Nostalgic memory-

Remembering all the wins and loses of memory corruptions

Shubham Dubey

DEEPSEC
IN-DEPTH SECURITY

$whoami

- Security Researcher @ Microsoft
- Expertise in low level security
- Follow @nixhacker

DEEPSEC

Agenda

- Basics of memory corruption
- Generation 1 mitigations
- Generation 2 mitigations
- Tools – Memory error detection
- Future of Memory corruptions
- Memory corruption matrix
- Conclusion

Categorizing Mitigations:

1

Gen 1 – (before 2010)

- Mostly discovered/added in early years when memory corruptions where at peak.
- Currently Stable state.

2

Gen 2 (after 2010)

- Covers the missing gaps of Generation 1 mitigations.
- Still been improved

Category 2

TOOLS

TECHNIQUES

Quick intro to memory corruption

```
[6984963.972864]  [<ffffffff81128160>] __perf_remove_from_context+0x60/0xd0
[6984963.973155]  [<ffffffff8112824b>] __perf_event_exit_context+0x7b/0xe0
[6984963.973427]  [<ffffffff811281d0>] ? __perf_remove_from_context+0xd0/0xd0
[6984964.009699]  [<ffffffff810bc4c7>] smp_call_function_single+0x147/0x160
[6984964.010030]  [<ffffffff811256bc>] perf_event_exit_cpu+0xbc/0x100
[6984964.010296]  [<ffffffff81126f77>] perf_reboot+0x27/0x50
[6984964.163576]  [<ffffffff816f665d>] notifier_call_chain+0x4d/0x70
[6984964.163871]  [<ffffffff81085938>] __blocking_notifier_call_chain+0x58/0x80
[6984964.164144]  [<ffffffff81085976>] blocking_notifier_call_chain+0x16/0x20
[6984964.164532]  [<ffffffff81072f7d>] kernel_restart_prepare+0x1d/0x40
[6984964.164799]  [<ffffffff81072fb6>] kernel_restart+0x16/0x60
[6984964.165055]  [<ffffffff810731d9>] sys_reboot+0x1b9/0x280
[6984964.260791]  [<ffffffff81140d30>] ? do_writepages+0x20/0x40
[6984964.261155]  [<ffffffff811b6ab2>] ? iput+0x32/0x50
[6984964.261613]  [<ffffffff811d6862>] ? iterate_bdevs+0x112/0x130
[6984964.261869]  [<ffffffff816f3435>] ? do_device_not_available+0x15/0x20
[6984964.262145]  [<ffffffff816faf5d>] system_call_fastpath+0x1a/0x1f
[6984964.262405] Code: 89 c7 48 89 d0 44 89 06 48 c1 e0 20 89 f9 48 09 c8 5d c3
66 90 55 89 f0 89 f9 48 89 e5 0f 30 31 c0 5d c3 66 90 55 89 f9 48 89 e5 <0f> 33
89 c7 48 89 d0 48 c1 e0 20 89 f9 48 09 c8 5d c3 0f 1f 84
[6984964.265746] RIP  [<ffffffff81045006>] native_read_pmc+0x6/0x20
[6984964.266078]  RSP <ffff88003a819b40>
[6984964.266405] ---[ end trace f2da148daa0c6e41 ]---
Segmentation fault
_
```

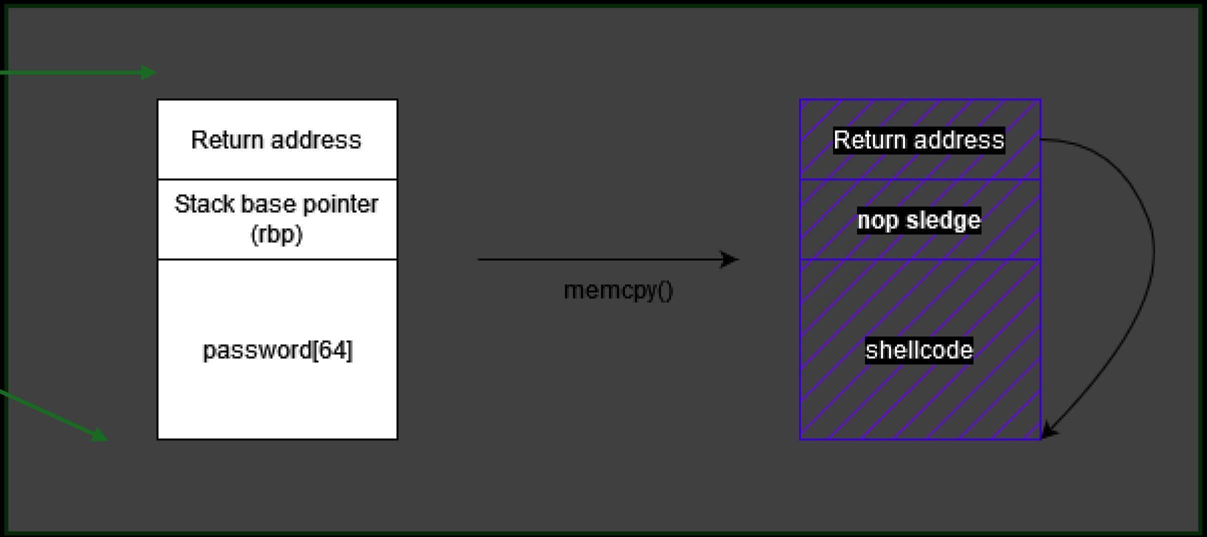
Intro to Memory corruption

```c
int admin_login()
{
        char password[64];
        strcpy(password, argv[1]);
        if (strcmp(password, "admin123#$") == 0)
        {
            printf("Sucessfully login\n");
            admin();

        }
        else
        {
            printf("Password doesn't match. Unable to login.\n");
            exit();
        }
        return 0;
}
```

DEEPSEC

Intro to Memory corruption

```c
int admin_login()
{
	char password[64];
	strcpy(password, argv[1]);
	if (strcmp(password, "admin123#$") == 0)
	{
		printf("Sucessfully login\n");
		admin();

	}
	else
	{
		printf("Password doesn't match. Unable to login.\n");
		exit();
	}
	return 0;
}
```



DEEPSEC

Other types of memory corruption

- Heap overflow
- Double free
- Indirect function calls modification
- OOB read/write
- NULL pointer dereference

When it all started

fingerd unix application which was exploited by Morris worm.

1988

1996

phrack edition 49 “Smashing the stack for fun and profit” in 1996

Gen 1 timeline

- BoundCheck — 1992
- StackGuard — 1997
- Stack Shield — 2000
- StackGhost — 2001
- Propolice — 2004
- NX stack — 2004
- ASLR — 2005

BoundCheckers [Tool][1992]

- Memory leaks detection suite released by NuMega Corp.
- Capable of detecting array and buffer overrun conditions.
- Currently part of DevPartner studio in Visual studio.

BoundCheckers – Capability and Working

- memory corruption problems caused by the following
  - Overrun allocated buffers
  - Continued access to memory after it has been deallocated
  - Deallocating a resource multiple times (e.g. double free)

- works by doing instrumentation to perform memory tracking and and API validation.

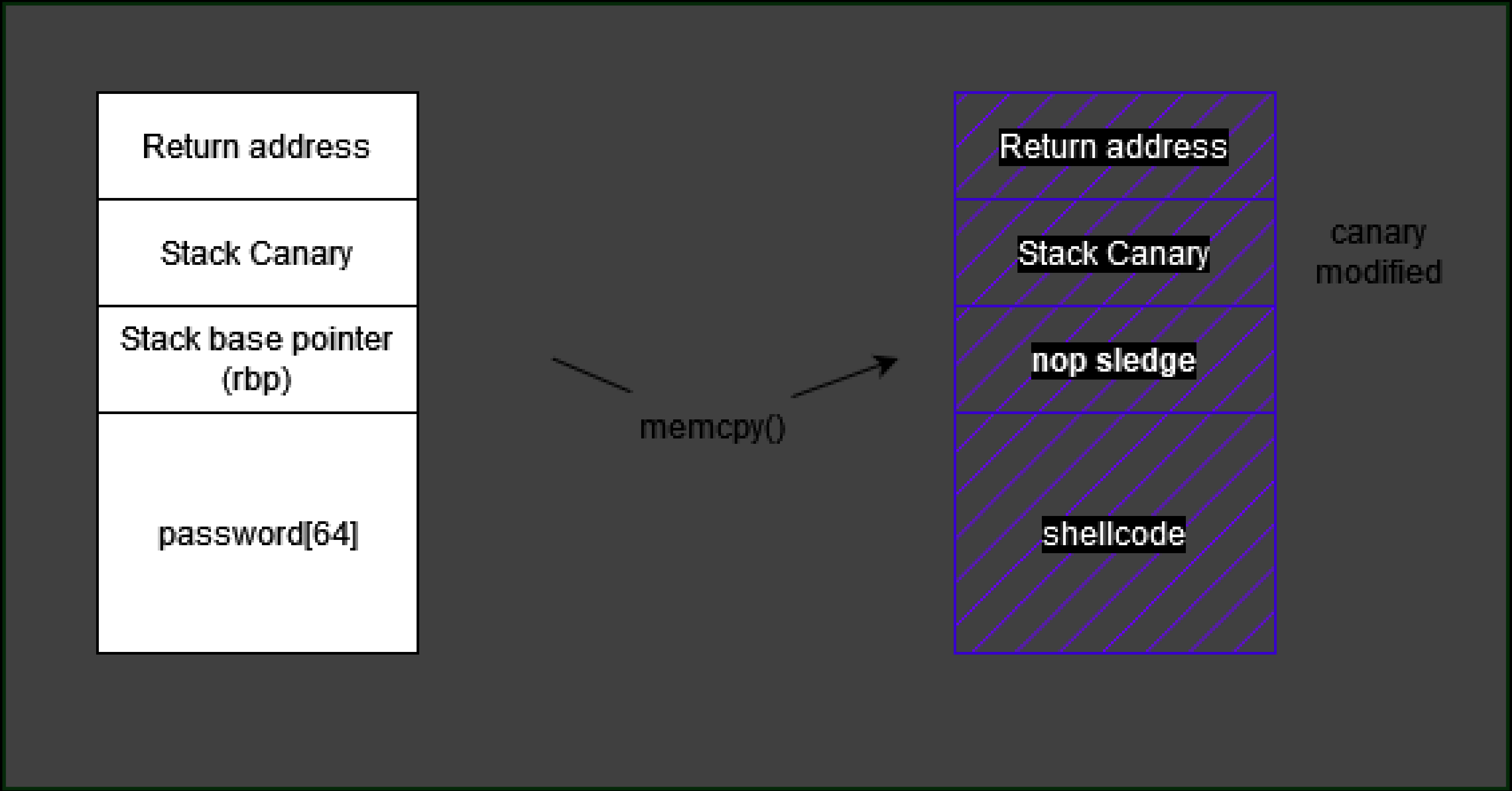
Limitation

- closed source nature caused less implementation and usage.
- Performance impact due to heavy instrumentation.
- Poor maintenance.

