

motivation: beyond stack canaries

stack canaries: try to make sure we return to genuine place

should make return-oriented programming hard

problem: defeated with information leak

alternate idea: make sure we aren't returning to gadget

...by checking that it actually called our function

a simple way to check returns?

observation: places we return to usually after call instructions

exception: 'tail calls' — we'll ignore this for now

we could check for one...

replace return with:

```
return address ← PopFromStack()  
if DecodeInstruction(return address - 5) == "call thisFunction":  
    goto return address  
else:  
    CRASH
```

a simple way to check returns?

more practical: `label $ID` instruction with encoding:
TWO-BYTE-OPCODE FOUR-BYTE-CONSTANT
(real version: can reuse some sufficiently nop-like instruction)

```
...  
    call foo  
    label $0xf19279bb // random ID for function foo  
...
```

```
foo:  
...  
    pop %rdx // RDX <- return address  
    cmp $0xf19279bb, 2(%rdx)  
    jne CRASH  
    jmp *%rdx
```

looks like canaries? (1)

what attacks does this stop that canaries don't?

looks like canaries? (1)

what attacks does this stop that canaries don't?

ID does not need to be secret!

assuming non-executable writeable memory, no!
attacker can't write new places for return to go

looks like canaries? (1)

what attacks does this stop that canaries don't?

ID does not need to be secret!

- assuming non-executable writeable memory, no!
- attacker can't write new places for return to go

avoids “stack pivoting” attacks

- attacker can't make stack pointer point to wrong part of stack...
- and expect it to return differently

looks like canaries? (2)

what attacks does this NOT stop that canaries do?

example: SortList can be called from Innocent,
then return from Dangerous

assumption: attacker can overwrite return address at right time
(running on another core? problem with sortFunc1?)

```
void Innocent() {  
    ...  
    SortList(someList1,  
             sortFunc1);  
    Use(someList1);  
    ...  
}
```

```
void Dangerous() {  
    ...  
    SortList(someList2,  
             sortFunc2);  
    UseDangerously(someList2);  
    ...  
}
```


checking a VTable call

```
class A { public:  
    virtual void bar() { ... }  
};  
class B : public A { public:  
    void bar() { ... }  
};  
void example(A *obj) {  
    obj->bar();  
}
```

example:

```
// rax <- vtable address  
movq (%rdi), %rax  
// rdx <- first vtable entry  
movq (%rax), %rax  
// call using vtable entry  
call *%rax
```

checking a VTable call

```
class A { public:  
    virtual void bar() { ... }  
};  
class B : public A { public:  
    void bar() { ... }  
};  
void example(A *obj) {  
    obj->bar();  
}
```

example:

```
// rax <- vtable address  
movq (%rdi), %rax  
// rdx <- first vtable entry  
movq (%rax), %rax  
// call using vtable entry  
call *%rax
```

example uses VTable to call method
target for memory corruption attacks
just like return addresses
so, apply same strategy

checking a VTable call

```
class A { public:  
    virtual void bar() { ... }  
};  
class B : public A { public:  
    void bar() { ... }  
};  
void example(A *obj) {  
    obj->bar();  
}
```

```
A::bar():  
    label $0xe0c5df0b  
...  
B::bar():  
    label $0xe0c5df0b  
...
```

```
example:  
    // rax <- vtable address  
    movq (%rdi), %rax  
    // rdx <- first vtable entry  
    movq (%rax), %rax  
    // call using vtable entry  
    call *%rax
```

```
example:  
    movq (%rdi), %rax  
    movq (%rax), %rax  
    cmpq $0xe0c5df0b, 2(%rax)  
    jne CRASH  
    call *%rax  
...
```

checking a VTable return

```
A::bar():  
    label $0xe0c5df0b  
    ...  
    pop %rdx // RDX <- return address  
    cmp $0x64a0cfe3, 2(%rdx)  
    jne CRASH  
    jmp *%rdx  
B::bar():  
    label $0xe0c5df0b  
    ...  
    pop %rdx // RDX <- return address  
    cmp $0x64a0cfe3, 2(%rdx)  
    jne CRASH  
    jmp *%rdx
```

```
example:  
    movq (%rdi), %rax  
    movq (%rax), %rax  
    cmpq $0xe0c5df0b, 2(%rax)  
    jne CRASH  
    call *%rax  
    label $0x64a0cfe3  
    ret
```

