

Integrating address space layout randomization
and make it compatible with memory
de-duplication in Unikraft

Master Thesis



University of Liège - School of Engineering and Computer
Science

Author: Terry Loslever

Supervisor: Prof. Dr. Laurent Mathy

Co-Supervisor: Gauthier Gain

Master's thesis carried out to obtain the degree of Master of Science
in Computer Engineering

Academic year 2021-2022

Acknowledgement

First, I would like to express my gratitude towards Laurent Mathy who gave me the opportunity to work on this pioneer subject on, unikernels. This project helped me learn and discover things that I had never thought about.

Then, I would like to thanks Gauthier Gain, my co-supervisor, who has always been available when needed and gave me really precious advice. Without him, i probably wouldn't have had half of the brilliant ideas that drove me to the complete solution.

Further, I am addressing my affectionate greetings towards the whole Unikraft community which helped me solving problems or bugs and especially Dinca Daniel for the time discussing ASLR implementations.

Finally, I would like to dedicate this thesis to my family, particularly to my mother and grand-father, who always encouraged me to push my limits further.

Abstract

During the past years, people's online services usage kept growing which increases the load on the servers of cloud services and content distribution network. Those servers often run on containers that run on top of monolithic operating systems or simply on monolithic operating systems which embeds libraries, abstractions and codes that may not be needed nor be used by the application it runs.

Unikernels give the opportunity to the developer to build a specific operating system that contains only features that will be further used by the application and run it directly on top of the hypervisor. Furthermore, simplifying the operating system, on which the application runs, often results in a performance gain.

However, if unikernels were to be used more frequently by the industry, we have to be certain that they are as secure as the other technologies available on the market. Throughout this thesis, we have implemented address space layout randomization inside Unikraft in order to make memory related vulnerabilities harder to exploit.

Nevertheless, address space layout randomization comes at a cost which is its memory usage. We addressed that problem and found a way to mitigate the overhead : through page sharing between the unikernels thanks to indirection tables, which were implemented in two different manners. In the first, problematic instructions were set in a table that is appended directly to its corresponding library while the second creates a global table at a specified address that holds instructions from every libraries. Gauthier Gain, the co-supervisor of this thesis, implemented the appended method thus this thesis addresses the implementation of the global table and the comparison between the two manners.

Finally, we compared the performances of our address space layout randomization with Unikraft's previous implementation of it, and we discussed the two indirection methods. We came to the conclusion that the appended tables gave satisfying results when there was enough images running on the hypervisor, while the other was not giving any memory savings due to the constraints induced by the x86 64 bits CPU architecture.

Contents

1	Introduction	1
1.1	Address Space Layout Randomization	3
1.2	Unikraft	3
1.2.1	Running Unikraft KVM	4
1.2.2	ASLR In Unikraft	4
1.2.3	Memory De-duplication In Unikraft	6
2	Theoretical Background	7
2.1	Memory Layout	8
2.2	Calling convention	9
2.3	Operating System	9
2.3.1	Process	10
2.3.2	Virtual Address Space	11
2.3.3	Thread	11
2.4	Buffer Overflow Attack	12
2.5	Position Independent Executable	15
2.6	Object File	15
2.7	Compilation	15
2.8	Linker	16
2.9	Unikernel	17
2.10	Shannon Entropy	18
3	State Of The Art	19
3.1	Linux	20
3.2	Compilation	20
3.3	ELF loading	20
3.4	ASLR in Linux	20
3.4.1	Heap	21
3.4.2	Stack	21

4	Solution	22
4.1	ASLR Implementation	23
4.1.1	Unikraft Virtual Memory Mapping	23
4.1.2	Linking Script ASLR	26
4.1.3	Stack/Heap ASLR	32
4.1.4	ASLR scripts	34
4.2	Merging ASLR With Memory De-duplication	35
4.2.1	Creating a GCC Plugin	36
4.2.2	Using Daniel's Bootloader	37
4.2.3	Binary Rewriting	37
4.2.4	Patching Calls	40
4.2.5	Patching Jumps	42
4.2.6	Patching Mov And Lea	43
4.2.7	Page Sharing	44
5	Results	46
5.1	ASLR performances	47
5.1.1	Entropy	47
5.1.2	Text	47
5.1.3	Stack	49
5.1.4	Heap	50
5.1.5	Boot Time	51
5.1.6	Image Size	52
5.2	Comparing To Previous ASLR	53
5.3	ASLR-Deduplication Methods Comparison	55
5.3.1	Page Sharing	55
5.3.2	Selection	57
6	Conclusion	58
6.1	Conclusion	58
6.2	Further Work	59
6.2.1	ASLR	59
6.2.2	ASLR - De-duplication Compatibility	59
A	Appendix	60
A	Uniformity Verification	60
B	Requirements	60
B.1	ASLR	61
B.2	Deduplication	61
C	Machine	61

D	Linker Modifications	61
B	Bibliography	63

List of Figures

1.1	Two different Unikraft instances	6
1.2	Two different Unikraft with memory de-duplication	6
2.1	Program memory layout	8
2.2	Stack convention	9
2.3	Linux system representation	10
2.4	Frame page mapping	12
2.5	Vulnerable C code	12
2.6	Program compilation	13
2.7	Main's return address	13
2.8	Hidden function address	14
2.9	Compiler steps	16
4.1	Unikraft's memory overview	23
4.2	Unikraft's memory order before and after ASLR	25
4.3	Portion of link64.lds	27
4.4	Linker's modified text section	28
4.5	Unikraft's kvm folder	30
4.6	Resolving sections	30
4.7	Makefile's patch	31
4.8	Stack and heap randomization strategies	33
4.9	Indirection table methods	36
4.11	Call instruction indirection	40
4.12	Jump instruction	42
4.13	Problematic instruction comparison	43
4.14	Mov instruction indirection	43
5.1	Entropy measure on the text section	48
5.2	ASLR entropy on [12-20] w.r.t [20-28]	50

5.3	Entropy measure on the heap	51
5.4	Booting time ASLR vs no ASLR	52
5.5	Image size ASLR vs no ASLR	52
5.6	Indirection tables comparison	56
5.7	Memory on ten ftp images	56
A.1	Histogram on 100000 generated numbers	60
A.2	Linker.uk modifications	62

Chapter 1

Introduction

In 2020, O'Reilly surveyed their subscribers about the micro-service technology's adoption[\[1\]](#).

The latter pointed out that 46% out of 1502 respondents had their company moving at least one quarter of their solutions to micro-services while 15% are migrating the remaining (75% to 100%) of their solutions on them. Moreover, they also gathered information on the success rate of such migrations, thus the survey indicated that 91% of the participants' declarations ranged from "Some success" to "A complete success" regarding their use of micro-services.

This seems to indicate that micro-services architecture is already a well spread system in the IT business but this also shows that it is here to stay. More practically, these are run either in containers or monolithic operating systems depending on the company's choices but a new player could change that way of doing : unikernels.

Unikernels are modular operating systems built out of a set of libraries. Their goal is to embark only the necessary code base to run their application, which differentiates them from traditional monolithic (general purpose) operating systems such as Linux, that contains the code of all its features even if the application does not use them. Hence, unikernels benefit usually from lower image's sizes, lower memory footprint and better performances because they are built with the strict minimal code base that allows their payload to run.

Those pros are really interesting for cloud and micro-services providers, because unikernels fit the micro-service architecture which aims at splitting the solution in a set of small services acting together[\[2\]](#). Therefore, the providers could increase the performance of their servers by switching from general purpose operating systems or containers to unikernels.

As the industry will potentially deploy more and more unikernels in their solutions, we have to address security concerns in order to protect users from any attack on their data. This is even more important that we know that bugs are inevitable[3, p.521], the latter may cause vulnerabilities that could be exploited by an attacker. Thankfully, unikernels already partially solve that problem by their design, indeed, since they aim at reducing the amount of embarked code, it also reduces the amount of vulnerabilities (*a.k.a the attack surface*). However, the remaining ones can be mitigated by making them harder to exploit.

In this thesis, we discussed and implemented a well known solution named "address space layout randomization" in the Unikraft unikernel. This mechanism makes memory exploits harder by randomizing the memory of the kernel and/or the application. Then, we searched a way to make that technique compatible with memory de-duplication which had never been done before.

Finally, we showed that even if the current implementation of Unikraft imposes constraints on our solution, we managed to make address space layout randomization more secure than the one implemented by FreeBSD while having no impact on the boot time of the kernel. Further, we drew conclusions on which of the two different implementations could be integrated to Unikraft with regard to memory de-duplication and address space layout randomization's compatibility mechanism.

1.1 Address Space Layout Randomization

Address space layout randomization (a.k.a ASLR) is a security mechanism implemented in the operating system that aims at making vulnerable software harder to attack.

Programs with fixed memory display are easily exploited because of their predictability, and that is exactly what ASLR tries to avoid. The latter can be implemented from numerous ways but the most frequent one is made per execution (on Linux, Windows and Mac OS). This technique consists in letting the operating system set the executable at random addresses in virtual memory, making sure that programs' position is not predictable anymore.

However, ASLR only addresses memory corruption vulnerabilities such as buffer overflow, format string, memory leaks[4]. Its robustness depends on two factors : the probability distribution used and the bit range on which it operates. The higher the bit range is the better, while the probability distribution should be as uniform as possible.

Thus ASLR does not prevent the program to have a vulnerability, nevertheless it ensures that the latter will be hard to exploit because addresses are very unlikely to be the same across the programs.

1.2 Unikraft



Unikraft[5] is an open-source project driven by multiple universities, in which the university of Liège is actively working. It was partly funded by the European Commission research and innovation program.

The project aims at providing and developing an easy way to build our own unikernel for a broad range of applications. Hence allowing to take the advantages of the unikernels while minimizing some of their drawbacks, which are their specificity and difficulty to build.

Like traditional unikernels, Unikraft is organised in a set of libraries that are compiled together in order to create the final executable code. The latter distinguishes[6] 'internal libraries' in which hardware drivers, file systems, schedulers, tracing, etc are handled, from 'external libraries' that are either abstractions

¹<https://github.com/unikraft/unikraft>

such as : thread managing, TCP/IP protocol, sockets or ported applications like nginx and sqlite. Still, its core element is the platform and architecture dependent code[6]. Finally, the application is compiled as payload with the desired internal and external libraries[7].

Regarding performances, Unikraft running on Qemu (KVM) benefits from a 1.7x-2.7x performance gain (depending on the application being run) when compared to Linux guests. The most noticeable elements are the memory footprint and the application's performances. When it comes to Unikraft compiled with nginx, the latter saves up to 2 MB of memory when compared to Docker (Figure 4)[8] and reaches 2.68 million GET requests per second against 1.95 million for Docker.

This is directly due to Unikraft's design[9] : since the image is made as small as possible such that only the necessary is included and that it runs directly on the hypervisor thanks to KVM. The application is able to take advantage of this and it results in a gain in performance.

1.2.1 Running Unikraft KVM

Unikraft is made to run on different platforms : linuxu (linux user's space), kernel based virtual machine (*a.k.a* KVM, an hypervisor) and xen (hypervisor).

Throughout this thesis, we worked with Unikraft running on top of KVM which is accessed with Qemu[10]. In order to run an Unikraft image, the following can be entered in the terminal,

```
qemu-x86-system (-enable-kvm -cpu host) -kernel
UNIKRAFT_IMAGE (-append args)
```

Where `UNIKRAFT_IMAGE` is the path leading to the file, `-append` can be used to pass arguments to the application and `-enable-kvm -cpu host` places the kernel on top of the hypervisor, which means that the latter runs in bare metal.

1.2.2 ASLR In Unikraft

ASLR had been already implemented by DINCA Daniel[11] as a thesis subject, its internal working is the following : since some assembly files could not be turned to PIE because they are CPU configuration files. Daniel created a minimal Unikraft image called the "bootloader"[11, p. 11] that includes assembly, platform and architecture dependent files[11, p. 18] which are statically and no PIE compiled. The purpose of such method is to configure the CPU and then load the "true" PIE compiled Unikraft image at a random memory offset, creating thus ASLR.

This method comes with some drawbacks,

- Creates image's size overhead, since the minimal set is loaded twice, once in the bootloader and once in the loaded image.
- The method is not flexible in the sense that the whole ELF file is moved at a unique random address in memory. An attacker that finds this base address and knows the offset of his target in the ELF could overcome the ASLR.

Regarding performances, this version of Unikraft has an increased boot time and slightly worst performances[11, p. 35-36] when compared to the no ASLR version of Unikraft. This is explained by the fact that the ELF was made PIE which was necessary to enable ASLR. This adds an additional level of indirection through plt and got tables.

1.2.3 Memory De-duplication In Unikraft

The problem with unikernel is that multiples instances of the same library are loaded in memory, when they are running on top of the same hypervisor, due to the fact that they are all statically linked. This linking method should be kept because it allows to enforce the isolation principle between the kernels by running them on top of an hypervisor. However, this creates some unnecessary overhead that dynamically linked programs solve with shared libraries which are loaded once in virtual memory and then shared among the applications that use them.

Gauthier Gain worked on memory de-duplication which allows to limit the overhead[12].

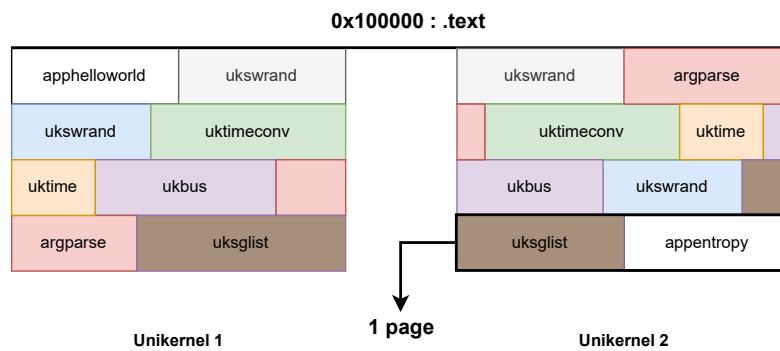


Figure 1.1: Two different Unikraft instances

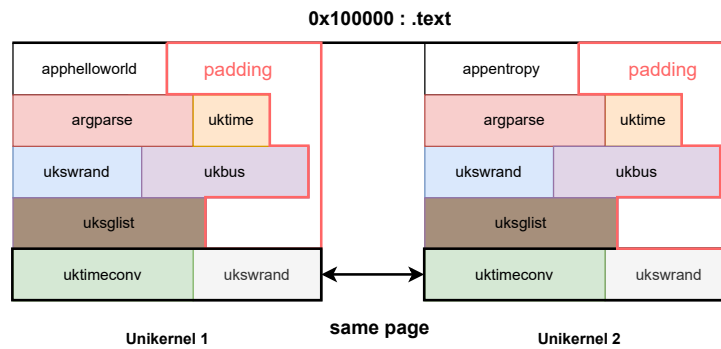


Figure 1.2: Two different Unikraft with memory de-duplication

The goal is to standardize the memory layout so that the hypervisor can share the same memory page to multiple running kernels. To do so, Gauthier's solution assigns a particular virtual memory position to each Unikraft's libraries which causes them to be at the same place in every image built. This allows the hypervisor to reuse the same page for multiple kernels thus reducing memory footprint, since one page is loaded instead of multiple.

This comes at the cost of heavier virtual images since libraries are aligned on pages and may not fill them completely. This method uses "kernel same page merging" which is a feature of KVM.

Chapter 2

Theoretical Background

For the reader convenience and as a reminder, the following section will review all the theoretical aspects that may, at some point, be addressed by this thesis. The elements are explained broadly but if the reader wants to get a deeper understanding, he is invited to have a look at the original papers.

2.1 Memory Layout

In modern OSes, a process represents a program loaded in memory which has been given a portion of the virtual address space. The structure of this address space consists in the following elements depicted in Figure.2.1.

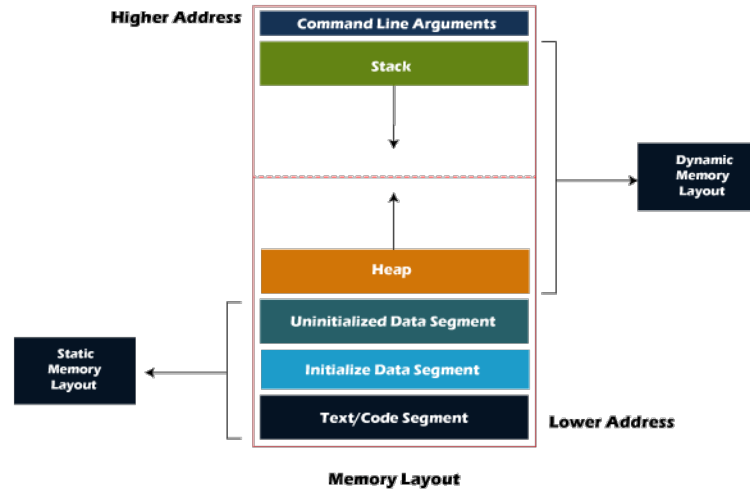


Figure 2.1: Program memory layout
1

The memory is split in 2 parts : the dynamic memory that can be allocated as the program runs and the static memory that is given at program start-up and can't grow any further.

