

# Increasing Unikraft security by implementing address space layout randomization

Master Thesis



**Author:** Loslever Terry (Student ID: S174777)

**Supervisor:** Univ.-Prof. Dr. Mathy Laurent

**Co-Supervisor:** Gain Gauthier

Department of Electricity, Electronics and Computer sciences

Faculty of applied sciences

University of Liège

February 9, 2022

## Acknowledgement

## Abstract

[Abstract goes here (max. 1 page)]

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Theoretical Background . . . . .	1
1.1.1	Memory Layout . . . . .	1
1.1.2	Calling convention . . . . .	2
1.1.3	Memory Virtualisation . . . . .	2
1.1.4	Operating System . . . . .	3
1.1.5	Buffer Overflow Attack . . . . .	4
1.1.6	Position Independent Executable . . . . .	7
1.1.7	Object File . . . . .	7
1.1.8	Compilation . . . . .	7
1.1.9	Linker . . . . .	8
1.1.10	Unikernel . . . . .	9
1.2	Unikraft . . . . .	9
1.2.1	ASLR in Unikraft . . . . .	10
<b>2</b>	<b>State Of The Art : Linux</b>	<b>11</b>
2.1	ELF loading . . . . .	11
2.2	Pie Or NoPie . . . . .	11
<b>3</b>	<b>Solution</b>	<b>12</b>
3.1	Implementation . . . . .	12
3.2	Results . . . . .	12
<b>4</b>	<b>Sharing Randomized Memory</b>	<b>12</b>
<b>A</b>	<b>Appendix</b>	<b>13</b>
	References	14

## List of Figures

1figure.caption.7

2     Stack convention . . . . . 2

3figure.caption.9

4     Linux system representation[4] . . . . . 4

5     Vulnerable C code . . . . . 5

6     Program compilation . . . . . 5

8figure.caption.16

## List of Tables

# 1 Introduction

## 1.1 Theoretical Background

For the reader convenience and in order to have a reminder, The following section will review all the theoretical points that may, at some point, be addressed by the 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.

### 1.1.1 Memory Layout

In modern OSes, programs that we can assimilate to process for this example are given a private virtual address space in order to run. The structure of this address space consist in the following elements depicted in [Figure.1](#)

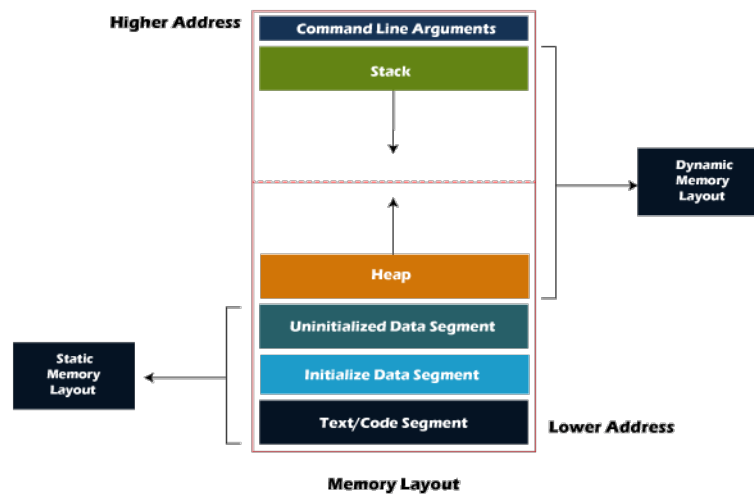


Figure 1: Program memory layout <sup>1</sup>

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, is located the stack which grows to lower addresses as we do functions calls or allocate local variables. There's also the heap which is responsible to store the dynamic allocated memory, growing 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 encapsulates 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 are given a value or not in the source code.

