Master research Internship

Bibliographic report

Software Fault Isolation using the CompCert compiler

Domaine: Cryptography and Security

Author:
Alexandre DANG

Supervisor:
Frédéric Besson
Team CELTIQUE

Abstract:

Contents

1 Introduction

- Secure malicious code through software solution
- Usage in applications which use modules from unknown origin (browsers, computer clusters)
- current appeal for SFI speed and small TCB
- SFI is still incomplete, especially with ROP attack =¿ our approach
- plan

2 *Software Fault Isolation*

We introduce here *Software Fault Isolation* (SFI) which inspired us the idea to protect return addresses through fixed stack frame size. SFI aims to protect a main program from the different modules that he will need to use. These modules will be loaded in the same memory space as the main program but in a confined area called *sandbox*. The SFI mechanism is composed of two elements: a code generator and a verifier. The generator transforms the assembly code of the hazardous modules so that they will be constrained in the sandbox. The verifier operates just before loading the modules in the memory. It checks the if SFI transformations introduced by the generator are still present and valid. For the rest of the document we will reserve the word ”program” to refer to the code protected by SFI and ”module” to refer to the hazardous code.

2.1 Principle

The main principle behind SFI was first presented in the work of Wahbe and al. ref . Later works ref, which will be introduced in the chapter 2.1.4, are all based on the foundations of SFI detailed here. The implementation described here was realised for a RISC architecture like MIPS or *Alpha*.

SFI considers that a malicious code is effectively contained in the sandbox if these three security properties hold true:

- **Verified code**, only instructions that have been checked by the verifier will be executed
- **Memory safety**, malicious modules won't do any *write* or *jump* operations out of the sandbox
- **Flow control integrity**, every flow control transfer from hazardous modules to the main program is identified and verified

The first property protects us against self-modifying code which could bypass the SFI measures. *Memory safety* prevents any illegal access to the memory of the protected program. The last property allows us to authorized only licit interactions between the program and its modules. SFI forbids any call from malicious modules that could modify the flow control of the program. If the flow control was fiddled with, it could lead to an unexpected behaviour of the program which we want to avoid.

The code generator transforms the assembly code of the hazardous modules so that respect the security properties presented before. The generator is integrated to the compiler which will create *sandboxed executable*. Afterwards this executable will be checked by the verifier before being loaded in the memory. It verifies that the transformations introduced by the generator are present and valid. If the verification fails the module will be rejected and won't be executes. We can note that we only need to trust the verifier to prevent running any dangerous module. It's one advantage of SFI, only the verifier needs to be in the *Trusted Computing base* (TCB).

2.1.1 Code generator

To protect the program from its modules, the generator will restrain every write and jump instructions of the modules to addresses of their sandbox. The generator has to face three issues to do so. Firstly, is to introduce protection mechanisms before every dangerous instructions. For example, assessing that the address of a jump instruction is an authorized one. Secondly, we have to make sure that these protection mechanisms can't be avoided. Finally, the transformations introduced have to authorized only legal calls out of the sandbox by using entry points specified by the protected program. For example, Google Chrome only allows its modules to use a specific interface to interact with the browser. This way the modules can't disrupt the flow control of Google Chrome easily.

Confining memory accesses The main program memory should avoid being corrupted by its modules. SFI aims to isolates these modules in a reserved of the program's memory called sandbox. The sandbox is a contiguous memory area which size is a power of two. Indeed, these requirements eases the confinement of the modules in their sandbox by using arithmetic operations on bits which accelerates the process.

Protection of sandboxing mechanisms

Controlled interactions with the protected program

2.1.2 Code verifier

2.1.3 Pros and cons

2.1.4 Implementations

NativeClient, SFI for Google Chrome

2.2 SFI using CompCert

2.2.1 CompCert the verified compiler

CompCert

Memory model of CompCert

2.2.2 SFI with CompCert

Cminor

Specification of the SFI transformation

Masking in CompCert

2.2.3 Evaluation of the approach

2.3 Limits of SFI

2.3.1 Return addresses

2.3.2 Proposed solution

3 Overview of the approach

Many attacks on software aims at diverting with the control flow of the targeted program. Among those, *Returned Oriented Programming* (ROP) attacks specifically try to overwrite the return addresses. By doing so the attacked function will return to a malicious piece of code that will get executed. Stack overflow is an example of such ROP attacks. We propose a solution against ROP attacks which combined with SFI would protect from most of control-flow interference attacks. Inspired from SFI techniques we aim to prevent any overwriting of the return addresses. To do so we need to know these return addresses locations in the memory. Therefore our approach consists of modifying the stack structure in order to have a way to distinguish the return addresses locations. With this knowledge we will be able to put a mask, as in SFI, before every dangerous write instructions and prevent any ROP attack.

3.1 ROP attack

We want to protect our program against ROP attacks. These attacks are directed against the stack and especially the function return addresses located in the stack. We will begin by a short introduction about the mechanisms behind the stack. Then we will explain how ROP attacks work with the example of a classical buffer overflow.

3.1.1 The stack

The stack is a specific area of the memory of a program. The memory allocated to a program is divided among multiple areas like the stack (which we are going to detail), the heap (where we put dynamically allocated or global variables), the code... The stack is composed of frames and each of them are linked to a function being executed. Frames are piled up on the stack following the FIFO rule (*First In First Out*). Explicitly, every time a function is called, a new frame is created and placed on the top of the stack. Reciprocally when a function terminates its frame will be popped out of the stack. Frames contain multiple kind of data related to their function like local variables, parameters of the function, return addresses... Return addresses indicates the point of execution to return to after a function terminates. When popping a frame the program is supposed to execute the code at the address matching the value contained in the return address. ROP attacks aims to overwrite these return addresses which enables them to execute malicious code hidden in another part of the memory instead of continuing the normal flow of the program.

3.1.2 Buffer overflow

Stack overflows are the most popular ROP attacks. In Figure 1 we can see an example of buffer overflow written in C. The goal of this code is to execute the function called *evil_code()* which just prints "Argh, we got hacked!\n" line 6 of Figure 1. We can see that in a normal execution *evil_code()* should not be executed since it is never explicitly called. This code was compiled with *gcc -m32 -fno-stack-protector* to remove all stack protections used by gcc. The output of the code of the successful buffer overflow can be seen in Figure 2.

