It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks

Stephen Crane, Stijn Volckaert, Felix Schuster,
Christopher Liebchen, Per Larsen, Lucas Davi,
Ahmad-Reza Sadeghi,Thorsten Holz,
Bjorn De Sutter, Michael Franz

Chair of IT Security
Department of Informatics
Technical University of Munich

December 3, 2018

Philip Holzmann

Goals

- Preventing Function-Reuse Attacks: COOP and RILC

  → Prevent disclosure of function pointers

  → Hide code layout

Adversary Model

Adversary Model

- Adversary can exploit a memory corruption vulnerability
  - → read and write arbitrary memory

Adversary Model

- Adversary can exploit a memory corruption vulnerability
  - → read and write arbitrary memory
- Adversary can adjust the attack payload at runtime (e. g. via a scripting environment in a browser)

Adversary Model

- Adversary can exploit a memory corruption vulnerability
  - → read and write arbitrary memory
- Adversary can adjust the attack payload at runtime (e. g. via a scripting environment in a browser)

+ W^X
+ X-only
+ JIT-cache protection

Outline

- Extended COOP
- Dynamic Linking
- PLT Randomization
- Vtable Randomization
- Implementation
- Performance
- Security Evaluation

Extended COOP

Extended COOP

- Regular main loop gadget (ML-G):
  - iterate over container of objects

Extended COOP

- Regular main loop gadget (ML-G):
  - iterate over container of objects
- Alternative ML-Gs:
  - Recursive: REC-G
  - Unrolled: UNR-G

Extended COOP: REC-G

```
class X {
public:
    virtual ~X();
};

class Y {
public:
    virtual void unref();
};
```

```
class Z {
public:
    X* objA;
    Y* objB;

    virtual ~Z() {
        delete objA;
        objB->unref();
    }
};
```

Extended COOP: REC-G

```
class X {
public:
    virtual ~X();
};

class Y {
public:
    virtual void unref();
};
```

```
class Z {
public:
    X* objA;
    Y* objB;

    virtual ~Z() {
        delete objA;
        objB->unref();
    }
};
```

arbitrary vfgadget

Extended COOP: REC-G

```cpp
class X {
public:
    virtual ~X();
};

class Y {
public:
    virtual void unref();
};
```

```cpp
class Z {
public:
    X* objA;
    Y* objB;

    virtual ~Z() {
        delete objA;
        objB->unref();
    }
};
```

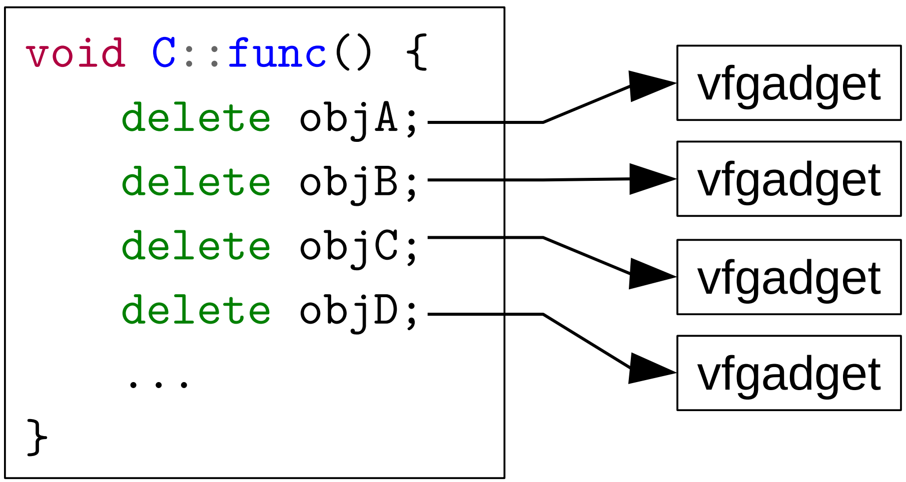
arbitrary vfgadget

recursion

Extended COOP: UNR-G

```
void C::func() {
    delete objA;
    delete objB;
    delete objC;
    delete objD;
    ...
}
```

Extended COOP: UNR-G



Dynamic Linking
(for ELF)

Dynamic Linking
(for ELF)

- Libraries can be loaded at runtime
  - Addresses of symbols not known at compile time

Dynamic Linking
(for ELF)

- Libraries can be loaded at runtime
  - Addresses of symbols not known at compile time
- Global Offset Table & Procedure Linkage Table are used to resolve addresses at runtime

Dynamic Linking:
Global Offset Table

some_lib.h:

```c
extern int foo;
```

```c
#include "some_lib.h"

...
foo = 3;
...
```

some_lib.so:

foo

Dynamic Linking: Global Offset Table

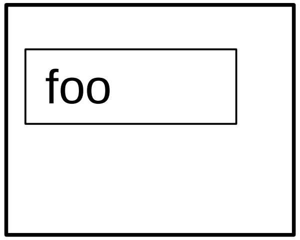
some_lib.h:

```
extern int foo;
```

```
#include "some_lib.h"


...
foo = 3;
...
```

```


...
6d4: movl    $0x3, 0x20095a(%rip)
...
```

some_lib.so:



Dynamic Linking: Global Offset Table

some_lib.h:

```
extern int foo;
```

```
#include "some_lib.h"

...
foo = 3;
...
```

