École Polytechnique Fédérale de Lausanne

A study on the overhead of memory-tagging in compression libraries

by Edouard Michelin

Bachelor Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Andrés Sanchez
Thesis Supervisor

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

June 10, 2023

Acknowledgments

I would like to extend my sincere gratitude to my supervisor, Andrés Sanchez, for his assistance throughout this project, and for his understanding during the times when I could not work on the project.

Lausanne, June 10, 2023 Edouard Michelin

Abstract

Intel *Memory Protection Keys* (MPK) is a hardware primitive that allows adding thread-local permission restrictions on group of pages. Although being acknowledged, MPK is a fairly new technology which is not extensively used on the application side.

Zlib Memory Protection (ZMP) is a lightweight wrapper of the zlib compression library that aims to offer heap-based memory isolation between zlib and the rest of the application using it.

Through an ampirical analysis, we run two versions of ZMP implementing two different ways of applying page-based protection, (1) using the `mprotect()` system call, and (2) using MPK. We then use minizip (a portable zip & unzip library using zlib) as our test application and display and compare the measured overhead introduced by these isolation techniques.

Our evaluation shows that the MPK-based ZMP-enhanced version of minizip carries a negligible performance overhead (< 0.5%) compared with the unaltered version, whereas the `mprotect()` version has been found to suffer from performance impact ranging from 5 to 25% with time spent in kernel space being up to 2.45x higher.

We also examine alternative compression libraries such as ZStd and discuss the portability of ZMP to other libraries.

Contents

Chapter 1

Introduction

When performance is a core element of an application or system, softwares are often written in performant but memory-unsafe languages (C, C++ or Assembly). However, this performance comes at the cost of possible exposure to memory bugs which can be hard to find and can be used to craft various attacks. According to recent reports, these memory vulnerabilities accounted for a noteworthy proportion of all bugs indentified within specific organizations. In particular, Android's Jeff Vander Stoep and Chong Zhang reported in a blogpost that more than 75% of the bugs discovered in Android were attributed to memory issues [7]. Similarly, Google and Microsoft indicated that 70% of analyzed vulnerabilities were memory-related [2, 11].

A possible strategy to mitigate the impact of these exploit lies in the isolation of the componenents within an application, which is traditionally done in an inter-process manner, involving the spawning and handling of child processes and *inter-process communication* (IPC). Nonetheless, this approach is resource-intensive [11] mainly because it requires the OS to take over. Conversely, there are usually no restrictions on memory-accesses happening inside the same process. For example, an application using a library that is vulnerable to *Out-Of-Bound* (OOB) access could leak sensitive information stored in a region that the said library may never need to access in the first place.

This is where intra-process memory isolation can greatly improve the situation; a setting in which we are particularly interested in the potential benefits of memory tagging. For this purpose, Intel's MPK provides a means to accomplish this by allowing a process for specific control over its own accesses to memory, which is done by tagging pages with a so-called protection key and controlling the access-restrictions inherited by the key via the *Protection-Key Rights for User pages* (PKRU) special register [6].

Despite the fact that MPK is an acknowledged technology (being incorporated into recent Intel CPUs for both client and server, and for which Linux offers an API [8]), its adoption remains limited on the application side. While our analysis focuses on studying the performance impact of

MPK, it is important to note that its integration into applications requires to take other factors into consideration, albeit the overhead is a significant one.

In this project, we present ZMP. ZMP acts as a wrapper of the zlib compression library while offering an interface to allocate buffers of arbitrary sizes on a safe heap whose access is restricted when entering zlib. Such restrictions are enforced in two different manners: (1) using the `mprotect()` system call, and (2) using memory-tagging as implemented by MPK. The purpose of this project is to analyze, report, and compare the performance overheads introduced by the application of intra-process (or domain-based) memory isolation, with the two approches cited above, in the context of isolating a compression library, which is a memory intensive application.

Chapter 2

Background

2.1 Inter-Process Memory Isolation

Modern operating systems provide strong, hardware-enforced, memory isolation between processes. This inter-process mechanism ensures that a failing or compromised process will not impact other processes running on the same system, and makes the private (as opposed to shared) memory used by process A completely opaque to process B. Partitionning an application into multiple processes in order to isolate untrusted components from trusted ones could provide a secure way to isolate these memory regions. However, splitting an application into different processes comes with a certain impact on performance (see results from [9, 15]) that time-sensitive applications cannot afford. Highly considering the performance cost leads to look for other alternatives which avoid the need of context-switching.

