

# Software Fault Isolation using the CompCert compiler

*Author:* Alexandre Dang

*Supervisor:* Frédéric Besson

Team Celtique

June 16, 2016

# Flash vulnerable plugin

Do you know this logo?

Flash is famous for its multiple vulnerabilities

- consequences on Flash
- but ALSO endangers your browser



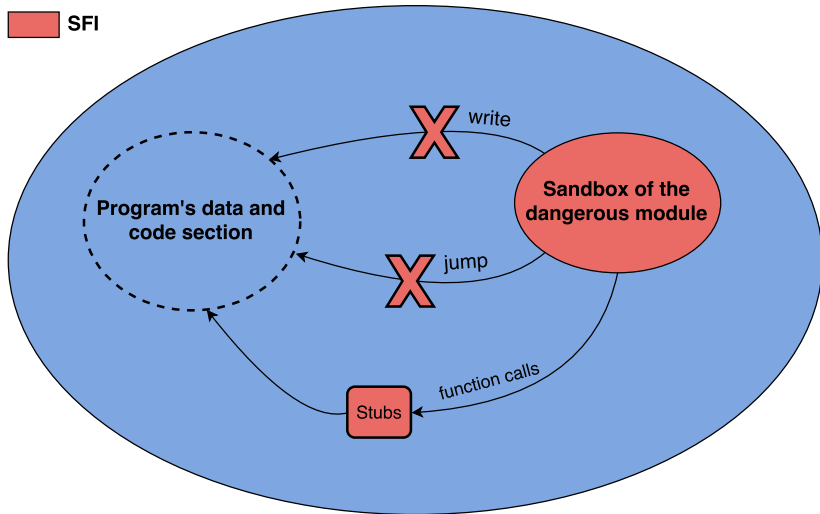
# Goals of Software Fault Isolation (SFI) [4]

- ▶ 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 memory space called sandbox

# Goals of SFI

 Memory of the protected program

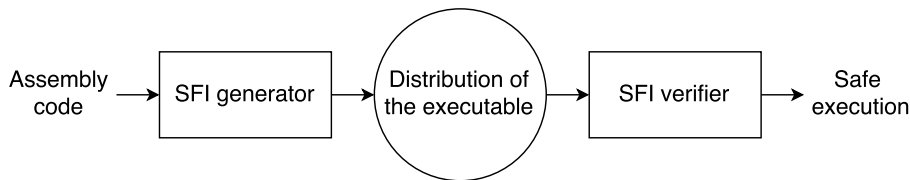
 SFI



# 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

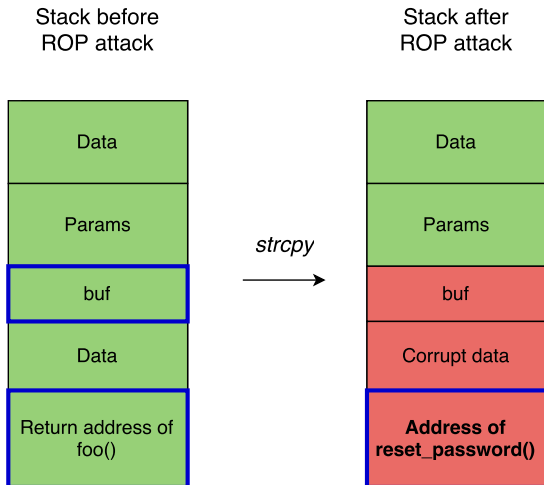
We want to prevent attackers from using vulnerable modules to compromise our system

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

## ROP attack example (1/2)

```
1 void reset_password() {  
2     ... reset password ...           // Sensitive code  
3 }  
4  
5 void foo(char* input){  
6     char buf[1];  
7     ... code ...  
8     strcpy(buf, input);              // Vulnerability  
9     ... code ...  
10 }
```