We see in the Figure 2 the consequences of the buffer overflow in red. The stack was overwritten and the return address was modified to the address of *evil_code()* which code was successfully executed.

```c
1 #include <string.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 void evil_code() {
6         printf("Argh, we got hacked!\n");
7 }
8
9 void foo(char* input){
10         char buf[1];
11         printf("\nStack before:\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n\n");
12         strcpy(buf, input);
13         printf("\nStack after :\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n%#010x\n\n");
14 }
15
16 int main (int argc, char* argv[]) {
17         void (*a)(void) = evil_code;
18         printf("Address of evil_code = %#010x\n", evil_code);
19         if (argc < 2) {
20                 printf("Need an argument\n");
21         } else {
22                 foo(argv[1]);
23         }
24         return 0;
25 }
```

Figure 1: Example of buffer overflow in C

We are going to explain how the attack works. In the function *foo()* the lines 11 and 13 print the stack. *%#010x* formats the output in hexadecimal with `0x` at the beginning for addresses. The vulnerability resides in the function *strcpy* line 12. *strcpy* just copies characters one by one until it finds “0”(which corresponds to the end of a string) in the source string. However our source string can contain many more characters than *buf* is supposed to have. Indeed *buf* is declared line 10 as an array of 1 character and our source string is the argument that we give to the program. If the source string is bigger than the destination *strcpy* will just continue to write the source string over others variables location in the stack and possibly reach the return address. After few tries and fails we found the correct input to successfully do the buffer overflow. This input can be seen on the first line of Figure 2 which is _python -c ’print 13*”a”+”\x7b\x84\x04\x08‘_ or _aaaaaaaaaaaaa\x7b\x84\x04\x08_
In our example we filled the stack with “a” which corresponds to “61” in ASCII until we reached the return address. We can see the consequence of the attack in the output Figure 2, where the stack is full of 61 after executing *strcpy*. When we reached the return address we overwrote it with

the address of *evil_code* which was `0x0804847b` given on the second line of Figure 2. This way, the next instruction that will be executed after *foo* finishes will be the function *evil_code*. At the end of the program we can see that we get a *Segmentation fault (core dumped)*, which is normal because we messed up the stack when we overwrote is with “a”. But since we managed to execute *evil_code* the attack is still successful.

```
terminal$  ./buffer $(python -c 'print 13*"a"+"\x7b\x84\x04\x08"')
Address of evil_code = 0x0804847b
Stack before:
0xf7712000
0xff957998
0xf7593d26
0xf7712d60
0x0804868c
0xff957978
0xf7593d00
0xf7713dc0
0xf77828f8
0xff957998
0x08048510            //Return address of foo

Stack after :
0xff958161
0xff957998
0xf7593d26
0xf7712d60
0x0804868c
0xff957978            //Buffer overflow

0x61593d00            //"a"
0x61616161            //"aaaa"
0x61616161            //"aaaa"
0x61616161            //"aaaa"
0x0804847b            //"\x7b\x84\x04\x08", evil_code address

Argh, we got hacked!   //Success! evil_code was executed
Segmentation fault (core dumped)
```

Figure 2: Output from buffer overflow example

3.2 Description of the approach

We want to protect programs against ROP attacks like the buffer overflow seen previously. We want to prevent any return address from being overwritten illegally. The only moment they should be written over is during a function call routine. We want to be able to add runtime checks in the code like SFI, therefore we need to be able to check if an address is the location of a return address.

The biggest difficulty is to be able to know if a location in the stack corresponds to a return address or not. Indeed the stack grows through function calls which pile up stack frames. These frames are constructed dynamically depending of the function, hence the locations of return addresses aren't easily known. As it is we don't have enough information to correctly protect return addresses since we don't know precisely where they are located.

Several solutions exists against this issue. We could for example add a lot of meta-data during the compilation to have extra information and then effectively protect the return addresses. Another solution is to create a second stack called *shadow stack*. We would then have complete control over the *shadow stack* which allows us defend against ROP attacks. maybe expose the cons of these approaches from our point of view

Our solution is to modify the current stack structure to be able to know the return addresses locations easily. The main idea is to fix a constant offset n between return addresses allowing us to exactly know where a return address is located relative to the others. We will explain thoroughly the approach that we want to apply in the following section.

3.2.1 Proposed solution

Fixing return addresses locations and stack alignment We want to be able to decide if a pointer points to a return address at runtime. With this knowledge we will be able to detect if an instruction may compromise our program. The idea is to modify the stack layout in order to have a constant offset n between neighbouring return addresses. This way we know that the neighboring return addresses are always located at a distance n from a frame return address. Furthermore all the other return addresses are separated by a distance which is necessarily a multiple of n. For example let's say we know the location of a return address, we call this location c. Since all neighboring return addresses are separated by n, we know that the following return addresses locations will be $c+n$, $c+2n\dots$ Reciprocally the previous return addresses will be located at $c-n$, $c-2n\dots$ Thus we have a global formula expressing the location of all return addresses:

c mod n, with ***c*** *the location of one of the return addresses and* ***n*** *the size of the frames*

The next hurdle is to choose n and c cleverly. For n, the most important thing is that frames have enough space to store all the needed data. Therefore we define the value of n as the biggest frame size of all the functions in a program. If the return addresses are separated by this amount we are sure that every function will have enough space in the stack for its frame. Afterwards we have to define c. The best way that we found is to modify the stack in order to have the first return addresses location to be equal to c. If we are able to do such a thing, we can also easily define the value of c and for simplicity we chose $c = 0$.

The Figure 3 pictures the transformation we want to apply to the stack. On the left we have represented an usual stack with return addresses all over the place. Since these locations are almost random it's really difficult to pinpoint their location. After transforming the stack (stack on the right) we can see that the different addresses are separated by the same constant n. We can also notice that we fixed the location of the first return address c with the value 0. Then we are able to know all the return addresses locations following the formula **0 mod n**.