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

### 1.1.2 Calling convention

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

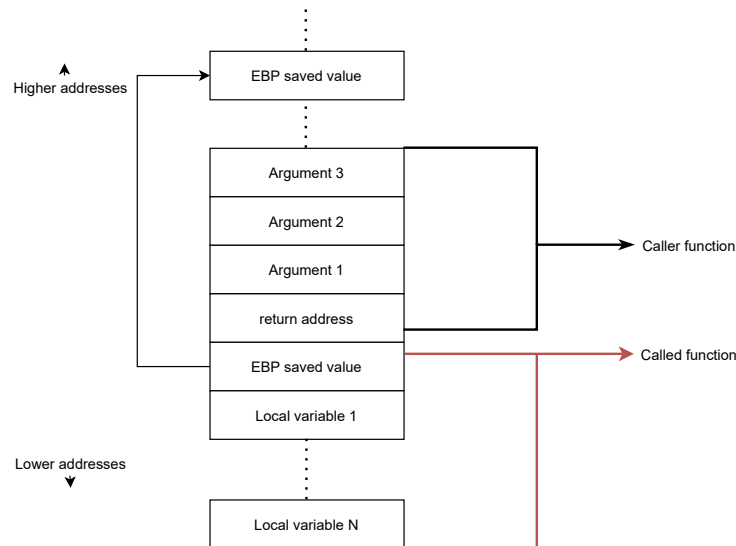


Figure 2: Stack convention

Figure.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 the local variables of the functions.

### 1.1.3 Memory Virtualisation

Programs are not given true physical memory. Modern OSes implement what is called "Virtualisation" which allows to emulate a possible infinite memory. It

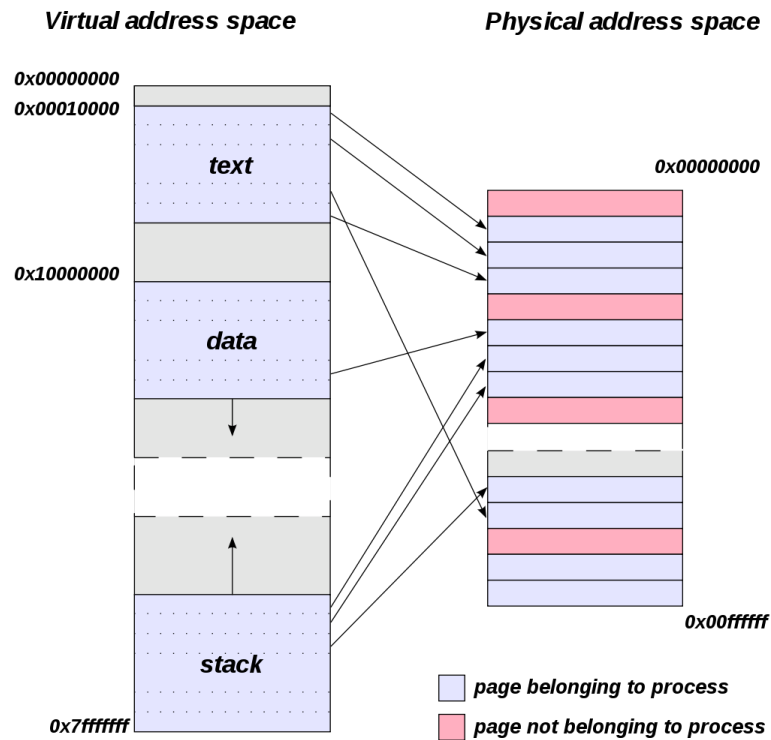


consists in a mapping between "pages" and "frames". Physical memory is split among multiples frames of fixed size which is a contiguous block this memory, usually it's 4KiB. However, recent instruction set architecture support multiple page sizes, which reduces the size of the page table.

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 the to 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 it to notice it, because from its 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.3 shows a mapping between the



page and the frame.

#### 1.1.4 Operating System

An operating system (*a.k.a OS*) is a "software that acts as an intermediary between user programs and the computer hardware"[7].

It aims at using efficiently the computer hardware by scheduling applications, caching data, providing services, allocating memory. The later can be considered as the base block which allows the softwares to run on the computer. It also

makes programming easier by providing a set of functions called "a programming interface".

Most common OSes can be split in two 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 think he have direct access if it runs on top of an hypervisor[1]. The second, runs in user mode and only has access to logical memory<sup>1.1.3</sup> 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.

The OS creates also a set of abstractions such as processes, threads, files, file system, sockets, ... is a non exhaustive list of these abstractions which are created in order to use the hardware conveniently.

Those services can be called through an API called "Syscalls"[6] which are a set of functions that switches the system from user to the kernel space. This way a program can get access to the abstractions made by the OS and described previously.