## ROP attack example (2/2)





# Modern ROP attacks

- ▶ ROP attacks are a common kind of attack in the industry
- ▶ Modern ROP attacks are much more complicated [1]

*“Skype URI handling  
routine contains a buffer overflow”, 2005<sup>1</sup>*

*“Apple Mail buffer  
overflow vulnerability”, 2006*

*“glibc vulnerable to stack buffer  
overflow in DNS resolver”, 2016*

---

<sup>1</sup>From CERT vulnerability database

# Contributions

- ▶ An approach protecting return addresses against ROP attacks
- ▶ An implementation of our approach with the compiler CompCert for the x86-32 architecture

# Problematic

We want to add additional checks at runtime to protect the return addresses.

How do we know the return addresses locations in the memory?

# Problematic

We want to add additional checks at runtime to protect the return addresses.

How do we know the return addresses locations in the memory?

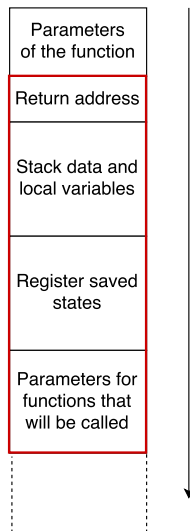
⇒ Modify the memory structure to have an easy way to distinguish return addresses locations

# Presentation of our approach

1. Presentation of the stack
2. Transformation of the stack structure
3. Insertion of runtime checks in the protected code
4. Evaluation of the approach

# 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

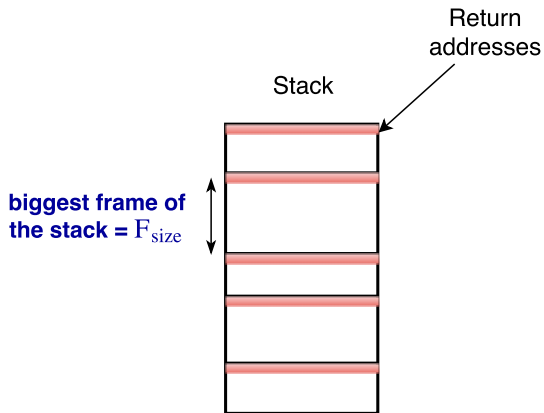


# Stack transformation objective

- ▶ We want a stack structure with a property on the return addresses locations
- ▶ Every return addresses location  $a$  verifies the equality  $a \bmod n = 0$
- ▶ The transformation is composed of two steps:
  1. Constant frames size
  2. Stack alignment

# Stack transformation (1/6)

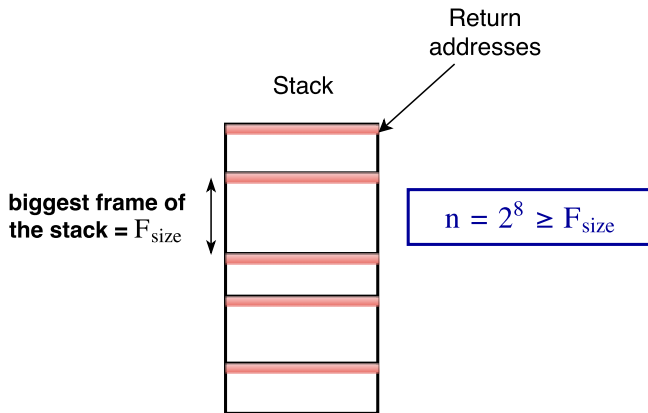
Find the biggest frames size





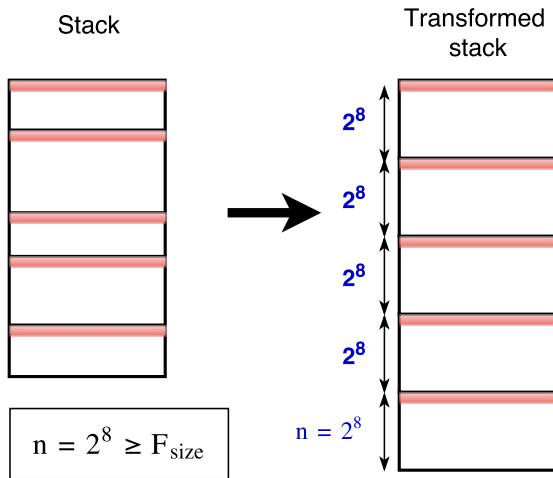
# Stack transformation (2/6)