In the first one resides the stack and the heap. The stack grows to lower addresses as we do function calls or allocate local variables while the other is responsible to store the dynamically allocated memory and grows towards higher addresses. Depending on the programming language, the following keywords allocate variables on the heap : `new` (Java,C++), `malloc/calloc` (C), ...

The second one holds the code segment which is the machine code that the CPU has to execute in order to run the program. Some OSes set this portion of memory in "read only" mode in order to avoid attacks that could rewrite the program's instructions. Then comes the initialized data segment which is the portion of memory that holds the global or static variables of the program from which a value is already assigned in the program's code. The only difference between the uninitialized and initialized data segment comes for the fact that variables have been assigned a value or not in the source code[13].

For efficiency reasons, most OSes share the code segment of some often used programs such as text editors,shells,..., in order to save memory space.

¹<https://www.javatpoint.com/memory-layout-in-c>

2.2 Calling convention

When functions are called, the caller places the arguments and the return address onto the stack [2.1] before the call[14]. Arguments are from convention, passed from Right-to-Left which means that the last argument will be pushed first.

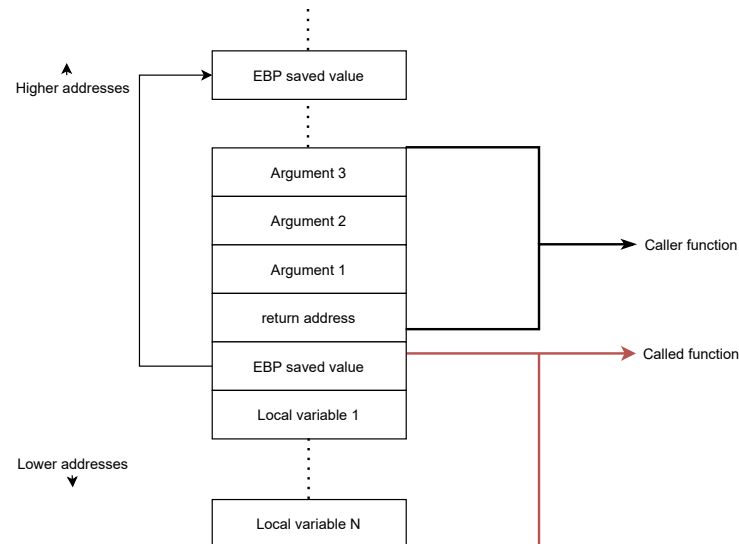


Figure 2.2: Stack convention

Figure.2.2 depicts what has been said upper with a function that takes 3 arguments. It's worth noting that the `call` instruction in x86 assembly is responsible to push the return address, which is the following instruction after itself, before jumping to the called function.

Then, the called function pushes the current stack base pointer value (EBP), changes EBP value to the current stack pointer value (ESP) and finally allocates memory for its local variables.

2.3 Operating System

An operating system (*a.k.a OS*) is a "software that acts as an intermediary between user programs and the computer hardware"[15].

The latter aims at using efficiently the computer hardware by scheduling applications, caching data, providing services, allocating memory. It can be considered as the base block which allows the softwares to run on the computer. The OS creates also a set of abstractions such as processes, threads, files, file system, sockets, ... which is a non exhaustive list of elements that are created in order to use the hardware conveniently.

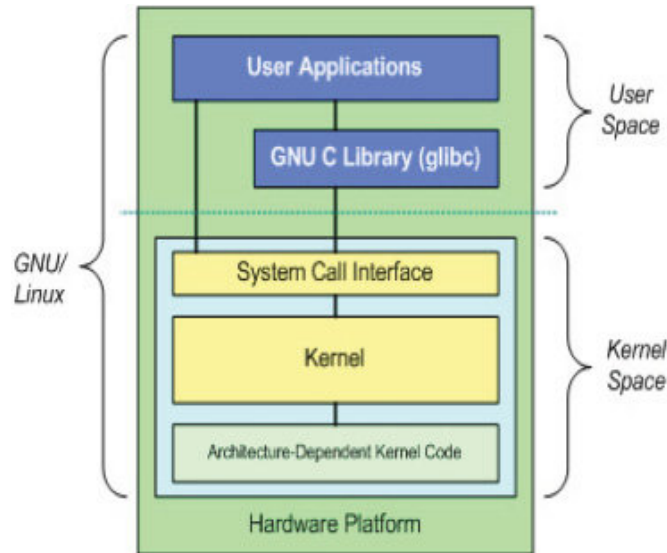


Figure 2.3: Linux system representation

2

Programming is also made easier because the OS provides a set of functions called "a programming interface". Those services can be called through an API called "Syscalls"[16] which are a set of functions that switches the system from user to the kernel space. The reader can observe the latter on Figure.2.3. This way a program can get access to the abstractions made by the OS that were described previously.

There exists various architectures : Monolithic (Linux), Microkernels (Mach, from which MacOS X is partially made), unikernels (Unikraft), hybrids, ...

They all include different abstractions and work in different ways.

2.3.1 Process

A process is an abstraction describing a program that has been loaded in memory by the OS and active. The reader can understand being "active" as currently being set on a core of the CPU or waiting to be scheduled on it.

Not only does the process contains the machine code, but it also holds a bunch of interesting data : process state, process number (pid), program counter, registers data, address space, open files, ... Which are all used by the OS for different purposes[17, p.86].

Every processes are independent from each other meaning that their address spaces are not shared and their memory is arranged as Figure.2.1 depicts it. However there exists no one-to-one mapping between a program and a process, indeed, some programs such as Google Chrome run a process for each tab. The

²https://www.researchgate.net/publication/276550505_Using_Network_Traffic_to_Infer_Hardware_State_A_Kernel-Level_Investigation/figures

latter shows that the same program can be loaded in multiple processes.[18]³

2.3.2 Virtual Address Space

Programs are not given access to true physical memory. Modern OSes implement what is called "Virtualisation" which allows to emulate a possible infinite memory[17, p.194]. It consists in a mapping between "pages" and "frames" that is held in the page table. Physical memory is split among multiples frames of fixed size which are contiguous blocks of memory, their size is usually 4KiB. However, recent instruction set architectures support multiple page sizes, which reduces the size of the page table.

Pages are blocks of contiguous logic memory which have the same size as the frames of the system. The mapping between the two is made thanks to the page table, held by the OS, which knows to what frame the page is stored.

This solution allows to spread the program's memory all across the physical memory without having him to notice it, because from the program's point of view it's like a block of contiguous memory.

The translation between the two types of memory is done by the memory management unit (MMU) of the CPU. Figure.2.4 shows a mapping between the page and the frame.

Most common OSes can be split in two memory spaces : the kernel space and the user space. The first one runs in privileged mode and has a **direct** access to the hardware, or at least thinks it does if it runs on top of an hypervisor[17, p.477-478]. The second runs in user mode and only has access to virtual memory addresses (*a.k.a VMA*) that was previously allocated to it by the OS. Any process going out of its allocated user space memory should eventually be detected by the OS and trigger an error.

2.3.3 Thread

An execution thread also called simply "a thread" [17, p.97-115] depicts an abstraction that tracks the execution state inside the process. Multi-threading thus occurs when a process holds multiple threads that runs concurrently different portions of its machine code. The OS achieves this by allowing the process to hold multiple stacks and registers' data copies in memory.

Threads live in a shared memory environment since they access the process' memory. Nevertheless, there exists thread local storage (TLS) that allows threads

³The following source talks about google chromium which is the source code on which google chrome is based. <https://fr.wikipedia.org/wiki/Chromium>

⁴https://en.wikipedia.org/wiki/Page_table#/media/File:Virtual_address_space_and_physical_address_space_relationship.svg

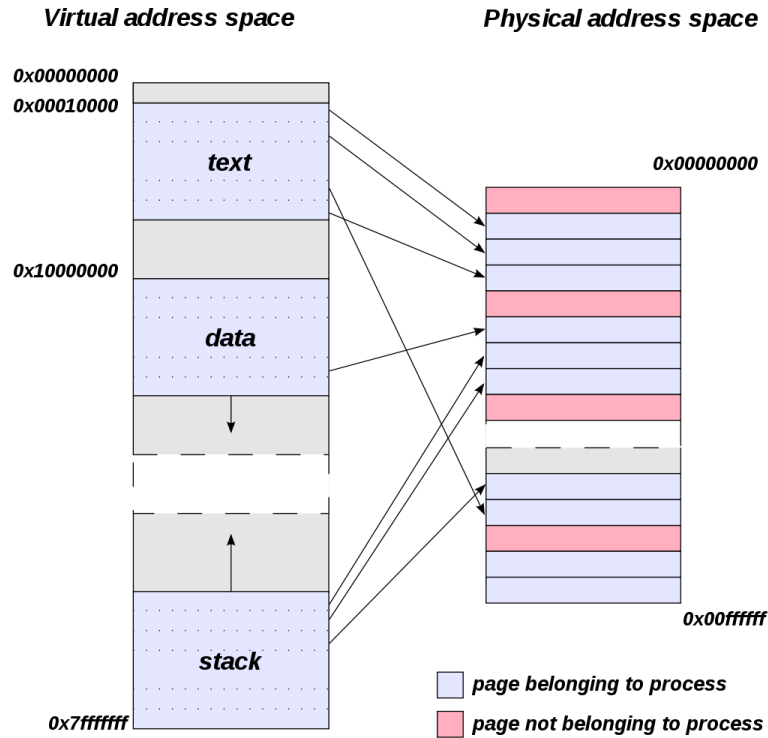


Figure 2.4: Frame page mapping

4

to keep data in non-shareable memory.

2.4 Buffer Overflow Attack

```
#include<stdio.h>
#include<string.h>

void root_privilege(){
    printf("You gained root access.\n");
}

int main(void){
    char password[10];

    printf("Enter the password:\n");
    gets(password);

    if(strcmp("LOVEASLR",password) == 0)
        printf("Hi user !\n");
    else
        printf("Wrong password. Try again !\n");
}
```

Figure 2.5: Vulnerable C code

This attack aims at either crashing a program or injecting malicious code in order to initiate a privilege escalation. The latter is made possible when the software writes data into a buffer without checking its size beforehand. So the intruder inputs a string long enough so that the program rewrites the return

address of the last function call, thus redirecting the program's execution towards the malicious code.

Figure.2.5 shows a simple C code that is vulnerable to such an attack. The software aims at greeting the user if he enters the correct password, otherwise tells him that he entered the wrong password. It also includes a secret function that prints a secret message, this message shouldn't be printed in a normal behavior since it is never called any function.

The attack will consist in modifying the address to which `main()` should return after its execution by the address of the function `root_privilege()`. To do so, we'll use the a vulnerability caused by `gets()` which naively copies the standard input to the given buffer without length verification, therefore one can rewrite the stack of the program.

For the attack to work, we have to compile the program with the flags from Figure2.6,

```

terrylos@terrylos: ~/thesis/thesis
(terrylos@terrylos)-[~/thesis/thesis]
(terrylos@terrylos)-[~/thesis/thesis]
$ gcc example.c -o example -fno-stack-protector -no-pie
example.c: In function 'main':
example.c:12:2: warning: implicit declaration of function 'gets'; did you mean 'fgets'? [-Wimplicit-function-declaration]
12 |     gets(password);
    |     ^~~~~
    |     fgets
/usr/bin/ld: /tmp/ccUC7tt4.o: in function 'main':
example.c:(.text+0x34): warning: the 'gets' function is dangerous and should not be used.

```

Figure 2.6: Program compilation

The reader can observe that gcc compiler warns the programmer that he should avoid using `gets()` function. Moreover, `-fno-stack-protector` allows to disable protections put by Linux on the stack such as canaries [19]. The other one, `-no-pie`, disables the position independent executable option, the reader can read more about it in its corresponding subsection [2.5].

[X] Disassembly (pd)	[Cache] Off	[X] Stack (pxq 256@r:SP)	[Cache] Off
0x00401155 b 55 push rbp		0x00007f583e629e4a 0x00007ffdd12a06a8 J.bX.....*	
0x00401156 4889e5 mov rbp, rsp		0x0000000013e629c27 0x00000000000401155 '.bX...U..@	
0x00401157 4883ec10 sub rsp, 0x10		0x00000000c0000000a 0x0000000000000000	
0x0040115d 488d3db80e00 lea rdi, str.Enter_the_password_	[1]	0x00000000000000000 0x00000000000000000	
0x00401164 e8c7feffff call sym.imp.puts	[1]	0x00000000000000000 0x00000000000000000	
0x00401169 488d45f6 lea rax, [rbp - 0xa]	[2]	0x00000000000000000 0x00000000000000000	
0x0040116d 4889c7 mov rdi, rax		0x00000000000000000 0x00000000000000000	
0x00401175 b830000000 mov eax, 0		0x00000000000000000 0x00000000000000000	
0x00401175 e8d6feffff call sym.imp.gets	[2]	0x00000000000000000 0x00000000000000000	
0x0040117a b840000000 lea rax, [rbp - 0xa]	[3]	0x00000000000000000 0x00000000000000000	
0x0040117e 4889c6 mov rsi, rax		0x00000000000000000 0x00000000000000000	
0x00401181 488d3da90e00 lea rdi, str.LOVEASLR	[0x4]	0x00000000000000000 0x00000000000000000	
0x00401188 e8b3feffff call sym.imp.strcmp	[3]	0x00000000000000000 0x00000000000000000	
0x0040118d 85c0 test eax, eax		0x00000000000000000 0x00000000000000000	
0x0040118f 750e jne 0x40119f		0x00000000000000000 0x00000000000000000	
0x00401191 488d3da20e00 lea rdi, str.Hi_user_	[0x4]	0x00000000000000000 0x00000000000000000	
0x00401198 e893feffff call sym.imp.puts	[1]	0x00000000000000000 0x00000000000000000	
0x0040119d eb0c jmp 0x4011ab		0x00000000000000000 0x00000000000000000	
0x0040119f 488d3da90e00 lea rdi, str.Wrong_password_Try_	[1]	0x00000000000000000 0x00000000000000000	
0x004011a6 e885feffff call sym.imp.puts	[1]	0x00000000000000000 0x00000000000000000	
0x004011ab b800000000 mov eax, 0		0x00000000000000000 0x00000000000000000	
0x004011ab c9 leave		0x00000000000000000 0x00000000000000000	

Figure 2.7: Main's return address

A quick look with `radare2`^[20] allows us to identify the return address of the `main()` function (framed in red). The return address is `0x7f583e629e4a` and is located on the stack at the address `0x7ffdd12a05b8`, as can be seen from [Figure.2.7](#).

```

;-- root_privilege:
0x00401142      55          push rbp
0x00401143      4889e5      mov rbp, rsp
0x00401146      488d3db70e00. lea rdi, str.You_gained_root_access.    ; 0x
0x0040114d      e8defeffff  call sym.imp.puts                      ;[2]
0x00401152      90          nop
0x00401153      5d          pop rbp
0x00401154      c3          ret

```

Figure 2.8: Hidden function address

At this state, the program still hasn't modified the stack base pointer and hasn't allocated the char buffer. There's also the hidden function at location `0x00401142` ([Figure.2.8](#)), it is the address towards which the program has to jump to print the secret message.

We set a break point after the `gets()` function in order to deduce how much characters we should put in the buffer before rewriting the return address.

We see that we have to rewrite 18 bytes of the stack before rewriting the return address. Therefore, entering 18 times "a" plus the hexadecimal address should unveil the vulnerability.

A simple python script such as `"print("a" * 18 + '\x42\x11\x40\x00\x00\x00\x00\x00')"` fed into the program with a piping : `"python3 script.py | ./example"` grants us access to the hidden function.

As planned, the message "You gained root access." appears on the standard output which means that the attack worked.

2.5 Position Independent Executable

A position independent executable (*a.k.a* *PIE*) is an executable in which the machine code can be placed anywhere in memory without needing any modification to the code. This is the exact opposite of absolute code that can only run at fixed memory addresses.

PIE works thanks to relative addressing which specifies the jump address with respect to the current instruction pointer position (RIP) [21].