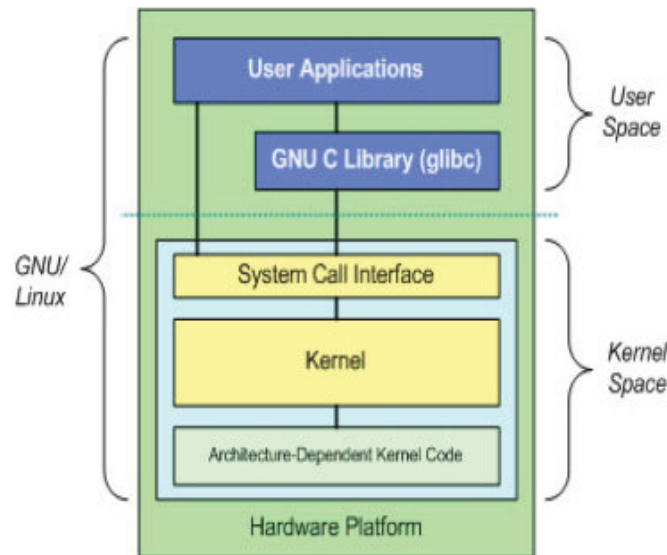


Figure 4: Linux system representation[4]

There exists various architectures : Monolithic (Linux), Microkernels (Mach, from which MacOS X is partially made), Unikernels (Unikraft), hybrids, ... Which all include different abstractions and works in different ways.

### 1.1.5 Buffer Overflow Attack

This attack aims at either crashing a program or injecting malicious code in order to initiate a privilege escalation. It's made possible when the software writes/copies data into a buffer without checking its size before hand. So the intruder gives an input 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.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, or tell him if 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.

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

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

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

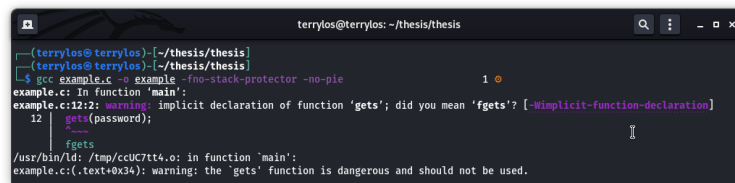
    printf("Enter the password :\n");
    gets(password);

