Master research Internship

Bibliographic report

Software Fault Isolation using the CompCert compiler

Domaine: Cryptography and Security

Author:
Alexandre DANG

Supervisor:
Frédéric Besson
Team CELTIQUE

CentraleSupélec

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 (see Figure **??**). 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 location 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()*. We can see that in a normal execution *evil_code()* should not be executed since it is never called. The 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 **??** We see in bold that the return address was modified to the address of *evil_code()* and that its code was executed.

We are going to explain how the attack works. In the function *foo()* the lines ?? and ?? print the stack. *%010x* formats the output in hexadecimal with `0x` at the beginning for addresses. The vulnerability resides in the function *strcpy* line ??. *strcpy* just copies character for character until it finds "0" in the source string. However our source string contains much more character than *buf* is supposed to have. Indeed *buf* is defined line ?? as an array of 1 character. In this case *strcpy* will just continue to write the source string over others variables location in the stack and possibly reach the return address. After few try and fails we found the correct input "" line ?? to successfully do the buffer overflow. In our case 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 **??**, 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 `??` in the example. This way, the next instruction that will be executed after *foo* finishes will be the function *evil_code*.

3.2 Description of the approach

We want to protect programs against ROP attack like the buffer overflow seen previously. We want to prevent any return address to being overwritten illegally. The only moment they should be written over is during a function call routine. We want to use a masking operation similar as in 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 piling up stack frames. These frames are constructed dynamically depending of the function, hence the locations of return addresses aren't easily guessed. 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 then have complete control over the *shadow stack* which allows us defend against ROP attacks.

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

The idea is to modify the stack layout in order to have a constant offset n between neighbours return addresses. This way we know that the neighbors 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

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

void evil_code() {
        printf("Argh, we got hacked!");
}

void foo(char* input){
        char buf[1];
        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");
        strcpy(buf, input);
        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");

}

int main (int argc, char* argv[]) {
        void (*a)(void) = evil_code;
        printf("Address of evil_code = %#010x\n", evil_code);
        printf("Address of evil_code = %p\n", a);
        a();
        //foo("aaaaaaaa");
        foo("aaaaaaaaa\x4b\x84\x04\x08");
        //0x804844b
        return 0;
}
```

Figure 1: Example of buffer overflow in C

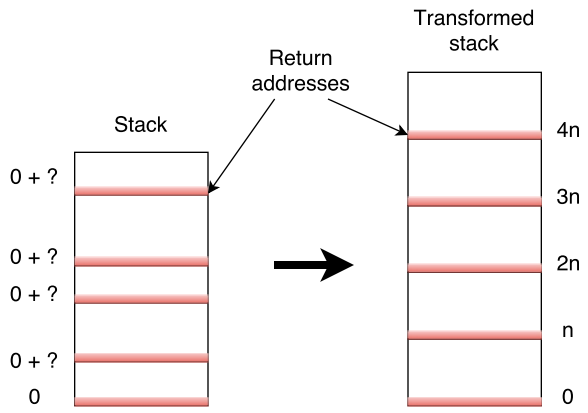

Figure 2: Stack modifications