2.2 Intra-Process Memory Isolation

Intra-process memory isolation is the concept of isolating components (also called domains) of the same process from each other by restricting their access to some memory regions. For this purpose, Linux offers the `mprotect()` system call which changes the access protections for the calling process's memory pages included in a given range [12]. Once set up, the program can continue in user-space and any illegal access will trigger a SIGV. Yet, as modification of the *Page Table Entry* (PTE) requires privileged access, there is no other way to apply these protections but to call `mprotect()`, which will naturally lead to a context-switch. In the context of intra-process memory isolation, particularly when there is tight interactions between domains, domain-based context-switches[1] can happen at a high rate and will need the intervention of the OS every time in order to change permission in PTEs. Newer techniques like Intel's MPK allow these changes to occur in user-space, thus removing the overhead resulting from context-switches.

2.3 MPK

In its Skylake architecture, Intel introduced a new way of updating the permissions assigned to a group of pages, without requiring modification of the PTEs. This means that access-restrictions applied to a group of pages, grouped under the same *protection-key* (pkey), can be modified while staying in user-space. All this is done by first getting an available protection-key (with the `pkey_alloc()` system call) and then by attributing this key to individual pages — i.e., the granularity of MPK, often called the granule when referring to memory-tagging, is a page — (with the `pkey_mprotect()` system call), which is often refered to as tagging the pages, and modifiyng the permissions that come with the key. The key is stored in four previously reserved and unused bits in the PTE. This means that we can assign up to sixteen protection-keys. In reality, the key 0 is reserved and used as a default value which only leaves fifteen keys to the programmer. Once a page has been tagged, the programmer can change the restrictions inherited by the key by writing to a special register (making these changes inherently thread-local) named PKRU, containing restrictions information for the sixteen protection-keys. To this effect, MPK introduces two new unprivileged instructions to read (`RDPKRU`), and write (`WRPKRU`) to the PKRU register. PKRU is a 32-bit register which holds restriction values (*Write-Disable* (WD) and *Access-Disable* (AD)) for all the sixteen keys; a quarter of its structure is shown in Figure 2.1. It is the combination of sixteen pairs of (WD|AD) bits with bit `2*i` being the AD bit of pkey `i`, and bit `2*i+1` its WD bit ($\forall i \in [0, 15]$), meaning that if `PKRU[2*i] == 1` then no data accesses on the pages tagged with pkey `i` are permitted, whereas if `PKRU[2*i+1] == 1` then user-mode write accesses are denied [6]. Supervisor-mode write accesses depend on additional conditions which will not be stated here as it has no purpose for the scope of this project.

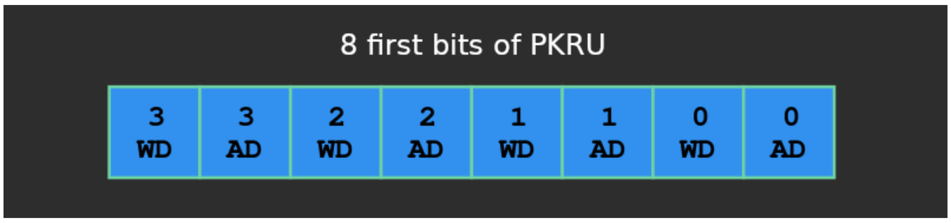

Figure 2.1: Schema of the eight first bits of the PKRU register [1].

Other system calls and interfaces are provided by the Linux kernel, all of which are displayed in Table 2.1. As measured by Soyeon et al., the `mprotect()` system call takes about 1,094 CPU cycles

[1] Domain-based context-switch from domain A to domain B refers to the process of changing permissions on the group of pages belonging to A and B so that the correct accesses are in place.

to complete whereas `pkey_set()` only needs 23.3 [13]; meaning that a single change in the access permission of a page with `mprotect()` is fifty times slower than the equivalent [2] using MPK.

Name	Cycles	Description
`pkey_alloc()`	186.3	Allocate a new pkey
`pkey_free()`	137.2	Deallocate a pkey
`pley_mprotect()`	1,104.9	Associate a pkey with a page
`pkey_get()`	0.5	Get the access rights of a pkey
`pkey_set()`	23.3	Update the access rights of a pkey

Table 2.1: List of MPK system calls and standard library APIs. [13].

2.4 Zlib

Zlib is a general-purpose, lossless data-compression library providing in-memory compression and decompression functions [3]. It is by essence a CPU-intensive and memory intensive application, hence having a large attack surface. Being wildly used (in Debian, iOS, OpenSSL, and nginx just to cite a few) it is the ideal target for attackers. Some of the applications using it are time-sensitive applications which could not afford one of their components to suffer from a high performance penalty, even at the cost of better security. Being a memory intensive library where speed is (for some) central to its effectiveness, it is a perfect candidate for being isolated in the fastest manner possible. It uses two stream data structures, one for zlib compression/decompression and one for gzip; these data structures are the only two for which both the main application and zlib need write access.

