Software Fault Isolation using the CompCert compiler

Author: Alexandre Dang *Supervisor*: Frédéric Besson

Team Celtique

June 17, 2016

Flash vulnerable plugin

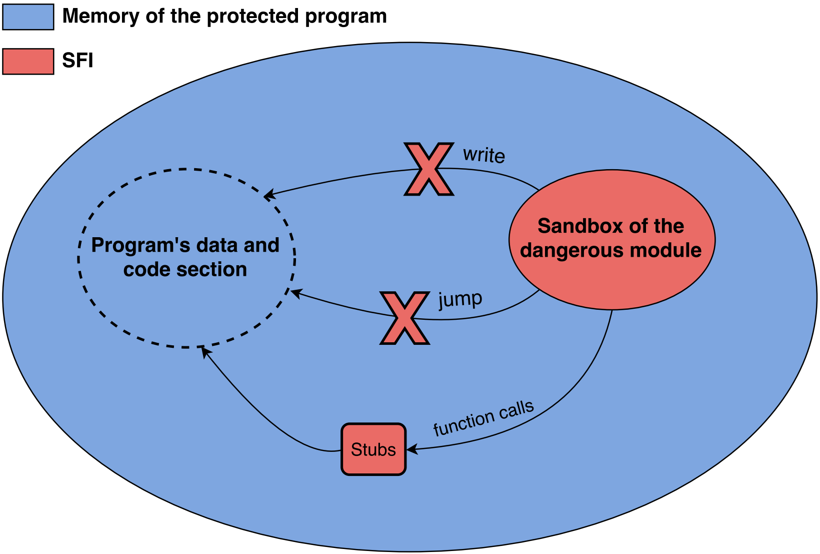
Do you know this logo?

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

Software Fault Isolation (SFI) [Wahbe et al, 1993]

- 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

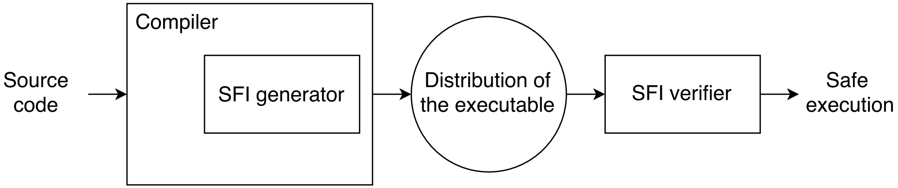


Memory of the protected program
SFI
write
jump
Program's data and code section
Sandbox of the dangerous module
function calls
Stubs

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



Limitations of SFI

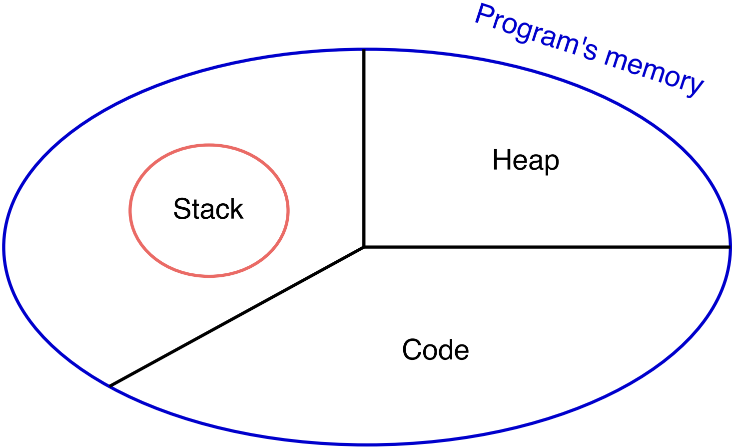
Strength
SFI prevent attackers from using vulnerable modules to compromise our system

Weakness
SFI has difficulties to protect return addresses from being targeted
→ Returned Oriented Programing attacks (ROP)

Contributions

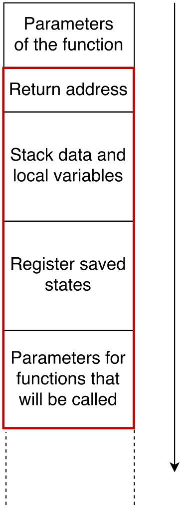
- An approach inspired from SFI protecting return addresses against ROP attacks
- An implementation of our approach with the compiler CompCert for the x86-32 architecture