Stack Guard [1997][Technique]

- First major buffer overflow protection added to gcc 2.7 in year 1998 in Immunix distribution.
- it adds a random 8bytes data (called stack canaries) at the starting of function stack frame
- During the function return, it match if the canary is same or not.
- During exit if the canary is found to be modified, the program gets abort.

Stack Guard



DEEPSEC

Stack Guard – Linux implementation

```
; int __cdecl main(int argc, const char
public main
main proc near

var_60= qword ptr -60h
var_54= dword ptr -54h
dest= byte ptr -50h
var_8= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 60h
mov     [rbp+var_54], edi
mov     [rbp+var_60], rsi
mov     rax, fs:28h
mov     [rbp+var_8], rax
xor     eax, eax
mov     rax, [rbp+var_60]
add     rax, 8
```

Function prologue

```
loc_11CE:
mov     eax, 0
mov     rcx, [rbp+var_8]
sub     rcx, fs:28h
jz      short locret_11E7

call    ___stack_chk_fail

locret_11E7:
leave
retn
; } // starts at 1165
main endp
```

Function epilogue

Stack Guard – Linux implementation

- What all are protected (based on gcc flag):
  - -fstack-protector
  - -fstack-protector-strong
  - -fstack-protector-all
- Canaries on linux kernel
  - CONFIG_CC_STACKPROTECTOR
  - CONFIG_CC_STACKPROTECTOR_STRONG
  - -CONFIG_CC_STACKPROTECTOR_ALL

Stack Guard – Windows implementation

- introduced in Windows in year 2003 with visual studio support for /gs flag.

```
mov     rax, cs:__security_cookie
xor     rax, rbp
mov     [rbp+130h+var_18], rax
```

Function Prologue

```
mov     rcx, [rbp+130h+var_18]
xor     rcx, rbp        ; StackCookie
call    j___security_check_cookie
lea     rsp, [rbp+128h]
pop     rdi
pop     rbp
retn
```

Function Epilogue

- The call j__security_check_cookie will verify if rcx is set to 0 or not. If not than it will abort the program otherwise return.

Stack Guard – Windows kernel

```
push    rbx
sub     rsp, 70h
mov     rax, qword ptr [ntkrnlmp!__security_cookie (fffff80
xor     rax, rsp
mov     qword ptr [rsp+68h], rax
```

Function Prologue

```
mov     rcx, qword ptr [rsp+68h]
xor     rcx, rsp
call    ntkrnlmp!__security_check_cookie (fffff8047dc6b0d0)
add     rsp, 70h
pop     Operation (rbx)
ret
```

Function Epilogue

Limitation of stack guard

- Detect overflow but not prevent it (Can be major issue in Kernel architecture).
- Guessed by brute force in certain implementation.
- Not prevent modification of local variable.

Stack Shield [2000][Tool]

- Consist of shieldgcc and shieldg++ to compile c/c++ binary with stackshield protection.
- Two main feature
  - the Global Ret Stack (default)
  - the Ret Range Check.
- GRS save the return address in a separate memory space named retarray.
- RRC detect and stop attempts to return into addresses higher than that of the variable shielddatabase

Stack Shield - Implementation

```
function_prologue:
    pushl  %eax
    pushl  %edx

    movl   retptr,%eax     // retptr is where the clone is saved
    cmpl   %eax,rettop     // if retptr is higher than allowed
    jbe    .LSHIELDPROLOG  // just don't save the clone
    movl   8(%esp),%edx    // get return address from stack
    movl   %edx,(%eax)     // save it in global space
```

```
function_epilogue:
    leave                  // copies %ebp into %esp,
                           // and restores %ebp from stack
    pushl  %eax
    pushl  %edx

    addl   $-4,retptr      // allways decrement retptr
    movl   retptr,%eax
    cmpl   %eax,rettop     // is retptr in the reserved memory?
    jbe    .LSHIELDEPILOG  // if not, use return address from stack
    movl   (%eax),%edx
    movl   %edx,8(%esp)    // copy clone to stack
```

DEEPSEC

StackGhost [2001][Technique]

- Hadware enforced stack overflow protection for sparc architecture.
- uses register windows in SPARC architecture to make stack overflow exploitation harder.

StackGhost Implementation – Protect return address

01

Encoded return address: Return address goes through reversable transform and then saved in stack. During access, transform is recalculated before access is complete.

02

XOR cookie - XORing the cookie with return address before it is saved and xoring again after it popped off preserve the legitimate pointer but distort the attack.

03

Encrypted Stack Frame – Corrupted return can be detected by encrypting part of the stack frame when the window is written to the stack and decrypting it during retrieval.

04

Return address stack: Having a return address stack as FIFO which is based on register windows concept.

StackGhost Limitation

- Randomness of XOR cookie is low that can be easily predicted.
- Techniques based on detection but not protection.

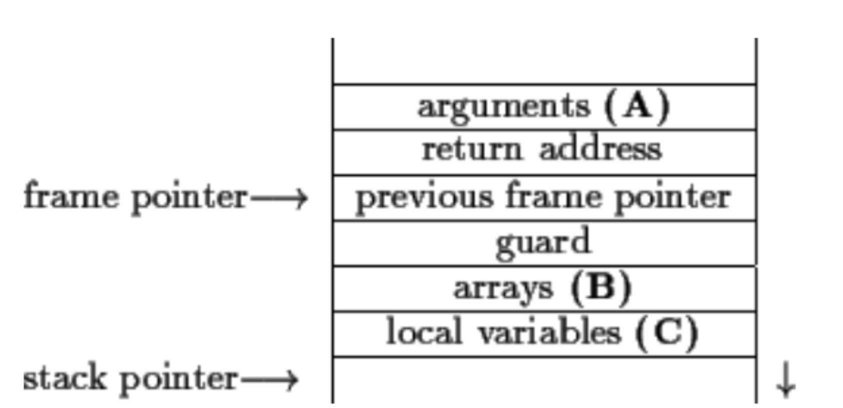
ProPolice [2004][Technique]

- Patches added by IBM in gcc to improve stack guard protection.
- It detects modification of local variable that stack guard doesn't support.

DEEPSEC

ProPolice Implementation

- Patch include the reordering of local variables to place buffers after pointers to avoid the corruption of pointers.



- For protecting function pointer, . It makes a new local variable, copying the argument `func1‘’ to it, and changing the reference to `func1‘’ to use the new local variable.

```
void bar( void (*func1)() )
{
        void (*func2)();
        char buf[128];
        .......
        strcpy (buf, getenv ("HOME"));
        (*func1)(); (*func2)();
}
```

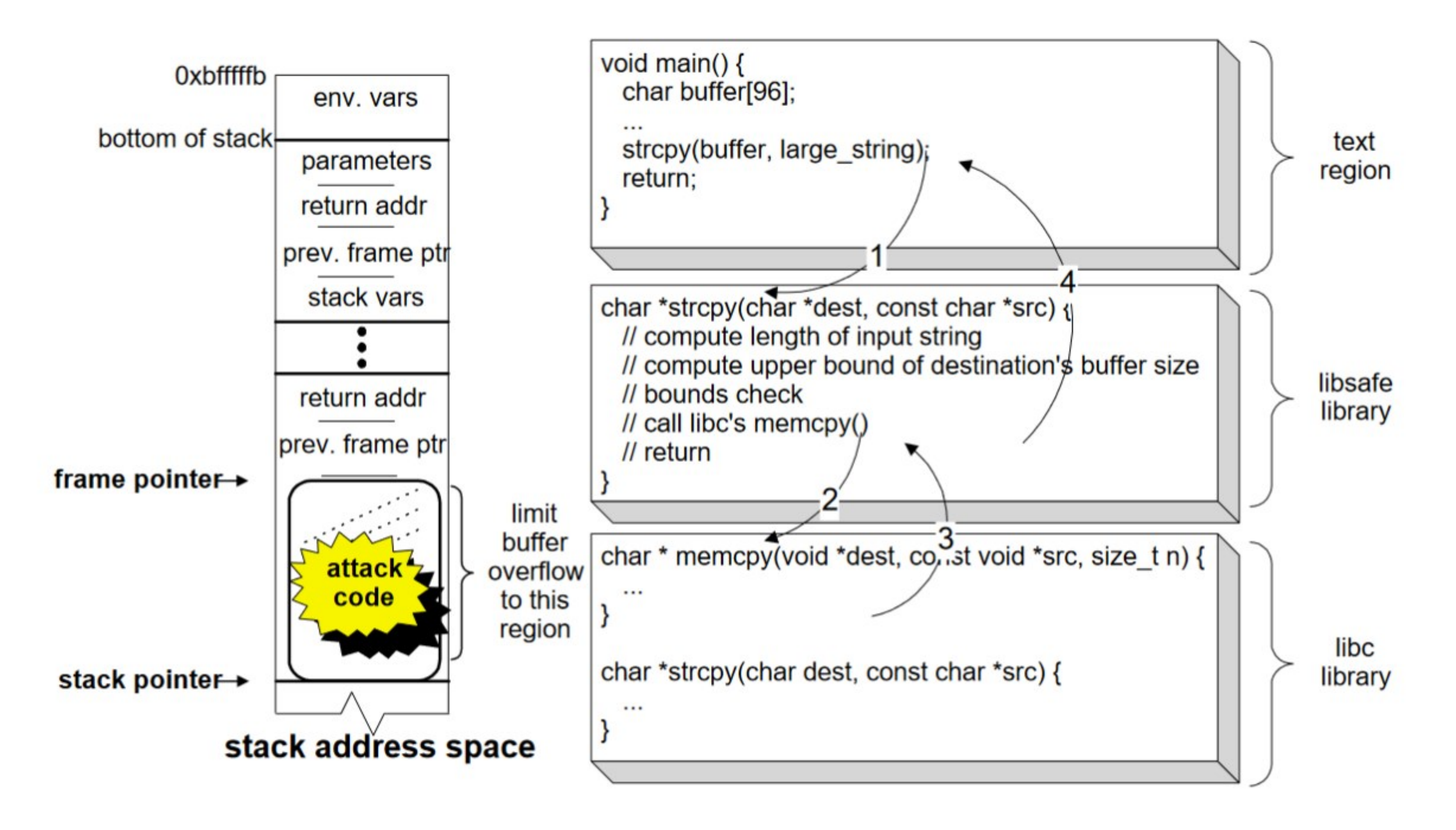
DEEPSEC

Libsafe and Libverify [2000][Tool]

- Used by loading precompiled dynamic library with any process.
- The libsafe intercepts all calls to library functions that are known to be vulnerable from the loaded library.
- The libverify library relies on verification of function's return address before it is used.
- It inject the verification code at the start of process execution via rewriting the binary after it is written on the memory.