The latter allows to share piece of code between multiple programs such as commonly used libraries, to do so, their addresses are resolved during run time in the plt/got tables that PIE executables contain. But this has a slight efficiency cost since we add an indirection, when compared to absolute code, in the calling process.

It also makes address space layout randomization (ASLR) possible since the OS couldn't place the machine code at random places if it was position dependent.

2.6 Object File

Object files can usually be found on computer with the ".o" or ".obj" extensions. It is an ELF file[22], containing additional information, that has not been linked yet and that can't be run by the OS. Moreover, useful meta-data generated by the compiler can be found : symbol held in tables that eases linking process by giving elements' type, name and length if it's a function. But also debugging and profiling information.

2.7 Compilation

In order to create the executable file (which is following the ELF format in Linux), the source code has to go through multiple steps in order to be run on the CPU. Those are depicted by **Figure.2.9**. The compilation step is the action which transforms a source code file written in a programming language such as : C, C++, GO, Java, Rust, ... To an object file. The underneath figure shows the diverse actions done by a compiler.

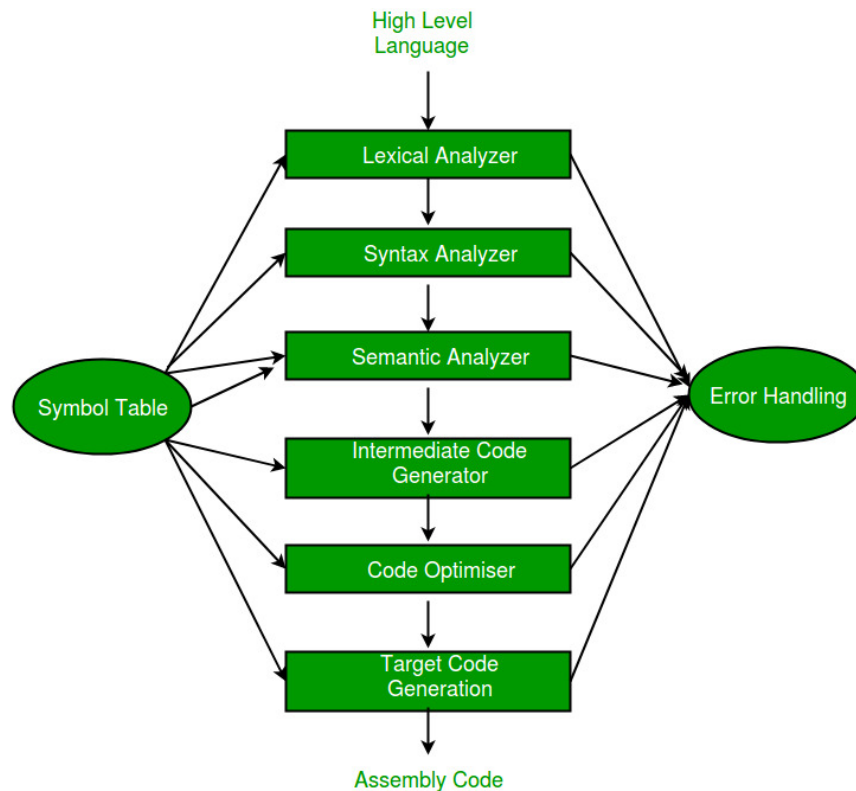


Figure 2.9: Compiler steps

5

During that process, the addresses assigned to the functions or pointers, by the compiler, are local symbolic addresses. They only make sense inside their file and do not yet refer to actual addresses managed by the OS. The addresses will only be resolved during the linking procedure, where their definitive values will be assigned.

As an example,

```
gcc -c foo.c
```

will generate the object file `foo.o` out of the C file `foo.c`.

Note that these are different from interpreted codes, which are compiled and linked on the run by the interpreter of the language.

2.8 Linker

Linking is the final step of the compilation procedure which generates the final executable file [23, chap.7]. It consists in loading all the object files [2.6] that are part of the software. Then the latter binds them together with libraries (if any are used) in a final file, where the symbols' addresses are mapped to addresses from the logical address space.

⁵<https://www.geeksforgeeks.org/phases-of-a-compiler/>

There exists two types of linking procedures : either static or dynamic. The first creates the executable as described upper but adds used libraries binary code into the executable.

The second allows to keep some undefined symbols in the executable file which are resolved when the latter is executed. Thus, we only have a reference to the used libraries which has multiple advantages :

- Allows to share the common libraries between the programs by only loading them once in memory.
- Can update the code of the libraries without necessarily compiling again all the programs that were using them.

2.9 Unikernel

Unikernels[24] are single memory space OSes meaning that there exists no distinction between the user and the kernel memory space. Moreover, this operating system is built to run a single application which means that everything runs, by default, in a single process. The system is compiled with its running application as payload and built with a set of strictly necessary libraries that will be used by the application.

This type of OS, despite being very specific, gives non negligible advantages:

- Secure environment : it brings the smallest possible set of libraries and thus code. Which decreases the attack surface for malicious softwares. When multiples unikernels run on the same hardware, the security of the whole system depends on the hypervisor since it tries to ensure that they are isolated from each other.
- Better performances : by recognizing only one memory space, the system does not have to switch between modes as it exists in traditional systems. Hence increasing the performances as the application gets direct access to the hardware.
- Fast start: since the OS image is as small as possible, the latter can be loaded in memory and executed rapidly (in the order of the 10's ms depending on the implementation)
- Low footprint: this is derived from the fact that the image is as small as possible. Thus, less function calls are made before running the application, which reduces stack's memory use.

Each time the reader will encounter the terms "traditional" or "general purpose" with regard to OSes, it refers to Monolithic kernel/Microkernels which are heavily deployed in desktop computers or servers.

Nevertheless, that is not the panacea since it comes with cons such as,

- Highly specific : the kernel is built to work on a single application, hence the OS does not fit regular desktop usage.
- Static : the OS can not be modified once built. If the user wishes to add new features, he has to recompile the kernel with its payload.

These advantages and drawbacks make them an arguably good choice for clouds solutions and micro-services.

2.10 Shannon Entropy

The entropy[25, p.45] characterises the amount of information that an event, described by a random variable, stores. Practically the rarer the event, the more information such an event contains.

It can be obtained thanks to that equation,

$$H(X) = \sum_{i=0}^N P(x_i) \log_2 \frac{1}{P(x_i)} \quad (2.1)$$

Where X is a discrete random variable, x_i its possible outcomes and $P(x_i)$ the probability of such outcomes.

If the random variable follows an uniform distribution, we can use the following : $\sum_{i=0}^N P(x_i) = 1$ and $P(x_0) = P(x_1) = \dots = P(x_N)$.

We get,

$$\sum_{i=0}^N P(x_i) \log_2 \frac{1}{P(x_i)} = \sum_{i=0}^N P(x_i) \times \sum_{i=0}^N \log_2 \frac{1}{P(x_i)} = \sum_{i=0}^N \log_2 \frac{1}{P(x_i)} \quad (2.2)$$

Practically in this thesis, entropy will be seen as "the amount of uncertainty that an attacker have about the location of a given object" which another way of describing it [26, p.10].

Chapter 3

State Of The Art

In this section, we will review how Linux implemented ASLR. However, the reader should keep in mind that Linux and Unikraft are really different OSes. In fact, Unikraft doesn't include process and kernel/user space abstractions while Linux does and also implements per process randomization. Hence the following sections should be considered as an example of ASLR's implementation.

3.1 Linux

Linux is an open-source operating system developed since 1991 that follows a monolithic design [17, p.720], it draws the line between kernel and user mode where the running programs are abstracted as processes [2.3.1].

Therefore, the latter has a way to load the ELF files in memory and execute them unlike Unikraft which doesn't abstract its running application, instead it considers the application as a part of its own code.

3.2 Compilation

In order to benefit from ASLR on Linux distributions, the user has to compile its executable as PIE [2.5] so that it can be placed anywhere in memory. If not, the produced executable will end up at the address given by the linker during the linking procedure.

3.3 ELF loading

First, Linux loads the ELF file in memory through its function `load_elf_binary` [27] which takes the `linux_binprm` structure as argument. Digging a bit deeper inside the code unveils some interesting things [28], the binary's structure possesses references to memory mappings through `vm_area_struct` that holds its pages and `mm_struct` which is the memory descriptor.

This function first checks the header field, verifying that it respects the format and extracts the information about the binary. Then it looks for an interpreter that fits it, once all the set-up is done, it flushes all traces of the same executable in the kernel thanks to `begin_new_exec`.

The loading process being done, it can continue to the mapping procedure which aims at mapping the file to its allocated memory segments.

3.4 ASLR in Linux

ASLR can be turned on or off by setting `randomize_va_space` macro to 0 [29], on a running Linux image this is feasible by writing in `/proc/sys/kernel/randomize_va_space`. More technically this results in rewriting the value of the Linux kernel macro that can be found in `mm.h` [30].

This macro is used in the source code here [27] and is checked before randomizing (or not depending on its value) the stack's top and heap's base addresses.

3.4.1 Heap

The heap is randomized first thanks to `setup_new_exec`, we can trace all the functions calls that leads to `mmap_base` which is the function that sets the heap at a random base address.

```
setup_new_exec -> arch_pick_mmap_layout -> arch_pick_mmap_base ->
mmap_base[31]
```

It is placed in memory based on a gap and a random number generated by linux's own randomization engine. The gap is computed from the maximal possible stack size derived from the task's size (in Linux' source code a task[32] is a process) and its current stack size. Then the final address is obtained by rounding to a page the difference between the task's size with gap and the random number.

3.4.2 Stack

Then, stack's randomization is made by adding a random offset generated by `get_random_long` which is defined again in Linux' engine to its default top address.

```
random_variable = (get_random_long()
& STACK_RND_MASK) << PAGE_SHIFT
```

The number is then limited by a mask and shifted so that the address is aligned to a page, as the reader can observe it upper. This new top address is then passed as argument to `setup_arg_pages` that initializes the stack.

In this Linux' implementation the mask is macro set at 0x7ff in x86.[33]

Chapter 4

Solution

The complete implementation will be split in two : ASLR alone and ASLR with memory de-duplication, which allows the user to configure the image at his/her will. The configuration is made through the *make menuconfig* command inside the build folder.

Regarding the first, a Python script adds randomness inside the linking script of Unikraft. This method allows to implement ASLR without adding any modification to the existing code, only the building procedure needs to be modified.

For the second, multiple approaches were considered and will thus be discussed. Rewriting the Unikraft binary by adding an indirection table was the only solution that worked.

There was two ways of implementing such a table. Either create a global one placed at a designated VMA, or appending an indirection table on each library of the kernel. Due to the great amount of work this involved, it was decided that we would go for the first one while Gauthier Gain would go for the second.

4.1 ASLR Implementation

Since Unikraft does not include and does not need a process abstraction by nature, one could think about another manner to implement ASLR than the one made by Linux. On one hand, the idea is to modify Unikraft's linker script before linking in order to modify the memory layout which will result in a multitude of different Unikraft images, despite having the same payload. On the other hand, the kernel will place its stack and heap at random addresses while booting, because the solution described upper only applies ASLR on the elements that are present in the ELF.

Hence, achieving ASLR without any modification on how the image boots and loads the application. This is deeper explained in the section [4.1.2].

4.1.1 Unikraft Virtual Memory Mapping

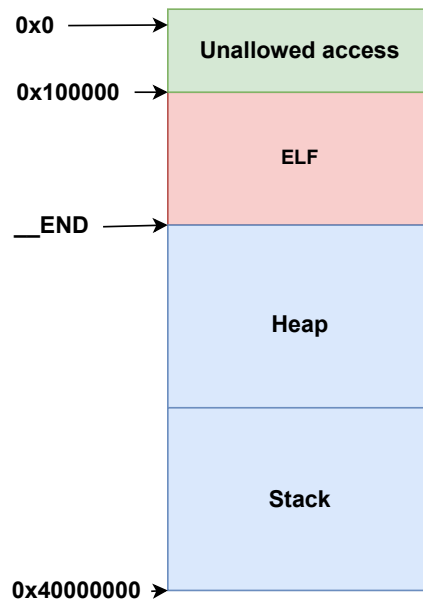


Figure 4.1: Unikraft's memory overview

First, let us have an overview on how memory is used by Unikraft. The Figure.4.1 shows that Unikraft runs on the 0x100000-0x40000000 addresses by default. It does not go higher than that (1 GB) because pages are not yet dynamically allocated and swapping is not implemented. That explains why Unikraft's memory is so restrained, it should be possible to fit it entirely in memory.

The addresses before 0x100000 are not mapped in the page table and left for Qemu. This address space is thus $\log_2(0x40000000 - 0x100000) \approx 30$ bits long, which indicates that the addresses before 0x100000 are not a great loss. However it could be great to use all that memory in order to increase ASLR's range.

The pointer `__END` points to the end of text section. The heap and stack's sizes are set by the user, the default stack's size is set at 4096×16 bytes which is 16 pages longs, as can be seen on the following code snippet[34],

```
#define __STACK_SIZE (__PAGE_SIZE * (1 <<
    __STACK_SIZE_PAGE_ORDER))
```

With `__PAGE_SIZE = 4096` and `__STACK_SIZE_PAGE_ORDER = 4`.

While the stack is given the remaining free space between the end of the stack and the `__END` pointer.

From now on, when comparing two Unikraft images, we will suppose that they were both build with same libraries and application.

As can be seen from **Figure.4.2**, Unikraft always loads its sections and libraries at the same virtual memory offsets[35] :

- The image is always loaded at the address 0x100000.
- Libraries that are included in the text sections have their order decided by the GNU linker. These lines from the script are responsible of the previous:

```
*(.text)
*(.text.*)
```

- The remaining memory sections have always also the same order.

These points are making the Unikraft image easy to attack. Imagine multiple instances of an Unikraft image running a web service, if a hacker finds a vulnerability in one of those images. He would be able to reproduce the exploit in every images that were built with the same payload and libraries.

The solution to that is the script : `link64_ASLR.lds` depicted by **Figure.4.2** which solves the upper problems. It places the text section at an address higher than 0x100000, then places the libraries in the text section in a random order and creates padding in-between them. In order to prevent any attack on other memory sections that already benefit from the padding in `.text`, the script also shuffles some of the memory sections.

There was no clear documentation indicating which section could or couldn't be moved, so we made it thanks to the trial-error methodology.



Figure 4.2: Unikraft's memory order before and after ASLR

4.1.2 Linking Script ASLR

The solution runs with the following arguments :

- `-setup_file` : path leading to `entry64.S` file.
- `-base_addr` : address at which the image should start.
- `-file_path` : path leading to `link64.lds`, the linking script for KVM x86.
- `-output_path` : path and name of the file in which the modified script should be written.
- `-lib_list` : list of the libraries that will be linked into the final ELF.
- `-deduplication` : path to the `aslr_dedu_config.txt` file.

Throughout ASLR's development, Python's functions `randint()` and `choice()` were used. They are following an uniform distribution[36] which means that all the possibilities in the given range have the same probability to appear. Tests were made beforehand on the library to verify that the probability distribution used is indeed uniform [Appendix.A]. Because if an attacker can predict the generated random numbers, it can also predict the memory layout made by the ASLR.

The program works in two different flavors. The first one reads the `entry64.S` file, determines a new random offset to add to the base address and patches the latter. To do so, only the argument `-setup_file` is required.

The second one is more complex since it's the ASLR itself and requires the arguments : `-base_addr`, `-file_path`, `-output_path`, `-lib_list` to work.

Reading The Linker Script

ASLR uses an analyzer that understands GNU ld's syntax and reads `link64.lds` at the path given in `-file_path`, the script creates a symbol table that contains either :

- Sections

`aSection ADDR : { content }`

Notifies that `content` should be included in `aSection` placed in virtual memory at `ADDR`

- Current address

`. = ALIGN(0x20)`

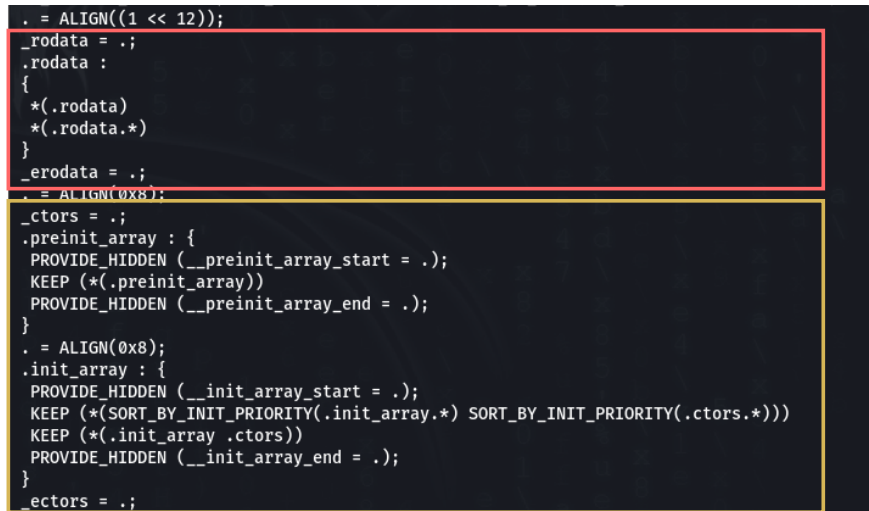
Sets the current virtual memory address (location counter) to a certain value.