if we want to use this label-checking on the return
need to choose the same label for A::bar and B::bar return, too

calls through function pointers

```
typedef int (*CompareFnType)(const char*, const char*)
void SortFunction(const char **items, CompareFnType compare)
{
    ...
    (*compare)(a, b);
    ...
}
```

here: call through explicitly passed function pointer

want to do the same thing we did for VTable calls

- all the compare functions have the same label

- all the returns from compare functions have the same label

yes, if we can somehow label all the compare functions

calls through function pointers

```
typedef int (*CompareFnType)(const char*, const char*)
void SortFunction(const char **items, CompareFnType compare)
{
    ...
    (*compare)(a, b);
    ...
}
```

here: call through explicitly passed function pointer

want to do the same thing we did for VTable calls

- all the compare functions have the same label

- all the returns from compare functions have the same label

yes, *if we can somehow label all the compare functions*

CFI overhead

Abadi et al's 2004 paper:

- used label-based approach

- 0-45% time overhead on SPECcpu2000 benchmarks

- best: compression program

- worst: chess engine

Tice et al's 2014 paper (clang-style impl, sometimes in GCC, sometimes in Clang)

- could separately enable different parts

- in tests on SPECcpu 2006 benchmarks:

- 0-10% slowdown for VTable dereference checks

 - but 20% without tuning

- 0-6% for other indirect call checking

looks like canaries? (2)

what attacks does this NOT stop that canaries do?

example: SortList can be called from Innocent,
then return from Dangerous

assumption: attacker can overwrite return address at right time
(running on another core? problem with sortFunc1?)

```
void Innocent() {  
    ...  
    SortList(someList1,  
             sortFunc1);  
    Use(someList1);  
    ...  
}
```

```
void Dangerous() {  
    ...  
    SortList(someList2,  
             sortFunc2);  
    UseDangerously(someList2);  
    ...  
}
```


concept: labels and control flow graph

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

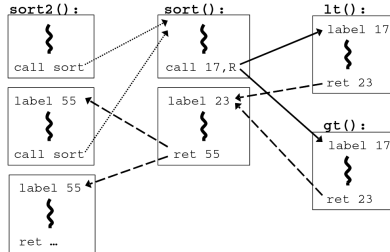


Figure 1: Example program fragment and an outline of its CFG and CFI instrumentation.

control flow graph

nodes = blocks of code

edges = *potential jump/call*

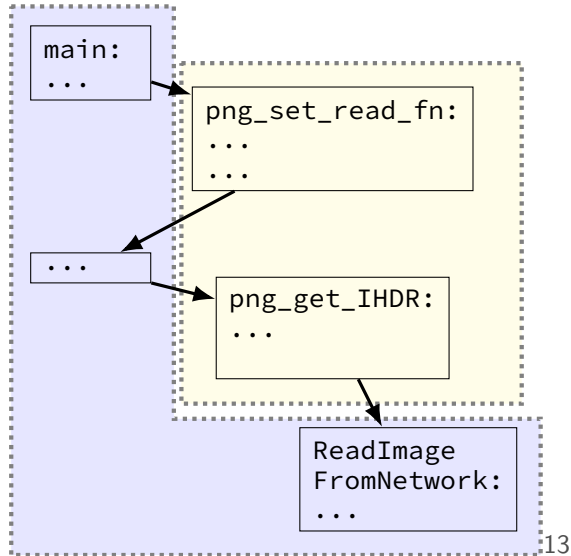
assigning labels: every in-edge needs to check same label at source

library-crossing CFGs

main.c

```
#include <png.h>
void ReadImageFromNetwork(
    png_structp libpng_handle,
    unsigned char *bytes,
    size_t size
) { ... }

int main() {
    /* init libpng */
    png_structp libpng_handle = ...;
    /* tell libpng how to read image data */
    png_set_read_fn(
        libpng_handle, ...,
        ReadImageFromNetwork
    )
    ...
    /* extract "header"
       information from image */
    png_get_IHDR(libpng_handle, ...)
    ...
}
```