HONORABLE MENTION

Libsafe and Libverify

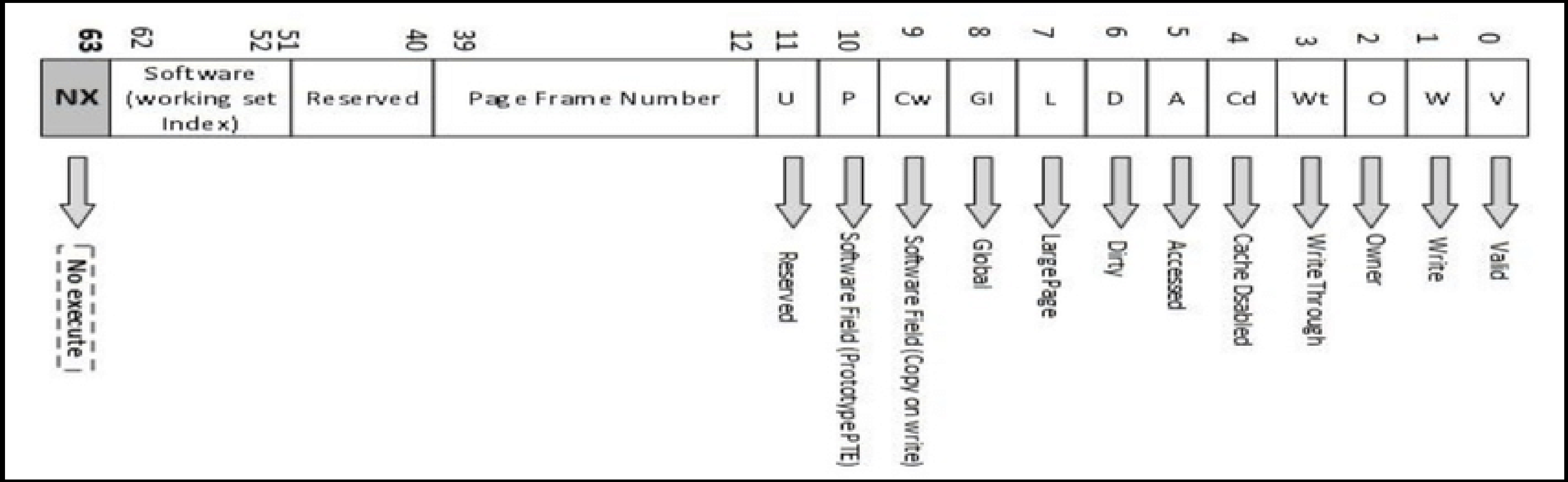


Non executable stack [2004] [Technique]

- Software mitigation added in 1998. Hardware support introduced in 2001 by Intel/AMD.
- Merged in gcc in 2004.
- Called NX stack (Non executable stack) in linux and DEP (Data execution prevention) in Windows.
- Focus on preventing exploitation of memory corruption by making stack non executable.

Non executable stack - Implementation

- Page table entries:



Non – executable stack limitations

- only protect case where attacker try to redirect the execution to process stack.
  - Bypassed by ROP.

ASLR [2005][Technique]

- Address space layout randomization
- first introduced in PAX project in year 2001. In an operating system introduced in OpenBSD in 2003, followed by linux in 2005 and Windows vista in 2007.
- randomize the address of most/all sections of a process memory so that attacker cannot predict gadget or shellcode address.

ASLR

```
shubham@MININT-1T2PIDD:~/memory_protection$ ./a.out
Address of variable in stack is 0x7fffc3c31f44
Address of variable in heap is 0x55629504e2a0
Address of variable in rdata is 0x556293764008
Address of variable in bss is 0x55629376603c
Address of variable in text is 0x556293763145
shubham@MININT-1T2PIDD:~/memory_protection$ ./a.out
Address of variable in stack is 0x7ffdb2f31a24
Address of variable in heap is 0x560c588042a0
Address of variable in rdata is 0x560c56aea008
Address of variable in bss is 0x560c56aec03c
Address of variable in text is 0x560c56ae9145
shubham@MININT-1T2PIDD:~/memory_protection$ ./a.out
Address of variable in stack is 0x7ffe2005a5c4
Address of variable in heap is 0x557fa68e72a0
Address of variable in rdata is 0x557fa5258008
Address of variable in bss is 0x557fa525a03c
Address of variable in text is 0x557fa5257145
shubham@MININT-1T2PIDD:~/memory_protection$ ./a.out
Address of variable in stack is 0x7ffe25223104
Address of variable in heap is 0x5580eaaae2a0
Address of variable in rdata is 0x5580e9c05008
Address of variable in bss is 0x5580e9c0703c
Address of variable in text is 0x5580e9c04145
shubham@MININT-1T2PIDD:~/memory_protection$ ./a.out
Address of variable in stack is 0x7fff40b10764
Address of variable in heap is 0x55d0a2fc42a0
Address of variable in rdata is 0x55d0a2a2c008
Address of variable in bss is 0x55d0a2a2e03c
Address of variable in text is 0x55d0a2a2b145
```

ASLR limitations

- Address prediction due to low entropy (specially kernel).
- Having module loaded with no ASLR support.
- Address leaks
- Heap spraying
- Advance attack like Side channel

Generation 2 mitigation

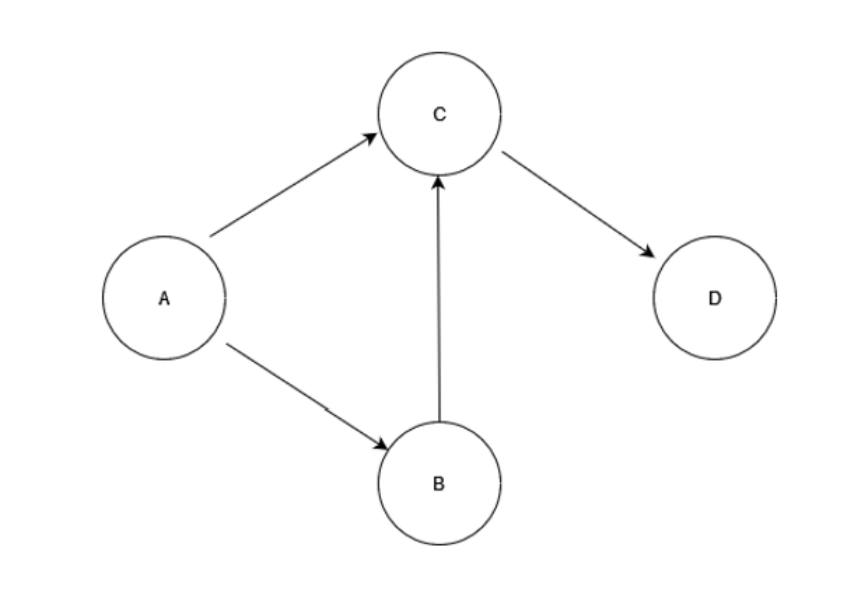
Overcome following limitation of gen 1 mitigations:

No heap based mitigation	No mitigation for indirect calls	Existence of ROP chaining

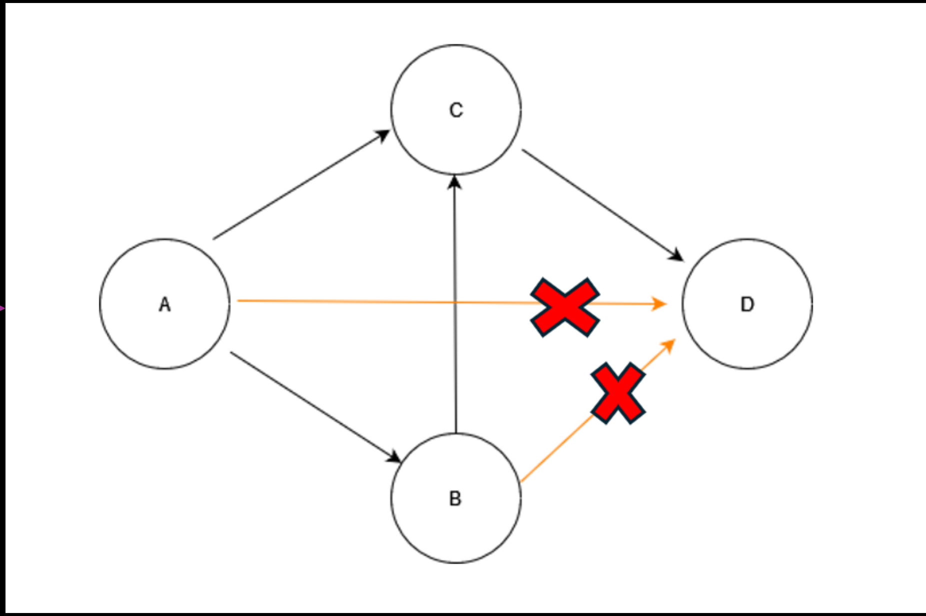
Control flow integrity

- CFI mitigate against exploitation of memory corruption by maintaining control flow by restricting illegal branch.
- For all generation 1 mitigation in place, there are cases where attacker cause arbitrary code execution using ROP chaining.