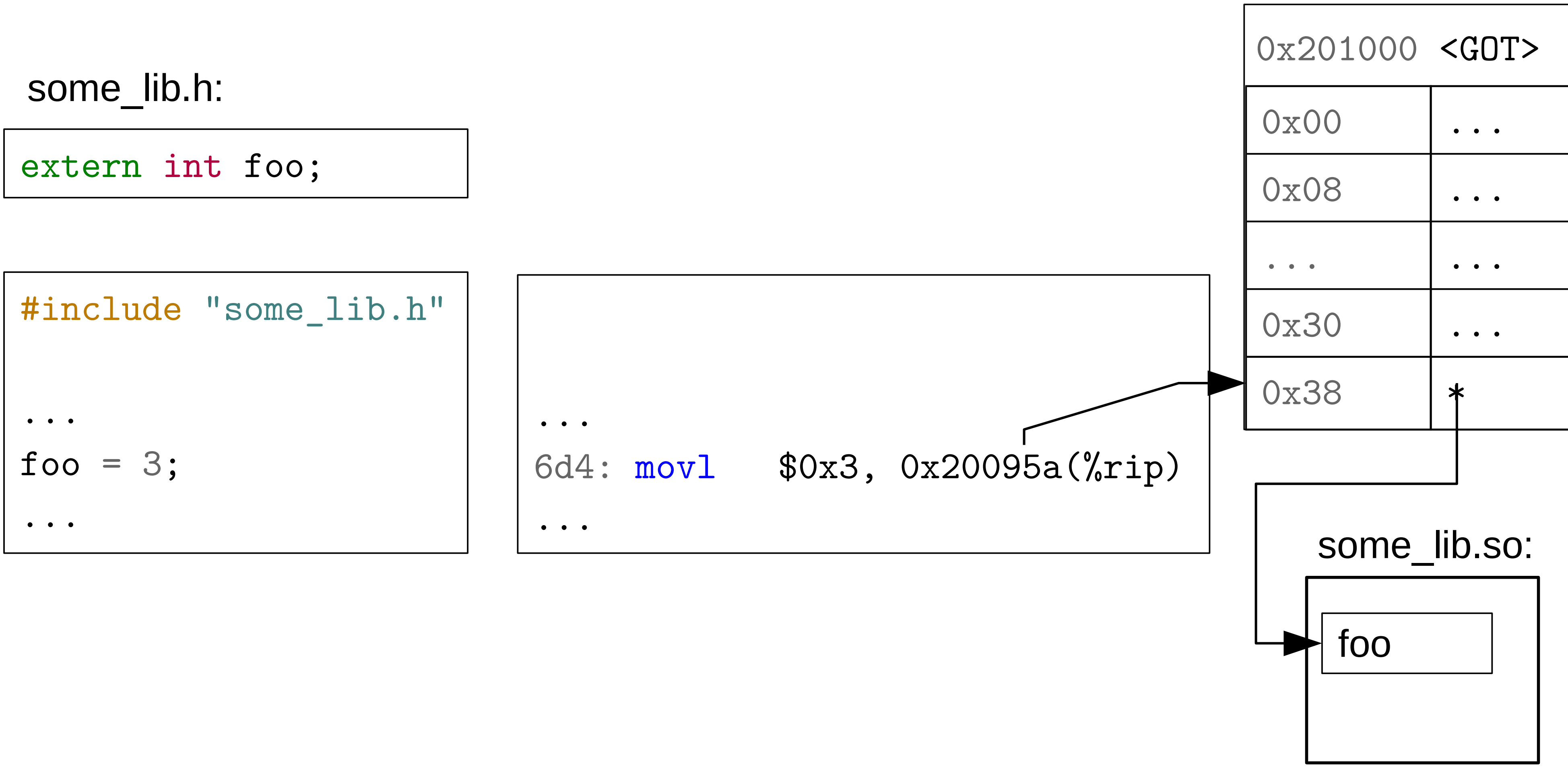
```
...
6d4: movl    $0x3, 0x20095a(%rip)
...
```

0x201000 <GOT>	
0x00	...
0x08	...
...	...
0x30	...
0x38	*

some_lib.so:

foo

Dynamic Linking: Global Offset Table



Dynamic Linking: Procedure Linkage Table

some_lib.h:

```
void fun(void);
void fun2(void);
```

```
#include "some_lib.h"


...
fun();
...
```

some_lib.so:

fun
fun2

Dynamic Linking: Procedure Linkage Table

some_lib.h:

```
void fun(void);
void fun2(void);
```

```
#include "some_lib.h"


...
fun();
...
```

```
690 <.plt>:
  690: pushq  0x200972(%rip)
  696: jmpq   *0x200974(%rip)
  69c: nopl   0x0(%rax)
6a0 <fun2@plt>:
  6a0: jmpq   *0x200972(%rip)
  6a6: pushq  $0x0
  6ab: jmpq   690 <.plt>
6b0 <fun@plt>:
  6b0: jmpq   *0x20096a(%rip)
  6b6: pushq  $0x1
  6bb: jmpq   690 <.plt>

...
819: callq  6b0 <fun@plt>
...
```

some_lib.so:

fun
fun2

Dynamic Linking: Procedure Linkage Table

some_lib.h:

```
void fun(void);
void fun2(void);
```

```
#include "some_lib.h"

...
fun();
...
```

```
690 <.plt>:
  690: pushq  0x200972(%rip)
  696: jmpq   *0x200974(%rip)
  69c: nopl   0x0(%rax)
6a0 <fun2@plt>:
  6a0: jmpq   *0x200972(%rip)
  6a6: pushq  $0x0
  6ab: jmpq   690 <.plt>
6b0 <fun@plt>:
  6b0: jmpq   *0x20096a(%rip)
  6b6: pushq  $0x1
  6bb: jmpq   690 <.plt>

...
819: callq  6b0 <fun@plt>
...
```

0x201000 <GOT>	
0x00	...
0x08	...
0x10	...
0x18	*
0x20	*
0x28	...