Detection of dangerous instructions The second step is to detect every possibly harmful instructions to return addresses. We consider as dangerous every instruction that can freely write to the memory. Our approach is mainly related to the C language. In C, instructions

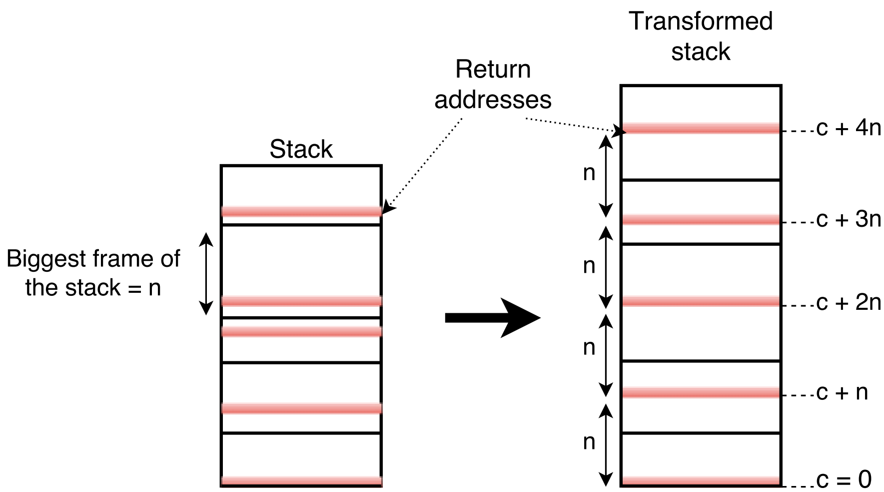

Figure 3: Stack modifications

that fit such criteria are assignment to pointer dereference in the form of *(pointer)** = *value* or *(pointer+offset)** = *value*.

In the previous example of buffer overflow Figure 1, the vulnerability resides in the function *strcpy* line 12. To pinpoint the dangerous instruction let's check the source code of *strcpy* Figure 4 from Apple. We can see in the *while* loop line 10 that *strcpy* copies characters one by one from the source string *s2* to the destination *s* until it finds a character equals to *0*. To copy the characters, *s* and *s2* are pointers which initially point to the memory area of the destination and the source string. Then until it finds a character matching *0* the value pointed by *s2* is copied to the location pointed by *s* and the pointers are incremented. The harm happens when the source *s2* is much longer than the destination. In this case we continue to copy to the location pointed by *s* even if the memory written to does not belong to the destination string anymore.

In this example we see clearly that it's the pointer dereferencing that allows one to write directly in the memory. For that reason we target such type of instructions in our approach.

```
1 char * strcpy(char *s1, const char *s2) {
2         char *s = s1;
3         while ((*s++ = *s2++) != 0)
4                 ;
5         return s1;
6 }
```

Figure 4: *strcpy* source code from Apple

Securing dangerous statements Finally when we have detected all the dangerous statements we transform the module code. Before each of this dangerous statement we add a protection mechanism similar to masking in SFI. The algorithm of the check is represented in Figure 5:

1. We check if the address is in the stack. Return addresses only exist in the stack, we don't need to concern ourselves with the other accessible memory area: the heap.

2. If the address is in the stack we check if the target address matches our formula **0 mod n**. If it does then it's a return address location.

3. If a target address abides by the two previous condition, it's an illegal instruction and we make the program crash. If it does not then the program just continue to run like normal.

We want our implementation to respect the property of transparency, if a program is safe then our transformation does not modify its behaviour.

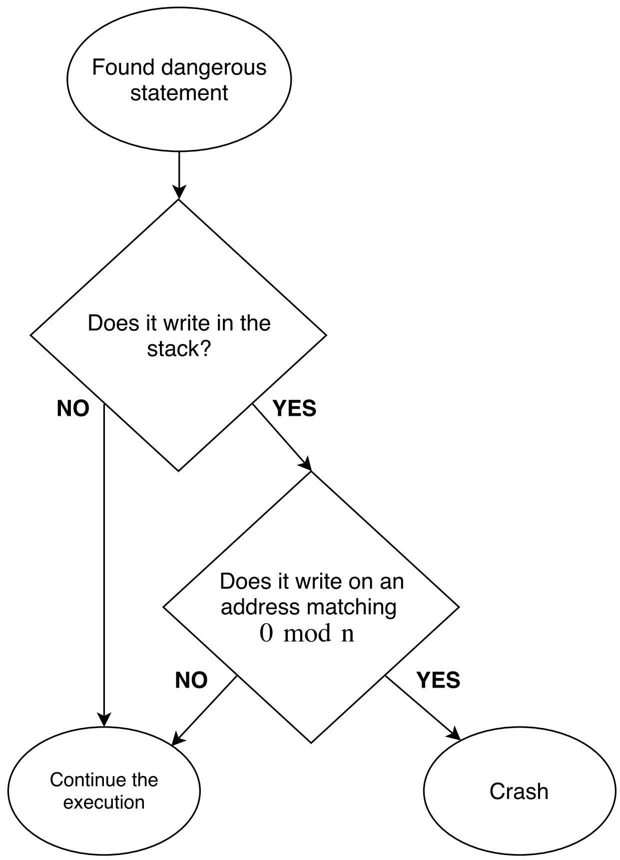

Figure 5: Runtime check algorithm

To sum it up, our approach aims to have an easy way to know return addresses location and then add a check at runtime before every dangerous instruction to prevent illegal writing on return addresses location. To do this we divided the approach into four phases:

1. Fix stack frames size

2. Align the stack

3. Detect dangerous statements

4. Secure the dangerous statements

3.3 Security properties

The approach we propose is composed of four phases, to get the confidence that our idea is effective in protecting return addresses we are going to formalize the properties we expect from each phase. Furthermore like we pointed earlier, we are going to work with the certified compiler CompCert. The ideal way to be sure of our idea would be to prove it with Coq the proof assistant the language used to build CompCert. By working with these tools we hope that one day we will be able to prove some security guarantees brought by our approach.

1. **Fixed stack frames size**
   - Return addresses locations are all separated by a constant offset bigger or equal to any frame of the stack
2. **Stack alignment**
   - The first return address location of the program has its least significant bits equal to 0
3. **Detection of memory write statements**
   - Every statement of the analysed code that might modify the stack memory state is detected
4. **Securing memory write statements**
   - The protection will trigger an error behaviour if we try to write on a protected address

1. and 2. combined give us the guarantee that the least significant bits of all the return addresses location will be equal to **0 mod n** with n the fixed offset between return addresses. Basically we make it so the protection mechanism prevents any write on addresses located in the **stack memory area** with their least significant bits equal to **0 mod n**.

Another property we didn't mention yet is that all our transformations need to be transparent. In other words, if we apply our methods on a program which is already safe then its behaviour is not affected. We will explain how we ensure this property more thoroughly in the Chapter 4.

For these properties to always hold true we need to place some conditions which we are going to list in the following section. Our approach guarantees that if all the properties mentioned are fulfilled the program will be protected against any ROP attack.

3.4 Analysis of the approach

3.4.1 Conditions

The solution we've just presented can bring very strong security properties against ROP attacks. However for this approach to work we need certain hypothesis to be true. Indeed some of the properties enumerated before become false after certain operations.