Control flow integrity



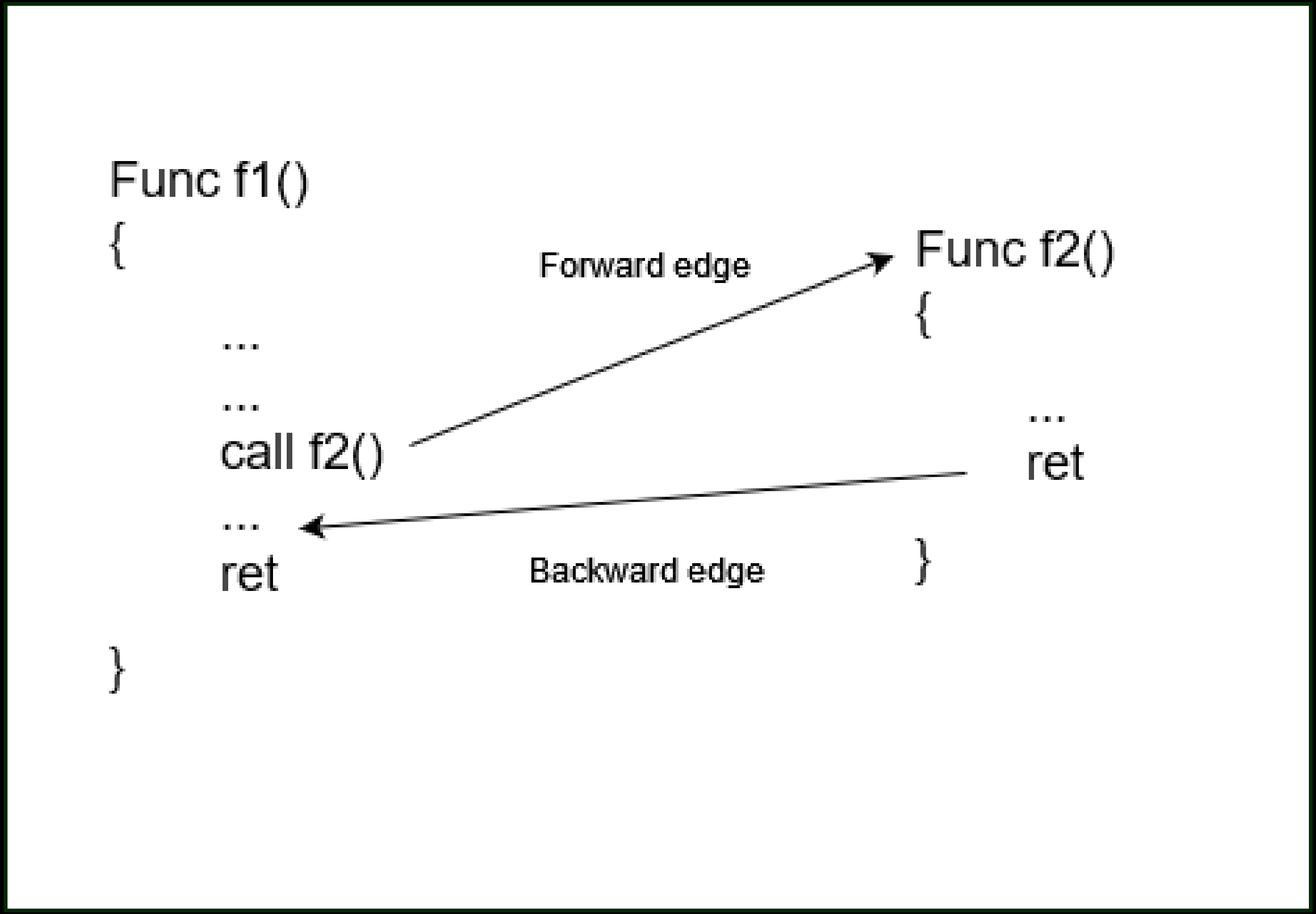
CFI in action



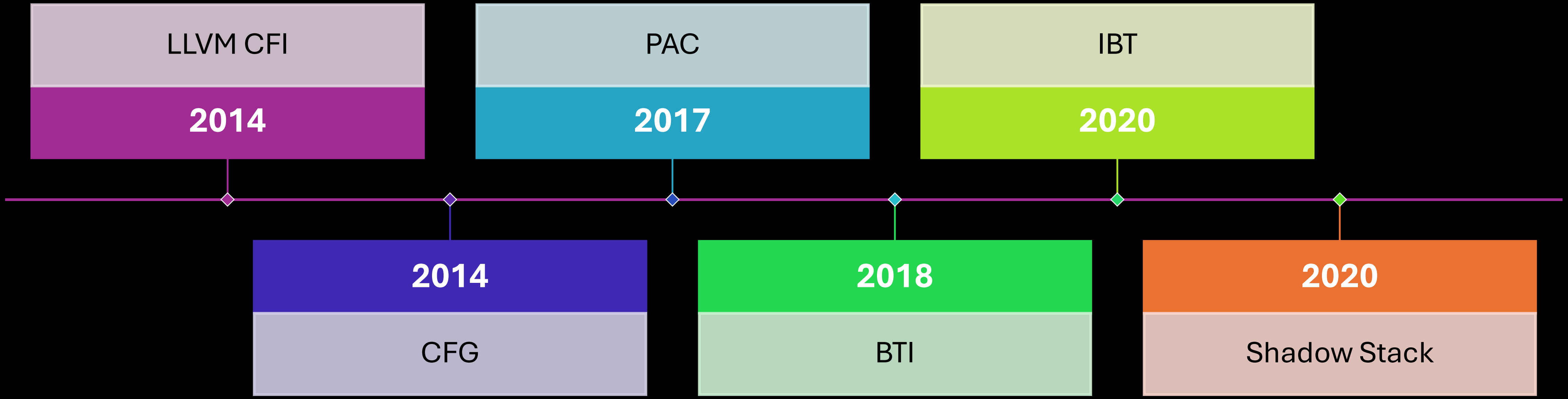
Types of CFI

Forward Edge Integrity

Backward Edge Integrity

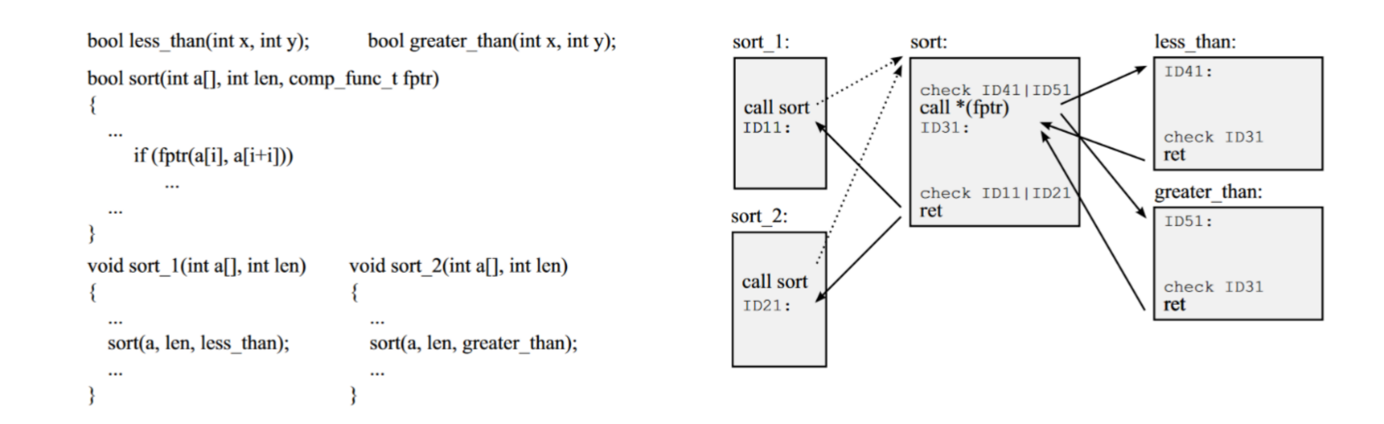


CFI timeline



Initial CFI implementation [2005] [Technique]

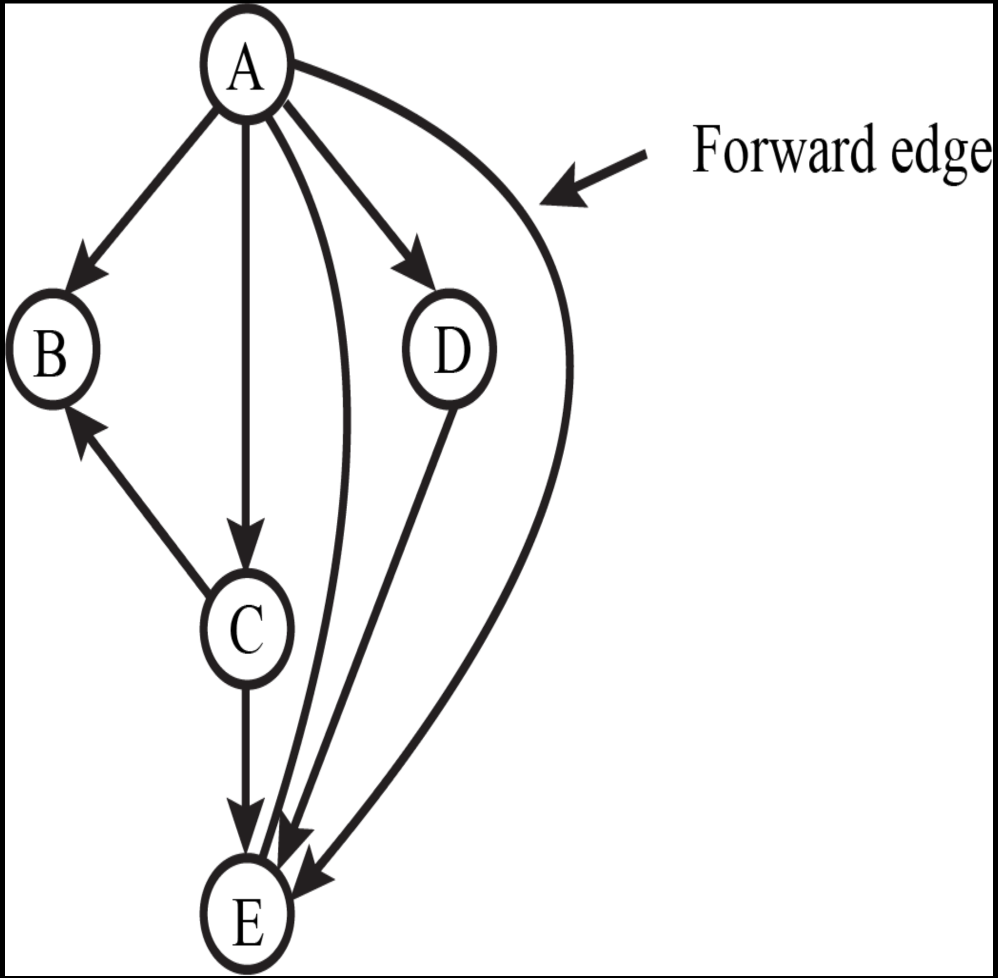
- CCFIR and bin-CFI.
- UID assigned to each valid target.
- Checks are inserted for indirect calls to ensure valid target are reached.



Initial CFI implementation Limitation

- Just Proof of concept. Not implemented at major compilers.
- Performance impact due to added checks and tags on each function calls.

Forward edge Integrity



LLVM CFI [2014][Technique]

- Aims for protecting heap and indirect calls from getting exploited.

- Contain two different methods:

VTV
- Virtual table verification

IFCC
- Indirect function call check

VTV – Virtual table

- Address of virtual functions for each function is present in Virtual table.
- When Tiger object created, first value in heap buffer is virtual pointer.

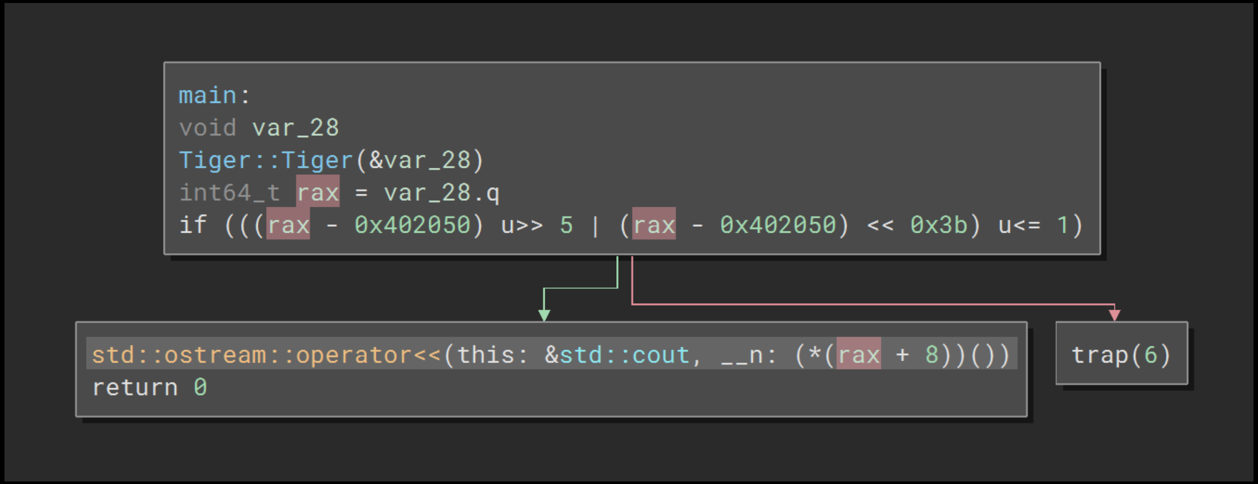
virtual pointer
weight
height
animal_name[]