some_lib.so:

fun
fun2

Dynamic Linking: Procedure Linkage Table

some_lib.h:

```
void fun(void);
void fun2(void);
```

```
#include "some_lib.h"

...
fun();
...
```

```
690 <.plt>:
  690: pushq   0x200972(%rip)
  696: jmpq    *0x200974(%rip)
  69c: nopl    0x0(%rax)
6a0 <fun2@plt>:
  6a0: jmpq    *0x200972(%rip)
  6a6: pushq   $0x0
  6ab: jmpq    690 <.plt>
6b0 <fun@plt>:
  6b0: jmpq    *0x20096a(%rip)
  6b6: pushq   $0x1
  6bb: jmpq    690 <.plt>

...
819: callq  6b0 <fun@plt>
...
```

0x201000 <GOT>	
0x00	...
0x08	...
0x10	...
0x18	*
0x20	*
0x28	...

some_lib.so:

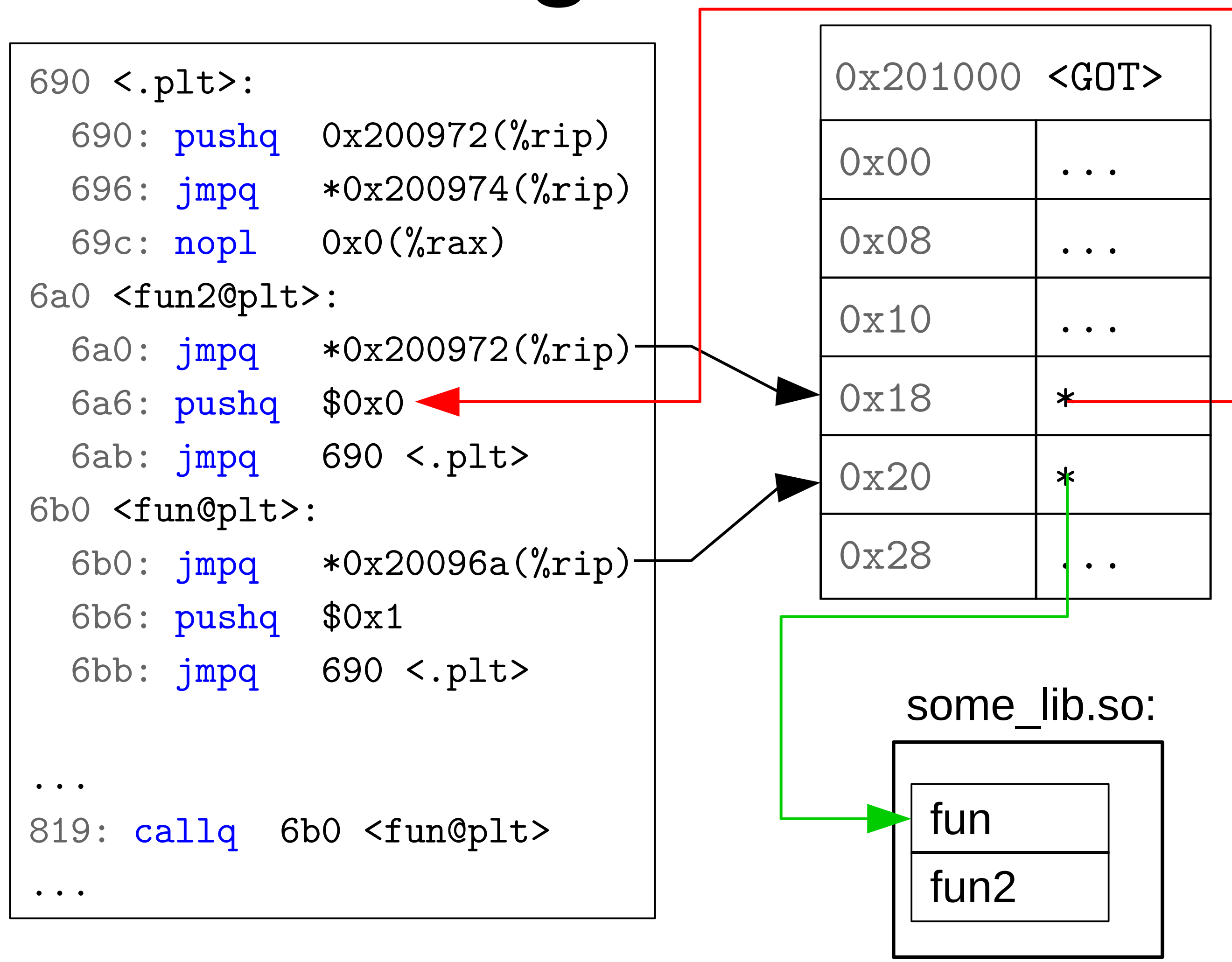
fun
fun2

Dynamic Linking: Procedure Linkage Table

some_lib.h:

```
void fun(void);
void fun2(void);
```

```
#include "some_lib.h"

...
fun();
...
```



Problems with GOT and PLT

- Global Offset Table has to be stored in read-writable memory

    → Adversary can read code pointers and infer memory layout

PLT Randomization

PLT Randomization

- Transform indirect jumps into direct jumps
  - Can now strip function pointers from GOT

PLT Randomization

- Transform indirect jumps into direct jumps
  - Can now strip function pointers from GOT
- Place PLT in X-only memory

PLT Randomization

- Transform indirect jumps into direct jumps
  - Can now strip function pointers from GOT
- Place PLT in X-only memory
- Eager binding (instead of lazy)

PLT Randomization

- Transform indirect jumps into direct jumps
  - Can now strip function pointers from GOT
- Place PLT in X-only memory
- Eager binding (instead of lazy)
- Insert booby traps