- **Stack modifications**, every operation that disrupts the stack structure may nullify our property that says "every return addresses are separated by a fixed offset". For example x86 architecture use the ESP register to keep track of the stack growth. If we fiddle with it we may introduce a shift in the return addresses location. Then our runtime check on **0 mod n** addresses would not be relevant anymore. For example, the Figure 6 shows a piece of inline assembly which disrupts the stack line 2. Inline assembly allows one to put some assembly code in the middle of C code. Here the assembly decrements the stack pointer stored in ESP. By doing so the stack will be shifted by an amount of 50 bytes and our formula to the locations of return addresses won't be correct anymore.

- **Unsecure libraries**, for our approach to work we need to have all dangerous write statements to contain our runtime checks. Hence all executed code must have been compiled with our transformation. For example, the *glibc* library of C contains multiple insecure functions like *printf, strcpy...* Furthermore those flawed functions are common vulnerabilities for *buffer overflows* attacks which are a type of ROP attack. To avoid this issue we would need to rewrite the glibc or compile it with our tools.

- **Modules need the same offset**, if a program uses multiple modules or library they need to be compiled with the same offset n. Indeed if the offset of the different modules are different we cannot use the previously defined formula **0 mod n** cannot be used anymore. Thus it's not possible to easily know if a location corresponds to a return address.

TODO substitute with the right syntax when finished

```
1 int foo(int a) {
2         asm(``\$sub 50, \%esp'');
3         //This line does the operation ESP = ESP - 50
4         //This disrupts the stack layout we establish in our
              transformation
5         printf("Hello world!");
6 }
```

Figure 6: C inline assembly

3.4.2 Discussion

We have presented the principle of our approach in this chapter. Then we mentioned some necessary conditions for our solution to work properly. In this section we are going to discuss about the pros, cons or remarks about the proposed solution.

The benefits of our transformation is clear, any code compiled with a compiler enforcing our methods is unable to interfere with the control flow of our program through return addresses. Furthermore if we combine our solution with the SFI presented earlier we can have some strong security properties on the execution of dangerous modules with our main program. Alas there are also some disadvantages to our approach that we are going to present here:

- **Architecture dependant**, our solution depends a lot of the stack layout of the program. Indeed fixing the size of the frames requires us to modify the original stack layout. Therefore since the stack layout vary depending of the architecture and compiler you are using, the modifications that have to be done are also different. We can then easily comprehend that we would need a different implementations for every existing stack layout. Moreover since these layouts can be really different it might be very gruesome to implement our solution on certain of them. In the implementation we present after we focus solely on x86-32 architecture with the compiler CompCert.

- **Memory consumption**, since we are fixing the size of the frames instead of having dynamic sizes the memory usage of the stack is bigger. We have the issue of choosing an adequate size for the frames in our solution. The easiest one is to take the maximum frame size of the program as the constant size for all the frames. The downside is that we might have a memory usage explosion from our stack. We didn't encounter any issue about memory during the tests we did but the impacts may be visible on especially big programs. It might be interesting to study the cost of our approach on the growth of the stack.

Despite the cons presented we believe the benefits we gain from this method is worth it. We are going to present in the following section the implementation we made based on the ideas we introduced here. This implementation was made with the compiler CompCert for the x86-32 architecture. We are targeting programs written in C, which explains that all the examples we used were related to the C language.

4 Implementation

For the implementation of our idea we chose to work with the compiler CompCert. CompCert already had an implementation of SFI presented earlier. Thus if we could combine our approach with the SFI, any program compiled with CompCert would have strong security guarantees. Furthermore CompCert is written with Coq the proof assistant, we eventually hope that we will be able to prove these security properties. In this section we are going explain in details how we implemented the approach and the different choices we did during the process. Afterwards we will discuss these choices and evaluate the results and performance obtained.

4.1 Implementation

Our approach is separated in four phases: "Fixed stack frames size", "Stack alignment", "Detection of memory write statements" and "Evaluation of the implementation". We are going to detail the implementation of these phases in the following sections. These transformations are deeply linked to the stack layout, hence to have a better understanding we are going to start by introducing the CompCert stack structure.

4.1.1 CompCert stack

The layout of the stack is dependent of the architecture and the compiler/interpreter used. For the sake of comprehension of the future sections we describe here the stack layout of x86-32 in CompCert. The stack layout of CompCert x86-32 is pictured in Figure 7. First of all we can notice that the stack grows downwards, it means that the stack grows from the highest addresses to the low ones. As we can see the usual data are stored in this stack like local variables, parameters, register states and the return address.

Each frame is built when a function is called, the different steps related to the creation of a frame is called *function call routine*. CompCert function call routine is described in the Figure 8. Each phase of the function call routine of the Figure 8 is explained just here:

1. Write the return address
2. Allocate enough memory for the rest of the stack
3. Save registers states in the stack
4. Execute the function body (use the memory for local and stack data)
5. When calling another function, place its parameters at the end of the stack and repeat the process

When returning from a function, the return routine is pretty much the opposite:

1. Restore registers state
2. Deallocate the stack until the return addresses
3. Pop the return address memory and jump to the value stored in it