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

Figure 5: Vulnerable C code

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 following flags :



```
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:12:22: 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 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 the OS on the stack such as canaries [10]. The other one, `-no-pie` disables the position independent executable option, the reader can read more about it in its corresponding subsection [red ??]

A quick look with **radare2**<sup>3</sup> allows us to identify the 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**. At this state, the program still haven't modified it's stack base pointer and hasn't allocated the char buffer. There's also the hidden function at location **0x00401142**, it is the address towards which the program has to jump to print the secret message.

(a) Main's return address

(b) Hidden function address

Let's put a breakpoint 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 (framed in red) of the stack before rewriting the return address (framed in blue). Therefore, entering 18 times "a" then 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.

<sup>3</sup><https://rada.re/n/>

### 1.1.6 Position Independent Executable

A position independent executable is an executable in which its 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.

PIC works thanks to relative addressing which specifies the jump address with respect to the current instruction pointer position (EIP) [9] and global offset tables (GOTs). The last binds the code symbols set by the compiler to their memory addresses. GOTs are placed to a certain offset from the code, but it is only known in the linking phase. Therefore, in runtime, PIC execute an indirect jump to the GOTs table which is placed at a known offset from the code, before reading the address of the wanted function and jumping to it.

PIE allows to shared piece of code between multiple programs such as commonly used libraries. 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. This has a slight efficiency cost since we add an indirection when compared to absolute code in the jumping process.

### 1.1.7 Object File

Object files can usually be found on computer with the ".o" or ".obj" extensions. It contains the machine code of the application but can not yet be run by the OS. Moreover, useful meta-data generated by the compiler can be found : symbol tables that eases linking process by giving elements' type, name and length if it is a function. Debugging information and profiling information.

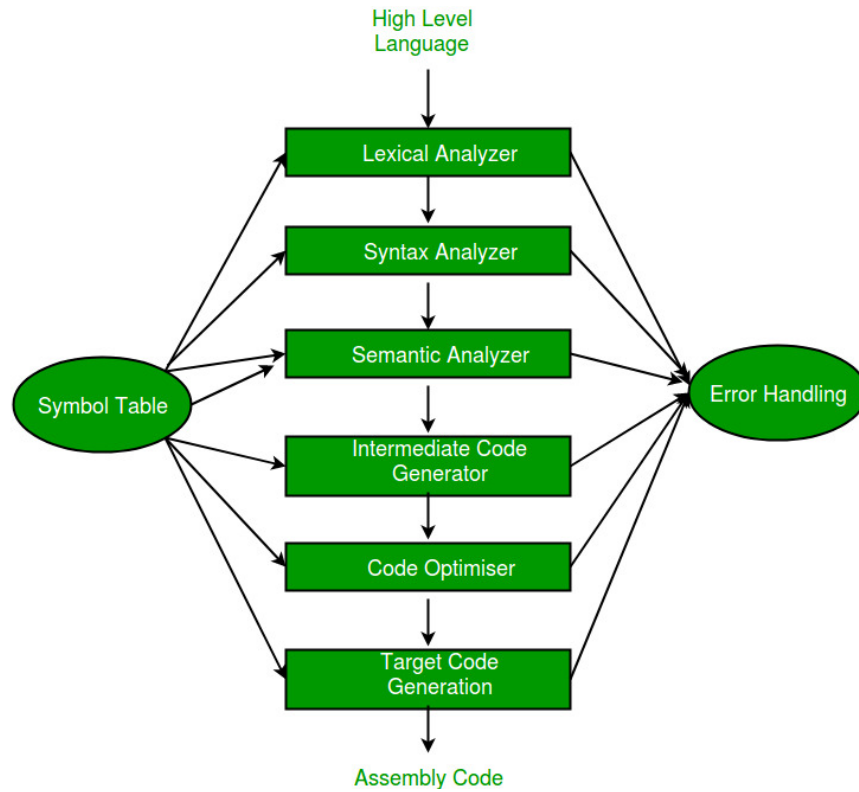
### 1.1.8 Compilation

In order to create the executable file (which is `elf` format in Linux), the source code has to go through multiple steps in order to be run on the CPU. The compilation step is the action to transform a program code file such as : C,C++,GO,Java,Rust,... To an object file. The underneath figure shows the diverse actions done by a compiler.

The addresses given to the functions or pointers, by the compiler, are local symbolic addresses. They only make sense inside the file and doesn't yet refers to actual addresses managed by the OS. As an example :

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

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

Figure 8: Compiler steps <sup>4</sup>

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

### 1.1.9 Linker

Linking is the final step of the compilation phase which generates the final executable file. It consists in loading all the object files[1.1.7] that are part of the software. Then it binds them together with libraries (if any are used) in a final file, where the symbol addresses are mapped to logical address space.

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 files which are resolved when the following is executed. Thus, we only have a reference to the used library which has multiple advantages :

- allows to share the common libraries between the programs by only loading them once in memory.
- can update libraries code without necessarily compiling again the whole programs that were using it.

### 1.1.10 Unikernel

Unikernels[2] are single memory space OSes meaning that there exists no distinction between an user or a kernel 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 later.

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

- Secured environment : brings the smallest possible set of library and thus code. Which decreases the attack surface for malicious softwares. When multiples Unikernels run the same hardware, the hypervisor is the one from which depends the security on the whole system since it assures 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<sup>5</sup> 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 later can be loaded in memory and executed rapidly (in the order of the 10's ms depending on the implementation)
- Low footprint: Derived from the fact that the image is a small as possible. Thus, less function called are made before running the application, therefore reducing stack memory use.

These advantages makes them an arguably good choice for certain clouds solution.