PLT Randomization

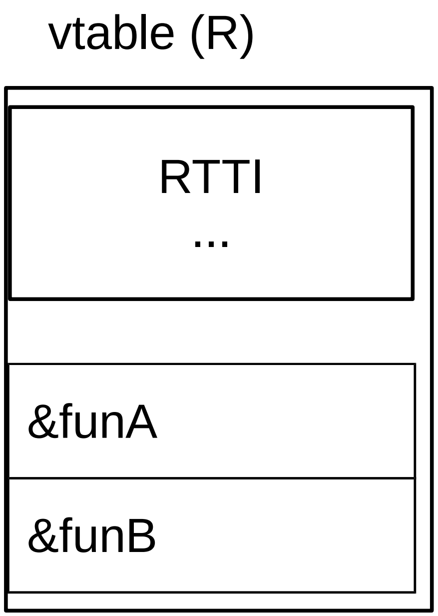
- Transform indirect jumps into direct jumps
  - Can now strip function pointers from GOT
- Place PLT in X-only memory
- Eager binding (instead of lazy)
- Insert booby traps
- Randomize order of PLT entries

PLT Randomization

- Transform indirect jumps into direct jumps
  - Can now strip function pointers from GOT
- Place PLT in X-only memory
- Eager binding (instead of lazy)
- Insert booby traps
- Randomize order of PLT entries
- Call using trampolines

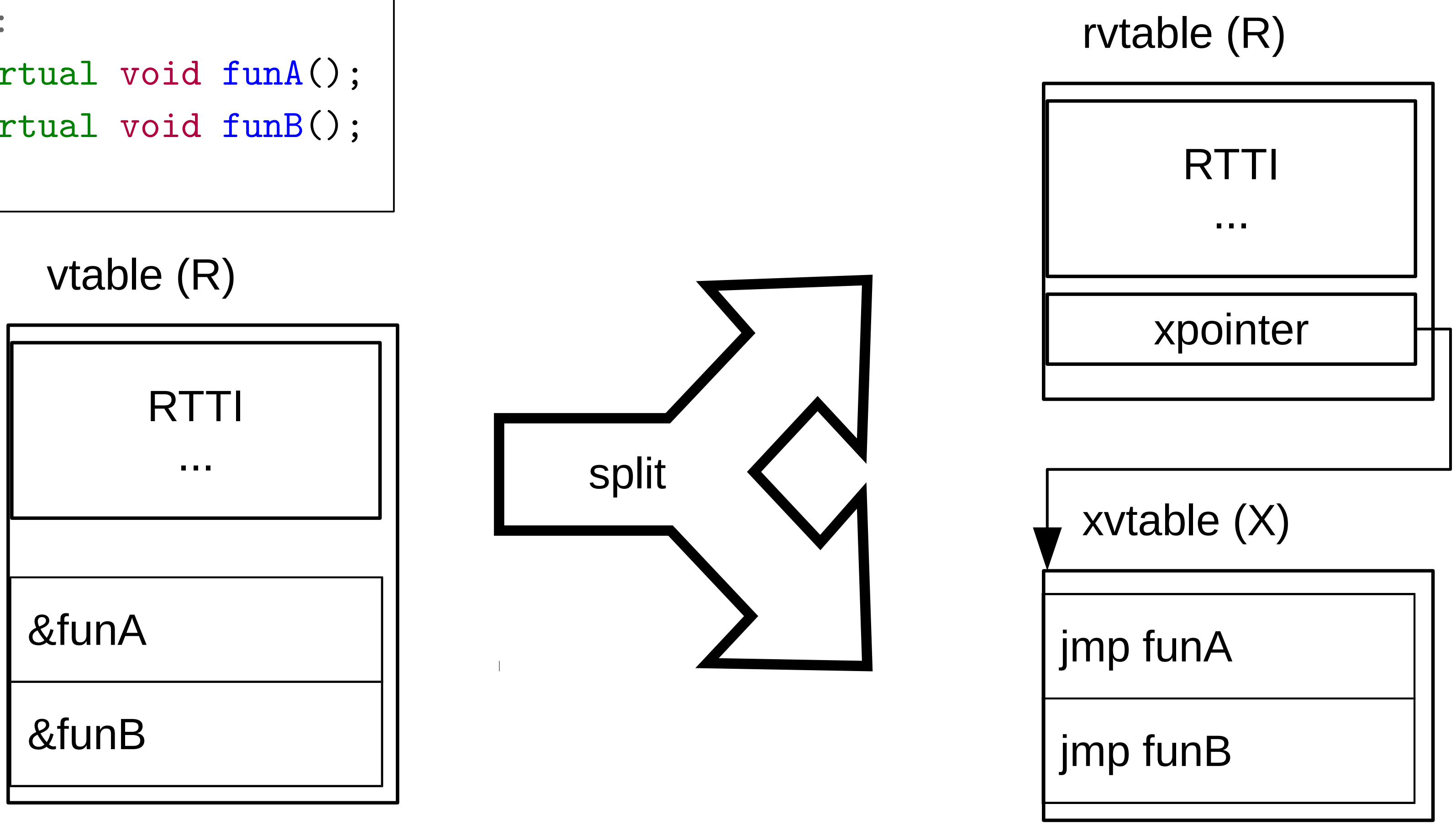
Vtable Splitting

```
class A {
public:
    virtual void funA();
    virtual void funB();
};
```



Vtable Splitting

```
class A {
public:
    virtual void funA();
    virtual void funB();
};
```



Vtable Randomization

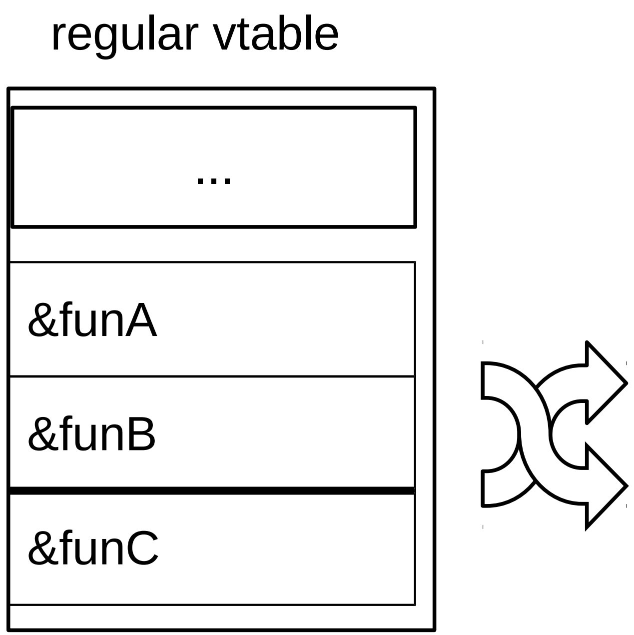
```
class A {
public:
    virtual void funA();
    virtual void funB();
};

class B : public A {
public:
    virtual void funC();
};
```