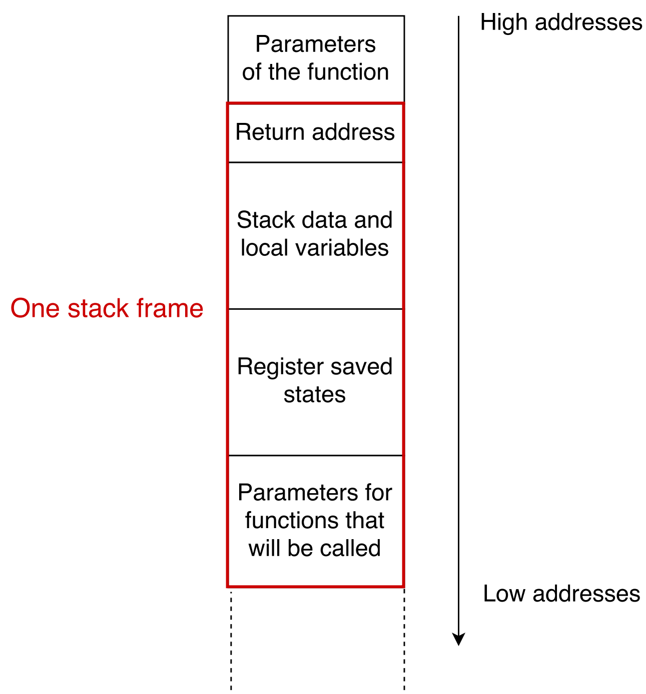

Figure 7: CompCert x86-32 stack layout

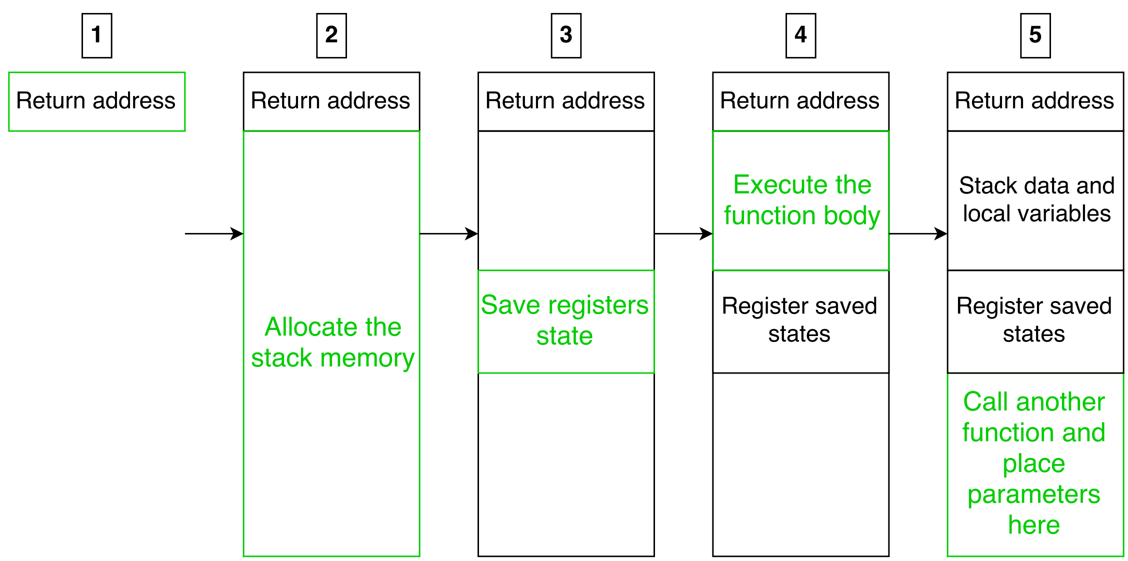

Figure 8: CompCert function call routine

4.1.2 Fixed stack frames size

During this phase we want to ensure these two properties:

- Return addresses locations are all separated by a constant offset bigger or equal to any frames of the stack

Fix the frames size Fortunately in the function call routine of CompCert the return addresses are always at the top of their frames. This particularity makes the task easier, indeed, since the location of the return address is fixed in the stack we can simply fix the size of the frames to have a constant offset between the return addresses. This special trait is not always true, for example in x86 architecture with the compiler *gcc* the location of the return addresses changes relatively to the frame depending of the parameters of the function.

To fix the size of stack we had to find the description of the stack in CompCert. Then we just had to put a constant in the attribute *size* of the stack and readjust the alignment of the different parts of the frames. We told CompCert to keep the return address of the stack as the first location in the frame and that all the extra space introduced by the fixed size will be taken for *stack data and locals*. The remaining parts keep the same alignment as before.

Choice of frames size We chose to limit the choice of the frames size to powers of two. Indeed, since addresses are written in binary having a power of two as constant offset will ease the runtime checks implementation. Indeed by using powers of two, all the return addresses will have the same least significant bits. For example if the first return address location is `0xffffff911` (in hexadecimal) and our constant frame size is 2^8 = `100` (in hexadecimal). Then all the following return addresses locations will be `0xfffff811` (the stack grow downwards), `0xfffff711`, `0xfffff611`... And their least significant bits are actually always equal to `11`. This particularity will help during the phase of runtime checks that we will explain in section 4.1.5.

To prevent having compiled programs with too small stack frame, we added in CompCert a check. This test verifies that the chosen size is bigger than any dynamically calculated one. If it's not, the compilation fails. This way the chosen size corresponds to **the smallest power of two which is bigger or equal to any dynamically calculated frames size of the program**.

We can see in Figure 9 the effect of our implementation. The left stack is the usual layout of CompCert stacks with the return addresses located at the top of the frames. We call F_{size} the size of the biggest frame of the whole program.

For our transformed stack we have to chose a fixed size for the frames and it needs to be a power of two, bigger or equal to F_{size}. In Figure 9 we chose 2^8 to continue the examples we gave before. We can see that the stack on the right has fixed size frames equal to 2^8 and the return addresses are all separated by the same offset dues to CompCert stack layout. The implementation effectively fulfills the property of having constant offset between return addresses. Furthermore we can see that the location of the return addresses are `0xfffff911`, `0xfffff811`, `0xfffff711`... Hence all the return addresses have the particularity of always having the same least significant bits (`11`). This particularity will be used later for the implementation of the protection mechanism.

4.1.3 Stack alignment

The implementation in this section has to do a transformation that makes the following property true:

- The first return address location of the program has its least significant bits equal to 0

We had multiple choices for the implementation of this property. One of them was to modify the main function of the protected program in order to align the return addresses locations correctly. Another idea was to modify the prelude of a program, the prelude is a piece of code created by

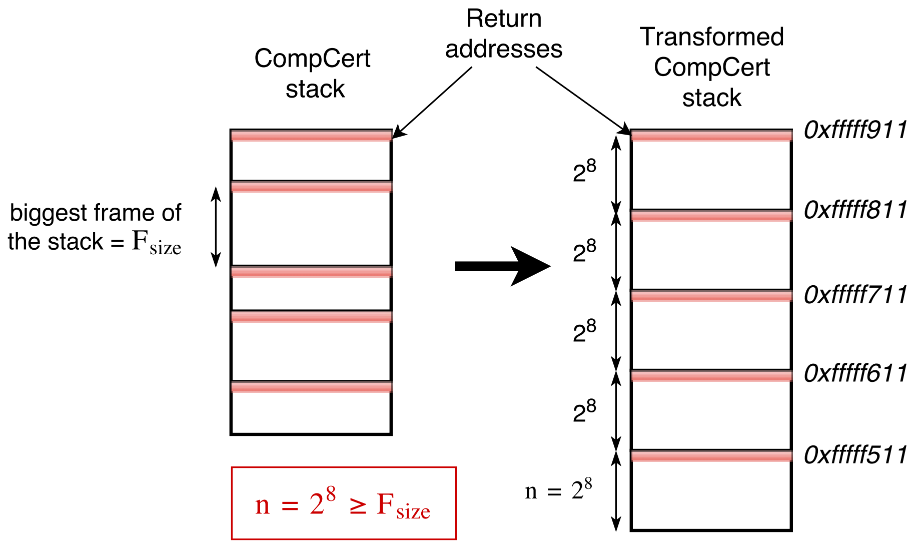

Figure 9: Fixing the size of CompCert stack frames

the compilers which is executed before the program. It is necessary for any program to have this prelude to work correctly.