[2]Here, equivalent is in the context of a single thread, pkey's permissions being thread-local where mprotect() is process-wide.

Chapter 3

Design

3.1 Original idea

The original idea was to isolate the zlib compression library from the application by splitting the process memory into two regions, the `SAFE` one and the `SHARED` one. The `SHARED` region was to be the one used by zlib and the `SAFE` one was the one used by the rest of the application, for which zlib would have either no access or read-only access. ZMP would therefore consist of a library that acts as an intermediary layer between zlib and the main application, bridging them together, as shown in Figure 3.1. In order to later compare their performance but also to realize the different challenges that would arise from switching from one system to the other, we wanted to first implement the memory isolation using `mprotect()`. Once correclty implemented, we would make use of MPK.

3.2 Isolating the heap

Taking into consideration the granularity of protections, isolating the heap was quite straighforward: we needed to allocate the memory from the two regions in two distinct memory areas so that `mprotect()` could not be called accidentally twice on the same page. For these two regions, we decided to first create a `SAFE` heap, (for which all allocations would be made via a custom memory allocator exported by ZMP) so that we could more keep track of memory to be isolated from the untrusted component of the application. As both the main application and zlib have read-and-write access to the `SHARED` memory area, there is no need for keeping track of its allocations as it will not benefit from any access restrictions of any kind. We therefore consider any memory allocation made with the standard library allocator to be assigned to the `SHARED` heap. One thing we did not consider was multi-threading. Indeed, multi-threading with `mprotect()` must be carefully implemented so that we do not lock a memory region that is being used by another thread; the fact that MPK is

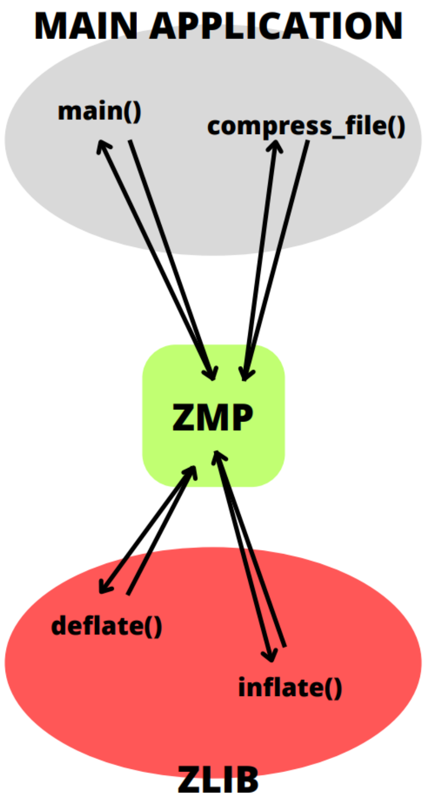

Figure 3.1: Schematic layout of ZMP in-between zlib and the main application.

thread-local makes it easier to handle multi-threaded applications. But due to the fact that this is a key difference between the two techniques, we decided not to assess it as we want to compare almost strictly equivalent versions of ZMP.

3.3 Isolating the main application

When the main application calls one of zlib routines, it is redirected to the ZMP's wrapper for that function (first step of Figure 3.2). Right before actually calling zlib's API, the wrapper locks the memory reserved to the main application (second step of Figure 3.2). When the function returns, restrictions on the locked region are lifted (third phase of Figure 3.2), making the program ready to return to the main application (fourth phase of Figure 3.2). In the meantime, any attempt to access the `SAFE` region made by the isolated component will raise a SIGSEGV signal, which will in turn terminate the process.

3.4 Inserting the wrapper

The wrapper is loaded dynamically but, in its current form, requires that some modifications of the source code be applied. These modifications include adding the header file after the zlib one and replacing all allocations to be made secure by the exported `zmp_safe_alloc()` routine made for this purpose.