Vtable Randomization

```
class A {
public:
    virtual void funA();
    virtual void funB();
};

class B : public A {
public:
    virtual void funC();
};
```

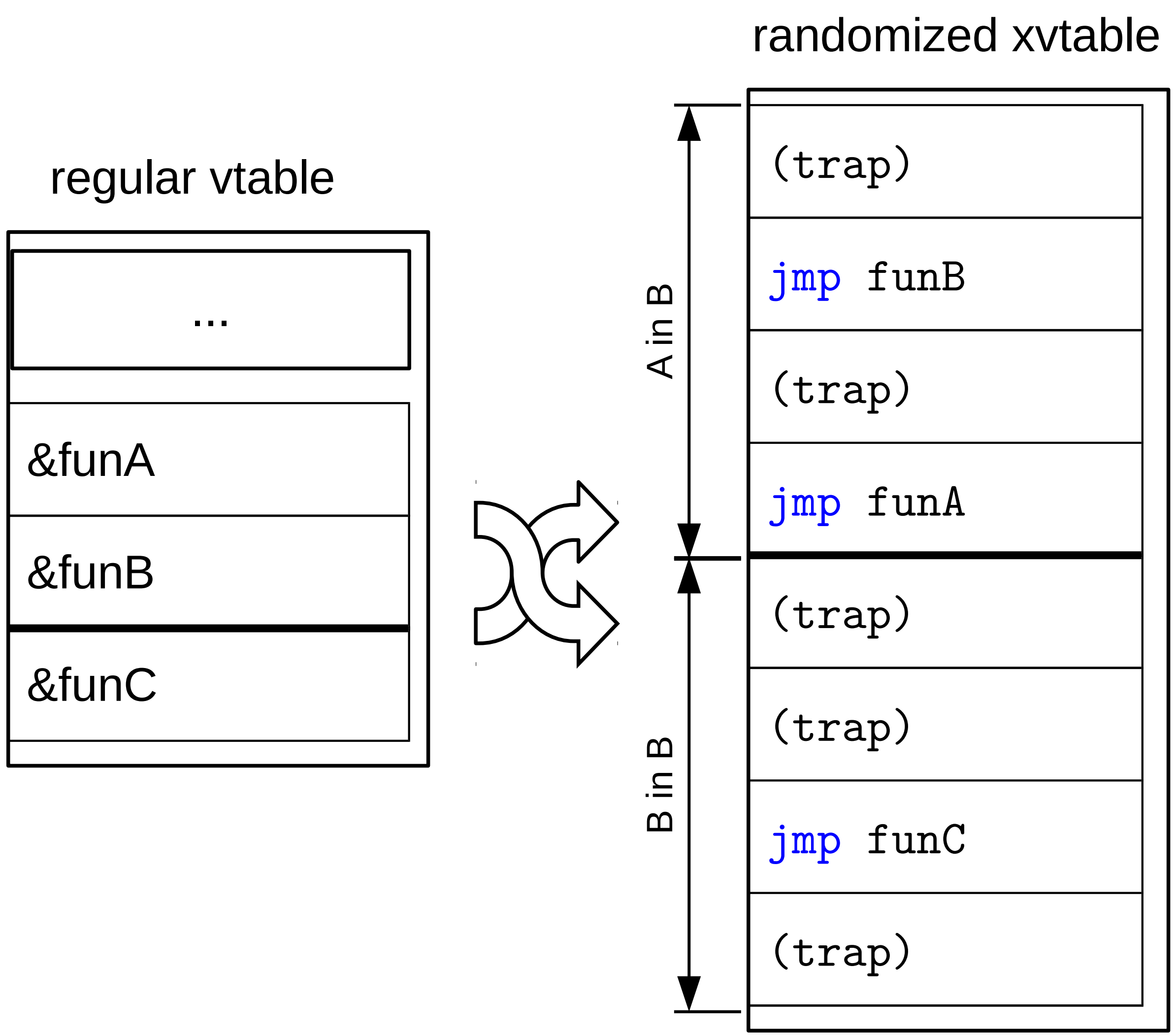
Vtable Randomization

```
class A {
public:
    virtual void funA();
    virtual void funB();
};

class B : public A {
public:
    virtual void funC();
};
```



Vtable Randomization

```
class A {
public:
    virtual void funA();
    virtual void funB();
};

class B : public A {
public:
    virtual void funC();
};
```



Vtable Randomization: Virtual Function Call

```
void example(void) {
    A* x = ...;
    x->funA();
}
```

xvtable (X)

```
(trap)
jmp funB
(trap)
jmp funA
```

Vtable Randomization: Virtual Function Call

```
void example(void) {
    A* x = ...;
    x->funA();
}
```

```
example:
    jmp <trampoline0>
return_site0:
    ...
```

```
trampoline0:
    call x->vtable->xvtable[3]
    jmp <return_site0>
```

xvtable (X)

(trap)
jmp funB
(trap)
jmp funA

Vtable Randomization: Virtual Function Call

```
void example(void) {
    A* x = ...;
    x->funA();
}
```