Calculate the new frames size



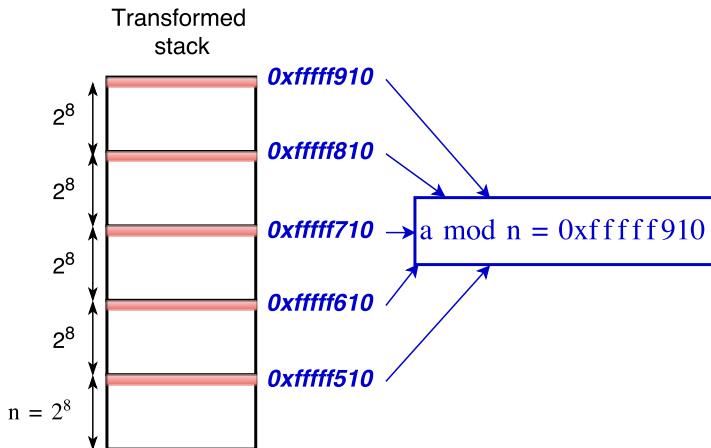
# Stack transformation (3/6)

Fix the size of the frames



# Stack transformation (4/6)

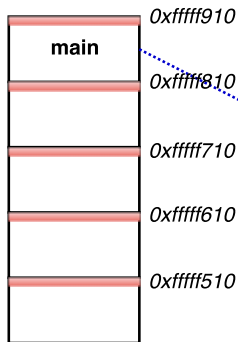
## Return addresses locations



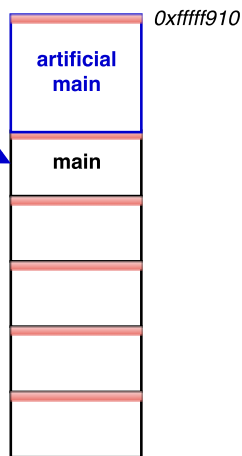
# Stack transformation (5/6)