Memory structure



Stack structure

- Return addresses are solely located in the stack
- The stack piles up frames
- Each frames matches a function

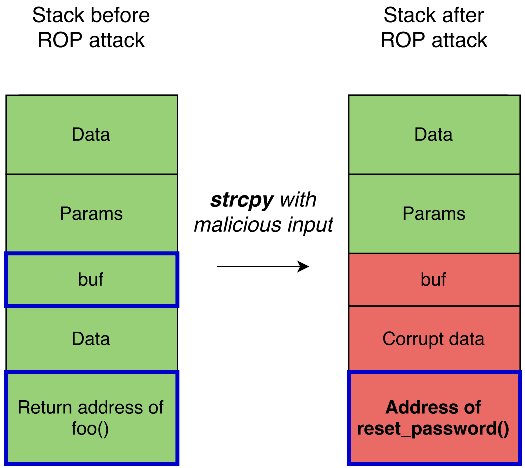


ROP attack example (1/2)

```
void reset_password() {
  ... reset password ...        //Sensitive code
}

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

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 [Buchanan et al,2008]

"Skype URI handling routine contains a buffer overflow", 2005[1]

"Apple Mail buffer overflow vulnerability", 2006

"glibc vulnerable to stack buffer overflow in DNS resolver", 2016

[1] From CERT vulnerability database

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?

$\Rightarrow$ Modify the memory structure to have an easy way to distinguish return addresses locations

Presentation of our approach

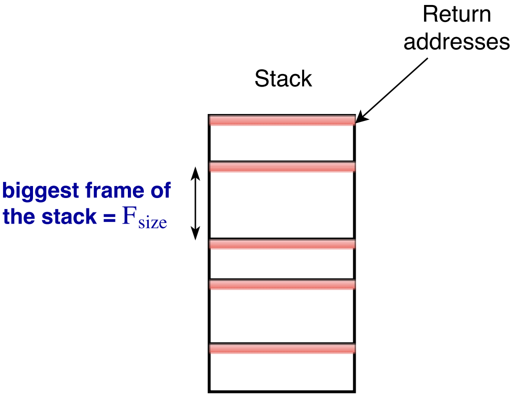
1. Presentation of the stack