- Assignment

pointer = .

Stores the current virtual memory address (location counter) in a variable.

Once the table is created, the script goes through it trying to create regions that are defined as being a set of commands surrounded by assignments.



```

. = ALIGN((1 << 12));
.rodata = .;
.rodata :
{
*(.rodata)
*(.rodata.*)
}
.erodata = .;
. = ALIGN(0x8);
.ctors = .;
.preinit_array : {
PROVIDE_HIDDEN (__preinit_array_start = .);
KEEP (*( .preinit_array))
PROVIDE_HIDDEN (__preinit_array_end = .);
}
. = ALIGN(0x8);
.init_array : {
PROVIDE_HIDDEN (__init_array_start = .);
KEEP (*(SORT_BY_INIT_PRIORITY(.init_array.*) SORT_BY_INIT_PRIORITY(.ctors.*)))
KEEP (*( .init_array .ctors))
PROVIDE_HIDDEN (__init_array_end = .);
}
.ctors = .;

```

Figure 4.3: Portion of link64.lds

This abstraction depicted by Figure.4.3, that shows two regions, allows to keep the coherence inside Unikraft’s internal working. Indeed, the upper pointers defined in the script are used to find sections’ position.

Pointers are paired together thanks to a longest prefix/suffix matching weighted by the distance between the two compared pointers in the symbol table. This is based on the hypothesis that regions are made in such a way that pointers circle one or a few sections, not the whole program.

However, this enforces Unikraft’s developers to keep consistency between their pointers’ name. They’re recommended to use either `NAME_start/NAME_end`, or `_NAME/_eNAME` where `NAME` is their pointer’s name in order have it properly working.

Modifying Text Section

Once the symbol table is filled and the regions are found, the program can start randomizing the text section which is responsible for the code of the libraries and the application.

All the libraries that are contained in the ELF are fed to the program as argument through `-lib_list`, they are all retrieved thanks to a built-in function defined in the `Makefile` of Unikraft that lists their name. Therefore, getting the corresponding name of their object file is simple, `gcc`'s default behavior keeps the original file's name and change its extension to `.o`[37], allowing us to place the object files in the linking script.

Thus the program picks at random which library it should place out of the list, then select a random integer between 0 and 65536 which is used to increment the location counter[38]. This results in a padding of the size of that integer.

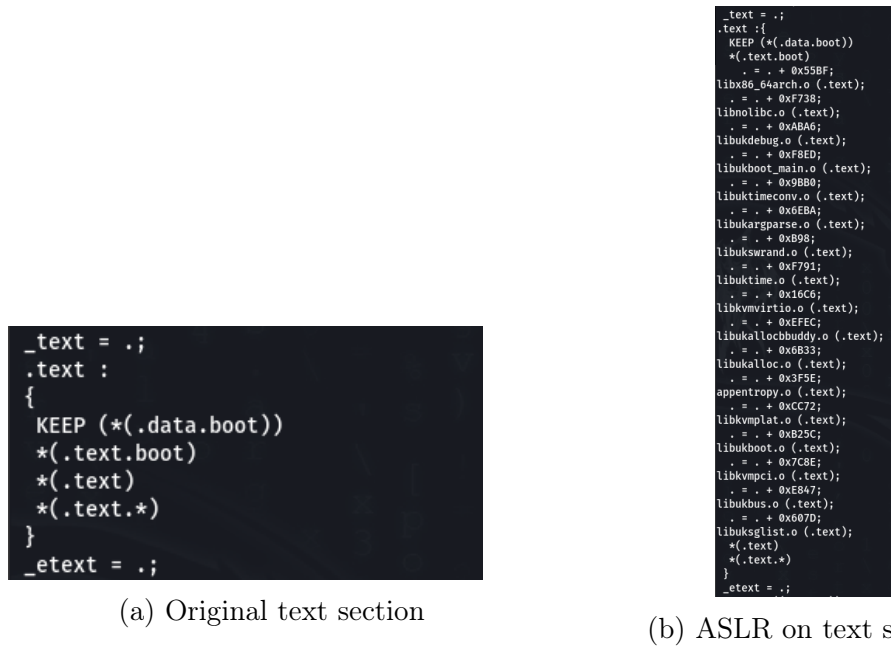


Figure 4.4: Linker's modified text section

The expressions of the original linking script,

```

*(text)
*(text.*)

```

are kept and set at the end of the text section, thus if any library has been forgotten in the list, the program will still be linked properly and work. Nevertheless, those libraries would only benefit from the previous added padding but not from shuffling.

Shuffling Other Sections

The remaining task is to randomize, if possible, other sections. This has been achieved by shuffling regions that could be moved and modifying values set to the location counter. Thus, the program runs over the whole symbol table a last time. Each time the program encounters expressions between regions such as,

`. = Address`

They are changed to,

`. = Address + offset`

Where the offset is drew the same way as the padding in the text section [4.1.2]. Note that the starting address has the same form, but its address range differs as we pick it between 0x100000 and 0x400000 which is a larger range.

Regions are taken at random, following again a uniform distribution, out of a list and placed in the linking script which is equivalent to shuffling them.

However, the solution faced some difficulties.

Since one needs Qemu [1.2.1] to run Unikraft on KVM, the solution needs to comply with Qemu and Unikraft's hypotheses and internal working. First, the region containing `.bss` and `.intrstack` can not be swapped out of the end of the script because it holds the pointers `__END` and `__bss_start` which are expected to be at the end of the ELF section. Then, text section's region has to be placed at the starting address so that Qemu can jump to the booting code, hence this region has to be the first of the ELF. The same way, `uk_inittab` and `uk_ctortab` which are used at Unikraft's start are not allowed to be moved. Displacing them causes page faults that are not handled during the booting procedure.

Then, the ASLR can't use memory space before 0x100000 because it is not allocated in Unikraft's page table and because Qemu loads the multiboot header at 0x95000. This explains why the starting address can not be placed below that value.

Adding The Indirection Table

When enabling ASLR with memory de-duplication, the program is fed with a new argument : `-deduplication` which indicates the path to a formatted file which gives the configuration of the table. The ASLR will create a section in the ELF file according to the configuration and fill the latter with NOP instructions. The size of the table is the sum of all the libraries' sizes. See the section [4.2.3] if you want to know more about its inner working.

Integration In The Build System

Unikraft uses multiple files in its build system and mainly custom files which all serve different purposes,

- `Config.uk` files are read during *Make menuconfig* which creates a configuration file that defines C preprocessor macros. Disabling or enabling features

and libraries that are imported to build the unikernel.


- `Makefile.uk` is used to hook the code to the build system. It gives information on which files should be compiled and linked and where they are located in the sub-folders.
- `Linker.uk` are files that can only be found in the sub-directories of Unikraft's 'plat' folder[39]. The latter gives the rules to apply while linking the final executable.
- `Makefile` is the traditional GNU Makefile used to create building rules.



```
(terrylos@terrylos)-[~/Thesis/unikraft/plat/kvm]
$ ls
arm Config.uk include io.c irq.c Linker.uk Makefile.uk memory.c shutdown.c x86
```

Figure 4.5: Unikraft's kvm folder

Figure 4.5 shows some of those files in the folder responsible for the KVM platform.



```
ENTRY(_libkvmplat_entry)
SECTIONS
{
    . = 0x100000;
    _text = .;
    .text :
    {
        KEEP (*(data.boot))
        *(.text.boot)
        *(.text)
        *(.text.*)
    }
    _etext = .;
    . = ALIGN((1 << 12)); __eh_frame_start = .;
    .eh_frame : { *(.eh_frame) *(.eh_frame.*) }
    __eh_frame_end = .; __eh_frame_hdr_start = .;
    .eh_frame_hdr : { *(.eh_frame_hdr) *(.eh_frame_hdr.*) }
    __eh_frame_hdr_end = .;
    . = ALIGN((1 << 12)); uk_ctortab_start = .;
    .uk_ctortab : { KEEP(*(SORT_BY_NAME(uk_ctortab[0-9]))) }
    uk_ctortab_end = .;
    uk_inittab_start = .;
    .uk_inittab : { KEEP(*(SORT_BY_NAME(uk_inittab[1-6][0-9]))) }
    uk_inittab_end = .;
    . = ALIGN((1 << 12));
    _rodata = .;
    .rodata :
    {
        *(.rodata)
        *(.rodata.*)
    }
    _erodata = .;
    . = ALIGN(0x8);
    _ctors = .;
    .preinit_array : {
        PROVIDE_HIDDEN (__preinit_array_start = .);
        KEEP (*(preinit_array))
        PROVIDE_HIDDEN (__preinit_array_end = .);
    }
    . = ALIGN(0x8);
}
```

```
ENTRY(_libkvmplat_entry)
SECTIONS
{
    . = 0x100000;
    /* Code */
    _text = .;
    .text :
    {
        /* prevent linker gc from removing multiboot header */
        KEEP (*(data.boot))
        *(.text.boot)
        *(.text)
        *(.text.*)
    }
    _etext = .;

    EXCEPTION_SECTIONS

    CTORTAB_SECTION

    INITTAB_SECTION

    /* Read-only data */
    . = ALIGN(_PAGE_SIZE);
    _rodata = .;
    .rodata :
    {
        *(.rodata)
        *(.rodata.*)
    }
    _erodata = .;

    /* Constructor tables (read-only) */
    . = ALIGN(0x8);
    _ctors = .;
    .preinit_array : {
        PROVIDE_HIDDEN (__preinit_array_start = .);
        KEEP (*(preinit_array))
        PROVIDE_HIDDEN (__preinit_array_end = .);
    }
}
```

(a) A Unikraft's link64.lds portion

(b) A Unikraft's link64.lds.S portion

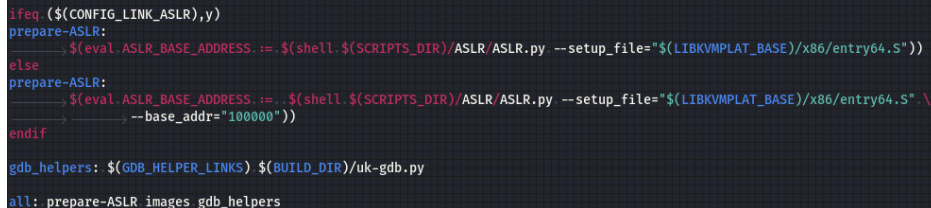
Figure 4.6: Resolving sections

Unikraft's linking script can be found incomplete in its sub-directories as `link64.lds.S`[35], which is further completed during the compiling process where the sections in the assembly file will be resolved. The process is described by Figure 4.6. Once the changes are made, it is copied as `link64.lds` in the Unikraft's build folder.

30

Therefore, ASLR script should be run two times : the first before compiling, in order to apply a patch to `entry64.S` which sets the base address, and the second right before the linking phase in order to benefit from all the modifications that were made during the compiling procedure.

The patch was added directly in Unikraft's `Makefile` and holds the new base address in a variable, as can be seen below in [Figure 4.7](#).



```

ifdef $(CONFIG_LINK_ASLR),y
prepare-ASLR:
    $(eval ASLR_BASE_ADDRESS := $(shell $(SCRIPTS_DIR)/ASLR/ASLR.py --setup_file="$(LIBKVMLAT_BASE)/x86/entry64.S"))
else
prepare-ASLR:
    $(eval ASLR_BASE_ADDRESS := $(shell $(SCRIPTS_DIR)/ASLR/ASLR.py --setup_file="$(LIBKVMLAT_BASE)/x86/entry64.S" \
    --base_addr="100000"))
endif

gdb_helpers: $(GDB_HELPER_LINKS) $(BUILD_DIR)/uk-gdb.py

all: prepare-ASLR images gdb_helpers

```

Figure 4.7: Makefile's patch

The patch makes sure to modify the `entry64.S` file with the new base address if ASLR is enabled, but it also resets this file if not. To do so, we feed the script with '100000' which is the base address of a common Unikraft image.

Scripts can be easily added to Unikraft since it includes a folder namely for this purpose[40]. Multiples scripts are used throughout the building process, hence we could use the `SCRIPTS_DIR` macro which was already defined in Unikraft's main `Makefile`.

The goal here is to run the ASLR script once the final linking script `link64.lds` and all the object files are generated. Thus, `Linker.uk` had to be modified in order to integrate the solution into the linking process.

Source code is compiled and regrouped per library into a single object file which is placed in the build folder. Once that's done, Unikraft links all libraries together in a `.dbg` file which is later stripped away in order to keep the image as small as possible.

On [Appendix A.2b](#), the reader can notice a few changes. First the script has been added right before the linking procedure with,

```

$(call build_cmd,ASLR,,,$@, \
$(SCRIPTS_DIR)/ASLR/ASLR.py $(ASLR_args))

```

Then, our modified linking script is passed to the linker thanks to the `-T` option[41] which tells the linker to link the objects according to the given script. Thus, feeding our modified script `link64_ASRL.lds` (the macro `KVM_LD_SCRIPT_FLAGS_ASRL` contains the flag and the path to the script) will tell the linker where to place the sections and libraries in memory. This macro also stores paths to other linking scripts if any.

The bash scripts that allows to link again the ELF file without compiling it back are copied in the application's folder with,

```
$(call build_cmd,COPY,,ASLR scripts, \
cp $(SCRIPTS\_DIR)/ASLR/relink_elf.sh ../../;\
cp $(SCRIPTS\_DIR)/ASLR/string_script.py ../../)
```

Therefore the solution only required to change `Makefile`, `Config.uk` and `Linker.uk`. The first, to set the base address in a variable, the second in order to let the user choose to enable ASLR or not thanks to macros and the last to insert the script into the linking process.

4.1.3 Stack/Heap ASLR

Implementation

In order to have a complete ASLR implementation, the unikernel needs to randomize its stack and heap. For instance, the upper example [2.4] is an attack made through the stack. Thus applying ASLR in the text section only results in avoiding the attacker to know where are located the functions. So, he can not set a proper function address on the stack. Unfortunately it doesn't secure them from vulnerabilities it only makes the exploit harder.

Unlike ASLR on the text section, this part will be done at run-time in the `setup.c` [42] file of the unikernel. Meaning that we will rely on Unikraft's own randomization engine from `libukswrand` to generate the random offsets.

In `setup.c`, the reader can observe that the function `_mb_init_mem` is in charge of placing the stack and heap in the remaining memory. The goal of this section is to find a way to fit the stack and the heap randomly in a fixed memory range.

As discussed upper [4.1.1], the heap is given the remaining space which, for small images with default configuration, leads to a heap considerably greater than the stack. According to Unikraft's performances on Helloworld[8], the image's size is 192.7KB.

Once transposed into the static virtual address, we can compute the length of the stack and heap. The `__END` pointer is located at $(0x100000+0x302CC) = 0x1308CC$ which leaves us with $0x40000000-0x1308CC = 0x3FECF734$ remaining space. The stack has a fixed size of 16×4096 bytes, thus $0x10000$ in hexadecimal, hence the heap is left with a length of $0x3FECF734-0x10000 = 0x3FEBF734$ that is 0.9988 GB.