Tiger heap chunk structure

```
class Animal // base class
{
    public:
        int weight;
        virtual int getWeight() { return 12;};
        virtual int getMass() { return 120;};
};

// Obviously, Tiger derives from the Animal class
class Tiger: public Animal {
    public:
        int weight;
        int height;
        char animal_name[64];
        int getWeight() {return weight;};
        int getMass() { return height;};
        int getname() {return animal_name;};
};

int main()
{
        Tiger t1;
        /* below, an Animal object pointer is set to point
           to an object of the derived Tiger class   */
        Animal *a1 = &t1;
```

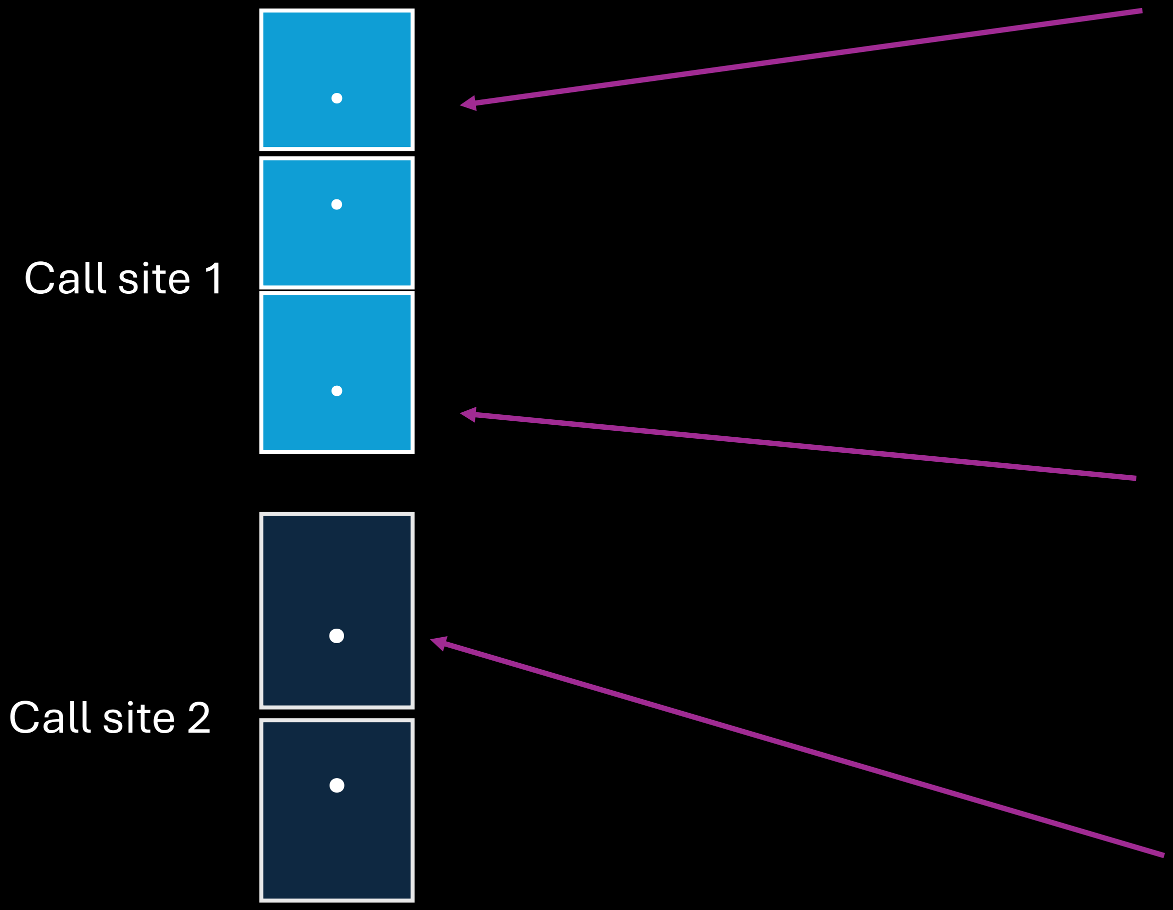
VTV - Working

- Can be used by passing following flag *-fsanitize=cfi-vcall* with clang++.



- Before using virtual function (rax+8), it is checked if the target is in range of valid call site.
- Valid call sites are added during IR phase based on object signature.

VTV –valid call site




```
Class human
{
    int height;
    int age;
    int get_age();
}

Class animal
{
    int legs;
    int weight;
    int get_weight();
}
Class car
{
    int weels;
    char brand[64];
    bool is_new();
}
```

DEEPSEC

IFCC: Indirect Function-Call Checks

- Protects integrity of indirect function calls.
- Generates jump tables for indirect-call targets.
- On indirect call site, instrumented code is added to verify if target points to correct jump table entry.

IFCC: Indirect Function-Call Checks

- Can be used by passing *-fsanitize=cfi-icall* to clang.

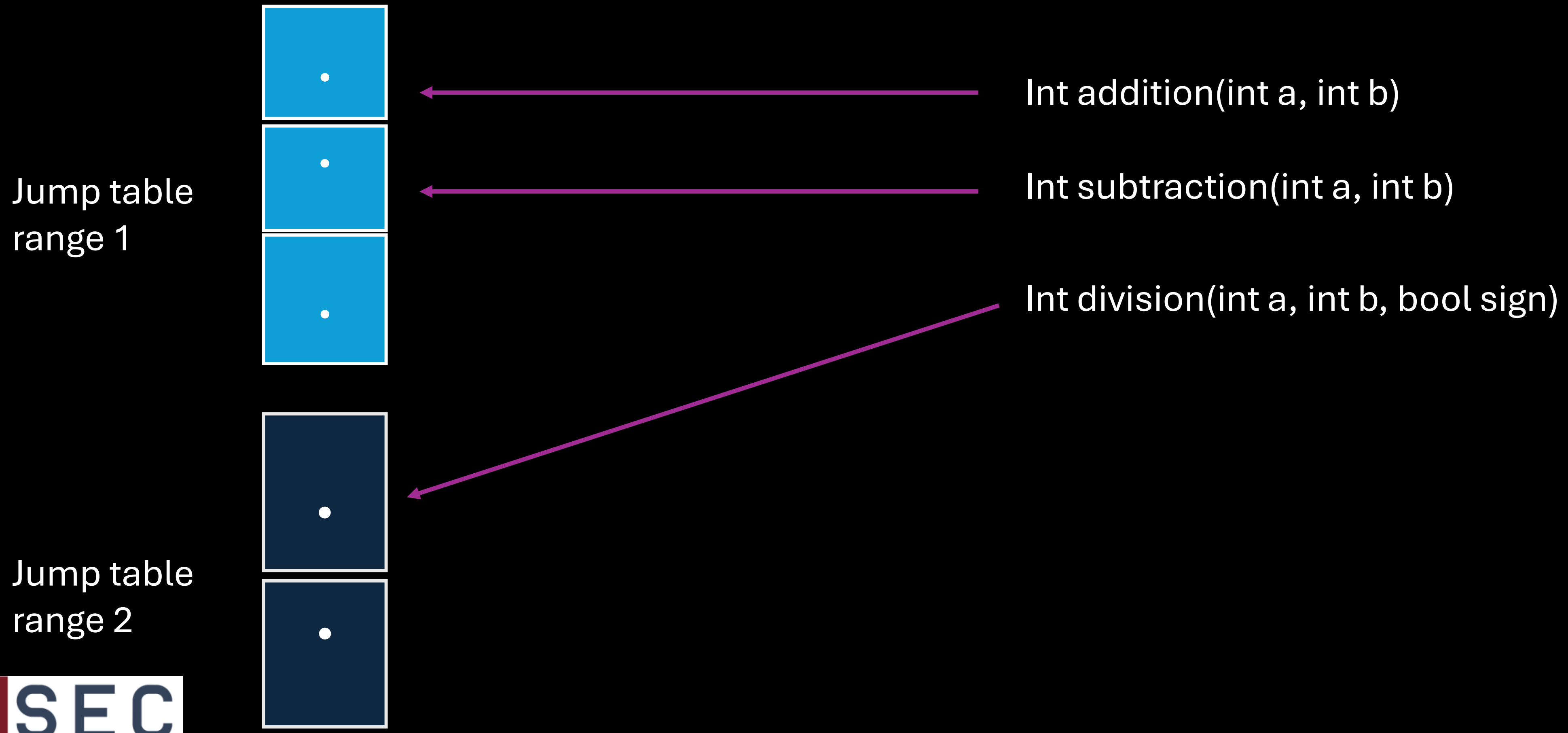
```c
int add(int a, int b) {
    return a + b;
}

int perform_operation() {
    // Declare a function pointer that points to
        taking two int arguments and returning in
    int (*operation)(int, int);
    char data_buffer[64];

    // Let's perform addition
    operation = add;
    int result = operation(10, 5);
    printf("Result of addition: %d\n", result);
```

```asm
; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+var_14], 0
mov     rax, offset add
mov     [rbp+operation], rax ; copying the address of add to oepration
mov     rax, [rbp+operation]
mov     rcx, offset add ; getting the address of add from jmp table
cmp     rax, rcx        ; comparing if the address match before the call
jz      short loc_401162

ud2

loc_401162:
mov     edi, 0Ah
mov     esi, 5
call    rax
mov     [rbp+var_4], eax
mov     esi, [rbp+var_4]
mov     rdi, offset format ; "Result of addition: %d\n"
mov     al, 0
call    _printf
xor     eax, eax
add     rsp, 20h
pop     rbp
retn
; } // starts at 401130
```

DEEPSEC

IFCC – Jump table generation

- Jump table ranges are generated based on function parameters



Clang CFI limitation

- Performance penalty – upto 20% for VTV, upto 4% in IFCC
- Not protect against certain Code reuse attack
  - COOP – At high level, it rely on finding protected targets in the application binary which can legitimately called and doesn't cause CFI violation.
  - Certain call sites covers more than 50% of function coverage – void foo(void)

kCFI – Fine gain CFI for linux kernel

- Limitations of CLANG CFI (IFCC)
  - Performance bottleneck due to jump table based CFI implementation
  - huge number of kernel function with similar prototype like void foo(void)
  - Support for self-modifying code and LKMs
  - Support for inline assemble code

kCFI – Fine gain CFI for linux kernel

- kCFI use tag based insertion. Tags are added using long nops.

(a) Prologue(s) instrumentation (tag).

```
...
<func>:
nopl    0xbcbee9
...
```

(b) Indirect call site(s) instrumentation (guard).

```
...
cmpl    $0xbcbee9,0x4(%rax)
je      <7>
push    %rax
callq   <kcfi_vhndl>
pop     %rax
callq   *%rax
nopl    0x138395f
...
```

kCFI – Fine gain CFI for linux kernel

- kCFI uses call graph detaching to reduce similar call sites.

```
<A>:
call b
tag 0xdeadbeef

<Z>:
if(something) ptr = &B
else ptr = &C
call *ptr
tag 0xdeadbeef

<B>:                  <C>:
check 0xdeadbeef      check 0xdeadbeef
ret                   ret
```

→

```
<A>:
call b_clone
tag 0xdeadc0de

<Z>:
if(something) ptr = &B
else ptr = &C
call *ptr
tag 0xdeadbeef

<B>:                  <C>:
check 0xdeadbeef      check 0xdeadbeef
ret                   ret

<B_clone>:
check 0xdeadc0de
ret
```

kCFI – Fine gain CFI for linux kernel

- By employing tag-based assertions, kCFI supports self-modifying code and LKMs.
- kCFI support inline assembly by rewriting of the assembly sources using information extracted during code and binary analysis.