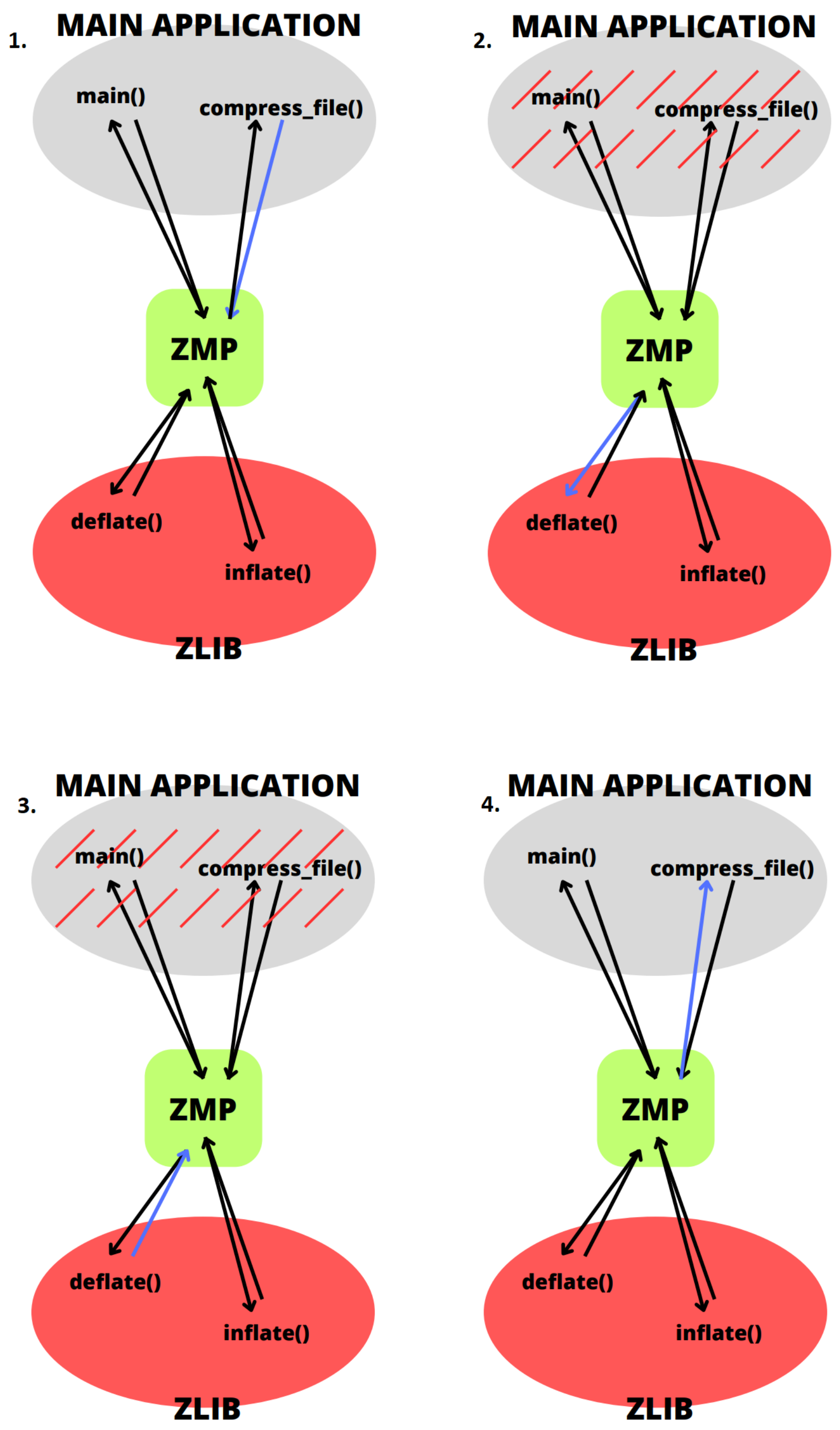

Figure 3.2: Schematic view of the four phases happening on a redirected call to a zlib routine by the main application.

Chapter 4

Implementation

At program start, ZMP's constructor function is called. Its role is to initialize the `SAFE` memory region, request a protection-key for the said region if necessary (with the `pkey_alloc()` syscall) and load the zlib's functions that will be wrapped. ZMP interacts with the main application in two manners. First, it exposes routines to allocate and free [1] memory on the `SAFE` heap. Second, it wraps the zlib API. See Table 4.1 for the list of functions exposed by ZMP. Note that not all zlib APIs are exported; this is because the current implementation only loads (greedily) routines that are used by the minizip application, presented in chapter 5, in order to not pollute the code with yet unneeded wrappers.

Name	Description
`zmp_safe_alloc()`	Allocates memory on the `SAFE` heap
`zmp_safe_free()`	Frees memory previously allocated on the `SAFE` heap
`zmp_crc32()`	Wrapper for `crc32()`
`zmp_get_crc_table()`	Wrapper for `crc_get_table()`
`zmp_deflateInit2()`	Wrapper for `deflateInit2_()`
`zmp_deflateBound()`	Wrapper for `deflateBound()`
`zmp_deflate()`	Wrapper for `deflate()`
`zmp_deflateEnd()`	Wrapper for `deflateEnd()`
`zmp_inflateInit2()`	Wrapper for `inflateInit2_()`
`zmp_inflate()`	Wrapper for `inflate()`
`zmp_inflateEnd()`	Wrapper for `inflateEnd()`

Table 4.1: ZMP's API. First group of rows are functions for allocation interactions, second group are wrapping functions of the zlib interface.