2. Transformation of the stack structure
3. Insertion of runtime checks in the protected code
4. Evaluation of the approach

Stack transformation objective

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

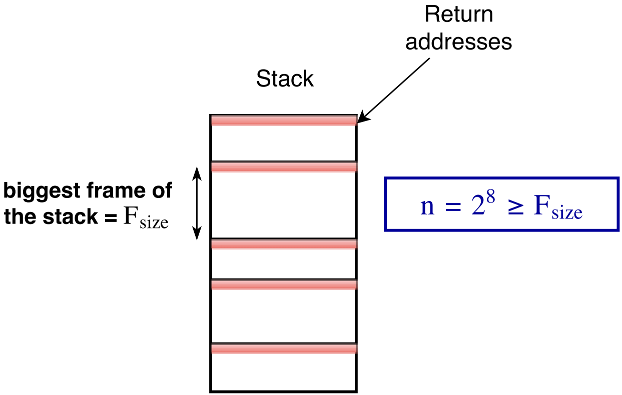
Stack transformation (1/6)

Find the biggest frames size



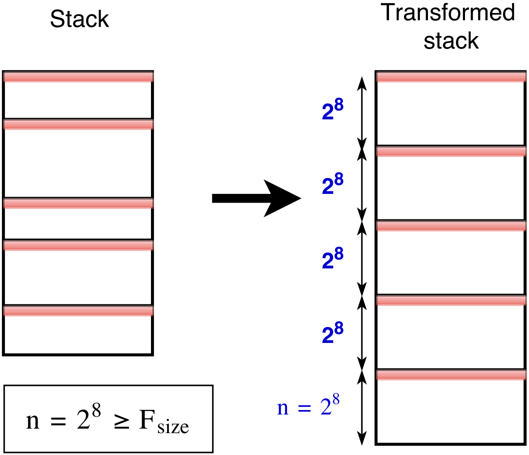
Stack transformation (2/6)

Calculate the new frames size



Stack transformation (3/6)

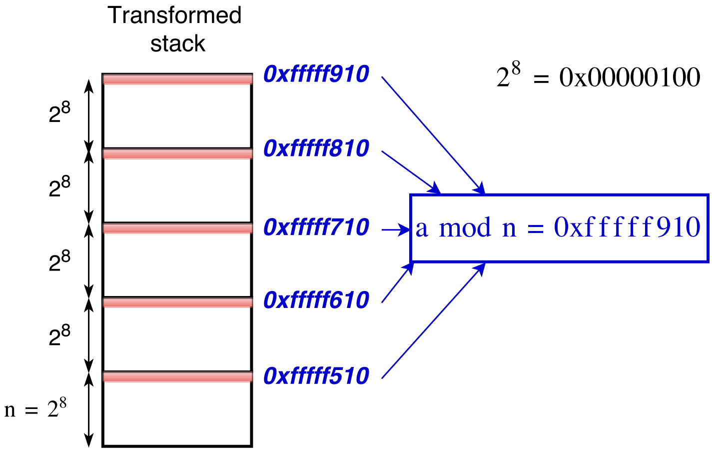
Fix the size of the frames



$$n = 2^8 \geq F_{size}$$

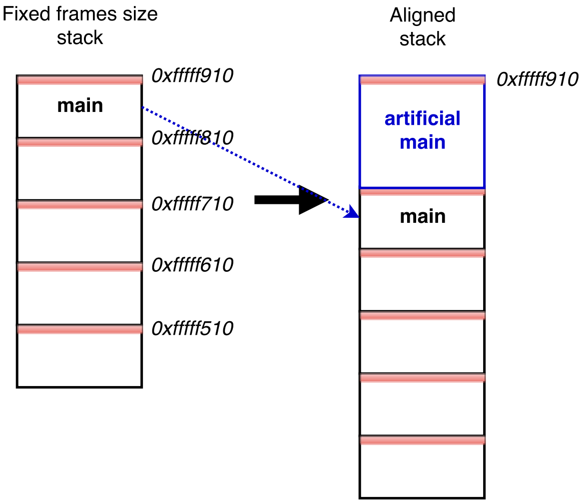
Stack transformation (4/6)

Return addresses locations



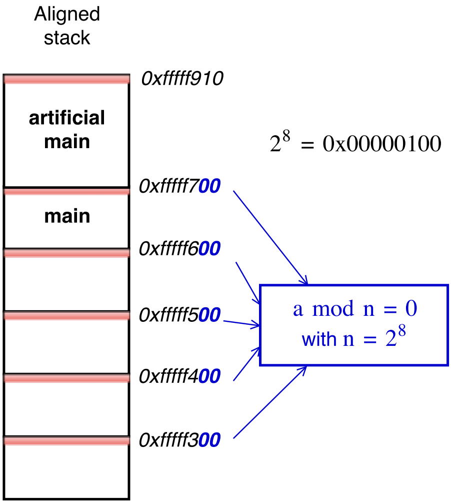
Stack transformation (5/6)

Insertion of a new artificial main



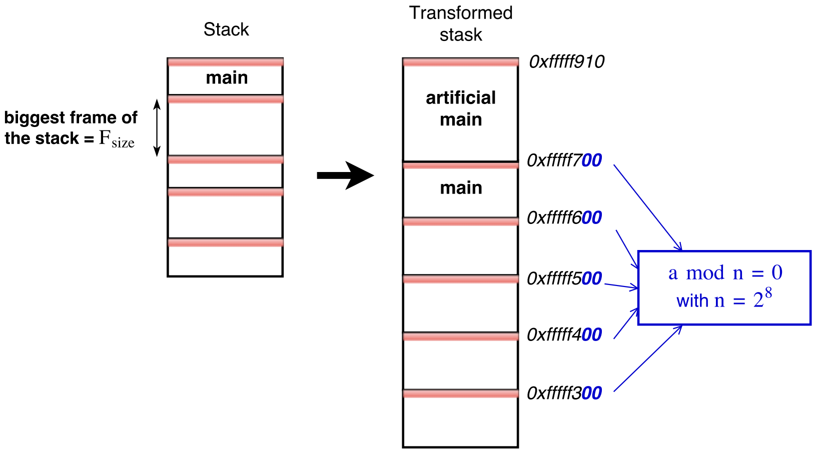
Stack transformation (6/6)

Return addresses locations



Stack transformation

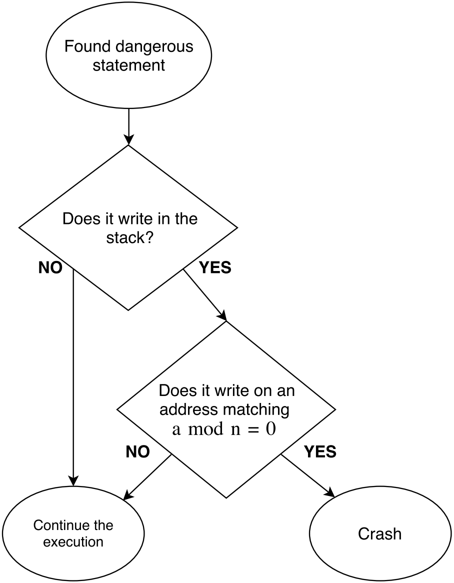
Summary



Code transformation