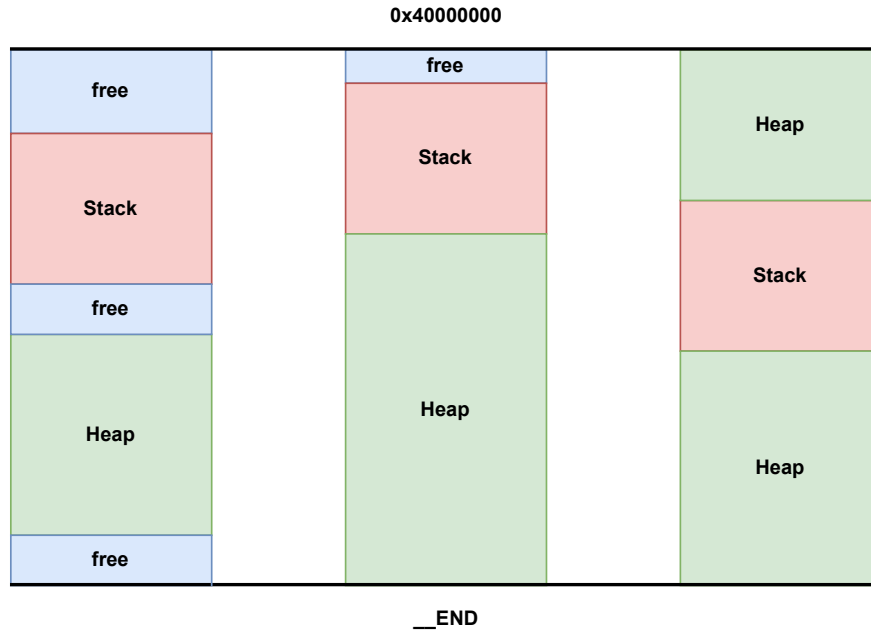


Figure 4.8: Stack and heap randomization strategies

There is multiple possible ways of placing those elements to achieve ASLR on the dynamic memory. Relevant ones are depicted by Figure.4.8 where free marked spaces are unallocated memory due to the random placement of the stack and heap.

The left solution allows to increase randomness and thus makes it harder for an attacker to guess their absolute position. However, it's achieved at the cost of losing memory that could have been given to the heap.

The centered one limits the amount of lost memory, it does so by using an unique random offset. That is less secure than the previous solution since either the stack or the heap starts at its standard address.

The last one allows to have a complete usage of the memory without any wasting. However, one could argue about a few drawbacks :

- Only the stack is truly randomized because the heap's starting address is always at `__END`
- It breaks the memory layout [2.1] model.
- It requires a special data structure that allows the heap to be fragmented in memory.

The first solution has been chosen and implemented because we could mitigate the amount of lost memory by implementing dynamic paging and swapping which will allow Unikraft to extend its memory limit above `0x40000000`. Nevertheless, the free space between the stack and the heap is set to 0 in order to limit memory waste, not doing that would result in randomizing two times the heap's end.

Indeed, since the stack is always allocated with a fixed size given the configuration process. Starting with its placement was the best idea. The first byte of the stack is placed at random following an uniform distribution along $[0x30000000, 0x40000000]$. Then heap's start address is placed randomly in the range $[__END, __END + 0x10000000]$, followed by its end address that is set in to **stack_end**. Note that all of these placements are rounded up to an offset corresponding to a page, avoiding to have these areas in the middle of a memory page.

In the worst case scenario, memory usage wise, the stack is placed at $0x30000000$ which is almost one quarter of the available memory that is left unused. Supposing its default size, only the range $[__END, 0x2FFF0000]$ could be used to place the heap. Then, this would result with a heap's length of $0x2FFF0000 - __END - 0x10000000 = 0x1FFF0000 - __END$. Now, taking the same example as upper $__END$ is located at $0x1308CC$ when considering no ASLR on the text section, with that value the remaining address space would be : $0x1FFF0000 - 0x1308CC = 1FFBF734$.

The best case scenario happens when the offset are set to 0 which is equivalent to an Image without ASLR, its heap size was already computed upper. Comparing the two shows that we discarded $100 - (0x1FFBF734 / 0x3FEBF734) * 100 \approx 50\%$ of the total usable virtual memory for the heap.

Libukswrand Initialization

Like every random number generators, the library needs to be initialized with a seed in order to produce random numbers[43]. By default, the library sets the seed based on : the kernel time, CPU's **rdrand** or a constant. The first is initialized later in the booting procedure thus we can't use it, the second is only allowed in the configuration file when the following condition is met : **ARCH_X86_64 && MARCH_X86_64_COREI7AVXI** and the last does not fit the application because we would always have the same seed across the images, resulting in the same pseudo-random pattern on the numbers.

Therefore, we found a workaround and obtained the seed thanks to the **rdtsc** assembly function that gives the number of CPU cycles.

4.1.4 ASLR scripts

The script called **relink_ELF.sh** is generated in the folder of the application after the building procedure when ASLR is enabled in the configuration files. It allows to relink and rerun ASLR which leads to a new memory layout for the image.

Script usage,

```
./relink_ELF.sh BUILD_DIR NAME_EXEC  
UNIKRAFT_FOLDER
```

The reader should be aware that this script was made to run ASLR without compiling again the whole program.

However, it does not work on complex applications that possess multiple linking scripts and the script does not modify the base address of the text section [4.1.2], as we would need to recompile some files for that.

4.2 Merging ASLR With Memory De-duplication

Memory de-duplication was made possible in Unikraft [1.2.3] by making libraries' position in virtual memory predictable. That is the complete opposite of what ASLR tries to achieve, nevertheless, combining the two (while being challenging) would have non negligible benefits on the OS : making it harder for attackers to exploit a vulnerability while mitigating the memory overhead caused by ASLR.

One way to make those two techniques coexist is through an indirection table which purpose is to contain all the elements that make memory pages different from one kernel to another. Those ties down to some assembly instructions [44, Chapter 5,6]:

- Calls
- Jumps
- Mov/Lea

The upper instructions cause pages to be different because they are able to work with addresses. Since their positions are made random with ASLR, memory de-duplication can not work without making sure that they all contain the same operands on in their .text sections so that their pages can be shared.

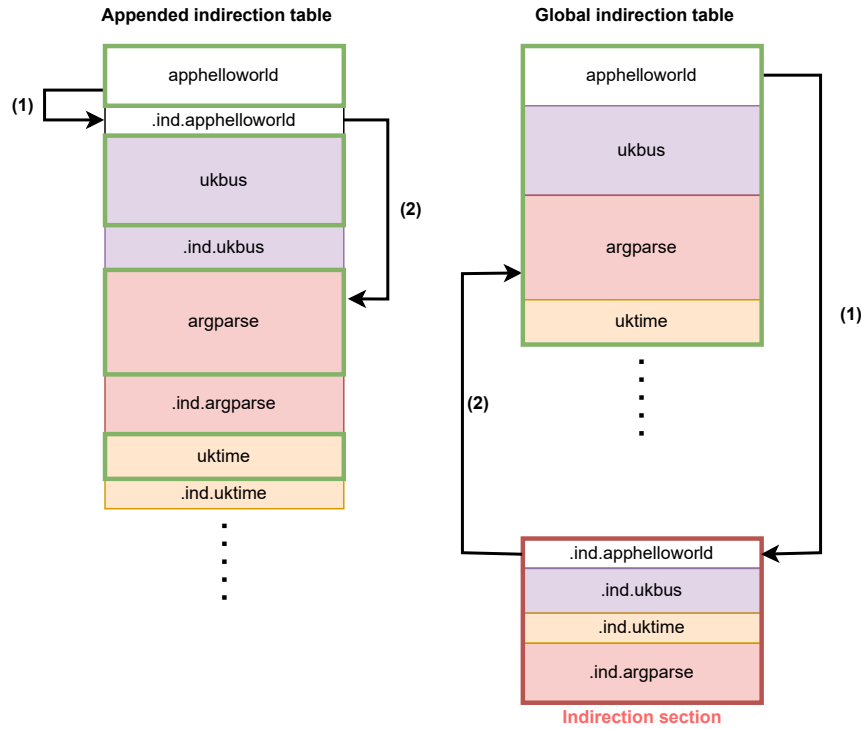


Figure 4.9: Indirection table methods

On Figure.4.9 green squares indicate that pages are made shareable.

There was two possible ways of implementing such tables, either a fragmented one (left) where each library would be given its own table or a global one (right) held in a specific memory section which would contain all the indirections from all the text section. With Gauthier, we decided to implement both in order to be able to compare them and chose the best one, hence, it was decided that he would implement the first and we would do the second. Therefore all the following developments and conclusions will concern the global indirection table.

4.2.1 Creating a GCC Plugin

At first, investigations were made on a compiler/linker combination that could allow such tables. We knew that gcc was able to produce plt and got tables which are indirection tables, therefore we tried to find a way to use that mechanism at our advantage.

Diving in gcc's documentation teaches us that it allows loadable modules[45], which gives the users the opportunity to add features to the compiler. However it is restricted to the compiler **only**, but that's not enough for what we try to accomplish since creating such a solution requires to move instructions to another memory section, thus the linker needs to be involved.

Hence, a solution based on a gcc was discarded since it didn't meet our needs.

4.2.2 Using Daniel's Bootloader

Instead of building the tables by ourself, the idea was to use the plt/got mechanism which is already existing and manipulating it so that the latter is placed where we want it to be in memory. To generate plt/got tables, the files have to be compiled and linked together with the PIE flag to create a position independent executable.

Most of the work was already done by Daniel DINCA because he found a way to turn Unikraft into a PIE ELF in his thesis. Hence, we just had to combine his work with our implementation of ASLR and place the tables at a fixed virtual memory address, this could have been done again by scripting in the linker file.

Finally, this did not work for one simple reason : libraries were not located inside the plt/got tables even though the Unikraft image was made as PIE, a small subtlety slipped through our fingers. Libraries in Unikraft are statically linked inside the text section, therefore from the linker's eyes, they do not need to be loaded at run-time which explains why plt and got sections are empty. A simple solution to that was to make the libraries dynamic by feeding the `-shared` flag to gcc that turns them to shared library. But a problem arises with this modification : the latter breaks the isolation. Sharing the libraries with the dynamic linker means that all the images will be able to use the same instance of the library directly in memory. This obviously breaks the isolation between the kernels.

The solution should thus be given up since isolation is an important feature of Unikraft that we could not break.

4.2.3 Binary Rewriting

The last method investigated was binary rewriting, which consists in patching the binary file itself. This method is not trivial since it requires to change low-level code which is eluded of all the abstractions and information that high-level code can contain.

This solution came up directly with some real troubles. The main one is that x86 64 bits CPU only uses addressing relative to the rip register[46], hence jumps with a constant operand indicate the address at `%rip+constant`. That leads to huge complications in this implementation for each jumps, but this is also true for calls since they are nothing more than,

```
1    PUSH rip
2    JMP %constant
```

Where constant is the offset to the address where the program has to jump and rip the instruction pointer, so calls are directly impacted by the way jumps are made.

This major problem can not be dodged because absolute addressing is needed in order to share pages. Due to ASLR, code is placed randomly in virtual memory which means that each instance of the code has a very high probability to have different memory offsets when compared to the other instances of the same code.

Another one comes from the fact that the table should guarantee that all the elements it contains are at a known address, otherwise patched instructions wouldn't match from one image to another. This is solved by letting the user decide where to place the table and which reserved space is attributed to every library. The `aslr_dedu_config.txt` file gives a description of the table and follows a strict layout:

```
Starting_address : default_size
Lib_name : given_size
Lib_name : given_size
Lib_name : given_size
...
```

`Given_size` can be omitted and the program will replace it with `default_size` but there should still be a space character after the colon. It does not allow space or any type of character except the carriage return after `default_size` or `given_size`. However, comments can be placed thanks to `"#"` which causes the loader to ignore the line.

Users are asked to update this file each time a new library is added to Unikraft and also make sure that every Unikraft image is built with the same version of this file. Because a simple swap between two libraries inside the indirection table can cause pages to mismatch in the images.

Image Extender

The solution runs with the following arguments,

- `-file_path` : path leading to `NAME_kvm-x86_64.dbg`, the Unikraft KVM debug file.
- `-build_path` : path leading to the build folder.
- `-conf_file` : path leading to the `aslr_dedu_conf.txt` file.

First, it extracts function symbols that are left in the Unikraft debug image and gathers symbols from imported libraries. With that data the latter builds a dictionary that maps the libraries' name with the tuple (function_name, function_pos, function_elf_pos) or with (function_name, function_pos, None) if the symbol was not linked in the debug image.

Then, it fills the indirection table with jump instructions that branch to the functions' address in text section. It is based on the order in which the libraries are displayed in `aslr_dedu_config.txt`.

Finally, the program goes through the whole code and patches the instructions that we discussed upper. That's also during this step that `mov/lea` instructions are added in the table in the function's corresponding library. Of course, it does not patch blindly all those instructions, they are only moved into the table if the address found in the operands :

- Is a direct access to memory.
- The memory address matches sections that could have been impacted by ASLR.

If the reader is willing to share pages of his custom libraries because he runs multiple images on the same hypervisor. He is advised to append it in the configuration and keep the original order of the native Unikraft's libraries.

Image Extender does not rewrite the image it works on, instead the script saves the modified version of the image under the name `NAME_deduplication`, where `NAME` is the debug image's name. Thus, the user still have access to the unmodified version of the debug image.

Integration In The Build System

We will follow the same methodology as the integration for the ASLR mechanism. The table should be filled and the text section patched right after we called the ASLR.

```
# register images to the build
ifeq ($(CONFIG_PLAT_KVM),y)
UK_DEBUG_IMAGES-y ..... += $(KVM_DEBUG_IMAGE)
ifeq (y,$(CONFIG_MEMORY_DEDUP_ASLR))
UK_DEBUG_IMAGES-y ..... += KVM_DEDU_DEBUG
endif
endif
UK_IMAGES-y ..... += $(KVM_IMAGE)
UK_IMAGES-$(CONFIG_OPTIMIZE_SYMFIL) += $(KVM_IMAGE).sym
UK_IMAGES-$(CONFIG_OPTIMIZE_COMPRESS) += $(KVM_IMAGE).gz
endif

KVM_DEDU_DEBUG: $(KVM_IMAGE).dbg
_____ $(call build_cmd,DEDU,, $(KVM_IMAGE).dbg,\
_____ $(SCRIPTS_DIR)/ASLR/ImageExtender.py \
_____ --file_path="$(KVM_IMAGE).dbg" --build_path="$(BUILD_DIR)" \
_____ --conf_file="$(SCRIPTS_DIR)/ASLR/aslr_dedu_config.txt")
```

The upper code snippets show that we added a new rule in the file which is added to `UK_DEBUG_IMAGES`, the latter is executed right after `KVM_DEBUG_IMAGE` which is the rule building the debug image.

Regarding the new rule itself, it calls our script that populates the table and patches the instructions.

4.2.4 Patching Calls

Call instruction varies from 5 to 7 bytes depending on its type. The ones we aim to modify are the 5 bytes long which are formatted such as : Opcode + 4 bytes of relative offset. Those are calls with a constant operands, others are not and thus will be similar in every Unikraft images that enabled ASLR.

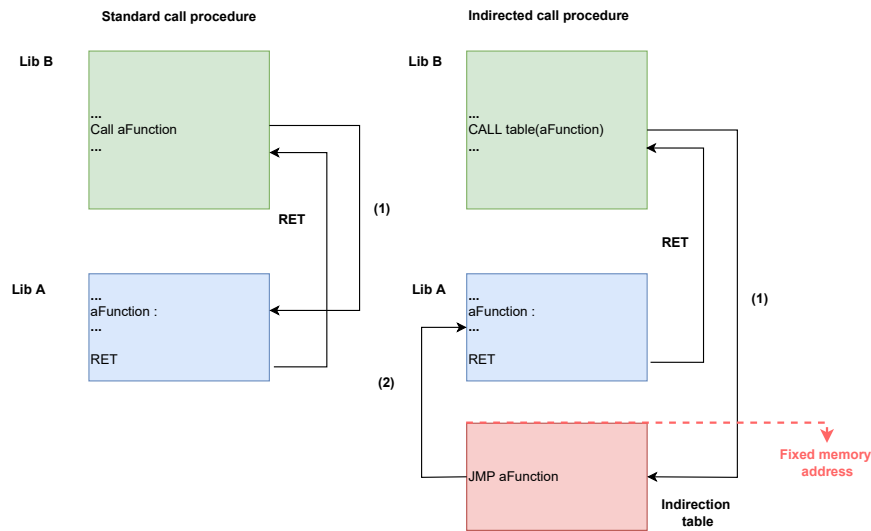


Figure 4.11: Call instruction indirection

We have to find a trick that ideally fits in 5 bytes which allows us to call in an constant absolute manner, so that we can implement the indirection such as depicted in Figure.4.11

Using The Stack

A first idea was to use the stack to fix the indirection problem with that code snippet,

```
1  PUSH imm32
2  RET
```

The upper assembly pushes a constant "imm32"^[47] on the stack and then jumps to it thanks to the RET instruction which pops the top of the stack and branches to it. The solution is however 1 byte too long since this patch is 6 bytes long : 5 bytes (PUSH) + 1 byte (RET).

Moreover, this solution does not saves the context of the calling function meaning that returning to the caller from the called is not possible. We had to discard this way of fixing the calls.