[1] In the current implementation, freeing memory allocated on the SAFE heap has no effect.

4.1 ZMP's heaps

In ZMP, heaps are represented by a simple structure, shown in Listing 4.1. Its allocation process is very primitive as memory buffers (or allocation blocks) are always allocated on top of each other, demanding new pages from the OS when the current one has been completely filled, which will not be freed for the duration of the process. When new pages are requested with the `mmap()` system call, they are given read-and-write accesses. The only difference between the `mprotect()` and the MPK version is that MPK requires an additional system call to `pkey_mprotect()` in order to tag the newly allocated page(s) with the previously reserved protection-key.

```c
struct zmp_heap {
  void *base_address; // starting address of the heap
  size_t size; // max size of the heap
  size_t used; // currently used size
  int protection_key; // used only in MPK mode
  struct zmp_allocation *allocations_table; // unused
};
```

Listing 4.1: Internal structure containing custom heap's metadata.

4.2 Isolating zlib

The second type of interaction that the main application has with ZMP is completely transparent to it, meaning that zlib APIs are wrapped without needing to change any line of code in the main application, further more, ZMP's wrapping functions have exactly the same signature, except for the name. The wrapping and isolating process is the same for all zlib wrapped functions, at the difference of the return type; its implementation is shown in Listing 4.2. In order to isolate and release the `SAFE` memory region when entering and leaving the real zlib's interface, we make use of two internal routines, respectively, `zmp_mem_lock()`, and `zmp_mem_release()`. Actual implementation of these functions are different depending on whether the isolation mechanism is implemented with `mprotect()` or MPK. Regarding the former, we make use of the `mprotect()` system call for modifiyng the protection bits in the PTE, whereas for the latter, we only need to overwrite the PKRU register with new access rights for `SAFE` key. This marks the major difference between the two systems, as there is no intervention from the kernel in the second version. We expect to observe the consequences of such a difference in our evaluation.
Independently of the applied technique, any unauthorized access to the `SAFE` memory area will raise a signal, as displayed in Listing 4.3.

```c
int wrapper(params)
{
  zmp_mem_lock(); // locks the MAIN memory region
  int result = wrapped_function(params); // calls zlib's wrapped
   routine
  zmp_mem_release(); // remove restrictions on MAIN
  return result;
}
```

Listing 4.2: Example of isolating the MAIN region when calling untrusted zlib routine.

```c
int main()
{
  char *shared = malloc(10);
  char *private = zmp_safe_alloc(10);
  private[0] = '1'; // write-access permitted

  zmp_mem_lock(); // SAFE memory locked (No-Access or Read-Only)

  shared[0] = '1'; // write-access to shared region permitted
  private[1] = '2'; // write-access denied | fault
}
```

Listing 4.3: Example of a faulty program that attempts to write in locked SAFE memory.

Chapter 5

Evaluation

We conduct a comparative evaluation of the two isolation mechanisms by augmenting the minizip application with ZMP. Minizip is a small, portable zip and unzip library than can be used in the command line interface to zip and unzip files. Minizip is tightly bound to zlib, which is at its core. This makes it an ideal candidate for our evaluation, as it allows to perceive the raw performance of our wrapper. Moreover, being centered around zlib, it facilitates the process of adjusting the input in order to benchmark the wrapper under different but specific conditions. Through a series of tests, we aim to provide a thorough analysis that demonstrates the impact on performance of each system. We measure (1) an unaltered version of minizip which serves as a reference, (2) the `mprotect()` version of the ZMP-enhanced minizip application, and (3) the MPK version.

For that purpose, we had to slightly modify minizip; first, to augment it with our design, but also to allocate the stream structure (cf. section 2.4) on the heap. This is simply to have a more heap-centered design, although it is not a necessary change.

We ran our performance tests using the *perf stat* tool, which allowed us to collect detailed performance counters. More precisely, we collected the time spent in user-space, the time spent executing in kernel mode, and the total elapsed time. Our macro-benchmark, which is based on the minizip's test suite, is constructed as follows: (1) remove the archive, (2) zip the content of `readme.txt` (the input) to `test.zip`, (3) rename `readme.txt` to a different name, and (4) unzip the content of `test.zip`. The steps stated above account for one round. For a comprehensive analysis, we run 100 rounds in order to have meaninful results. and display the average of all runs. We run the benchmark with different input sizes in order to measure the impact on performance of normal to excessive interaction between the two domains. It is important to note that, in the tested version of minizip, we allocated the input buffer on the `SAFE` heap, and that the locking mechanism only restricts write-access to the `SAFE` memory region.