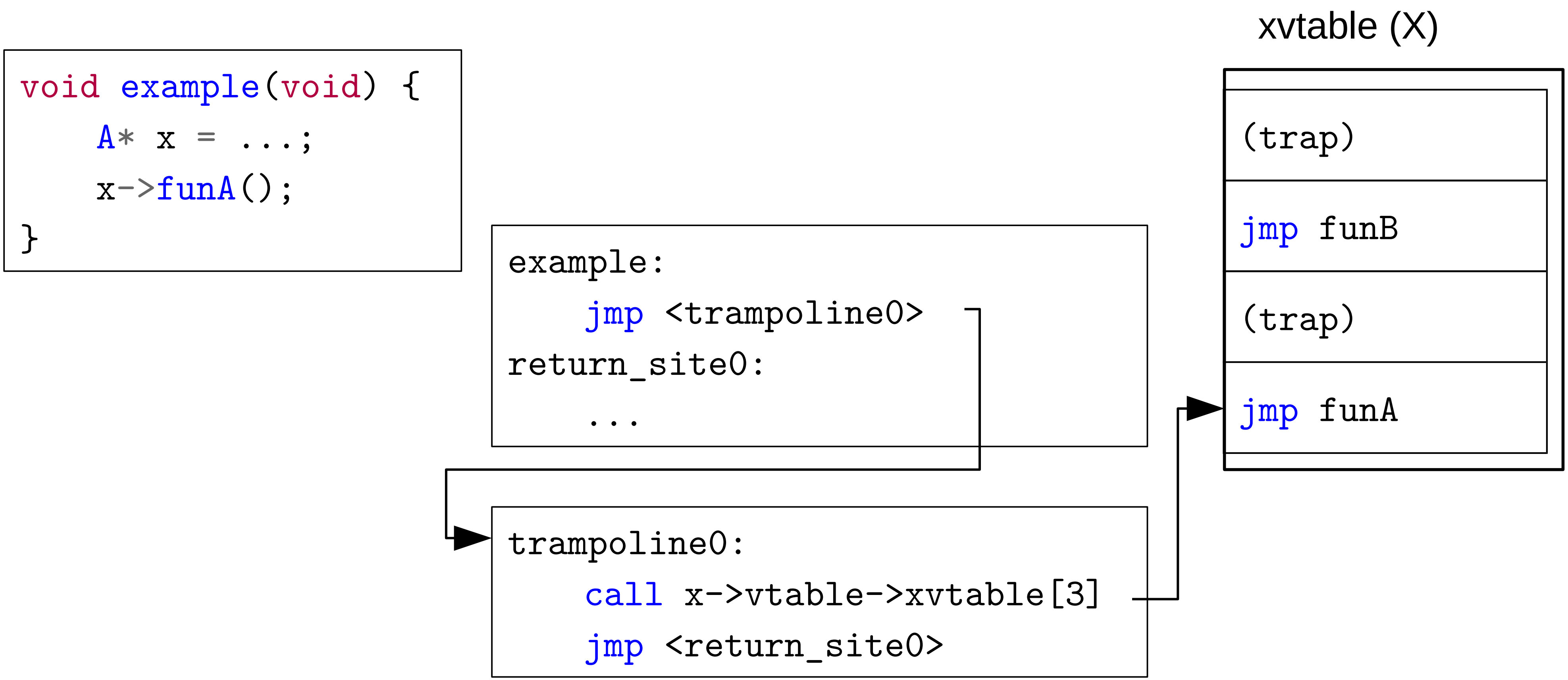
```
example:
    jmp <trampoline0>
return_site0:
    ...
```

```
trampoline0:
    call x->vtable->xvtable[3]
    jmp <return_site0>
```

xvtable (X)

