# Software Fault Isolation using the CompCert compiler

Alexandre Dang

Team Celtique

June 10, 2016

# Flash vulnerable plugin

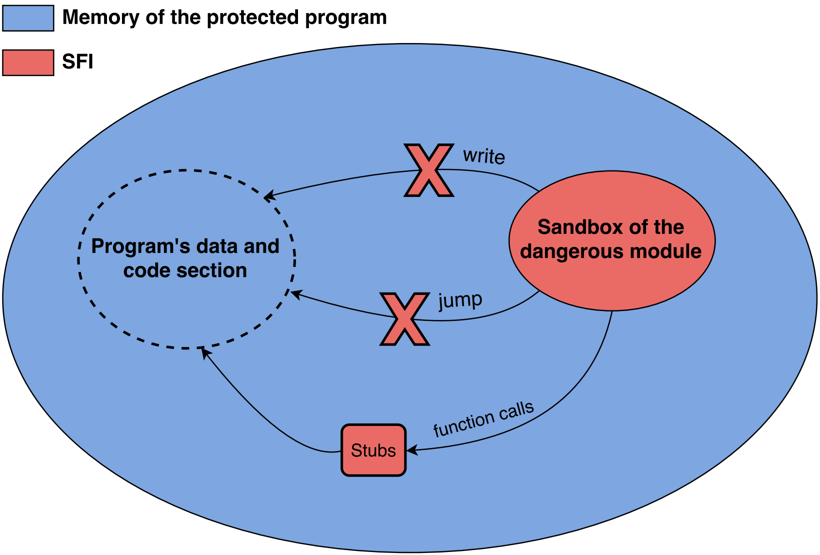
Do you know this logo?

Flash is famous for its multiple vulnerabilities
→ consequences on Flash
→ but ALSO endangers your browser

# Goals of Software Fault Isolation (SFI)

- SFI aims to allow a protected program to execute dangerous modules in its own memory space without dangers.
- SFI confines the execution of the dangerous modules in a reserved area called sandbox
- `jump` and `write` instructions are protected by runtime checks
- function calls to the protected programs are controlled by SFI

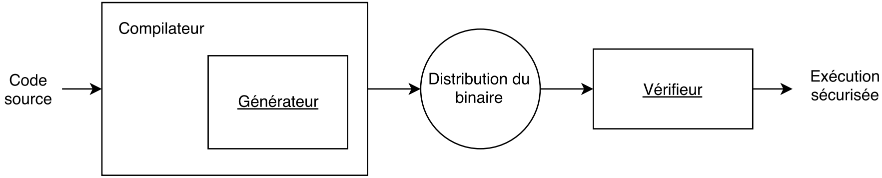
# Goals of SFI


Memory of the protected program
SFI
write
jump
Program's data and code section
Sandbox of the dangerous module
function calls
Stubs

# Overview of SFI

SFI chain is composed of two elements:

- the **generator** transforms the assembly code of the dangerous modules in order to confine the modules in their sandbox
- the **verifier** checks that the SFI transformations are present and valid before loading the code in memory



# Problematics of SFI

We want to prevent attackers from using vulnerable modules to compromise our system

- SFI gives us a way to face such issue
- However SFI is currently lacking against Returned Oriented Programing attacks
- ROP attacks focus function return addresses to execute malicious code they injected

# ROP attack example (1/3)

```
void reset_password() {
  ... reset password ...
}

void foo(char* input){
  char buf[1];
  ... code ...
  strcpy(buf, input);    //Vulnerability
  ... code ...
}
```

# Modern ROP attacks

- ROP attacks are a common kind of attack in the industry
- Modern ROP attacks are much more complicated
- *Return-to-libc* attacks uses code from the *glibc* library to construct malicious code and uses return addresses to execute it

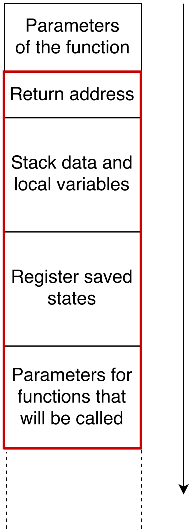
# Goals of our approach

We want to have a way to protect return addresses at runtime.

- Modifications of the memory layout in order to have an easy way to know return addresses location
- Code transformations which add runtime checks on the dangerous instruction in order to forbid any illegal `write` on the return addresses locations

# Stack structure

- Programs memory is separated into multiple area like the heap, the stack or the code section
- Return addresses are solely located in the stack
- The stack is composed of piled up frames each related to a function being executed
- Frames store data of their respective function



# Transformations of the stack layout

All the return addresses locations $a$ verify the equality $a \ mod \ n = 0$, with $n$ a constant offset between the return addresses locations.