Another important point to note is that when receiving a file to compress or decompress, minizip reads it in smaller chunks. Minizip will then call zlib's functions for each of these chunks, each call triggering two domain switches. Meaning that the bigger the input is, the more the zlib APIs are called. For example, zipping a file containing a simple `Hello World!` message will require 5 of these calls, a 32kiB 9, whereas an approximately 100MB-long message will require no less than 12500 calls. This gives us a good insight on how to measure excessive interaction between zlib and the main application, i.e., excessive amount domain switches.

5.1 First results

We run the our workbench test with three different inputs of size, 32kB, 100MB, and 1GB. We display the results obtained with inputs of size 100MB and 1GB in Figure 5.1. We decide not to include the results from the smaller input as they were too fast to be meaningful. We argue that at such speed (< 5ms), the impact of other events (e.g., from the OS scheduler, from hardware interrupts) is too high to perceive a real interest. Indeed, our measurements indicated performance impacts ranging from -1% to 5% when comparing the MPK version with the reference application, which is without considering excessively high runtimes that were slowed down by an order of magnitude. However, we noticed a trend averaging around 4-5%, which could be explained by the initialization of ZMP, having a higher, noticeable weight.

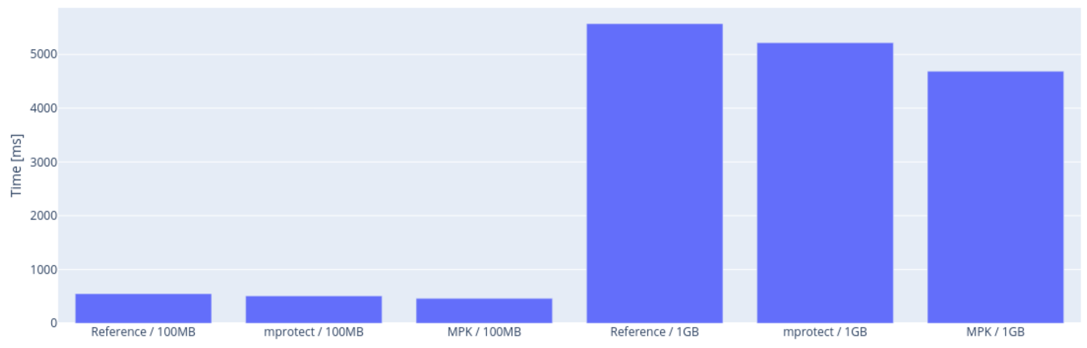

Figure 5.1: Average runtimes for the reference application, the mprotect, and the MPK versions with input size of 100MB for the left and 1GB for the right part.

Figure 5.2 displays the average time that each instance spent in kernel-space and were measured in the same run as the ones shown in Figure 5.1. Time spent in kernel mode is an important metric of our project as it exhibits a key advantage of using memory tagging, MPK in our context, for partionning a process into multiple regions.

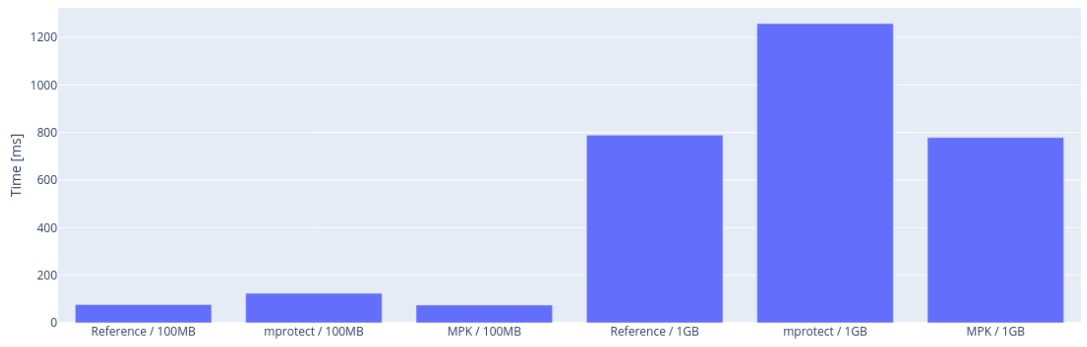

Figure 5.2: Average time spent in kernel-space for the reference application, the mprotect, and the MPK versions with input size of 100MB for the left and 1GB for the right part.

5.2 Adding large allocations

Our analysis found that, for a 100MB input file, 70 to 95% of the heap-allocated memory goes to the `SHARED` region. The results showed above being quite promising, we wanted to test if they held in extreme conditions. We therefore decided to further modify minizip by allocating 4MiB, 128MiB, and 1GiB on the `SAFE` heap prior to any interaction with zlib; which would make a significantly higher amount of pages to manage. Results of these benchmarks, with a 100MB input, are shown in Figure 5.3.