CFGs will be imprecise

```
FunctionPtr p = functionA;
Example() {
    while (true) {
        ...
        if (SomethingComplicated()) {
            (*p)();
        } else if (SomethngElseComplicated()) {
            foo();
        }
        ...
    }
}
foo() {
    ...
    if (AnotherComplexThing()) {
        p = functionB;
    }
}
```

finding possible function pointer values?

given call using function pointers

how do we find the **legitimate** possible values?

one idea:

```
for each fptr constant X: PossibleValues[X] = {X}
for each fptr variable X:
    PossibleValues[X] = empty set
until PossibleValues stops changing:
    for each fptr assignment LHS=RHS:
        for each fptr variable/constant Y
            that RHS could evaluate to:
                PossibleValues[LHS] = Union(
                    PossibleValues[LHS],
                    PossibleValues[Y]
                )
```

finding possible function pointer values?

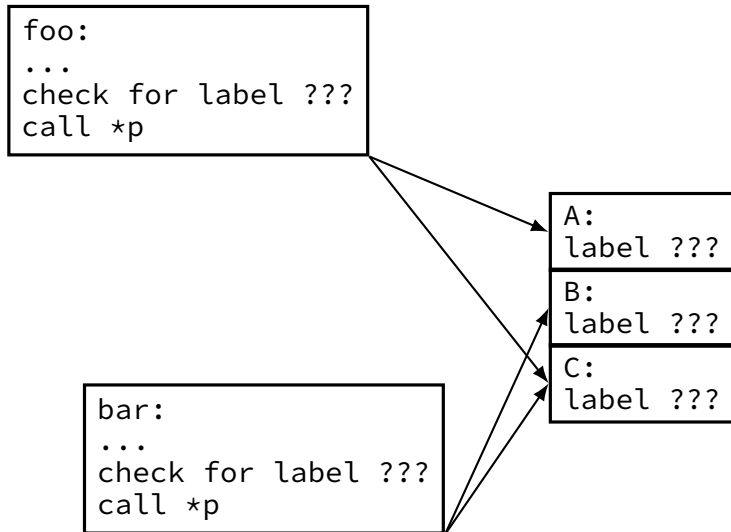
given call using function pointers

how do we find the **legitimate** possible values?

one idea:

```
for each fptr constant X: PossibleValues[X] = {X}
for each fptr variable X:
    PossibleValues[X] = empty set
until PossibleValues stops changing:
    for each fptr assignment LHS=RHS:
        for each fptr variable/constant Y
            that RHS could evaluate to:
                PossibleValues[LHS] = Union(
                    PossibleValues[LHS],
                    PossibleValues[Y]
                )
```

labels aren't enough?



labels aren't enough?

```
foo:  
...  
check for label ???  
call *p
```

```
bar:  
...  
check for label ???  
call *p
```

A:	label ???
B:	label ???
C:	label ???

The diagram shows two boxes on the left, 'foo:' and 'bar:', each containing code that checks for a label and then calls a pointer '*p'. Arrows from the 'call *p' lines in both boxes point to a central box on the right. This central box is divided into three sections labeled A, B, and C, each containing 'label ???'. This represents a jump table or a set of possible targets for the indirect call.

two possible fixes:

make checks scan
for multiple labels
(more overhead)

allow foo to call B
and bar to call A
(easier to attack)