## Insertion of a new artificial main

Fixed frames size  
stack

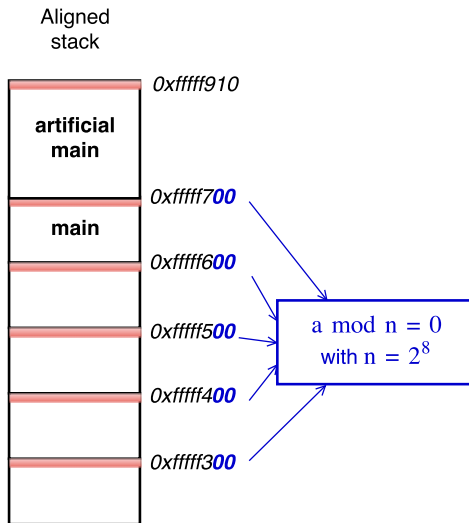


Aligned  
stack



# Stack transformation (6/6)

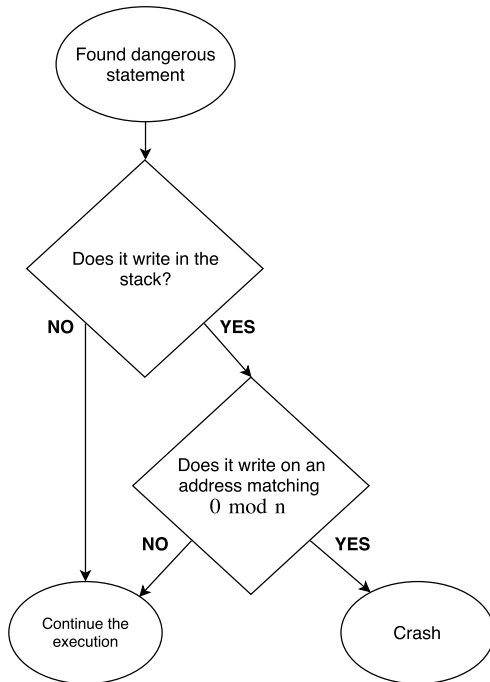
## Return addresses locations



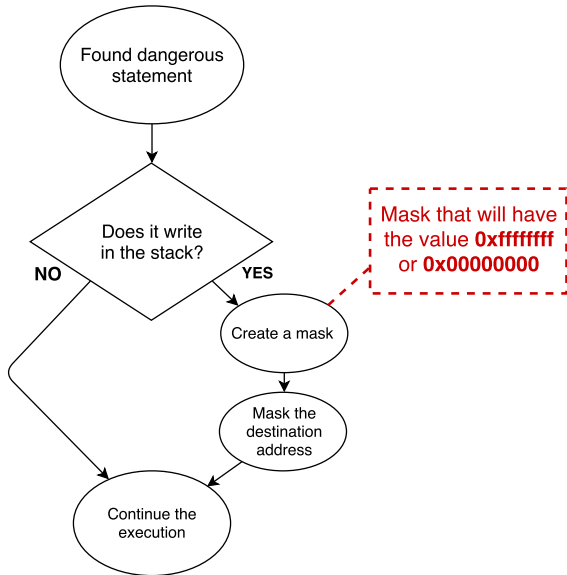
# Code transformation

- ▶ We have now an easy way to differentiate return addresses locations with  $a \bmod n = 0$
- ▶ We need to insert additional runtime check to protect these locations from being overwritten illegally
- ▶ Thus we transform the code adequately during the compilation phase

# Runtime check



# Branchless check

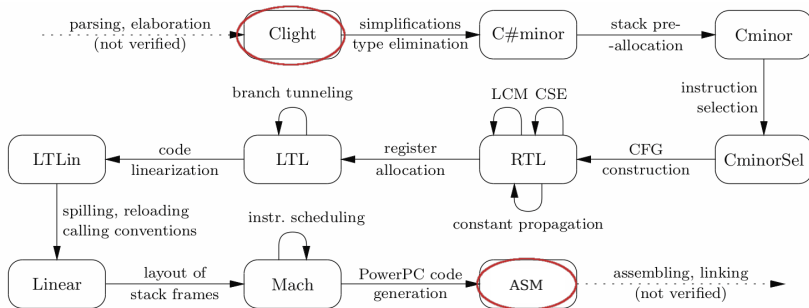




# Implementation environment

## CompCert the certified compiler [2]

- ▶ CompCert has been proven with the proof assistant Coq
- ▶ CompCert has performance similar to gcc -O1