Control flow guard [2014][Technique]

- Used by passing */cfguard* flag through msvc compiler (visual studio compiler) .
- Adds new data directory “Load Configuration” for storing CFG configurations.
- Functions that are valid indirect call targets are listed in the *GuardCFFunctionTable*

CFG Internals

- Windows perform following task for CFI:
  - Instrument around all indirect call with *_guard_check_icall* check.
  - Mapping CFG bitmap in process memory during Process initialization
- NT loader generate CFG bitmap storing all the valid targets address from the CFG whitelist in the module.
- __guard_dispatch_icall_fptr calls *ntdll!LdrpValidateUserCallTarget* which during execution verify the call to be valid using CFG Bitmap.

CFG internals

- CFG Bitmap working: Let's assume address target addr: 0x00b01030

- Encircled blue(3 bytes): used to find offset in CFGBitmap.
- 1: valid address 0:invalid address.
- Encircled red: used to find if the address is 0x10 aligned or not.

CFG internals

CFG - Limitations

- Require ASLR and guard functions to be aligned.
- Unsupported 3rd party module presence.
- Not supported in JIT code.

Hardware enforced forward edge Integrity

Intel and AMD has introduced hardware supported Control flow integrity to overcome software CFI performance impact.

IBT – Indirect branch tracking	BTI – Branch target identification

BTI [2018][Technique]

- Added in ARM v8.5, goal is to protect indirect jump to reach unintended location.
- When enabled, the first instruction encountered after an indirect jump must be a special BTI instruction.
- type of branch is store in PSTATE.BTYPE bits.

BTI - Internal

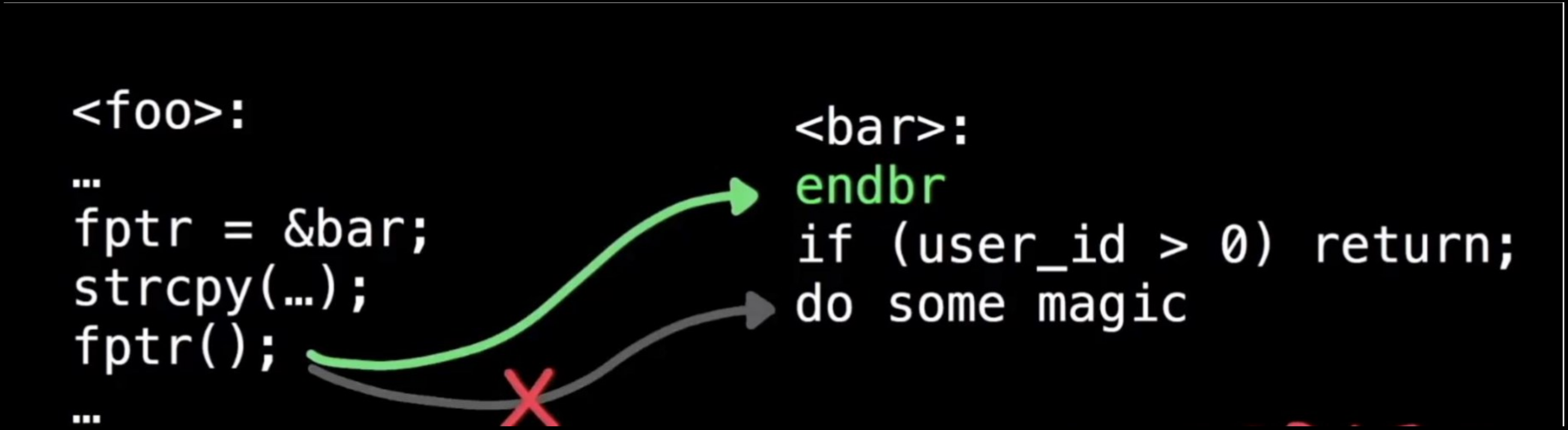
- adding -*mbranch-protection=bti* in gcc
- There are 3 variants of the BTI instruction:
  - c -Branch Target Identification for function calls
  - j - Branch Target Identification for jumps
  - jc - Branch Target Identification for function calls or jumps.

```
000000000000086c <add>:
 86c:   d503245f        bti     c
 870:   d10043ff        sub     sp, sp, #0x10
 874:   b9000fe0        str     w0, [sp, #12]
 878:   b9000be1        str     w1, [sp, #8]
 87c:   b9400fe1        ldr     w1, [sp, #12]
 880:   b9400be0        ldr     w0, [sp, #8]
 884:   0b000020        add     w0, w1, w0
 888:   910043ff        add     sp, sp, #0x10
 88c:   d65f03c0        ret

0000000000000890 <subtract>:
 890:   d503245f        bti     c
 894:   d10043ff        sub     sp, sp, #0x10
 898:   b9000fe0        str     w0, [sp, #12]
 89c:   b9000be1        str     w1, [sp, #8]
 8a0:   b9400fe1        ldr     w1, [sp, #12]
 8a4:   b9400be0        ldr     w0, [sp, #8]
 8a8:   4b000020        sub     w0, w1, w0
 8ac:   910043ff        add     sp, sp, #0x10
 8b0:   d65f03c0        ret
 8b4:   d503201f        nop
 8b8:   d503201f        nop
```

IBT [2020][Technique]

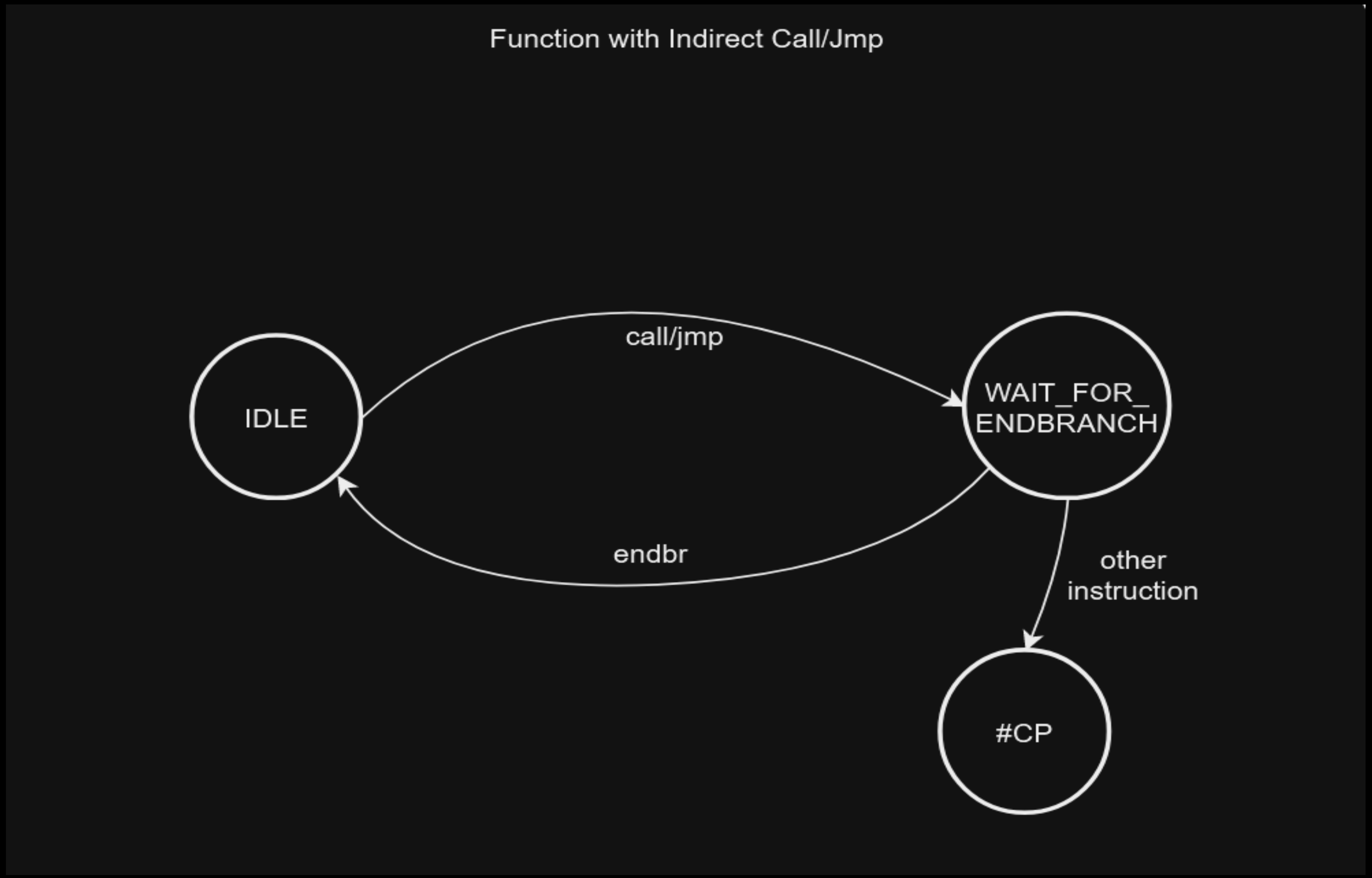
- Added as part of Intel CET in tigerlake processors.
- When enabled, the CPU will ensure that every indirect branch lands on a special instruction (endbr32 or endbr64).