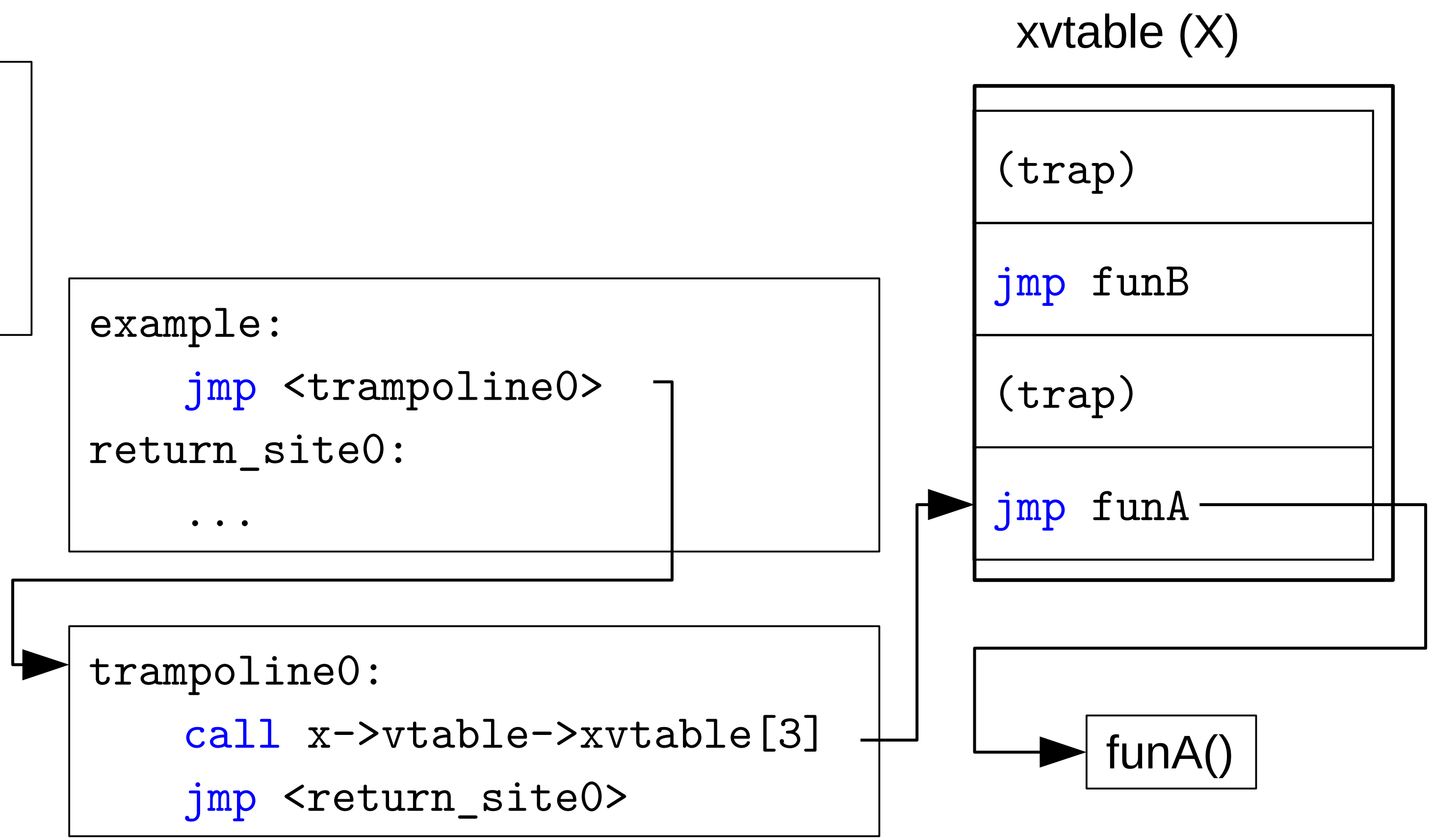
```
(trap)
jmp funB
(trap)
jmp funA
```

Vtable Randomization: Virtual Function Call

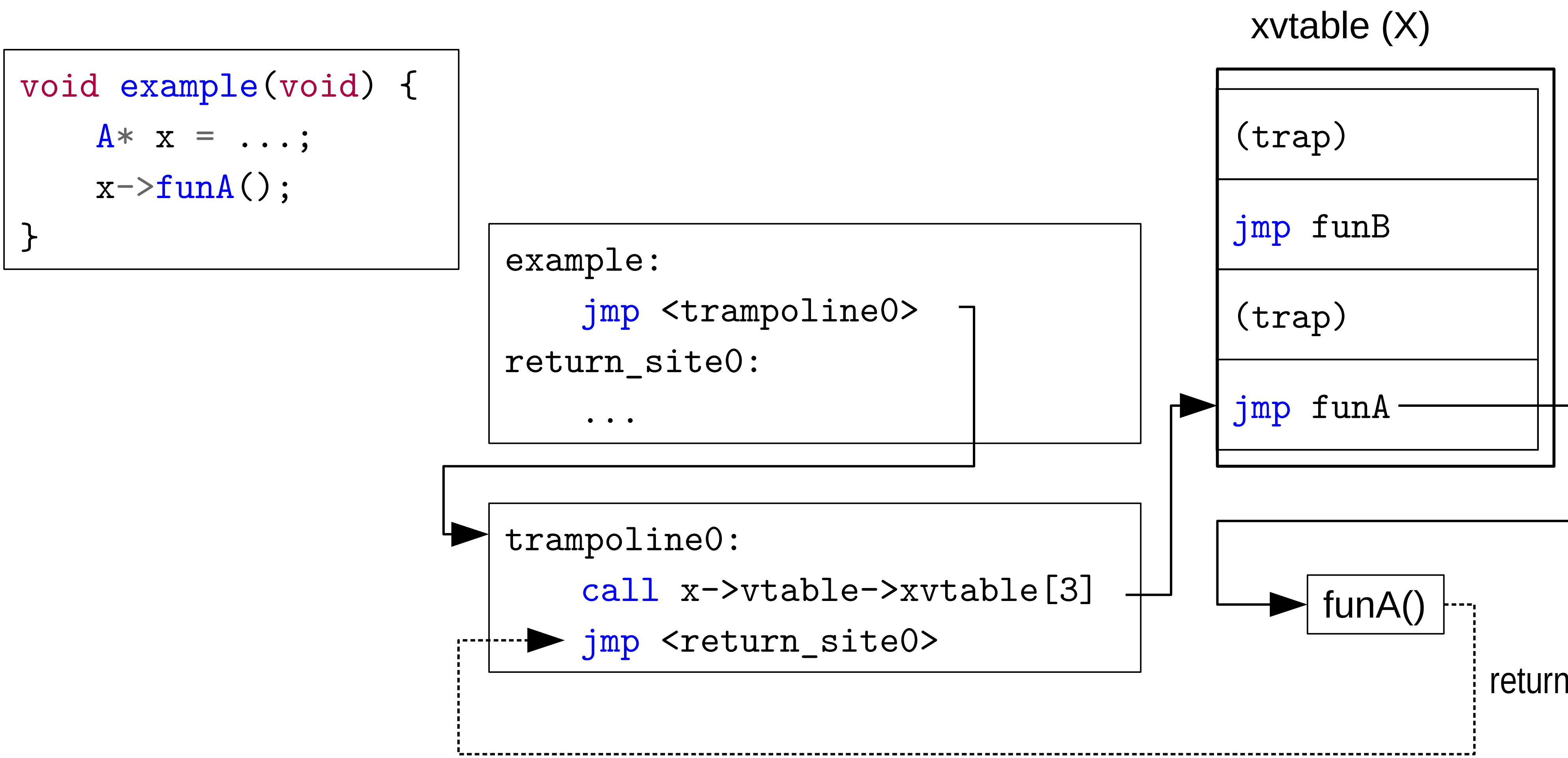


Vtable Randomization: Virtual Function Call

```
void example(void) {
    A* x = ...;
    x->funA();
}
```