Eventually we chose to introduce before the *main* function of the program an artificial *main*. Its role is to align the stack in order to make all the incoming return addresses locations to match our formula $0 \ mod \ n$. This approach has one definite advantage over the others solutions listed before. Creating a whole new function prevents us to modify recklessly core parts of the program like the original *main* or the prelude.

Since we have to modify the stack structure we did our transformation at the assembly level(ASM). Indeed the stack pointer ESP which is responsible for the stack growth is only available in ASM. ASM is the lowest level before binary code, though it is difficult to modify ASM correctly since you have to manipulate low level objects. By creating a separate artificial *main* function we avoid taking the risk of bugging the prelude or the program's *main* function.

Figure 10 represents the stack alignment transformation. The left stack is CompCert stack with fixed frames size equal to $n = 2^8$ like we had in the previous section. From this stack we show the consequences of our operation. We insert before the *main* function of the program an artificial *main*. Thus the frame of this artificial *main* is the first frame of the whole stack. The artificial *main* objective is to align the stack in order to have the next return address equal to **0 mod n**.
We can see on the left stack the effect of the transformation. The return address of *main* was previously at the address `0xfffff911` and is now at `0xfffff700`. Since the frames size remained constant we now have all the following return addresses locations matching **0 mod n**. This was the objective of the whole stack transformation which is now completed. The downside of this implementation can be seen clearly on the Figure 10. Indeed all the return addresses locations are equal to **0 mod n** except the return address of the artificial *main* we introduced. Since our approach aims to protect the locations matching **0 mod n**, this return address is vulnerable. Nevertheless, to reach this location an attacker would need to either know the exact location either overwrite the

whole stack.

- To pinpoint the location of vulnerable return address is really difficult, it would requires a lot of tries and fails or luck. Furthermore, nowadays most of the systems have a security feature called ASLR which inserts randomness in the memory addresses like the stack location. It means that every time a program is executed the location of this return address will be different which complicates the attack. Another possibility is to add in our runtime checks an extra condition to protect this specific return address.
- The other way to reach this unprotected return address is prevented by our implementation. The attacker would need to overwrite other frames return address to reach the vulnerable one. In this case our approach will make the program crash before it can arrive at the artificial *main* frame.

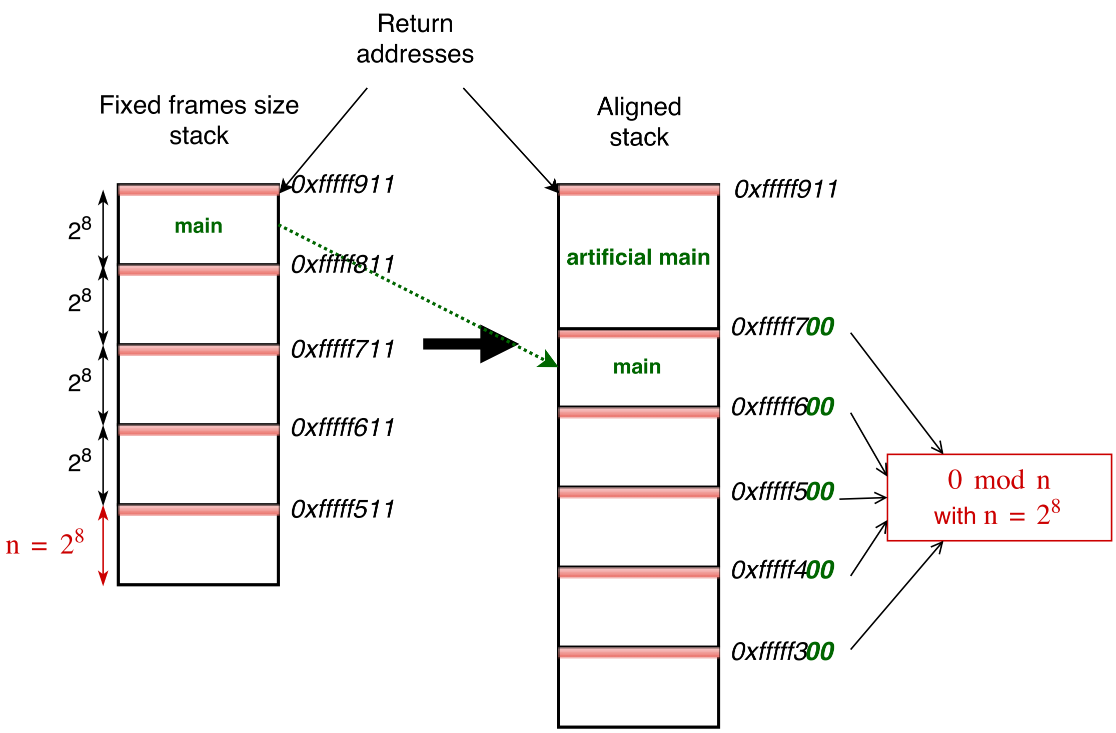

Figure 10: Aligning CompCert stack frames

Alignment algorithm We present in Figure 11, the algorithm used to calculate the right size in order to have the next return addresses aligned. The algorithm is written in pseudo assembly code.