## 1.2 Unikraft



Unikraft[12] 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 Union research and innovation program.

---

<sup>5</sup>Each time the reader will encounter this term with regard to OSes, it refers to Monolithic kernel/Microkernels which are heavily deployed in desktop computers or servers.

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

As traditional unikernels, it is organised in a set of libraries that are compile together in order to create the final executable. In which there is core elements being some platform and architecture dependent code[11], then comes "internal" libraries[11] which are composed of elements that are embedded in traditional OSes : hardware drivers, filesystems, schedulers, ... And "external" libraries which are applications ported to Unikraft in order to be run on the system. Ported libraries can be found at this address<sup>6</sup>.

### 1.2.1 ASLR in Unikraft

ASLR has previously already been implemented by DINCA Daniel[3] as a thesis subject, Its working is the following : it generates a valid random address where the kernel can be placed and then loads it at the later. That implementation shifts the whole kernel from the same offset. This required to modify Unikraft bootloader and give corrections to the constructor vector at runtime.

---

<sup>6</sup><https://unikraft.org/apps/>



## 2 State Of The Art : Linux

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 implements per process randomization. Hence the following section should be considered as an example of ASLR's implementation.

ASLR can be turned on or off by setting `randomize_va_space` macro to 0[8], 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` [5].

### 2.1 ELF loading

First, the ELF file is loaded into memory thanks TODO

### 2.2 Pie Or NoPie

Depending on the flat set during the ELF's compilation, ASLR won't behave the same way as depicted upper. Indeed, when the ELF is compiled as PIE, the whole program and its shared libraries are loaded at a random address in the virtual address space TODO

In `include/Linux/randomize_kstck.h` are the macros which compute the random offset.

In `arch/x86/kernel/process.c` -> `arch_align_stack` `arch_randomize_brk`  
 -> au lieu d'aller direct dans la structure ,peut-être regarder à `brk` -> set nbr dans l'elf pour pouvoir disable, modifier `process.c` -> Utilise des éléments de `sys/prctl.h` qui est dans `include` de la `mm` lib (invest sur `va_start`) -> Contient la struct et des defines -> `va_start`, `va_end` ne sont que de la gestion de listes de param -> `/fs/binfmt_elf.c` sont les fonctions qui chargent les elf -> `fs/exec.c` qui se charge de charger le programme et de le mettre en mémoire -> l'espace mémoire est donnée dans la structure `Linux_binprm` de l'`exec` et on lui attribue un espace mémoire `mm_struct` et `vm_area_struct` -> `setup_arg_pages` : update les flags et la position du stack peut être relocatilisée -> Ils jouent direct avec le stack depuis la structure `vma` et font du page align. Il faut trouver l'équivalent en Unikraft pour trouver la `vma` et le stack `Linux_binprm` est la structure qui est utilisée pour le chargement des librairies.

### 3 Solution

Since Unikraft doesn't include and doesn't need the process abstraction by nature, one could think about another way than the Linux one when implementing ASLR. The idea is to modify Unikraft's linker script before linking in order to modify the memory's disposition. This will result in a multitude of different Unikraft image despite having the same payload because each of them will have a different memory layout.

Hence, achieving ASLR without any modification on how the is loaded OS and how it loads the application.

#### 3.1 Implementation

#### 3.2 Results

### 4 Sharing Randomized Memory

## A Appendix

...

## References

- [1] ANDREW S.TANENBAUM, H. B. Modern operating systems. *Fourth Edition* (2015), 474–478 pg.
- [2] ANIL MADHAVAPEDDY, D. J. Unikernels: Rise of the virtual library operating system. <https://queue.acm.org/detail.cfm?id=2566628> (2014). [Online; accessed January, 2022].
- [3] DANIEL DINCA. Memory randomization support in unikraft, 2020. [Online; accessed January, 2022].
- [4] LANIER WATKINS, WILLIAM H.ROBINSON, RAHEEM BEYAH. Using network traffic to infer hardware state: A kernel-level investigation, 2015. [Online; accessed January, 2022].
- [5] LINUX. Linux mm.h, 2022. [Online; accessed February, 2022].
- [6] MAN PAGE, L. Linux syscall’s manual. <https://man7.org/linux/man-pages/man2/syscalls.2.html> (2021). [Online; accessed January, 2022].
- [7] MATHY, L. Lecture notes in operating systems, February 2021.
- [8] MICHAEL BOELEN. Disabling aslr in linux, 2018. [Online; accessed February, 2022].
- [9] SRISHTIGANGULY1999. Difference between relative addressing mode and direct addressing mode. <https://www.geeksforgeeks.org/difference-between-relative-addressing-mode-and-direct-addressing-mode/> (2020).
- [10] STYGER, E. 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).
- [11] TEAM, U. D. Unikraft core. <https://github.com/unikraft/summer-of-code-2021/blob/main/content/en/docs/sessions/02-behind-scenes/index.md> (2021). [Online; accessed January, 2022].
- [12] UNIKRAFT. Unikraft. <https://unikraft.org/> (2021).