IBT internals



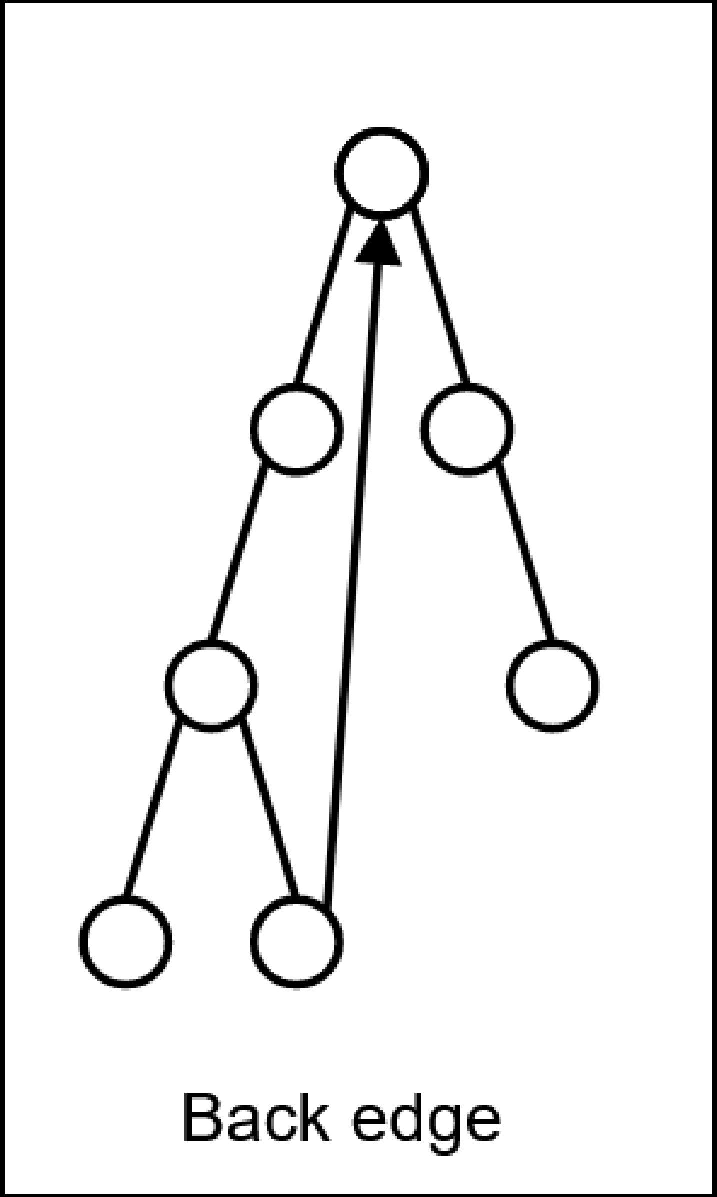
DEEPSEC

IBT working



DEEPSEC

Backward Edge Integrity



Backward edge mitigations

SHADOW STACK

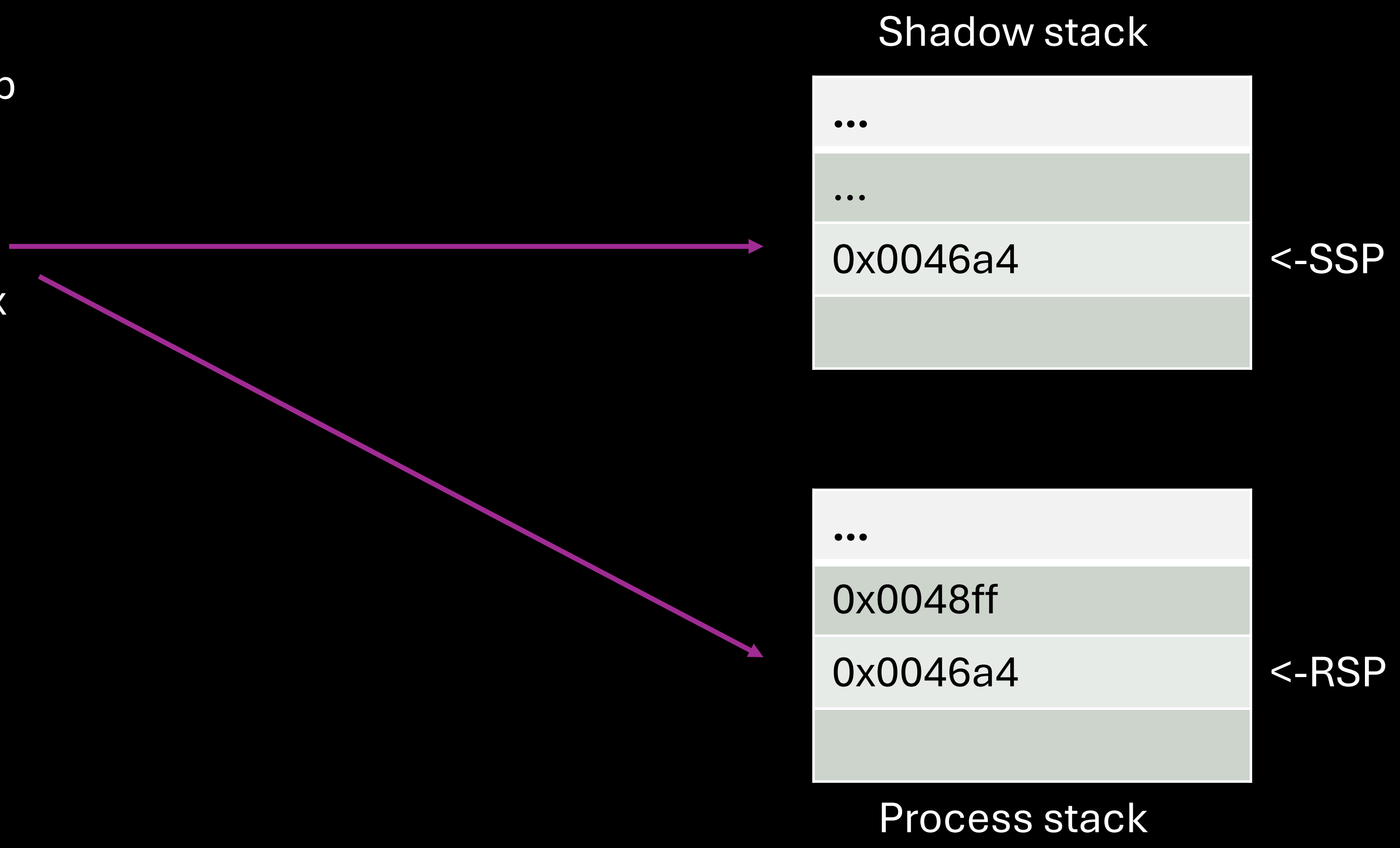
PAC

Shadow stack [2020][Technique]

- Added in intel TigerLake, use to address backward edge violation.
- replicates the return addresses that are pushed by the *CALL* instruction.
- during *ret* stack and shadow stack value is matched, generates INT #21 (Control Flow Protection Fault) in case of mismatch.
- protected from tamper through the page table protection.

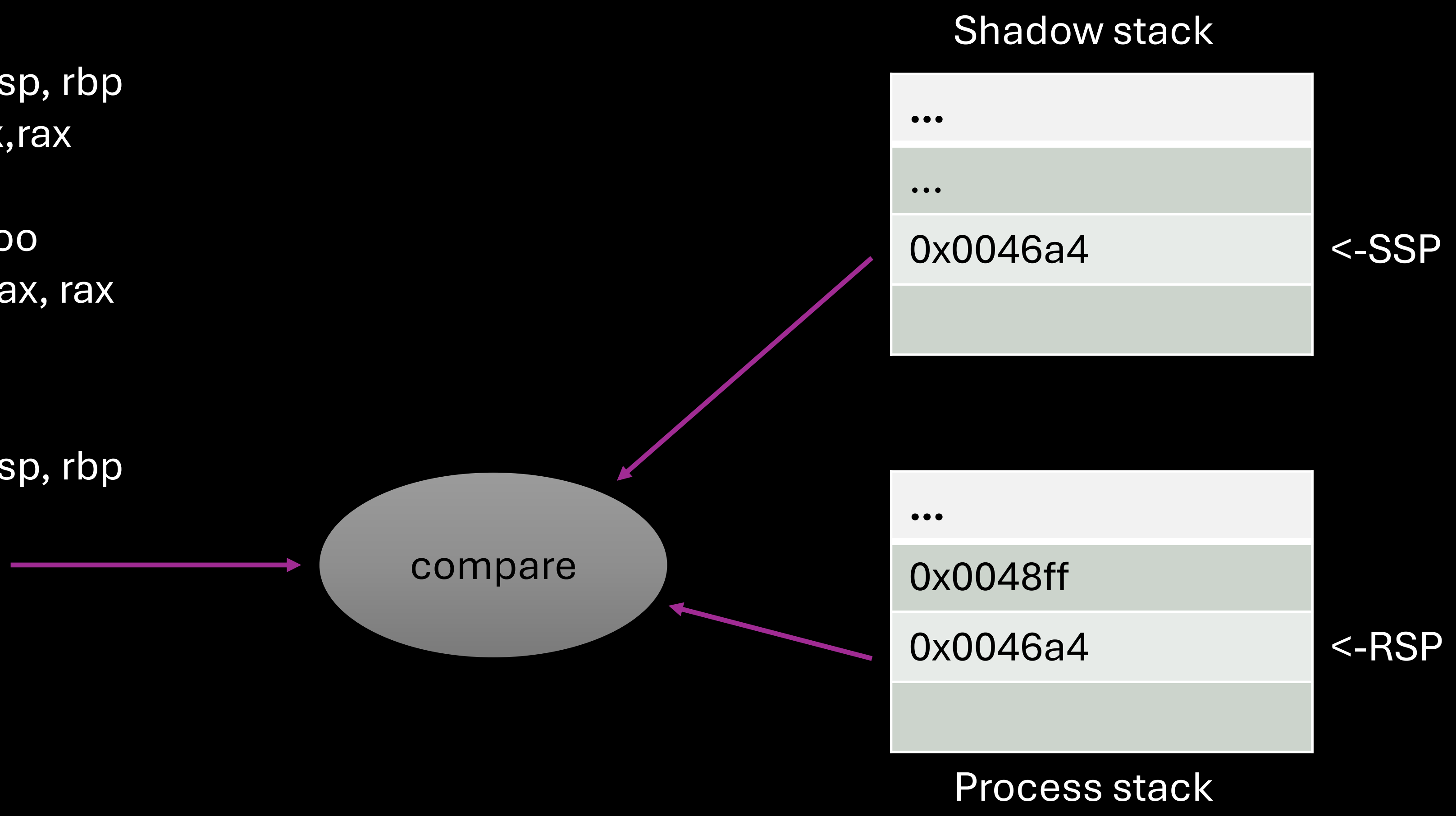
Shadow stack Implementation

```
        main()
        0x0044aa   mov rsp, rbp
        0x0044af   xor rax,rax
        ...
rip->   0x0046a0   call foo()
        0x0046a4   test rax, rax
```



Shadow stack Implementation

```
main()
0x0044aa   mov rsp, rbp
0x0044af   xor rax,rax
...
0x0046a0    call foo
0x0046a4    test rax, rax

foo()
0x0064aa   mov rsp, rbp
...
rip-> 0x0046a0    ret
```



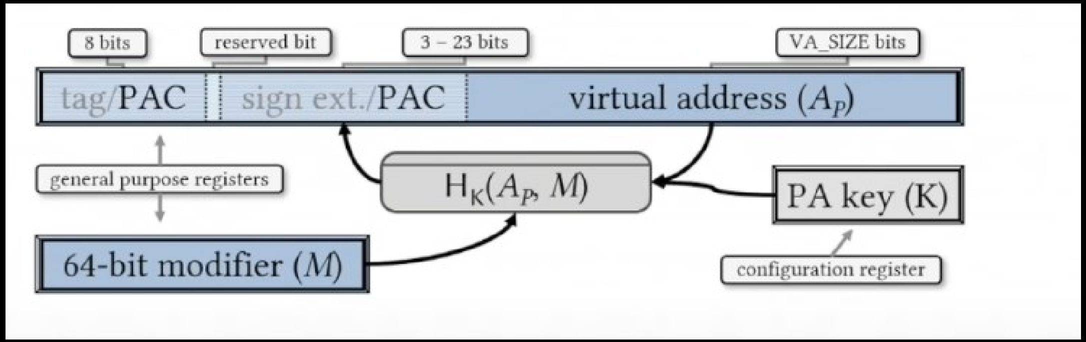
DEEPSEC

PAC [2017][Technique]

- Pointer Authentication Code
- ARM hardware feature. first added in Linux(Android) kernel in 2018.
- Ensure pointer in memory remains unchanged.
  - return address pointer.
  - data pointers