The easiest way to understand it is to go through it with an example. On the previous examples Figure 9 and Figure 10 our stack started at the value `0xfffff911`. Hence for the continuity we will keep this value. In Figure 10 we want to have our next frame starting at the address `0xfffff700`. We go through the different lines of Figure 11 to explain the algorithm:

1. we copy the current *next_frame* location into a register called *reg*, for the example we take randomly *next_frame* = `0xffff840`.
   The operation is then *reg* = *next_frame* = `0xffff840`

2. *reg* = *reg* & $(n-1)$ = `0xffff840` & `0x000000ff` = `0x00000040`
   In our examples we have $n - 1 = 2^8 - 1 =$ `0x000000ff`

3. *reg* = *reg* $+(n-4)$ = `0x0000040` + `0x000000fc` = `0x00000140`

4. *next_frame* = *next_frame* − *reg* = `0xffff840` − `0x00000140` = `0xffff700`

5. We start the function call routine, here we save registers state in the stack

6. We store the parameters of the original *main* in the stack

7. We call the original *main* function, its frame will start at the location stored in *next_frame* = `0xffff700`

A small remark on the example is that our algorithm only works if the last bit of the first value of *next_frame* = `0xffff840` is `0`. However this particularity is already present in all compilers since it improves the speed of execution and then our algorithm works with all standard addresses.

```
move    reg          next_frame
and     reg          n-1
add     reg          n-4
sub     next_frame   reg
store   regs_state
store   parameters
call    main
```

Figure 11: Alignment algorithm

4.1.4 Detection of memory write statements

Clight implementation We chose to implement the detection of dangerous statements and also the runtime checks of those statements at the Clight level. This choice is explained by the fact that Clight is a high-level language in the compilation steps of CompCert (it is the closest to C so the syntax is really similar). Indeed, doing our transformations at a high-level is much easier since all the complicated compilation operations are done later in the process. For example by using Clight we don't need to bother with low-level objects like registers which if misused can modify the program unexpectedly. Furthermore Clight is a compilation step placed before any optimization of CompCert. This mean that our implementation can be optimized automatically by CompCert which can improve our performances.

Clight semantic We have to make sure that we cover all possibly harmful statements with our runtime protection. Since we are working with the compiler CompCert we are going to take advantage of it. CompCert has multiple compilation steps which have all been proven from C to assembly language. To make these proofs a semantic was defined for each language of the compilation process. The semantics relate to the memory model briefly described in section 2.2. To detect all dangerous statements we looked at the semantic of Clight and found all statements that in the memory model could write freely in the memory.

```
Inductive statement : Type :=
  | Sskip : statement
    (**r do nothing *)
  | Sassign : expr -> expr -> statement
    (**r assignment [lvalue = rvalue] *)
  | Sset : ident -> expr -> statement
    (**r assignment [tempvar = rvalue] *)
  | Scall: option ident -> expr -> list expr -> statement
    (**r function call *)
  | Sbuiltin: option ident -> external_function -> typelist -> list
     expr -> statement
    (**r builtin invocation *)
  | Ssequence : statement -> statement -> statement
    (**r sequence *)
  | Sifthenelse : expr  -> statement -> statement -> statement
    (**r conditional *)
  | Sloop: statement -> statement -> statement
    (**r infinite loop *)
  | Sbreak : statement
    (**r [break] statement *)
  | Scontinue : statement
    (**r [continue] statement *)
  | Sreturn : option expr -> statement
    (**r [return] statement *)
  | Sswitch : expr -> labeled_statements -> statement
    (**r [switch] statement *)
  | Slabel : label -> statement -> statement
  | Sgoto : label -> statement
```

Figure 12: Clight statements

In Figure 12, we have exposed all the Clight statements. Among them we are going to focus on the ones that change the state of the memory. When looking at the semantic given to these statements, only four of them can change the state of the memory: *Sassign*, *Sbuiltin*, *Sreturn* and *Sskip*.

- ***Sassign***, is used to assign value to variables, it could be considered as an equivalent of "=" in C. These statements will be targeted by our approach.
- ***Sbuiltin***, is used to call builtin functions, which are functions created by CompCert that will

be expanded later in the compilation. These statements call functions we trust, that's why we won't consider them as dangerous. We could also look at the builtin functions and modify their code to make them safe.

- ***Sreturn***, these statements invoke the function call routine. They are also trusted statements, we won't need to add runtime checks on them.
- ***Sskip***, in certain cases these statements are used to pop the stack. This does not endanger return addresses, we won't concern ourselves with them.

Among all the statements, our security checks will only apply to the *Sassign* statements. Furthermore we can limit ourselves to *Sassign* statements whose left expression can write directly in the memory ("Sassign left_expr right_expr" $\leftrightarrow$ "left_expr = right_expr"). The left expressions targeted are then mostly pointers dereference. To be sure that we have all the dangerous instructions, we reiterate the same approach and we take a look at the semantic of the left expressions in Clight. After checking the semantic of the left expressions, only two types of expression are able to reference a location in the memory.

- ***Ederef***, as we predicted these expressions dereference pointers and will be targeted by our approach.
- ***Efield***, they refer to fields of structure and can also point to locations in the memory. These expressions will also be secured with runtime checks.

We finally have defined the profile of the dangerous statements that have to be targeted by our approach. To sum it up, the targeted statements are all the *Sassign* whose left expression is either *Ederef* or *Efield*.

Now that we can detect the dangerous statements we will now add the runtime checks in the Clight code which will terminate our implementation.

4.1.5 Securing memory write statements

We want to add a protection mechanism which prevents any dangerous statement from writing on a return address location. These return addresses locations have two special traits:

1. they only exist in the memory area of the stack
2. thanks to our previous modifications, their locations match the formula **0 mod n**

If a statement try to write on an address with these two properties then it's an illegal execution and our mechanism will trigger an error behaviour like crashing. The Figure 5 explained earlier can be used as a reminder of the principle of these runtime checks.

Distinguish stack and heap addresses A program can only use the stack and the heap to store data during runtime. Since return addresses are only present in the stack we need a way to differentiate stack and heap addresses. In x86 architecture, the stack usually grows downwards in direction of the heap. Therefore the addresses from the stack occupy the high addresses and the heap the low ones. The idea is to divide the memory space for the program's data into two distinct

part. The high addresses for the stack and the low for the heap.
We defined the clear separation at the address `0xff000000`. Every address bigger than `0xff000000` is considered as part of the stack. Reciprocally every address smaller is part of the heap. To ensure that either the stack or the heap grows too much and exceeds their designed area we want to put a guard area in the memory. The idea is to define a specific area in the memory, for example [`0xee000000 - 0xff000000`], where it's forbidden to write. If an instruction writes illegally in the guard area the program will detect it and will crash.
The principle is represented in Figure 13. Indeed we can see that the stack and the heap are clearly separated by the guard area located between [`0xee000000 - 0xff000000`]. Thus we are sure that every address above `0xff000000` are part of the stack.