clang's CFI implementation

<https://clang.llvm.org/docs/ControlFlowIntegrity.html>

also <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice>

only checks calls via VTables or function pointers

stable implementation requires libraries compiled with support

label information is placed in separate data structure

looked up using function or VTable addresses

trick: keep functions in one region of memory

clang's CFI implementation

<https://clang.llvm.org/docs/ControlFlowIntegrity.html>

also <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice>

only checks calls via VTables or function pointers

stable implementation requires libraries compiled with support

label information is placed in separate data structure

looked up using function or VTable addresses

trick: keep functions in one region of memory

clang idea for CFI indirect calls

```
start_funcs_with_two_string_args:  
    .align 8  
compare_alpha:  
    jmp real_compare_alpha  
    .align 8  
run_command_with_arg:  
    jmp real_run_command_with_arg  
    .align 8  
print_two_strings:  
    jmp real_print_two_strings  
    .align 8  
move_file:  
    jmp real_move_file  
    .align 8  
compare_reverse_alpha:  
    jmp real_compare_reverse_alpha  
end_funcs_with_two_sting_args:
```

functions of same type
placed together

every func's address
is multiple of 8

clang idea for CFI indirect calls

```
start_funcs_with_two_string_args:
.align 8
compare_alpha:
    jmp real_compare_alpha
.align 8
run_command_with_arg:
    jmp real_run_command_with_arg
.align 8
print_two_strings:
    jmp real_print_two_strings
.align 8
move_file:
    jmp real_move_file
.align 8
compare_reverse_alpha:
    jmp real_compare_reverse_alpha
end_funcs_with_two_sting_args:
```

check pseudocode:
round fptr to multiple of 8
if fptr < start or fptr > end:
 CRASH
allowed $\leftarrow$ [1,0,0,0,1]
 'mask' for compare funcs
offset $\leftarrow$ fptr - start
if bit (offset/8) of allowed
 is not set:
 CRASH

clang idea for VTables

check VTable element address instead of function address

otherwise

- place all VTables for related classes together

- check start/end address for VTables

- bit mask indicating which VTable entries are okay for call

CFI prevents?

```
class Foo { public: virtual void f() { } };  
class Bar : public Foo { public: virtual void f() { g(1); } };  
class Quux : public Foo { public: virtual void f() { } };  
void g(int x) { if (x == 0) { danger(); } }  
int h(int x) { return 0; }  
int (*ptr)(int) = &h;
```

with clang's CFI, which likely can end up calling `danger()` if an attacker can first write to arbitrary memory locations?

- A. `(*ptr)(1);`
- B. `(*ptr)(0);`
- C. `Foo *q = attacker_controlled(); q->f()`
- D. `Quux *q = attacker_controlled(); q->f()`
- E. none of these

CFI prevents?

```
class Foo { public: virtual void f() { } };  
class Bar : public Foo { public: virtual void f() { g(1); } };  
class Quux : public Foo { public: virtual void f() { } };  
void g(int x) { if (x == 0) { danger(); } }  
int h(int x) { return 0; }  
int (*ptr)(int) = &h;
```

with clang's CFI, which likely can end up calling `danger()` if an attacker can first write to arbitrary memory locations?

- A. `(*ptr)(1);`
- B. `(*ptr)(0);` *if compiler thinks ptr set to g ever, yes; otherwise, no*
- C. `Foo *q = attacker_controlled(); q->f()` *can only call real f() methods; could call Bar::f() but how to change g's arg?*
- D. `Quux *q = attacker_controlled(); q->f()` *can only call real f() methods of Quux and subclasses, so can't even call Bar::f()*
- E. none of these

CFI: dynamically loaded libraries?

what about dynamically loaded libraries...

problem: precomputed control flow graph now invalid

Intel hardware CFI support

Intel adds 'endbr64' instruction

special NOP instruction that acts as a label

means: only one label for everything

prevents gadgets from existing

“Control Flow Enforcement”: if enabled

computed jump to non-endbr64 triggers segfault-like error

ARM has similar feature called Branch Target Identification

authenticated pointers (1)

```
some_function:
    authentication_code <- MAC(
        secret key,
        return address
    )
    ... dangerous function code ...
    assert(authentication_code ==
        MAC(secret key, return address))
    jump to return address
```

authenticated pointers (2)

```
some_function:
    authentication_code <- MAC(
        secret key,
        stack pointer,
        return address
    )
    ... dangerous function code ...
    assert(authentication_code ==
        MAC(secret key, stack pointer, return address))
    jump to return address
```

authenticated pointers (3)

```
some_function:
  return address <- encode(
    secret key,
    stack pointer,
    return address
  )
  ... dangerous function code ...
  return address <- decode_or_crash(
    secret key,
    stack pointer,
    return address
  )
  jump to return address
```

authenticated pointers (4)

```
some_vtable[index] <- encrypt(  
  secret key,  
  label,  
  address of some of function  
)  
... dangerous code ...  
function pointer <- decrypt(  
  secret key,  
  label,  
  object->vtable[index]  
)  
call function pointer
```