1. Set a constant offset $n$ between all the return addresses
2. Align the stack

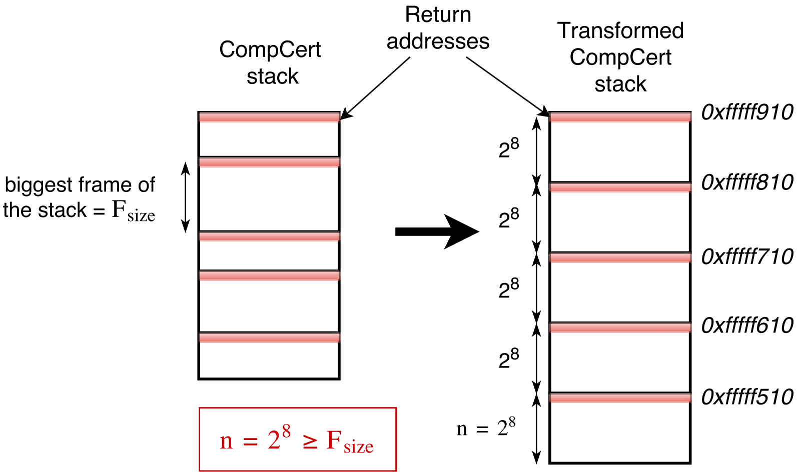
# Constant offset $n$ between return addresses (1/2)

Constant offset $n$ between return addresses locations

- Fix frames size to $n$
- Pick $n$ as the biggest frame size of the previous stack
- Pick $n$ as a power of two

With this we have $a \ mod \ n = c$, with $c$ the location of the first return address in the stack

# Constant offset $n$ between return addresses (2/2)

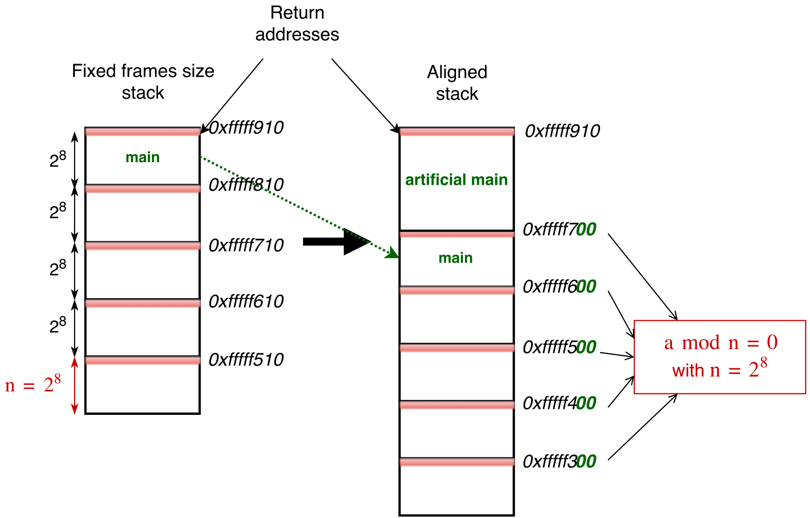


# Stack alignment (1/2)

We currently have the equality $a \ mod \ n = c$ but we want $a \ mod \ n = 0$ with $a$ any return address locations and $c$ the first return address location in the stack.

- introduce an artificial function at the beginning of the program
- the function align the stack as we wanted
- the function then calls the *main* of the program

# Stack alignment (2/2)



# Injection of runtime checks

1. Check if the address is part of the stack
2. Check if the address verifies $a \; mod \; n = 0$

```
if (targeted_address > 0xff000000) {
  temp_var = targeted_address & (n-1);
  if (temp_var < 3) {
    Error behaviour
  }
}
*targeted_address = value;
Continue execution...
```

# Branchless runtime checks

In certain cases branchless code shows much better performance

```
if (targeted_address > 0xff000000) {
    temp_var = targeted_address & (n-1);
    temp_var = temp_var - 3;
    temp_var = temp_var >> 31;
    temp_var = ~temp_var;
    targeted_address = temp_var & targeted_address;
}
*targeted_address = value;
Continue execution...
```

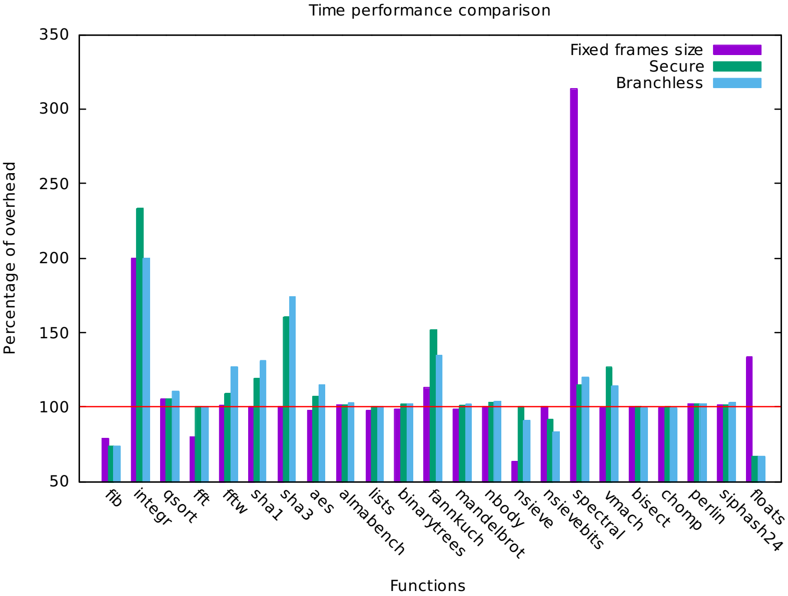
# Conditions of our approach

- No modifications of the stack (inline assembly)

```
int foo(int a) {
  asm(``\$sub 50, \%esp'');
   printf("Hello world!");
}
```

- Need to recompile extern libraries with the same frames size

# Evaluation of performance (1/2)



# Conclusion

- Test our implementation against more complicated ROP attacks
- Reduce the number of runtime checks with static analysis
- Improve the performance of the runtime checks with a super-optimizer
- See the impact of our approach on memory consumption