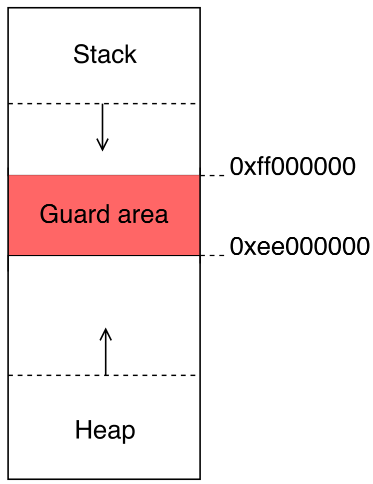

Figure 13: Guard area for the memory

To detect the write in the guard one possible way is to initialize the area with 0 for example. If we detect a bit in the guard area with the value 1 then we know that the guard has been corrupted and we make the program crash. Since our approach protects against ROP attacks, which takes effect when returning from a function, it is adequate to check the integrity of the stack at the end of each function before returning. To be honnest the guard area hasn't been implemented yet. Most of the programs don't have a stack or a heap which grows enough to exceed the limit of `0xff000000` so we were able to do satisfactory tests nevertheless. This guard area is necessary for our implementation to be complete and we hope that we will be able to do it during the remaining time of the internship.

Check the equality to 0 mod n The second step of the runtime check is to see if the targeted address location equals **0 mod n**. The algorithm used is represented in Figure 14 with pseudo Clight.

- At line 1 we reduce the targeted address to its $log_2(n)$ least significant bits (8 if $n = 2^8$). Since n is a power of two just comparing the least significant bits to 0 is equivalent to compare the whole address with $0 \ mod \ n$. For example we know that the return addresses locations of our previous example Figure 10 were `0xfffff700`, `0xfffff600`, `0xfffff500`... We can notice that we only need to check if the last eight bits (the last two digit in hexadecimal `00` in our

example) of an address is equal to 0. If it is we know for sure that the targeted location is a return address.
For a random authorized location `0xfffff7a2` and a return address location `0xfffff700` with $n = 2^8$, we would have to compare `0x000000a2`(authorized) and `0x00000000`(unauthorized) with 0.

- Line 2 we actually see that we don't compare the previous calculated value with 0 but 3. The reason is that return addresses are four bytes long. It means that the space taken by a return address located at `0xfffff700` would be [`0xfffff700` - `0xffff703`]. It is logical for us to protect the whole return address space, that's why we check if the targeted address least significant bits are smaller than 3.
- Line 3 is executed when there is an illegal behaviour. Currently our error behaviour is to make the program crash by trying to write over some protected memory located at the address 0 which triggers a *Segmentation fault*.
- Finally line 4 if we successfully pass the verification we are allowed to write on the dereferenced pointer *temp_var*.

```
1        temp_var = targeted_address & (n-1);
2        if (temp_var < 3) {
3                Error behaviour
4        }
5        *temp_var = value;
6        Continue execution...
```

Figure 14: Second test of the protection mechanism

Currently our protection mechanism use two *if . . . then . . . else* operation in a row to protect the return addresses. The first to check if the address is in the stack and the second to check the equality to **0 mod n**. However, this kind of control-flow operations are usually quite expensive for the processor. Furthermore we might have to inject our protection mechanism a considerable number of times for a program using a lot of pointer operations. Hence we present in the next section an alternative to the classic *if then else* called branchless statement.

Branchless check We wanted to limit the overhead introduced by our approach by trying another way to make the protection mechanism.

Branchless code allows one to create code with the same behaviour as a classic *if then else* but without creating any branch for the processor. In other words the processor won't need to execute different code depending of the condition, the code will be linear.
The best way to understand it is to have an example and we show the branchless version of implementation of the protection mechanism in Figure 15.

The branchless code presented reproduces the second test which checks if the least significant bits of an address is smaller than 3. Branchless uses a lot of bit arithmetic which may be unfamiliar so we are going to go through every line of Figure 15:

```
lsb = lsb - 3;
lsb = lsb >> 31;
lsb = ~lsb;
lsb = lsb & targeted_address;
*lsb = value;
Continue execution...
```

Figure 15: Branchless version of the second check

1. *lsb* (least significant bits) is a variable containing the least significant bits of the targeted address. For the example we will take two different values, one coming from a return address location and not for the other. We keep $n = 2^8$ for the chosen frames size.

   - *lsb* comes from a return address location, we take *targeted_address* = `0xfffff700` which gives *lsb* = `0x00000000`.
     We get *lsb* = *lsb* - 3 = `0x00000000` - `0x00000003`= `0xfffffffd` (signed representation)
   - *lsb* not from a return address location, *targeted_address* = `0xfffff7a2` which gives *lsb* = `0x000000a2`
     lsb = *lsb* - 3 = `0x000000a2` - `0x00000003` = `0x0000009f`

2. We make a right shift bit operation (`0100` → `0010`) thirty-one times. If the value is negative then the new bits introduced on the left are equal to 1 else they are equal to 0.

   - *lsb* = *lsb* >> 31 = `0xfffffffd` >> 31 = `0xffffffff` (because *lsb* was negative)
   - *lsb* = *lsb* >> 31 = `0x0000009f` >> 31 = `0x00000000`

3. We inverse the value of *lsb* (`0100` → `1011`)

   - *lsb* = ∼*lsb* = ∼`0xffffffff` = `0x00000000`
   - *lsb* = ∼*lsb* = ∼`0x00000000` = `0xffffffff`

4. Classic *and* operator (`0101` & `1100` = `0100`)

   - *lsb* = *lsb* & *targeted_address* = `0x00000000` & `0xfffff700` = `0x00000000`
   - *lsb* = *lsb* & *targeted_address* = `0xffffffff` & `0xfffff7a2` = `0xfffff7a2`

5. We write on the location we just calculated in *lsb*.

   - *targeted_address* was the location of a returned address, it was an illegal behaviour.
     We write to the memory location `0x00000000` which is a protected area. The operation makes the program crash which is what we wanted.
   - *targeted_address* was an authorized location. We have the equality *lsb* = *targeted_address*. It means that our operation didn't change the behaviour of the initial program, the transformation is satisfactory.

This way we can notice that we got the same result as with the *if then else* check but without creating any branch. In our case it's not possible to make two branchless instead of two *if then else* since even heap addresses would go through the second check which we don't want.
Unfortunately for us, the test that we made showed that the branchless version of our approach was not faster than the classic version it was even slower. Those results will be discussed in details in the next section where we evaluate our implementation.

4.2 Evaluation of the implementation

5 Conclusion