Using Register

Another way to circumvent the lack of direct far absolute calls in x86 64 bits assembly was to call the address from a register, making the call indirect.

```
1  PUSH rax
2  MOV imm32,rax
3  CALL rax
4  POP rax
```

That is how the upper solution was found, the latter pushes the value of rax on the stack, sets the address in it and then branches to the address. Finally, it gets back the value of rax from the stack. Nevertheless this method comes with major drawbacks:

- Its size, adding so much instructions increased the byte amount from 5 to 10 bytes.
- It discards the value of rax after the call instructions despite its important role in assembly x86 conventions [14]. Indeed this register is used to hold the return value of the called function.

We decided to drop the rax saving part, which leads us to the final solution,

```
1  MOV imm32,rax
2  CALL rax
```

After some research, we found a thread on StackOverflow talking about the same problem [48]. The saving part can be dropped because when the caller calls a function it expects it to be modified by the called, thus changing it right before the call instruction without saving it is completely fine.

However the patch's byte length still needs to be taken care of. There are two possible ways of doing so : either we handle that during the binary rewriting process which means that we have to re-compute every offset in the program, or we manage to modify the call instruction beforehand.

The second method was chosen because it appeared to be simpler, after some researches we found out that it was possible to modify the way gcc was compiling C code into assembly : the .md files.

Machine descriptions [49] are files that aim at indicating to gcc which assembly instruction it should map to its internal RTN insn template. The one responsible for x86 64 bits assembly is a large file named i386[?]. Wandering through the file and looking for the keyword "call", we found interesting information from line 14864 to 15330 where the file describes how calls should be translated. That's how we found out that call's assembly was generated in the

function `ix86_output_call_insn` from `i386.cc`. To shift the binary of three bytes at each direct call, 3 NOP instructions were added right after those. Now, the text section can be patched without any concern about shifting all the offsets of the binary.

The drawback of that method is that every direct call instruction of the binary is padded, thus we potentially inflated the call instruction with 3 additional bytes.

4.2.5 Patching Jumps

In legitimate code, jumps are mostly used by the compiler for conditional operations and loops. There exists `go to` statements, but it has been good practise to avoid using them since the paper of E.Dijkstra[50], which are translated to unconditional branches in assembly.

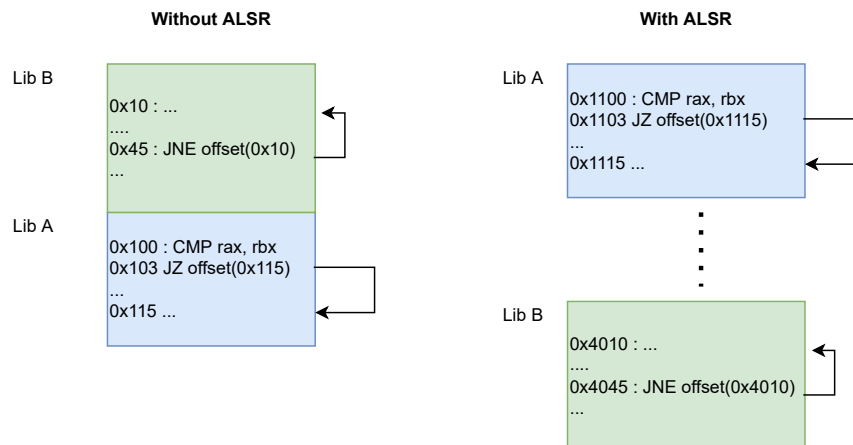


Figure 4.12: Jump instruction

Looking at jumps from that perspective, having them being relative is not a problem as we can observe from Figure 4.12. ASLR moves libraries and sections, not single functions and neither portion of functions' code. The only impact ASLR could have on jumps is if they were used to make direct jumps to functions which is mostly not the case because the compiler uses the call instruction that saves the context of the caller instead.

Moreover, jump instructions are 5 bytes long, if we were to patch them with the smallest sized patch, we would consume 6 bytes. That means we would have to pad again in gcc whenever a jump occurs in order to be able to rewrite it.

Based on the observations upper, we decided to skip jump instructions in the rewriting, because it appeared that there was very little gain to patch them when compared to what it would cost in terms of padding.

4.2.6 Patching Mov And Lea

Methodology

Our concern will be instructions which have the ability to load from memory or have direct access to memory addresses, thus mov and lea instructions were concerned.

Taking the MOV instruction as an example, the latter is problematic depending on its operands type.

Problematic MOV	Non-problematic MOV
MOV dword[0x140000], rax	MOV 0x61616161, rax
MOV 0x11570, rdx	MOV dword[rbp-0x10], rdi

Figure 4.13: Problematic instruction comparison

Where 0x140000 is the address of an element lying in .bss, 0x11570 a function's address.

From **Figure.4.13**, the reader can observe the comparison between loading the content of memory at 0x140000 into rax and setting rax to "aaaa" (since 0x61 corresponds to "a" in the ASCII table [51]).

The left ones are of course problematic since we try to access fixed memory addresses which may differ in every image with ASLR enabled. On the other side, it either represents a constant that is not an address of the program or one computed from a register.

Hence, to find out which mov/lea instructions should be displaced into the indirection table, it is enough to look at their operands to see how the latter tries to access memory.

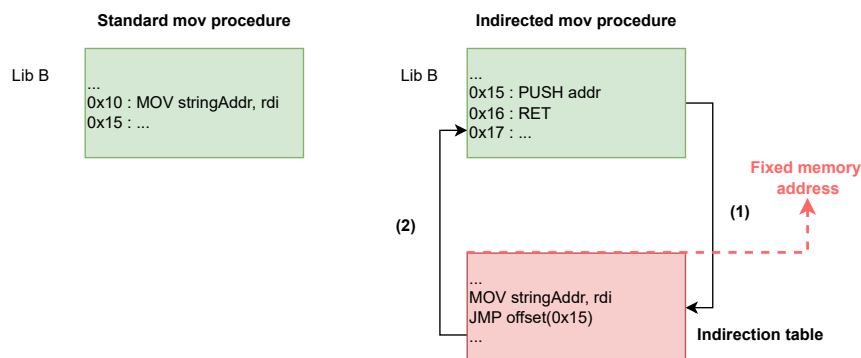


Figure 4.14: Mov instruction indirection

Now let's address how we should patch the instructions. As **Figure.4.14** shows it, the idea is to move the problematic instruction in the table and branch to it so that the instruction is executed in the table. However, we face again the relative addressing problem but this time, we are not using a function, therefore

there is no context change involved. This means that the upper solution [4.2.4] could now be used because its downside does not have any kind of importance anymore and its 6 bytes length is pretty small.

Unfortunately, `mov/lea` have variable sizes, which means that instructions which sizes are superior or equal to 6 bytes are fine but the others need to be padded. Contrarily to `calls`, we can not predict in `gcc` from the operands' form if padding is needed or not. Therefore we made the choice to skip the instructions that were too short to fit the patch.

Problems

We ran quickly into problems when patching these instructions. Due to the lack of absolute jumps, this solution uses the stack to create an indirection at low byte cost. However, the `mov/lea` instructions that work with addresses from the `bss` section at boot time causes the kernel to crash.

We had two solutions, either we do not indirect internal libraries that are called during the booting procedure or we don't patch `mov/lea` instructions that are having a `bss`' address as operand.

We went for the second solution, hoping that we could share pages even though some instructions were not patched.

This is a drawback linked to this method which does not appear in the append only tables because we can do indirections with simple relative jumps.

4.2.7 Page Sharing

To benefit from memory de-duplication, kernel same-page merging (*a.k.a KSM*) should be installed and launched since the solution relies on it to share the kernels' pages. Enabling it can be made with the following command line,

```
echo 1 | sudo tee /sys/kernel/mm/ksm/run
```

Finally, shared pages can be observed easily with,

```
grep -H '' /sys/kernel/mm/ksm/*
```

The latter reads the information generated by KSM about pages' state[52]. Three elements will retain our attention :

- Page shared: which indicates how many pages are currently shared by the kernels.
- Page sharing: which indicates how many pages could be shared if they were used.

- Page unshared: which indicates how many pages are unique

Results

Results are an important element of this thesis since it allows us to draw conclusions about the performance impact of the solution, but we can also assess how secure the application is.

Throughout this chapter we addressed ASLR alone and compared the latter with its previous implementation. We found out that our solution had a relatively good entropy when compared to other OSes such as FreeBSD, HardenedBSD and Debian, knowing that we had a limited VMA range. This ASLR implementation quadruple the image's size but offers the same boot time as a classic image, however we argued that the image's size was not a relevant problem since that's made of padding that is never read.

Then, we compared the two indirection methods based on the number of shared pages. From the measurements, we were able to conclude that indirection tables were a good solution to mitigate ASLR's memory consumption when a lot of images were run but this shouldn't be applied for a few Unikraft kernels.

Finally, we concluded that the appended tables should be used instead of the global one because the last was not performing well enough due to the constraints on its implementation.

5.1 ASLR performances

5.1.1 Entropy

During all this section, the reader should be aware that each time we will discuss entropy, we refer to it as the Shannon entropy [2.10].

Assumptions will be made in the other sections in order to assess the security and the performances of the solution.

Based on [4.1.1], the text, stack and heap share the memory space between 0x100000 and 0x40000000. Nevertheless, we remind the reader that the heap has its position dependent on the text's section size. On a standard Unikraft image we are giving the range,

$$\log_2(1.073741824 \times 10^9 - 1.048576 \times 10^6) = \log_2(1.072693248 \times 10^9) = 29.9985bits \quad (5.1)$$

This is almost 30 bits of entropy for the whole solution without considering constraints.

All the results under are based on a data-set made with the `entropy_kvm-x86_64` image which is an application that prints on the standard output the address of one hidden function, the address of a buffer allocated on the stack, the address of a structure allocated on the heap and finally the address of a global variable.

We recompiled every time the image with ASLR enabled before running the unikernel containing the application and then saved the addresses in a file. This way, 34500 images' addresses were gathered, which allows us to study the impact of ASLR on the unikernel.

The reader can produce a new data-set using the bash script `entropy_measure.sh` included in the submitted archive.

5.1.2 Text

Theory

In the solution, text section's beginning is placed between 0x100000 and 0x400000 which means that theoretically, the entropy on the first byte of the section should reach:

$$\log_2(4194304 - 1048576) = \log_2(3145728) = 21.585bits \quad (5.2)$$

Where numbers are changed from hexadecimal to decimal in the equation.

However as described upper, the method also allows to pad inside the section in order to virtually inflate it that causes to shift the following libraries with some other random offsets.

With the current configuration, a random pad drew between 0 and 65536 is added between each libraries. Considering app-entropy, as an example, which contains 17 libraries. We would have the following entropy on the last library included in the text section:

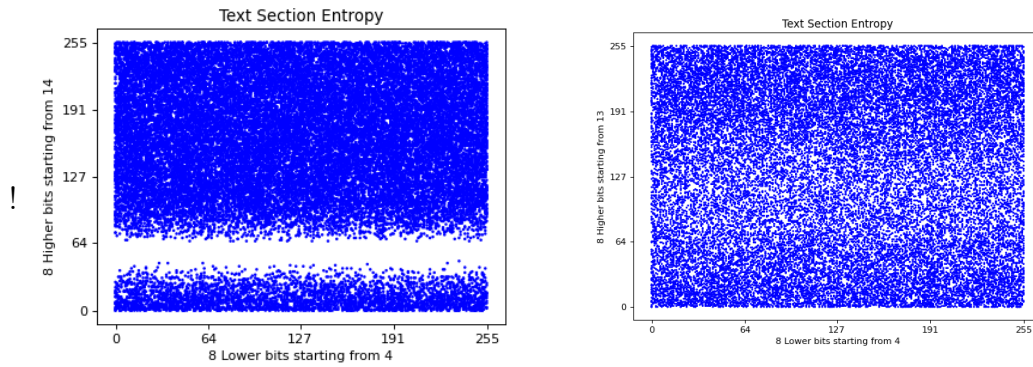
$$\log_2(3145728 + 17 * 65536) = 22.022bits \quad (5.3)$$

In reality, it doesn't only concern the last library of the text section, since the libraries are shuffled, there is no unfairness between the entropy on the libraries because their order is not the same in each images. Therefore we can consider that the entropy on the whole text section is about 22.022bits, and reaches the address 0x4AAEADe at maximum padding.

The solution is thus more secure than FreeBSD and HardenedBSD but less secure than Linux Debian since the last reaches a 29 bit entropy on the text section while ours theoretically reaches 22 bits[53, p. 11].

In Practise

Below, the figure depicts the address placement of a function in the text section from the entropy Unikraft image.



(a) ASLR entropy on [4,12] w.r.t [14,22] (b) ASLR entropy on [4,12] w.r.t [13,21]

Figure 5.1: Entropy measure on the text section

From now on, brackets [x,y] will express the bits taken from x to y.

Bits from [0-4] are discarded because they were always set to 0. After some investigations on the images with `radare2` and `readelf`, we could notice that the offsets added to the location counter were not strictly followed by the linker. Indeed, it tends to round up the addresses.

Having observed that, we can now understand and explain the results. The upper plots both depicts by a dot the function's address from an Unikraft image built with ASLR based on its highest and lowest bits.

The first **Figure.5.3b** looks at the range that we established upper [5.1.2]. An area from 45 to 65 along the y axis which are the most significant bits is never used by the images.

We can observe from **Figure.5.3b** that all points are scattered uniformly on the domain when we consider bits in range [13-21] as being the highest bits, however if we wanted to be picky we could notice that there seems to be a smaller concentration of it around the central area along the y-axis.

This shows that we have an effective entropy with this implementation on [4-21] which is 17 bits long. The result is worst than expected but it still makes the image harder to exploit.

Moreover, the maximum entropy reachable by the data set is the Shannon entropy [2.10] to which we can compare the one computed on the data set. This computation will only inform us about the dispersion of the memory in its available space, the further we are from this theoretical number the more we had redundancy in the data set, which we clearly do not want with ASLR.

The maximal value is $H_s(X) = \log_2(345000) = 15.07bits$, where all values are independent. The data set gets 14.92 bits which is close to it.

5.1.3 Stack

Theory

As described in [4.1.3], Unikraft places the start of its stack at its highest possible address *i.e* 0x40000000 to 0x40000000 - `__STACK_SIZE` which is up for the user to choose. We will consider the default value : 0x10000 bytes long.

As explained in [4.1.3], the first byte of the stack is placed in the range [0x30000000, 0x40000000] with the constraint that the address should be aligned to a page. This means that our 12 weakest bits will always be the same, thus the randomization occurs on the [12-28] bit range, leaving us with a 16 bit entropy on the stack. Comparing this value to what is done by other OSes[53, p. 7], the method is weaker than the one implemented in Debian and HardenedBSD which have respectively 30 and 41 bits of entropy. However our solution has the same entropy as FreeBSD.

Practise

Figure.5.2 is obtained using the same data set, we can now confirm the upper point. The stack addresses are indeed uniformly distributed on the [12-28] bit range as expected. The randomization also seems uniform despite using the Unikraft randomization engine instead of Python's library.

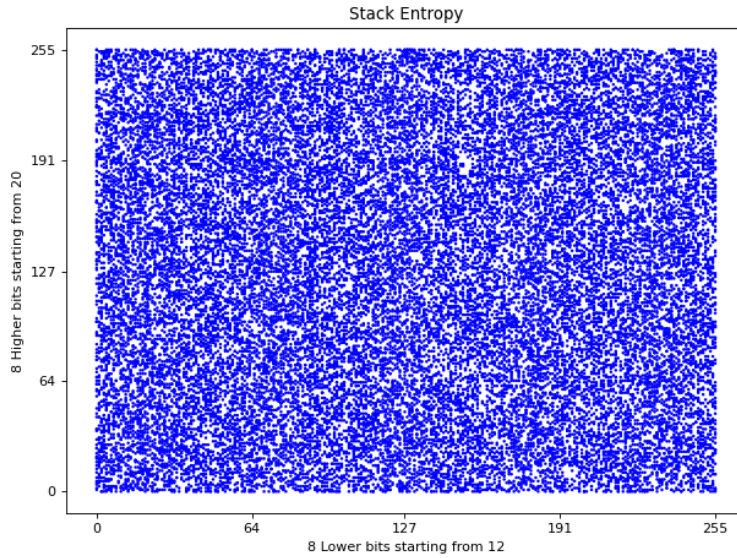


Figure 5.2: ASLR entropy on [12-20] w.r.t [20-28]

The maximal entropy value we can obtain in the data-set is $H_s(X) = \log_2(345000) = 15.07bits$, where all values are independent. The data set on the stack addresses gets 14.6 bits which is worst than the text section. We can conclude that we obtained more redundancy in the stack's addresses than the stack ones, meaning that the latter appears to be less secure.