- Differentiation of return addresses locations with $a \ mod \ n = 0$
- Insertion of additional runtime checks to protect these locations from being overwritten illegally
- Transformation of the code during the compilation phase

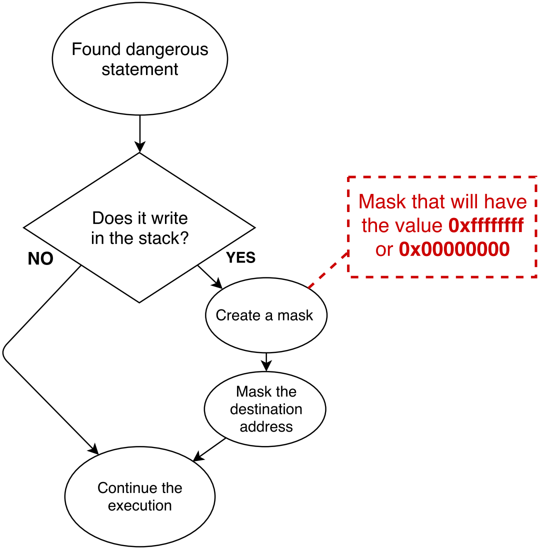
Runtime check



- The protection mechanism might be executed numerous times
- To improve performance we want to have this runtime check as fast as possible

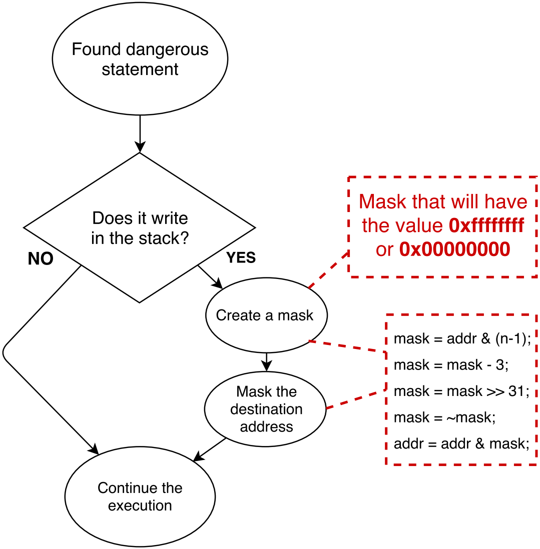
Branchless check

- Uses only a single *if…then…else* instruction
- In certain situation has shown better performance



Branchless check

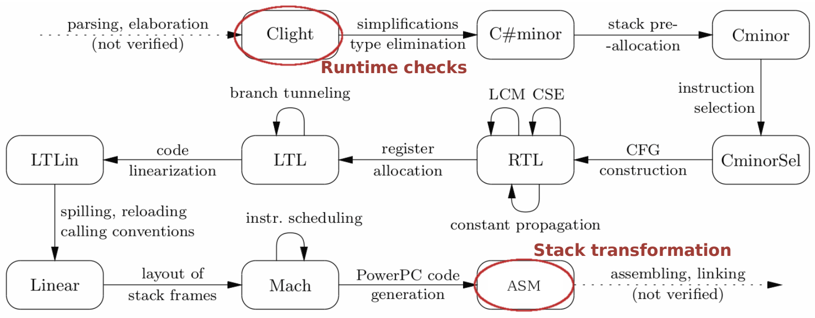
- Uses only a single *if...then...else* instruction
- In certain situation has shown better performance



Implementation environment

CompCert the certified compiler [Leroy, 2009]

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



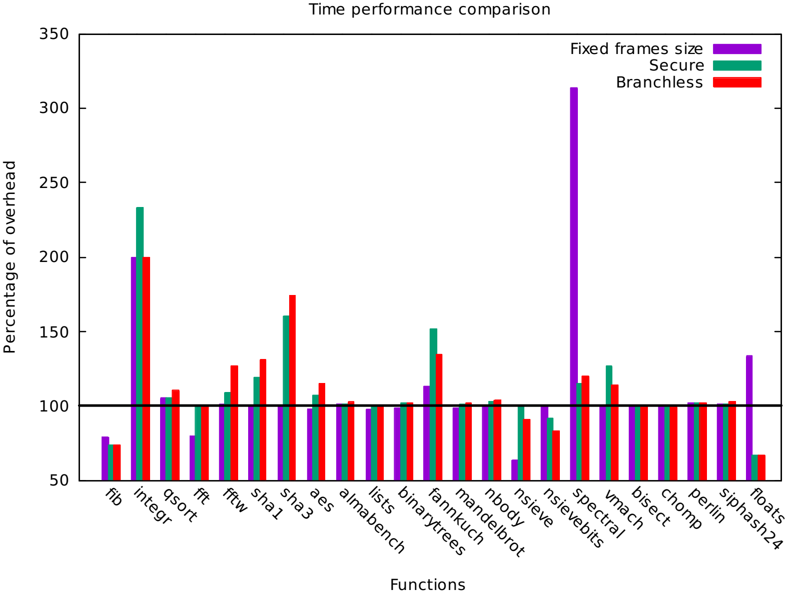