```
1 int foo(int a) {
2         int* pointer = &a;
3         //In pointer we get the address of the parameter a which is in
              the stack
4
5         *(pointer+4) = 0xff000000;
6         //We write the value 0xff000000 in the stack
7         //if this location is a return address, the ROP attack succeeds
8 }
```

Figure 3: Example of dangerous C statement

return address, we call this location c. With the property explained previously we know that the following return addresses locations will be $c+n$, $c+2n$... We then 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.

Afterwards we have to define c. The simplest 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 second step is to detect every possibly harmful to return addresses (Figure 3). All the operations which can write in the memory can be considered dangerous. Hence we target all instruction related to variable assignment. If we want more precision, we can also target assignments to variables located on the stack.

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. If the address written does not match the location of a return address, then the operation is allowed

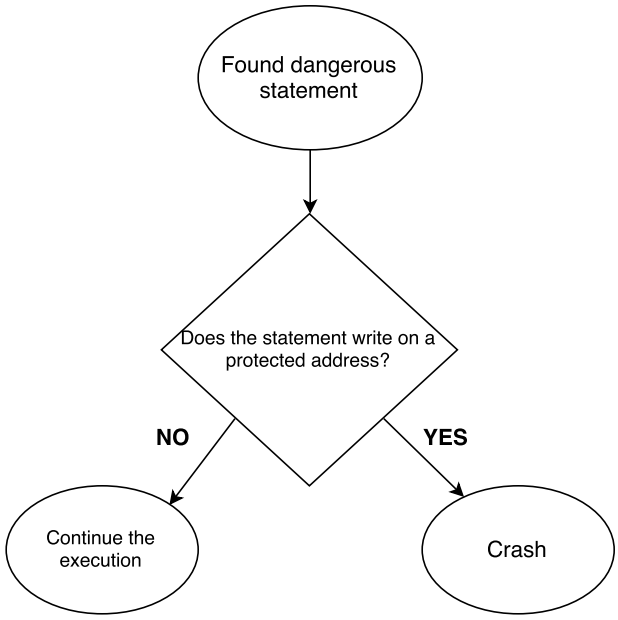

Figure 4: Runtime check algorithm

and the program continues to run casually. Whereas if there is an attempt to write on a return address we trigger an error behaviour like crashing (Figure 4).

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 equals to the size of the frames

- The transformation does not change the semantic of the module

2. **Stack alignment**
   - The first return address location of the stack have its least significant bits equals to 0
   - The transformation does not change the semantic of the module
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**
   - Statements that does not involve the protected addresses will keep their behaviour
   - 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 size of the frames .

We make it so the protection mechanism prevents any write on addresses located in the **stack memory area** and their least significant bits equal to **0 mod n**.

If all these properties are fulfilled we are confident that under certain conditions it's not possible for a hazardous module to modify the flow control through the return addresses. The necessary conditions for our approach to work will be discussed right after.

3.4 Analysis of the approach

3.4.1 Conditions

We believe that 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. An example of such operation in C is inline assembly which allows us to put some assembly code in C code (Figure 5). One simple solution to this flaw would be to forbid any usage of inline assembly.
- **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.

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 5: C inline assembly

Those conditions are necessary for our approach to be relevant. Yet we can't guarantee that these conditions are an exhaustive list of the hypothesis needed. They are the conditions we could think of but there might are some more.

3.4.2 Discussion

We have presented the principle of our approach in this chapter. Following we mentioned some necessary conditions for our solution to be relevant. 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. To put a cost on the impact of our solution on the stack size we would need to make numerous tests. Unfortunately during the span of the internship we were unable to do so but it's one of our objectives for the remaining time.

Despite the cons presented we believe the benefits we gain from this method is worth it. With regard to the negative impacts on stack memory consumption we personally didn't encounter any issue with the different tests we made our implementation go through. The impacts may be visible

on especially big programs which we did not test yet. 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 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 a better understanding of the future sections we describe here the stack layout of x86-32 in CompCert. First of all in x86 the stack grows downwards, it means that the stack grows from the highest addresses to the low ones. A lot of information are saved in the stack, we can find the parameters of the functions, local variables, saved state of registers and the return address of the function. In CompCert the stack layout is composed as in Figure 6.

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 7 and 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 routine is pretty much the opposite:

1. Restore registers state

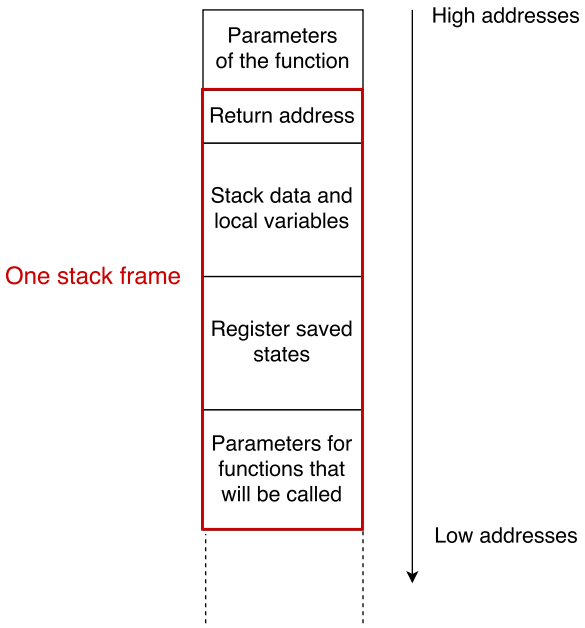

Figure 6: CompCert x86-32 stack layout

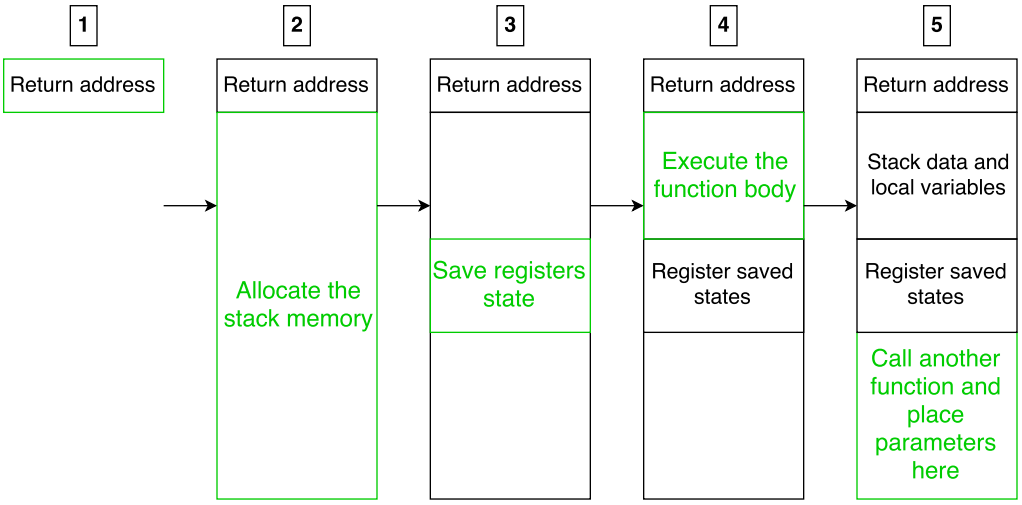

Figure 7: CompCert function call routine

2. Deallocate the stack until the return addresses

3. Pop the return address memory and jump to the value stored in it

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 equals to the size of the frames
- The transformation does not change the semantic of the module

Fortunately in the function call routine of CompCert the return addresses is always the first location in new frames. This peculiarity makes the task easier, indeed, since the location of the return address is fixed in the stack we can deduce that with just fixed size frame the offset between two return addresses locations will be constant. This peculiarity 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 in 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 part of a frame. 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 our implementation for the runtime check. By doing so, all the return addresses will keep the same least significant bits. For example if the first return address location is `0xffffff000` and our constant frame size is $n = 2^8 =$ `100` (in hexadecimal). Then all the following return addresses locations will be `0xfffff100`, `0xfffff200`, `0xfffff300`... And their least significant bits are always equal to `00`. This peculiarity will help during the phase of addresses checking 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 the chosen size is bigger than the 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 than any dynamically calculated frame size of the program**.

4.1.3 Stack alignment

4.1.4 Detection of memory write statements

4.1.5 Securing memory write statements

4.2 Evaluation of the implementation