5.1.4 Heap

Theory

Like the stack, the heap's starting address is rounded up to the next page, meaning that our 12 least significant bits will always be the same.

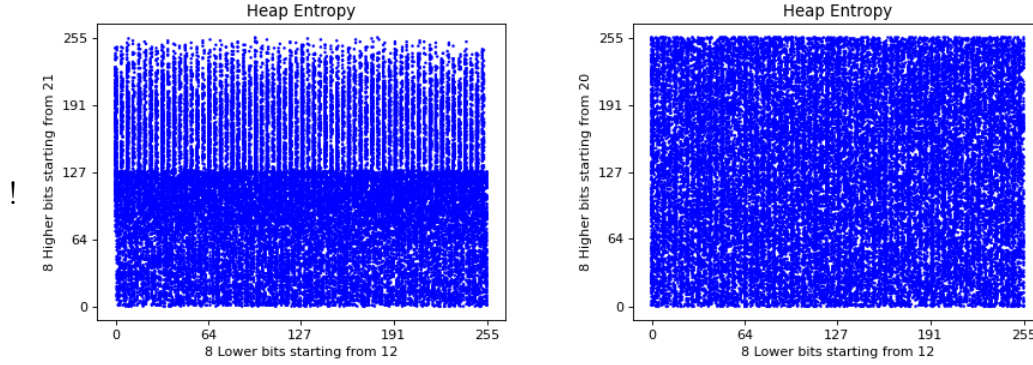
From section [4.1.3], we get that the randomization on the heap's first byte operates in $[_END, _END+0x10000000]$ and its end is placed at `stack_end`. Thus a variable allocated on the heap could be anywhere in memory between `_END` and `0x3FFF0000`, the latter is true if we suppose that the stack is placed at the maximum address.

Theoretically, we should have $\log_2(0x3FFF0000-_END) = \log_2(0x3FEBF734) \approx 30$ bits, with `_END = 0x1308CC` which is the text's size of our Helloworld image without ASLR.

However the upper scenario is unlikely and too optimistic so the range [12-29] is more suitable because the stack will almost never be at the maximum address.

The solution has theoretically 17 bits of entropy on its heap which is according to the same paper[53, p. 11] is better than FreeBSD but worst than the others.

Practise



(a) ASLR entropy on [12,20] w.r.t [21,29] (b) ASLR entropy on [12,20] w.r.t [20,28]

Figure 5.3: Entropy measure on the heap

We can clearly observe from Figure.5.3a that [12-29] is still too optimistic since the dots are not uniformly set above 127 along the y axis, unlike [12-28] which have dots in its whole addresses range. It can be concluded that we have 16 effective bits of entropy on the heap with this implementation.

Regarding data set's entropy, we expect as previously $H_s(X) = \log_2(345000) = 15.07$ bits while we obtained 14.69 bits from our computations. That is similar to what we had with the stack, showing that we have some small redundancy on some addresses in the data set.

5.1.5 Boot Time

For this experiment, we compared two Helloworld Unikraft images where one of them has been compiled and linked with our solution. The goal is to observe if there is any latency induced by the introduction of ASLR in the Image.

Each of the kernels were run 25 times and we computed their average boot time. From Figure.5.4, we can observe that no significant delay is induced by the solution, making Unikraft as fast as its vanilla version while providing an increased level of security.

The small difference is assumed to be the placement of the heap and stack which runs in a loop trying to set them at random memory offsets.

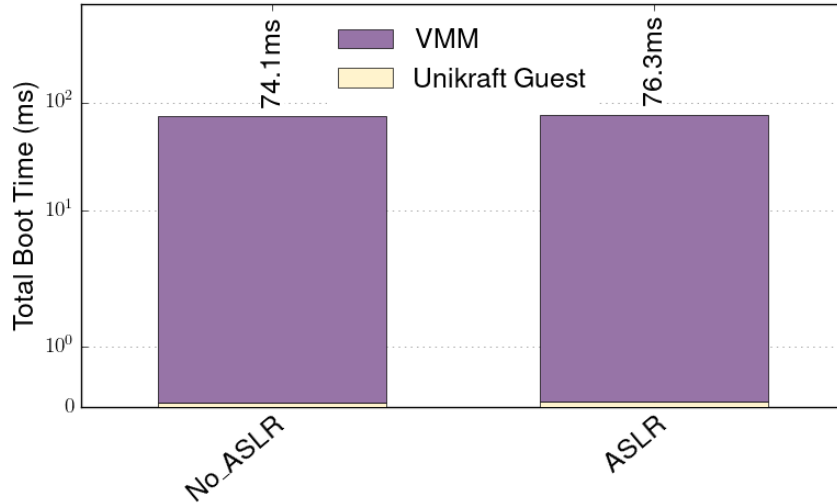


Figure 5.4: Booting time ASLR vs no ASLR

5.1.6 Image Size

Since the location counter in the Linker script follows the placement in virtual memory, one would assume that padding virtual memory as we do shouldn't have an impact on the binary size.

For this experimentation we compiled and linked 25 images with the ASLR enabled and we compared their average size with an image that has been compiled without it. Regarding the building configuration, they were all made with those options enabled,

- Optimized for size.
- Link time optimizations.
- Drop unused functions and data.

They can be set in the `make menuconfig` in the "Build options" menu.

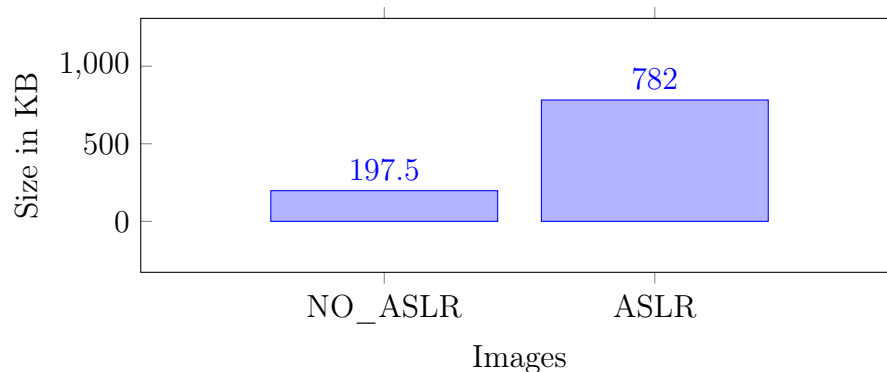


Figure 5.5: Image size ASLR vs no ASLR

The image size has significantly increased due to ASLR's enabling, it has almost quadruple. We can conclude that moving the location counter inside the linker script's sections has a cost which impacts the image's size since it directly adds padding in the output file. However that increase does not change Unikraft's performance as we have observed in [5.1.5] since the increase is only made by padding that is never read.

5.2 Comparing To Previous ASLR

Now that we have reviewed all the characteristics of our solution, we are able to compare it to the one already implemented [1.2.2]. In order to ease further discussions, we will distinguish Daniel's implementation from ours with the name `bootloader_ASLR`.

Security

Generally speaking, the approaches taken are truly different, `bootloader_ASLR` randomizes only the base address of the image which means that latter is using $\log_2(0x40000000) = 30$ effective bits of entropy.

Following the paper's taxonomy[26, chapter.2], it can be described as,

- Stack : per-exec, full VM but correlated-object since it keeps its offset w.r.t all the other objects.
- Heap : per-exec, full VM but correlated-object since it keeps its offset w.r.t all the other objects.
- Executable : per-exec, full VM but correlated-object since it keeps its offset w.r.t all the other objects.
- Libs : libraries are part of the executable and share its properties but are not randomized inside it.

On the other hand our solution places the text section randomly in a 17 bit range but randomizes the offsets of the libraries (remember that everything is a library in Unikraft), the heap and the stack. Hence finding the base address does not guarantee that the exploit works.

Taking the taxonomy into account, we can describe our solution as,

- Stack : per-exec, partial VM and isolated-object.
- Heap : per-exec, partial VM and correlated-object since its starting address is made out of an offset of the `__END` pointer from the executable.

- Executable : per-object, partial VM and isolated-object since it keeps its offset w.r.t all the other objects.
- Libs : libraries are part of the executable and share its properties.

However they are randomized inside it : Per-object, correlated-object since the offset is computed in between libraries but they are shuffled.

From a security point of view, the two approaches seem pretty equivalent in the sense that the `bootloader_ASLR` has a fixed image layout but can be placed in a huge search space (the first byte can be placed in 2^{30} places). While our solution has an unpredictable image where every of its pieces are placed in smaller search spaces.

It is also worth noting that the executable part is randomized per object, meaning that our implementation is secure when the attacker can not read the binary of the application it tries to attack.

One way to make our implementation better could be to increase the search space by breaking the usual abstraction [2.1]. This paper [26, p.] invites us to do so, arguing that it increases the entropy since memory sections do not follow a deterministic pattern anymore. That would change the characteristics of our objects from partial VM to full VM.

Boot time

This ASLR gave pretty good result on booting time as the difference with an original Unikraft image without ASLR is negligible [5.1.5]. On the other hand, the `bootloader_ASLR` suffers from a +37.68% increase on boot time which is caused by the PIE code and the boot loader's own initialization[11, p.35].

Image size

Daniel didn't give any information or metrics about the size but we could assess it with an upper bound taking into account the boot loader's size. According to his thesis[11, p.18], the boot loader size is about 25kB, meaning that his solution would be around 222.5kB while ours is 782 kB. Nevertheless, an important element to think about is that his solution includes twice the same code while ours is only made of padding that inflates the file's size but is never read in virtual memory.

5.3 ASLR-Deduplication Methods Comparison

As already discussed in [4.2], there was two ways to implement the indirection table. We will now compare the two implementations and observe which one is the most efficient, *i.e* the one which maximizes page sharing and minimizes memory footprint.

5.3.1 Page Sharing

Different applications

To assess the performances of the two methods, we will compare 3 different applications built with ASLR, which have libraries in common :

- nginx: which is a simple http server.
- sqlite: which is a database handler.
- ftp: which is a service that allows to retrieve files from a server.

These applications will be run at the same time, then we will measure the amount of page shared between the kernels each 0.5 second during 20 seconds so that we can observe the convergence. The scripts used in order to measure pages are `launch_vanilla.sh`, `launch_gauthier.sh` and `launch_terry`. Those bash scripts are running the kernels and measuring the amount of pages that are being shared.

We measured how many pages were shared between the applications. Three scenarios have been measured : an indirection table is appended at the end of each libraries (Gauthier's solution), the global indirection table (our solution), the applications without indirection mechanism (vanilla).

Metrics were obtained in MB in order to assess the total amount of memory used by the images.

At start-up, we can observe from Figure 5.6 that the images with the global indirection table have a higher memory consumption at start-up than the appended ones, but the consumption can be compared to the vanilla version.

Convergence's speed is roughly the same whether or not the images were compiled with indirection tables.

Once convergence is achieved, the global table does not perform as good as the appended tables, indeed, while the append method consumes only 0.02 MB of memory, the other consumes 0.19 MB of memory.

In that configuration, we can consider it normal that both indirection methods are not saving up memory, with only 3 images, we are adding more pages due to the tables than what we get back from de-duplication.

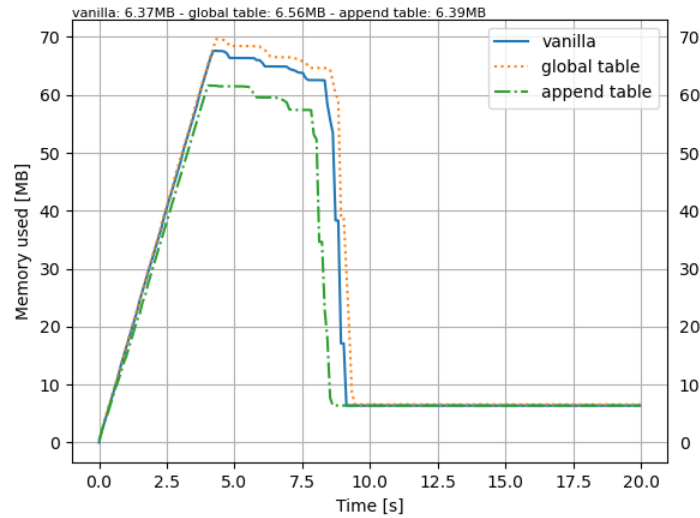


Figure 5.6: Indirection tables comparison

Same application

Now that we tested page sharing on 3 different applications, we will try 10 ftp images on which we also applied ASLR.

That means their memory layout will all be different while loading the same libraries and we will be able to observe if the memory de-duplication allows to mitigate ASLR's drawback which is its memory usage.

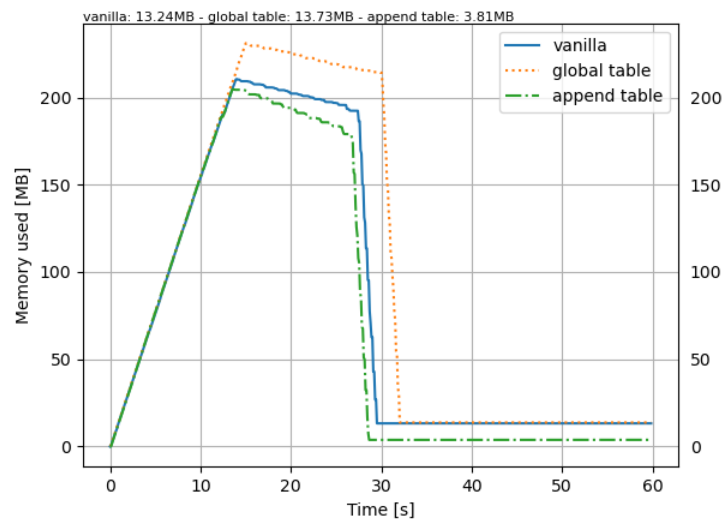


Figure 5.7: Memory on ten ftp images

We can observe from Figure 5.7 that results during start-up do not differ from what we saw previously. However, the global table suffers from a bigger spike that we attributed to measurement artifacts because the results of such

measurements are varying a lot regarding what is being run on the computer at the same moment.

Once converged, the solution with appended tables reduces the memory consumption of the 10 ftp images to 3.81 MB thus the latter saved 72.25% of memory footprint roughly 30 seconds after being run. On the other hand, the global table inflated the memory usage by 0.49 MB when compared to the vanilla images.

5.3.2 Selection

From the upper result, we can come to the conclusion that the append method should be used because it gives better results.

The other method didn't perform well because it must emulate far direct absolute jumps/calls, which are not supported by the modern x86 64 bits CPU.

That resulted in padding and poor page sharing, indeed, we were not able to patch all the required instructions because their size was either too small or their indirection caused crashes due to the tricks we used.

Moreover, the global table takes more memory as the libraries inside the latter should be always at the same addresses across kernels. It implies that images that do not use some libraries still have to pad the memory space in order to get the same absolute addresses, while this does not occur in the other implementation.

Finally, those conclusions should be tempered. Gauthier's solution is effective and useful only when we run multiple instances of Unikraft on the same hypervisor. As [Figure.5.6](#) shows it, using this method creates overhead when a few images are run.

Conclusion

6.1 Conclusion

After a lot of efforts and time, we managed to implement ASLR for Unikraft. It consists in two different steps : first, we randomize the code of the application by padding and shuffling the linker script before the linking procedure of the compilation. Then, we randomize the dynamic part of the program (i.e the stack and the heap) at run-time by generating random offsets that are added to their respective base address.

We showed that the solution was quite robust despite the limitations imposed by the current Unikraft design, indeed, we reached 17, 16 and 16 bits for the text section, stack and heap respectively. The latter does better than FreeBSD which is a well-known monolithic kernel. Regarding performances, the solution does not impact the boot time but quadruples the image's size.

In order to mitigate the drawbacks of ASLR, we did some research to make the latter compatible with memory de-duplication which is an hypervisor feature that shares similar pages across kernels. We investigated multiple approaches : a gcc plugin, using plt/got tables from PIE compiled executable and finally binary rewriting which was the one we used in the solution.

To obtain similar pages, we had to create indirection tables that were designed to contain instructions that made the pages unique. Those instructions were : calls, jumps and all the mov/lea instructions that refers to sections with their operands. With Gauthier, we discussed the solution and two methods were considered : a global table containing all the code and reserving space for all libraries or appended tables that are set at the end of their respective libraries. It was also decided that he would implement the second while I would be implementing the first.