In the worst case, it has been found that the `mprotect()` version of ZMP introduced a performance overhead of 25% compared to MPK, with time spent in kernel-space being 2.45x higher. The `mprotect()` syscall accounting for 74% and 53% of the time spent in system-space for the compression and decompression part respectively, followed naturally by `read()` and `write()`.

Results displayed in Figure 5.3 show how insensitive MPK is to the number of pages that need to be protected, for a fixed input. In comparison, `mprotect()`'s performance degrades as the amount of pages in the `SAFE` memory region increases.

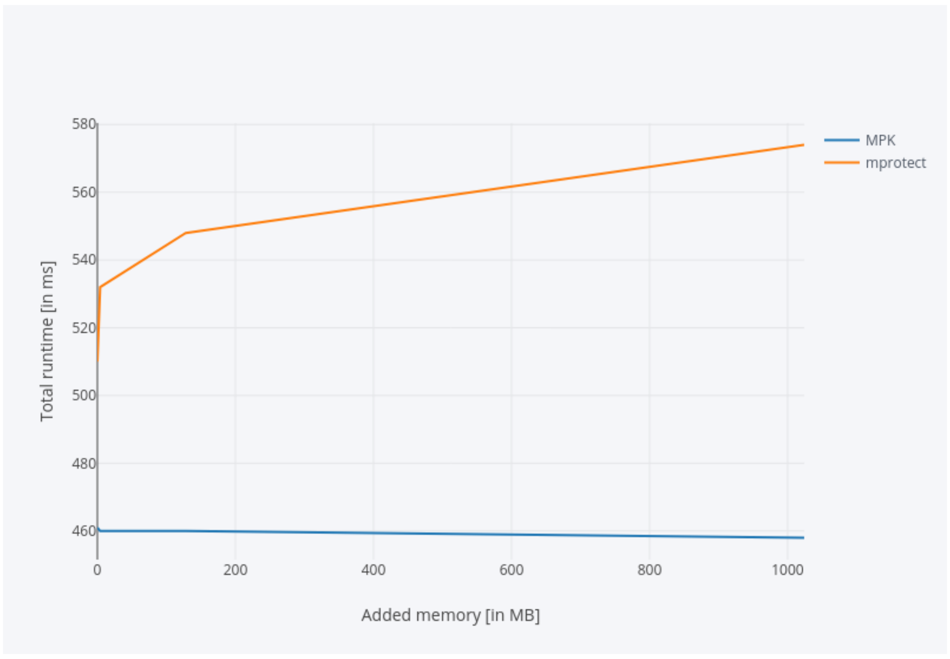

Figure 5.3: Average runtimes of the workbench for mprotect and MPK with increasing allocation size in the SAFE region, and a 100MB payload. MPK displays an almost constant 460ms runtime, whereas mprotect suffers from a 13% overhead in the assessed range.

Chapter 6

Discussion

6.1 Memory isolation with mprotect

Following the results presented in chapter 5, `mprotect()`-based intra-process memory isolation seem to rapidly reach limitations in terms of performance. While being a proven technique, it suffers from a non-negligible performance overhead, strictly increasing with input size [1]. In term of performance only, we argue that `mprotect()` is not a scalable way to enforce domain-based restrictions.

6.2 MPK, the new alternative?

We now need to verify, from our own evaluation, the validity of MPK in the context of intra-process memory isolation. As observed, a light interactivity between the main application and the wrapped library, especially in the context of an application living around said library, will make the ZMP initialization perceivable in the runtime metrics. However, results have shown, particularly in Figure 5.3, that MPK-based memory-isolation is a very scalable feature. Making it a good candidate for isolating libraries with either a high interaction with the rest of the application or a large set of data to encapsulate.

[1] Here, the input size denotes the amount of pages whose corresponding PTE will have to be modifed by the system call.

6.3 Portability of ZMP

In the past years, new compression libraries have emerged, of which we can cite LZ4 or Zstd. Being public, memory intensive, libraries written in an unsafe language, they are at high risk of presenting memory vulnerabilities and could therefore benefit from memory-tagging. In an attempt to evaluate the portability of ZMP, we examined Yann Collet's Zstd. At first sight, Zstd seems to be easily wrappable by a specially dedicated version of ZMP. However, Zstd's interface proposes what they call *Advanced experimental functions* which should not be dynamically linked. This defeats one of ZMP's principle which is to be as transparent as possible. Beyond this aspect, wrapping Zstd's API should not pose any problem.

6.4 Future work

6.4.1 Heap allocator and deallocator

As presented in section 4.1, heaps allocations are implemented in an utterly primitive manner. First and foremost, allocated blocks cannot be freed, which should not be a problem if `SAFE` memory should be kept for the lifespan of the application, but can rapidly become an issue otherwise. In a library whose purpose is to provide performant memory isolation, it appears as a necessity to make such feature available.