ARM authenticated pointers

ARM64 implements this idea with:

secret key kept in a special register (hard to leak to attacker)

authentication code placed in upper pointer bits

- makes pointer temporarily invalid

- can't “accidentally” use authenticated pointer without verifying authentication code first

authenticated pointer layout

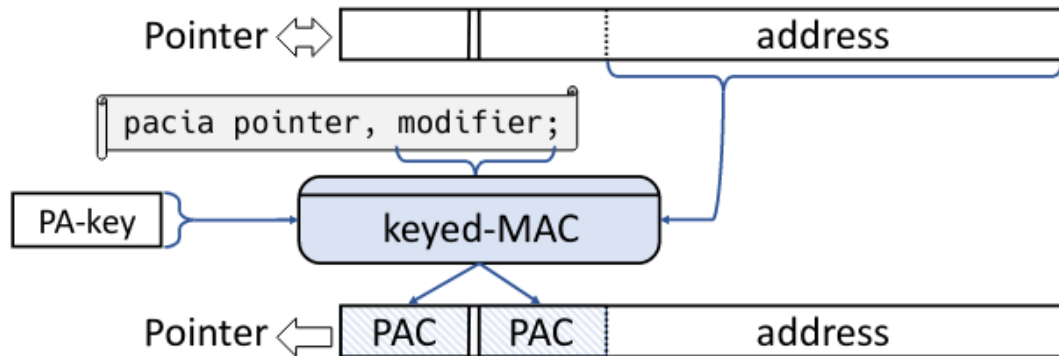


Figure 1: The PAC is created using key-specific PA instructions (`pacia`) and is a keyed MAC calculated over the pointer address and a modifier.

authentication keys

processes can have multiple authentication keys active

easy to use separate keys for

- return address pointers

- function pointers (in VTables, for example)

- any pointers to data

authentication keys are in special registers — need OS to read/set

also can “mix” in extra info like stack pointer

- similar to separate key for each stack address

different limits

CFI and pointer authentication both limit use of address

...but don't eliminate potential for abuse:

- can still call wrong function in CFI

- can still use wrong signed pointer

scope of labels/authentication keys+extra info important

pointer authentication use

commonly enabled on ARM64 for return addresses

otherwise:

on ARM64 OS X — used by OS X

two different compiler 'architectures': arm64, arm64e
can't use arm64e libraries in arm64 program or vice-versa

proposals to use on Linux in global offset table, etc.

GNU Linux linker option `-z pac-plt`; unclear if used 'for real'

macOS PAC for apple code

Item	Key	Salt
Function Return Address	IB	Storage address
Function Pointers	IA	0
Block Invocation Function	IA	Storage address
Objective-C Method Cache	IB	Storage address + Class + Selector
C++ V-Table Entries	IA	Storage address + Hash (mangled method name)
Computed Goto Label	IA	Hash (function name)
Kernel Thread State	GA	•
User Thread State Registers	IA	Storage address
C++ V-Table Pointers	DA	0

macOS PAC

‘new’ ‘preview’ architecture in computer

seems to be used internally by Apple

doesn't appear easily available/supported for others

incompatible with ‘old’ libraries

PAC bypasses

have been exploits in spite of PAC in iOS kernel

- only certain code pointers authenticated

 - too expensive to authenticate all pointers

- changing unsigned pointer values just before signed

 - get context switch to occur at just the right time

 - modify pointer value in saved context

- code that signs pointers they shouldn't

 - changing pointer without crashing if old pointer invalid

- 'gadgets' that allow brute-forcing MAC tag

 - not enough space in pointers for non-brute-forceable tag