Finally, both methods were compared and it turned out that the append one was performing better with regard to page sharing. The main reason is that the global table requires far direct absolute jumps/calls which are not supported on modern x86 64 bits CPU. This forced us to be inventive, indeed, we had to emulate such jumps/calls with multiple assembly instructions and patch the gcc compiler to correct offsets. However the latter was not the panacea, since all the concerned instructions could not be patched and those which could required excessive padding. Thus, we concluded that only the append method should be used in order to mitigate ASLR's memory usage.

6.2 Further Work

6.2.1 ASLR

The current implementation of ASLR could be improved and made more robust with two additional components that would result in a better entropy :

The first is memory range increase, indeed, we are currently restricted in `[0x100000-0x40000000]` which is quite limiting. If we were able to use the full potential of 64 bits computer, the search space would be `[0x100000-0xffffffff]`. Making it almost impossible for an attacker to find a relevant address given the size of the search space.

The second is breaking the memory layout abstraction made in [2.1] by letting the ASLR disrupt the order of the heap, stack and static memory. This is necessary because keeping the same order is a limiting factor for the ASLR, since it places the pieces of memory in VMA ranges in a deterministic order instead of being free to place them in the whole VMA space.

6.2.2 ASLR - De-duplication Compatibility

The placement of the global indirection table could be improved. We still rely on GNU ld to create the memory segments, our implementation only specifies at which address in virtual memory the section should be set.

As a consequence, it appeared that if the table was set with a starting address higher than `0x600000`. GNU ld seemed to create a new memory segment which had not the proper 'execute flag', thus the program stops on a Qemu error.

The only solution to that would be to do the segment mapping manually in the linker script, which is quite laborious.

Appendix

A Uniformity Verification

In order to assess the correctness of Python's documentation over its random functions, imported from the SystemRandom library. A script which generates random numbers in a given range and then plots the result in a histogram was made.

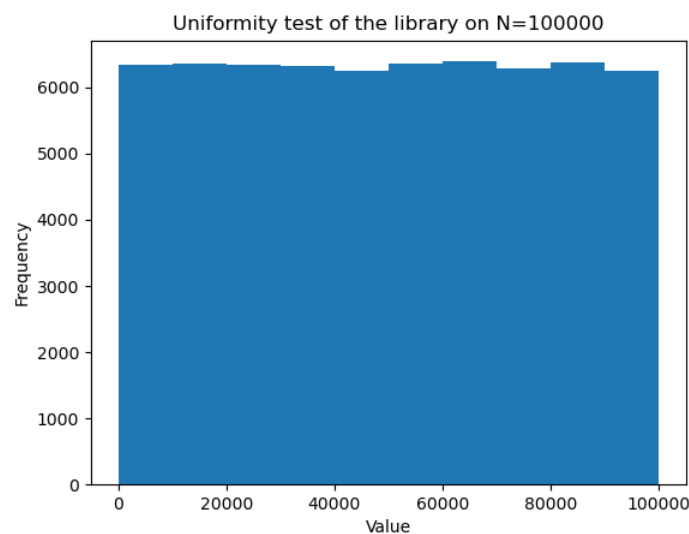


Figure A.1: Histogram on 100000 generated numbers

Figure.A.1 show us that the distribution of the random numbers is indeed uniform on the domain. We can thus safely use this library to implement ASLR.

B Requirements

To run this work, the user is advised to use the same versions of the tools that were used in this thesis. Furthermore, the solution is not guaranteed to compile

or run if any different version was used.

B.1 ASLR

To create an Unikraft image with ASLR enabled, python3 and python3.10 should be installed on the system. Moreover, the work was based on Unikraft 0.7.0 thus downloading it is recommended.

B.2 Deduplication

Multiples programs were used in order to make page sharing possible. The solution uses the different elements:

- Capstone version 4.0.1¹ (Python)
- Lief version 0.12.1² (Python)
- Python 3
- ld version 2.30
- gcc custom version, the latter can be found in the repository associated to this thesis.

C Machine

All the test were done on the following setup :

- CPU, Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
- RAM, 4x 4GO DDR4 corsair 2133 MT/s
- Motherboard, Asus PRIME Z370-P
- OS, Linux 5.16.0-kali6-amd64

D Linker Modifications

¹<https://www.capstone-engine.org/>

²<https://lief-project.github.io/>

```

$(KVM_DEBUG_IMAGE): $(KVM_ALIBS) $(KVM_ALIBS-y) $(KVM_OLIBS) $(KVM_OLIBS-y) \
    $(UK_ALIBS) $(UK_ALIBS-y) $(UK_OLIBS) $(UK_OLIBS-y)
$(call build_cmd,LD,, $(KVM_IMAGE).ld.o,\
    $(LD) -r $(LIBLDFLAGS) $(LIBLDFLAGS-y) \
    $(KVM_LDFLAGS) $(KVM_LDFLAGS-y) \
    $(KVM_OLIBS) $(KVM_OLIBS-y) \
    $(UK_OLIBS) $(UK_OLIBS-y) \
    -Wl$(comma)--start-group \
    $(KVM_ALIBS) $(KVM_ALIBS-y) \
    $(UK_ALIBS) $(UK_ALIBS-y) \
    $(KVM_LINK_LIBGCC_FLAG) \
    -Wl$(comma)--end-group \
    -o $(KVM_IMAGE).ld.o)
$(call build_cmd,OBJCOPY,, $(KVM_IMAGE).o,\
    $(OBJCOPY) -w -G kvmos_* -G libkvmplat_entry \
    $(KVM_IMAGE).ld.o $(KVM_IMAGE).o)
$(call build_cmd,LD,, $@,\
    $(LD) $(LDFLAGS) $(LDFLAGS-y) \
    $(KVM_LDFLAGS) $(KVM_LDFLAGS-y) \
    $(KVM_LD_SCRIPT_FLAGS) \
    $(KVM_IMAGE).o -o $@)

```

(a) KVM Linker.uk without ASLR

```

ifeq (y,$(CONFIG_LINK_ASLR))
$(KVM_DEBUG_IMAGE): $(KVM_ALIBS) $(KVM_ALIBS-y) $(KVM_OLIBS) $(KVM_OLIBS-y) \
    $(UK_ALIBS) $(UK_ALIBS-y) $(UK_OLIBS) $(UK_OLIBS-y)

$(call build_cmd,ASLR,, $@,\
    $(SCRIPTS_DIR)/ASLR/ASLR.py \
    $(ASLR_args))

$(call build_cmd,LD,, $(KVM_IMAGE).ld.o,\
    $(LD) -r $(LIBLDFLAGS) $(LIBLDFLAGS-y) \
    $(KVM_LDFLAGS) $(KVM_LDFLAGS-y) \
    $(KVM_OLIBS) $(KVM_OLIBS-y) \
    $(UK_OLIBS) $(UK_OLIBS-y) \
    -Wl$(comma)--start-group \
    $(KVM_ALIBS) $(KVM_ALIBS-y) \
    $(UK_ALIBS) $(UK_ALIBS-y) \
    $(KVM_LINK_LIBGCC_FLAG) \
    -Wl$(comma)--end-group \
    -o $(KVM_IMAGE).ld.o)

$(call build_cmd,OBJCOPY,, $(KVM_IMAGE).o,\
    $(OBJCOPY) -w -G kvmos_* -G libkvmplat_entry \
    $(KVM_IMAGE).ld.o $(KVM_IMAGE).o)

$(call build_cmd,COPY,, ASLR scripts,\
    cp $(SCRIPTS_DIR)/ASLR/relink_ELF.sh ../; \
    cp $(SCRIPTS_DIR)/ASLR/string_script.py ../)

$(call build_cmd,LD,, $@,\
    $(LD) $(LDFLAGS) $(LDFLAGS-y) \
    $(KVM_LDFLAGS) $(KVM_LDFLAGS-y) \
    $(KVM_LD_SCRIPT_FLAGS_ASLR) \
    -L$(BUILD_DIR) \
    -o $@)
else

```

(b) KVM Linker.uk with ASLR enabled

Figure A.2: Linker.uk modifications

Appendix B

Bibliography

- [1] S. S. Mike Loukides, “Microservices adoption in 2020.” <https://www.oreilly.com/radar/microservices-adoption-in-2020/>, July 15 2020. [Online; accessed May, 2022].
- [2] Aws, “Microservices.” <https://aws.amazon.com/fr/microservices/>, 2022. [Online; accessed May, 2022].
- [3] S. McConnell, “Code complete: A practical handbook of software construction,” *Second Edition*, 2015.
- [4] M. O. A. Mahmood Jasim Khalsan, “An overview of prevention/mitigation against memory corruption attack.” http://nectar.northampton.ac.uk/13420/1/Jasim_Khalsan_Mahmood_Okopu_Agyeman_Michael_ACM_2018_An_Overview_of_Prevention_Mitigation_against_Memory_Corruption_Attack.pdf, September 2018. [Online; accessed May, 2022].
- [5] Unikraft, “Unikraft,” <https://unikraft.org/>, 2021.
- [6] U. D. Team, “Session 01: Baby steps,” <https://usoc21.unikraft.org/docs/sessions/01-baby-steps/>, 2021. [Online; accessed January, 2022].
- [7] “Unikraft external libraries.” <https://unikraft.org/apps/>. [Online; accessed May, 2022].
- [8] Unikraft, “Unikraft performance.” <https://unikraft.org/docs/features/performance/>, 2021. [Online; accessed April, 2022].
- [9] Unikraft, “Hypervisor’s types.” [Online; accessed May, 2022], <https://usoc21.unikraft.org/docs/sessions/02-behind-scenes/>.
- [10] “Qemu.” <https://www.qemu.org/>. [Online; accessed May, 2022].

- [11] Daniel Dinca, “Memory randomization support in unikraft.” <https://drive.google.com/file/d/1wokQPLZKMJho4tocg0ANtjXTRlqZ6722/view>, 2020. [Online; accessed January, 2022].
- [12] G. Gain, “Memory deduplication with unikraft.” https://www.youtube.com/watch?v=HES65cNon44&t=340s&ab_channel=Unikraft, August 2021. [Online; accessed May, 2022].
- [13] G. TOLOMEI, “In-memory layout of a program.” <https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/>, 2015. [Online; accessed May, 2022].
- [14] C. staff, “Guide to x86-64.” <https://web.stanford.edu/class/archive/cs/cs107/cs107.1222/guide/x86-64.html>, 2021. [Online; accessed May, 2022].
- [15] L. Mathy, “Lecture notes in operating systems,” February 2021.
- [16] L. man page, “Linux syscall’s manual,” <https://man7.org/linux/man-pages/man2/syscalls.2.html>, 2021. [Online; accessed January, 2022].
- [17] H. B. Andrew S.Tanenbaum, “Modern operating systems,” *Fourth Edition*, 2015.
- [18] C. R. G. C. T. Adam Barth, Collin Jackson, “The security architecture of the chromium browser,” <https://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf>, p. 10, 2021. [Online; accessed May, 2022].
- [19] E. Styger, “Stack canaries with gcc: Checking for stack overflow at runtime,” <https://mcuoneclipse.com/2019/09/28/stack-canaries-with-gcc-checking-for-stack-overflow-at-runtime/>, 2018.
- [20] “Radare 2.” <https://rada.re/n/>. [Online; accessed May, 2022].
- [21] srishtiganguly1999, “Difference between relative addressing mode and direct addressing mode,” <https://www.geeksforgeeks.org/difference-between-relative-addressing-mode-and-direct-addressing-mode/>, 2020.
- [22] GNU, “Gnu object file format,” May 2022. https://www.gnu.org/software/guile/manual/html_node/Object-File-Format.html.

- [23] D. R. O. Randal E. Bryant, “Computer systems: A programmer’s perspective,” *Third Edition*, 2015.
- [24] D. J. Anil Madhavapeddy, “Unikernels: Rise of the virtual library operating system,” <https://queue.acm.org/detail.cfm?id=2566628>, 2014. [Online; accessed January, 2022].
- [25] L. Wehenkel, “Théorie de l’information et du codage.” <https://people.montefiore.uliege.be/lwh/Info/info-notes03.pdf>, September 2003. [Online; accessed May, 2022].
- [26] I. R. R. Hector Marco-Gisbert, “Address space layout randomization next generation,” *MDPI*, no. 2928, p. 25, 2019.
- [27] “Linux binfmt_elf.” https://elixir.bootlin.com/linux/latest/source/fs/binfmt_elf.c#L823. [Online; accessed May, 2022].
- [28] “Linux binary structure.” <https://elixir.bootlin.com/linux/latest/source/include/linux/binfmts.h#L17>. [Online; accessed May, 2022].
- [29] Michael Boelen, “Disabeling aslr in linux.” https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting/, 2018. [Online; accessed February, 2022].
- [30] Linux, “Linux mm.h.” <https://elixir.bootlin.com/linux/v5.16.7/source/include/linux/mm.h#L3180>, 2022. [Online; accessed February, 2022].
- [31] “Linux mmap.” <https://elixir.bootlin.com/linux/v5.17.9/source/arch/x86/mm/mmap.c#L82>. [Online; accessed May, 2022].
- [32] “Linux scheduler.” <https://elixir.bootlin.com/linux/v5.17.9/source/include/linux/sched.h#L728>. [Online; accessed May, 2022].
- [33] “Linux elf.h.” <https://elixir.bootlin.com/linux/v5.17.12/source/arch/x86/include/asm/elf.h#L337>. [Online; accessed May, 2022].
- [34] “Unikraft limit.h.” https://github.com/unikraft/unikraft/blob/staging/arch/x86/x86_64/include/uk/asm/limits.h. [Online; accessed May, 2022].
- [35] “Unikraft linker script assembly.” <https://github.com/unikraft/unikraft/blob/staging/plat/kvm/x86/link64.ld.s>. [Online; accessed May, 2022].

- [36] Python, “Random - generate pseudo-random numbers.” [Online; accessed May, 2022],<https://docs.python.org/3/library/random.html>.
- [37] GNU, “Creating object files.” [Online; accessed May, 2022],https://www.gnu.org/software/libtool/manual/html_node/Creating-object-files.html.
- [38] G. Organization, “Manuals, location counter.” [Online; accessed April, 2022],https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_3.html#SEC10.
- [39] “Unikraft plat folder.” <https://github.com/unikraft/unikraft/tree/staging/plat>. [Online; accessed May, 2022].
- [40] “Unikraft script folder.” <https://github.com/unikraft/unikraft/tree/staging/support/scripts>. [Online; accessed May, 2022].
- [41] GNU, “Command line options.” [Online; accessed May, 2022],https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html.
- [42] “Unikraft x86 setup file.” <https://github.com/unikraft/unikraft/blob/stable/plat/kvm/x86/setup.c>. [Online; accessed May, 2022].
- [43] “Unikraft random genetator.” <https://github.com/unikraft/unikraft/blob/staging/lib/ukswrand/swrand.c>. [Online; accessed May, 2022].
- [44] Intel, “Intel® 64 and ia-32 architectures software developer’s manual.” <https://www.intel.com/content/www/us/en/developer/technical-library/programming-guides.html?s=Newest>, March 2022. [Online; accessed May, 2022].
- [45] GNU, “Gcc plugins.” <https://gcc.gnu.org/wiki/plugins>, August 2018. [Online; accessed May, 2022].
- [46] F. Cloutier, “Call - call procedure.” <https://www.felixcloutier.com/x86/call>, May 2019. [Online; accessed May, 2022].
- [47] F. Cloutier, “Push — push word, doubleword or quadword onto the stack.” <https://www.felixcloutier.com/x86/push>, May 2019. [Online; accessed May, 2022].
- [48] P. Cordes, “Call an absolute pointer in x86 machine code.” <https://stackoverflow.com/questions/19552158/call-an-absolute-pointer-in-x86-machine-code>, April 9 2016. [Online; accessed May, 2022].

- [49] GNU, “Machine descriptions.” <https://gcc.gnu.org/onlinedocs/gcc-4.3.2/gccint/Machine-Desc.html>, May 6 2022. [Online; accessed May, 2022].
- [50] E. Dijkstra, “Go to statement considered harmful.” <https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf>, March 1968. [Online; accessed May, 2022].
- [51] V. Cerf, “Ascii format for network interchange.” <https://datatracker.ietf.org/doc/html/rfc20>, October 16 1969. [Online; accessed May, 2022].
- [52] H. D. Izik Eidus, “Kernel samepage merging.” <https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html>, November 7 2009. [Online; accessed May, 2022].
- [53] R. M. John Detter, “Performance and entropy of various aslr implementations.” <https://pages.cs.wisc.edu/~riccardo/736finalpaper.pdf>, 2015. [Online; accessed May, 2022].