Requirements

- No modifications of the stack (inline assembly)

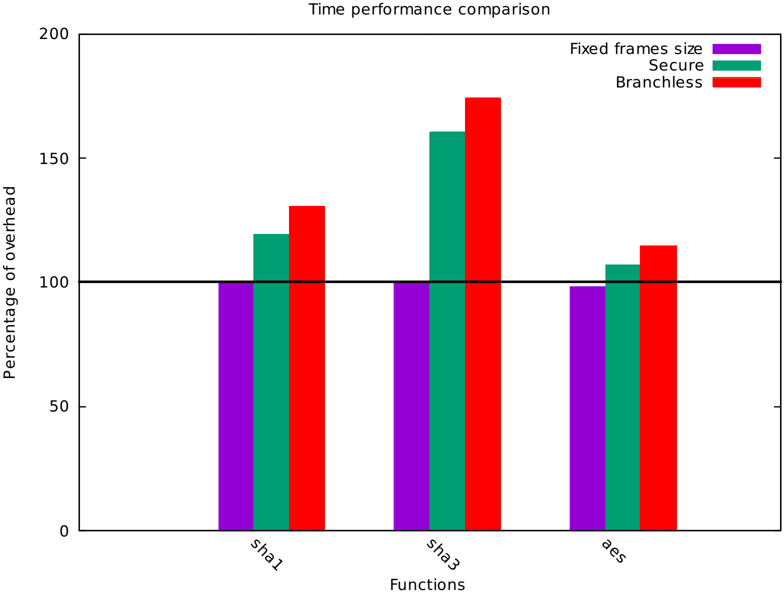
```
int foo(int a) {
asm(``\$sub 50, \%esp'');
printf("Hello world!");
}
```

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

Evaluation of performance



Evaluation of performance



Future work

Prospects

- Test our implementation against more complicated ROP attacks
- Reduce the number of runtime checks with static analysis [Zeng et al, 2011]
- Improve the performance of the runtime checks with a super-optimizer
- See the impact of our approach on memory consumption
- Prove the security properties of our implementation

Conclusion

- Analysis and inspiration from SFI
- Approach protecting programs against ROP attacks
- Implementation on CompCert x86-32
- Evaluation of the approach and future work

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.

Branchless example

Legal execution