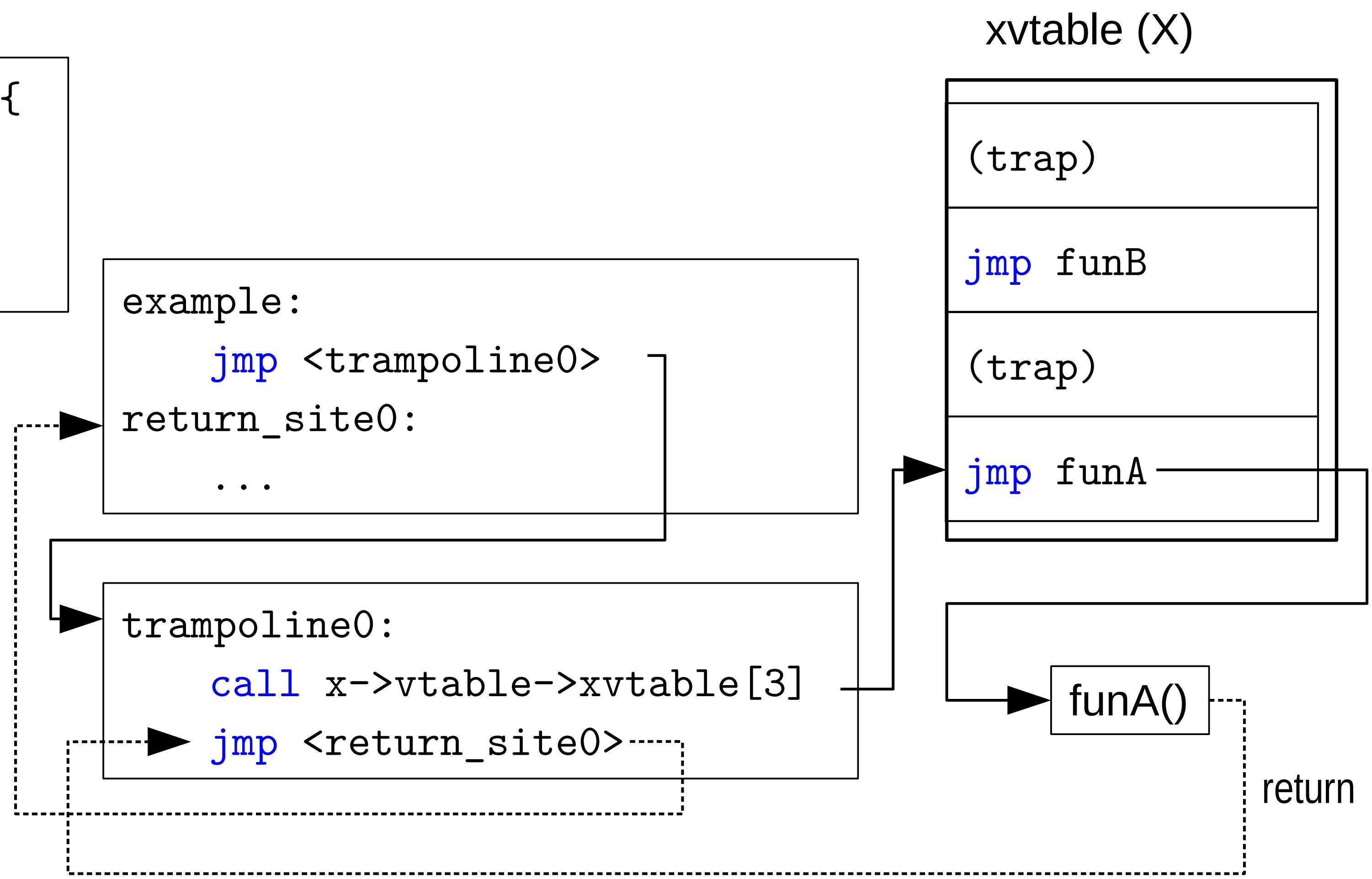


Vtable Randomization: Virtual Function Call



Vtable Randomization: Virtual Function Call

```
void example(void) {
    A* x = ...;
    x->funA();
}
```



Implementation: Readactor++

Implementation: Readactor++

- Extends Readactor (protects against ROP)

Implementation: Readactor++

- Extends Readactor (protects against ROP)
- Modified Clang:
  - Ensure separation of code and data
  - Collect *TRaP* information

Implementation: Readactor++

- Extends Readactor (protects against ROP)
- Modified Clang:
  - Ensure separation of code and data
  - Collect *TRaP* information
- At program start: RandoLib
  - Perform randomization
  - Rewrite call-sites
  - Unloaded afterwards

Performance

Performance

- Overall average performance overhead of 1.1 %

Performance

- Overall average performance overhead of 1.1 %
- Combined with Readactor: 8.4 %

Performance

- Overall average performance overhead of 1.1 %
- Combined with Readactor: 8.4 %
- Memory overhead negligible (?)

Security Evaluation

(Ignoring side channels)

Security Evaluation
(Ignoring side channels)

- Attacker can only disclose code pointers to trampolines
  - Cannot infer location of other functions from that

Security Evaluation
(Ignoring side channels)

- Attacker can only disclose code pointers to trampolines
  - Cannot infer location of other functions from that
- Program-allocated function pointer tables are unprotected

Security Evaluation

(Ignoring side channels)

- Attacker can only disclose code pointers to trampolines
  - Cannot infer location of other functions from that
- Program-allocated function pointer tables are unprotected
- Probability of correctly guessing 3 vfgadgets $< (1/16)^3 \approx 0.024$ %

References

– Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, Michael Franz. *“It’s a TRaP: Table Randomization and Protection against Function-Reuse Attacks.”*

– Michael Matz, Jan Hubička, Andreas Jaeger, Mark Mitchell. *“System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.6”*

Questions?