6.4.2 Isolating the stack

Functions that are restricted from writing or reading to the `SAFE` memory area should not be able to access the stack frame of trusted routines. The original idea for enforcing this additional protection measure consisted of padding the stack until the stack pointer is page aligned, so that we could protect the whole stack up to that point when switching domains. The question that persisted was, how to efficiently and precisely get the address of the bottom of the stack so that restrictions are correclty applied. After realizing that we were allowed to read and write to x86's `rsp`, and `rbp` registers, this questions became way easier to answer: reading the content of `rbp` in a constructor function would give good insight on the page the stack is starting from, and overwriting the content of `rsp` would allow us to efficiently redirect the stack to a `SHARED` stack, while enforcing access restrictions to pages in the range `[base_stack_addr, rsp_copy]`. However, due to our work focusing more on heap-based accesses, we decided not to implement this feature for the moment.

6.4.3 Large scale application as a test case

While minizip suited our needs for a highly configurable evaluation, it remains that its sole purpose is to use the wrapped library. An unresolved question is how much of a performance overhead there would be on large scale application, whose function does not revolve around zlib. As stated in section 2.4, there are numerous large scale, real-world applications using zlib, which could benefit from a performant way of isolating their sensitive components.

6.4.4 A more general library

In section 6.3, we discussed the potential portability to other compression libraries. However, there are more use-cases that could benefit from memory-tagging. Isolation of untrusted components has been an important topic in the field of software engineering and security for years, different approaches have been used in the past decades [9, 15] and many novel techniques are coming to light, which are already using or could use memory-tagging [4, 5, 10, 14].

Chapter 7

Conclusion

We have demonstrated that the `mprotect()` system call cannot be used in a scalable implementation of intra-process memory-isolation, primarily because of its internal functionnality, being a system call. Intel's MPK, on the other hand, provides very performant (cf. Table 2.1) means to modify page-based permission restrictions whilst remaining in user-space.

Evaluation of the MPK-enhanced version of our library has exhibited very promising results, even under heavy loads, with a negligible performance overhead (< 0.5%); at the exception of instances performing minimal operations, during which initialization steps may incur a perceptible overhead ranging from 0 to 5%, where the runtime is less than 3 milliseconds.

In conclusion, the incorporation of memory-tagging for intra-process memory isolation, particularly through the utilization of Intel MPK, presents a promising opportunity to mitigate memory-related vulnerabilities. Through the evaluation we conducted, it becomes evident that such approaches can be effective, albeit with certain considerations regarding security. As applications continue to demand higher security, the insight gained from this study can serve as a foundation for further exploration and optimization.

Bibliography

[1] Charly Castes. *Diving into Intel MPK.* URL: `https://charlycst.github.io/posts/mpk/`.

[2] Microsoft Security Response Center. *We need a safer systems programming language.* 2019. URL: `https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/` (visited on 06/07/2023).

[3] Jean-loup Gailly and Mark Adler. *zlib 1.2.13 Manual.* URL: `https://zlib.net/manual.html`.

[4] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. "Enclosure: Language-Based Restriction of Untrusted Libraries". In: (2021).

[5] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. "Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries". In: *USENIX Annual Technical Conference.* 2019.

[6] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual.* 4.6.2 Protection Keys.

[7] Android Security & Privacy Team Jeff Vander Stoep and Android Media Team Chong Zhang. *Queue the Hardening Enhancements.* 2019. URL: `https://security.googleblog.com/2019/05/queue-hardening-enhancements.html` (visited on 06/07/2023).

[8] The Linux Kernel. *Memory Protection Keys.* URL: `https://www.kernel.org/doc/html/next/core-api/protection-keys.html`.

[9] Douglas Kilpatrick. "Privman: A Library for Partitioning Applications". In: 2003.

[10] Volodymyr Kuznetsov, Làszlò Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-Pointer Integrity". In: *11th USENIX Symposium on Operating Systems Design and Implementation.* 2014.

[11] Google LLC. *Memory Safety.* 2018. URL: `https://www.chromium.org/Home/chromium-security/memory-safety/` (visited on 06/07/2023).

[12] *mprotect(2) — Linux manual page.* URL: `https://man7.org/linux/man-pages/man2/mprotect.2.html`.

[13] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)". In.

[14] 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*. 2019.

[15] Yongzheng Wu, Sai Sathyanarayan, Roland H. C. Yap, and Zhenkai Liang. “Codejail: Application-transparent Isolation of Libraries with Tight Program Interactions”. In: *European Symposium on Research in Computer Security*. 2012.