```
if (addr > 0xff000000) {
    mask = addr & 0x000000ff;
    mask = mask - 3;
    mask = mask >> 31;
    mask = ~mask;
    addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

addr is not a return address location, it should keep its value

<u>Values</u>

addr = 0xfffff718

mask = 0x00000000

Branchless example

Legal execution

```
  if (addr > 0xff000000) {
    mask = addr & 0x000000ff;
    mask = mask - 3;
    mask = mask >> 31;
    mask = ~mask;
    addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

addr is not a return address location, it should keep its value

Values

addr = 0xfffff718

mask = 0x00000018

Branchless example

Legal execution

```
if (addr > 0xff000000) {
  mask = addr & 0x000000ff;
  mask = mask - 3;
  mask = mask >> 31;
  mask = ~mask;
  addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

addr is not a return address location, it should keep its value

Values

addr $= 0xfffff718$

mask $= 0x00000015$

Branchless example

Legal execution

```
if (addr > 0xff000000) {
    mask = addr & 0x000000ff;
    mask = mask - 3;
    mask = mask >> 31;
    mask = ~mask;
    addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

addr is not a return address location, it should keep its value

Values

addr = 0xfffff718

mask = 0x00000000

Branchless example

Legal execution

```
if (addr > 0xff000000) {
    mask = addr & 0x000000ff;
    mask = mask - 3;
    mask = mask >> 31;
    mask = ~mask;
    addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

addr is not a return address location, it should keep its value

Values

addr = 0xfffff718

mask = 0xffffffff

Branchless example

Legal execution

```
  if (addr > 0xff000000) {
    mask = addr & 0x000000ff;
    mask = mask - 3;
    mask = mask >> 31;
    mask = ~mask;
    addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

addr is not a return address location, it should keep its value

Values

addr = 0xfffff718

mask = 0xffffffff

Branchless example

Legal execution

```
if (addr > 0xff000000) {
  mask = addr & 0x000000ff;
  mask = mask - 3;
  mask = mask >> 31;
  mask = ~mask;
  addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

addr is not a return address location, it should keep its value

<u>Values</u>

addr $= 0xfffff718$

mask $= 0xffffffff$

Branchless example

Illegal execution

```
if (addr > 0xff000000) {
    mask = addr & 0x000000ff;
    mask = mask - 3;
    mask = mask >> 31;
    mask = ~mask;
    addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

Forbidden `write` destination
the program should crash

<u>Values</u>

`addr` $= 0xfffff700$

`mask` $= 0x00000000$

Branchless example

Illegal execution

```
if (addr > 0xff000000) {
    mask = addr & 0x000000ff;
    mask = mask - 3;
    mask = mask >> 31;
    mask = ~mask;
    addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

Forbidden `write` destination
the program should crash

Values

`addr` = 0xfffff700

`mask` = 0x00000000

Branchless example

Illegal execution

```
if (addr > 0xff000000) {
    mask = addr & 0x000000ff;
    mask = mask - 3;
    mask = mask >> 31;
    mask = ~mask;
    addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

Forbidden `write` destination
the program should crash

Values

`addr` $= 0xfffff700$

`mask` $= 0xfffffffd$

Branchless example

Illegal execution

```
if (addr > 0xff000000) {
    mask = addr & 0x000000ff;
    mask = mask - 3;
    mask = mask >> 31;
    mask = ~mask;
    addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

Forbidden `write` destination
the program should crash

Values

`addr` $= 0xfffff700$

`mask` $= 0xffffffff$

Branchless example
Illegal execution

```
  if (addr > 0xff000000) {
    mask = addr & 0x000000ff;
    mask = mask - 3;
    mask = mask >> 31;
    mask = ~mask;
    addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

Forbidden `write` destination
the program should crash

Values
`addr` $= 0xfffff700$
`mask` $= 0x00000000$

Branchless example

Illegal execution

```
if (addr > 0xff000000) {
  mask = addr & 0x000000ff;
  mask = mask - 3;
  mask = mask >> 31;
  mask = ~mask;
  addr = mask & addr;
}
*addr = value;
Continue execution...
```

$n = 2^8 = 0x00000100$

Forbidden `write` destination
the program should crash

<u>Values</u>

`addr` $= 0x00000000$

`mask` $= 0x00000000$

Branchless example

Illegal execution

```
if (addr > 0xff000000) {
  mask = addr & 0x000000ff;
  mask = mask - 3;
  mask = mask >> 31;
  mask = ~mask;
  addr = mask & addr;
}
*addr = value;
Continue execution...
```

$$n = 2^8 = 0x00000100$$

Forbidden `write` destination
the program should crash

Values

`addr` $= 0x00000000$

`mask` $= 0x00000000$