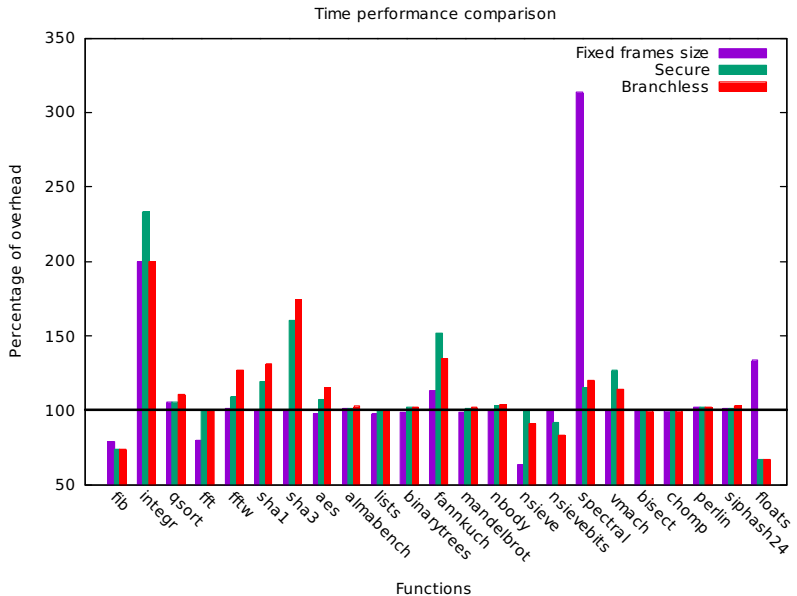
# Necessary requirements

- ▶ No modifications of the stack (inline assembly)

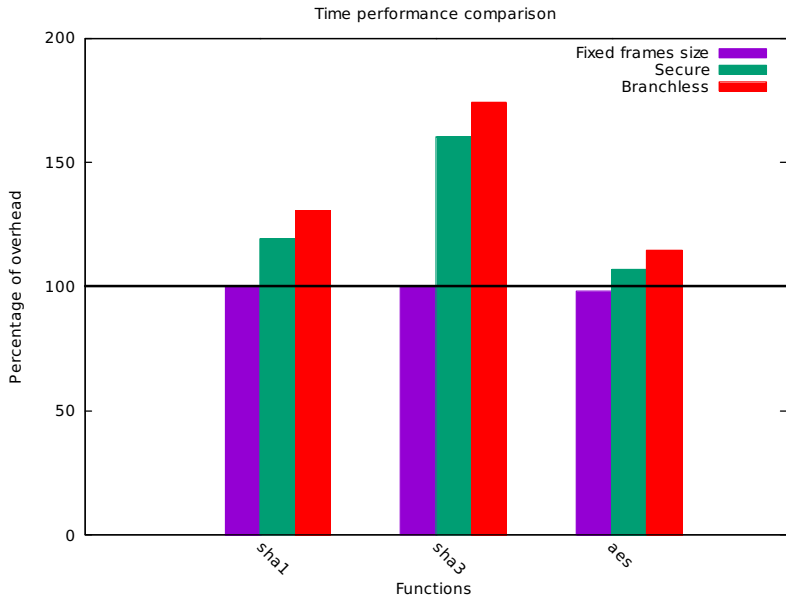
```
1  int foo(int a) {  
2  asm( ' '\ $sub 50, \%esp ' ' );  
3  printf( "Hello world!" );  
4  }
```

- ▶ No function pointers to protect our runtime checks
- ▶ Need to recompile external libraries with the same frames size

# Evaluation of performance



# Evaluation of performance



# Conclusion

## Prospectives

- ▶ Test our implementation against more complicated ROP attacks
- ▶ Reduce the number of runtime checks with static analysis [5]
- ▶ Improve the performance of the runtime checks with a super-optimizer
- ▶ See the impact of our approach on memory consumption

# References

- [1] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage.  
When good instructions go bad: Generalizing return-oriented programming to RISC.  
In Paul Syverson and Somesh Jha, editors, *Proceedings of CCS 2008*, pages 27–38. ACM Press, October 2008.
- [2] Xavier Leroy.  
Formal verification of a realistic compiler.  
*Commun. ACM*, 52(7):107–115, 2009.
- [3] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen.  
Adapting software fault isolation to contemporary cpu architectures.  
In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security’10, pages 1–1. USENIX Association, 2010.
- [4] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham.  
Efficient software-based fault isolation.  
*SIGOPS Oper. Syst. Rev.*, 27(5):203–216, 1993.
- [5] Bin Zeng, Gang Tan, and Greg Morrisett.  
Combining control-flow integrity and static analysis for efficient and validated data sandboxing.  
In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS ’11, pages 29–40. ACM, 2011.