PAC internal

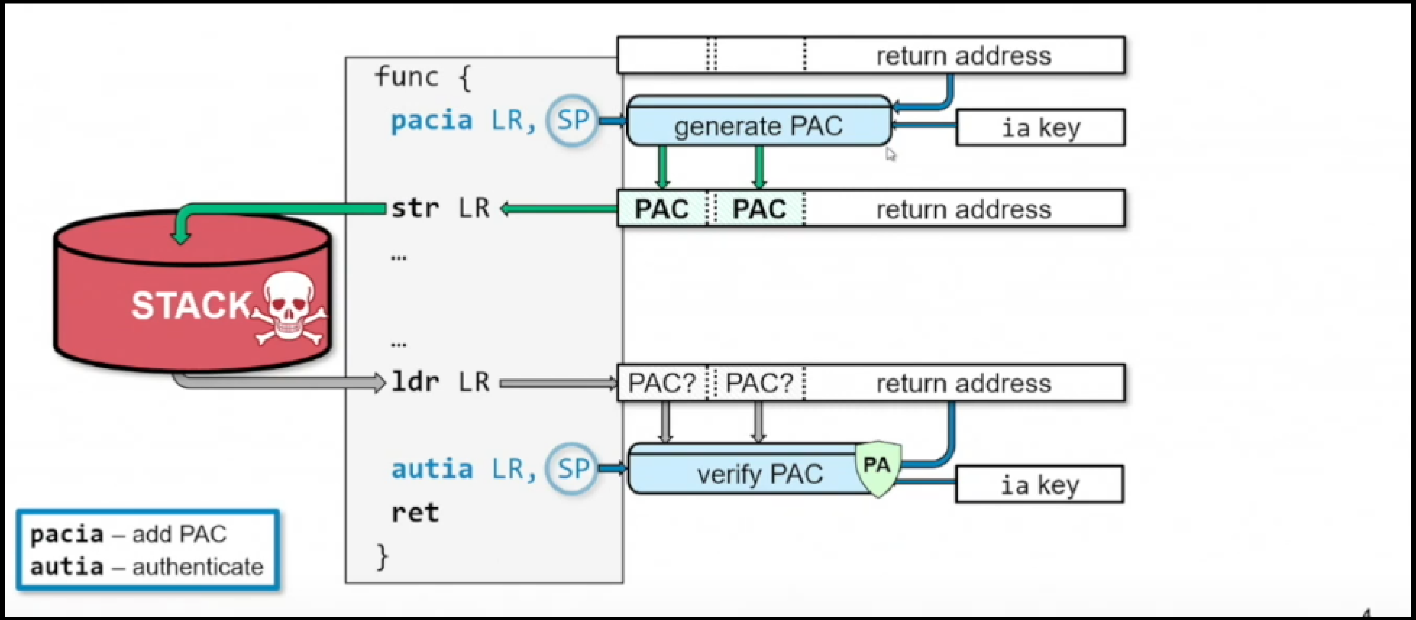
- Adds pointer authentication code to unused bits of pointer.



- PA key are protected by hardware. Modifier is created when pointer is used.
- Can be used by flag *–msign-return-address* in gcc and clang.

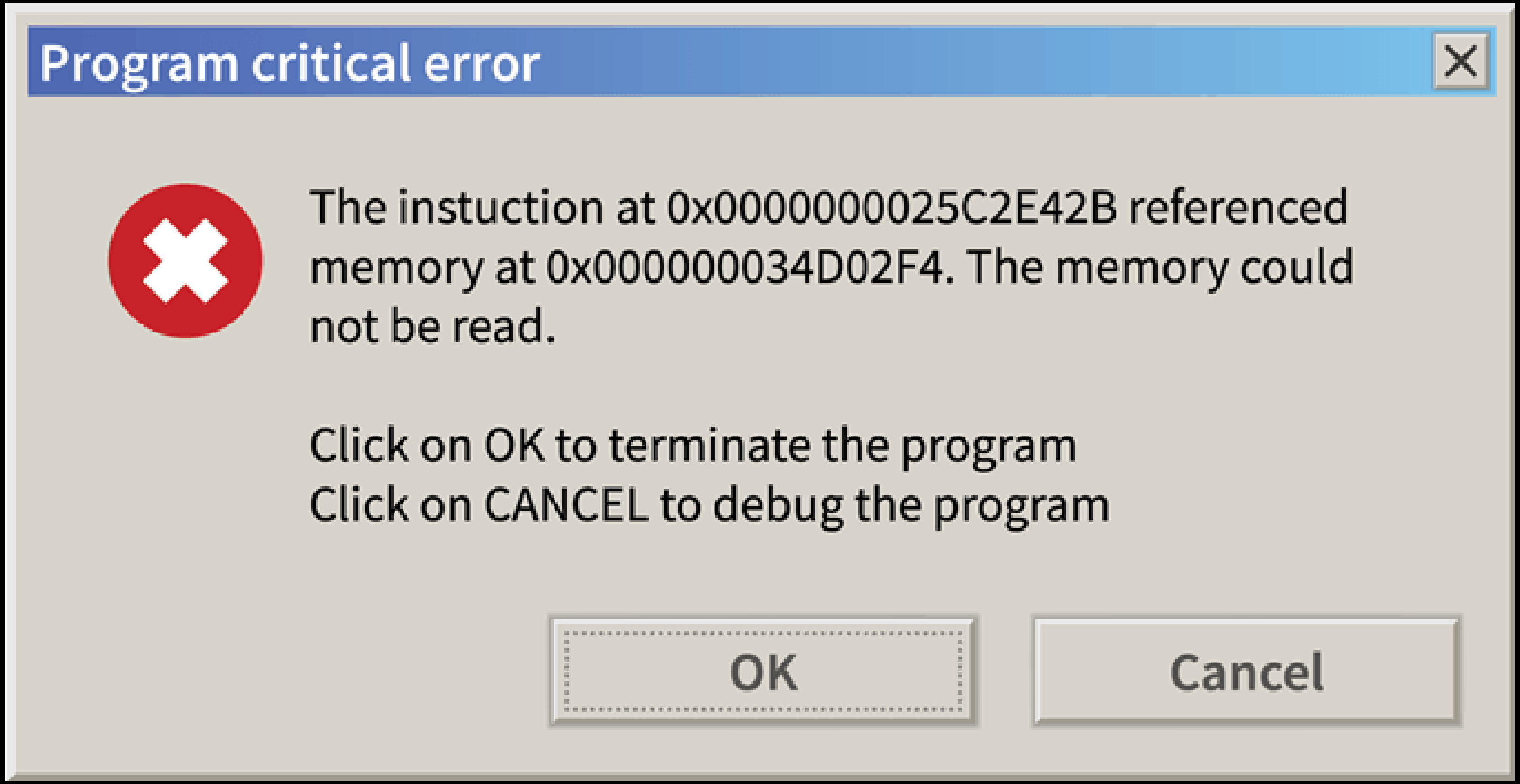
DEEPSEC

PAC internal



Source: https://www.youtube.com/watch?v=UD1KKHyPnZ4

Memory Error detection tools



Sanitizers for Compiler [2012+][Tools]

- Added as part of effort to detect memory corruption in debug environment before sending to production.
- Added in compiler like gcc, clang and msvc as tool.
- Usually rely on heavy instrumentation, hence impact performance.

List of known sanitizers

- ASAN (Address sanitizer)
  - Use after free (dangling pointer dereference)
  - Heap buffer overflow
  - Stack buffer overflow
  - Global buffer overflow
  - Use after return
  - Initialization order bugs
  - Memory leaks
- MSAN (Memory sanitizer)
  - Uninitialized memory

List of known sanitizers

- UBSAN (Address sanitizer)
  - Array subscript out of bounds
  - Bitwise shifts that are out of bounds for their data type
  - Dereferencing misaligned or null pointers
  - Signed integer overflow
  - Conversion to, from, or between floating-point types causing overflow
- Valgrind (Memcheck)

DEEPSEC

Memory error detection tools working

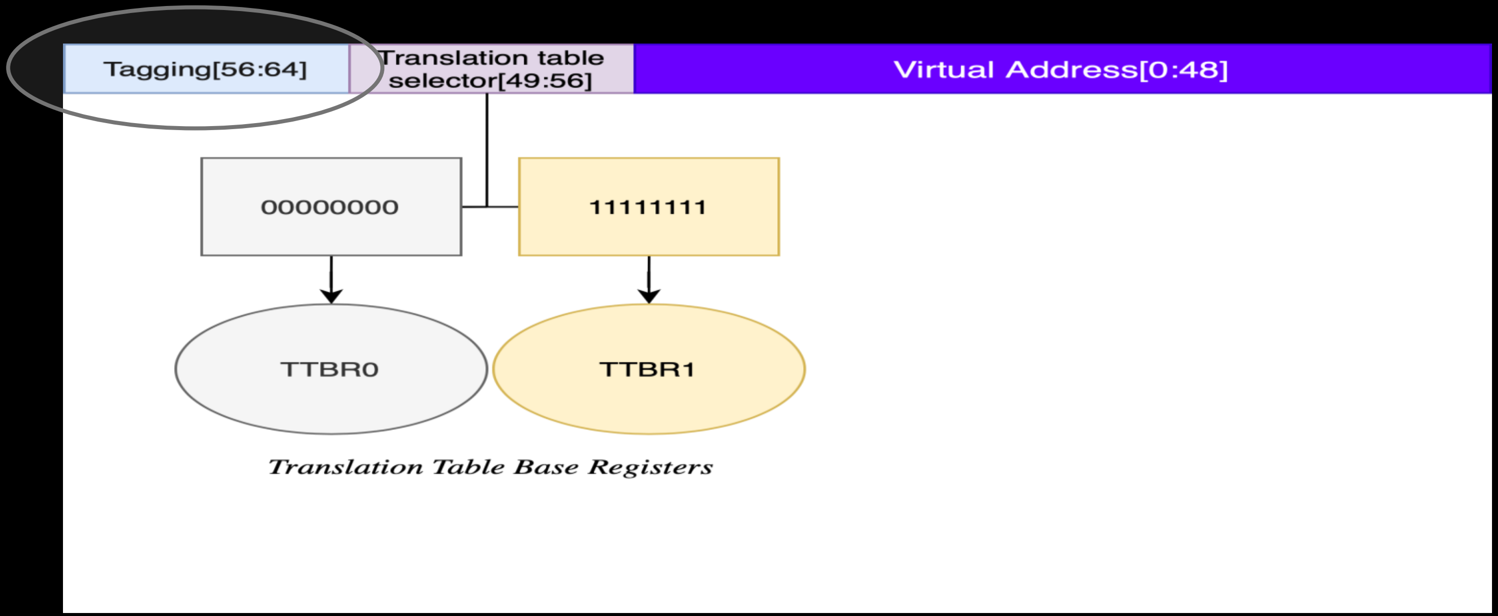
- Rely on three major components
  - Instrumentation around target instruction
  - Shadow memory
  - Runtime library

- Ex: ASAN

RedZone 1	Memory 1	RedZone 2	Memory 2	RedZone 3

MTE [2019] [Technique]

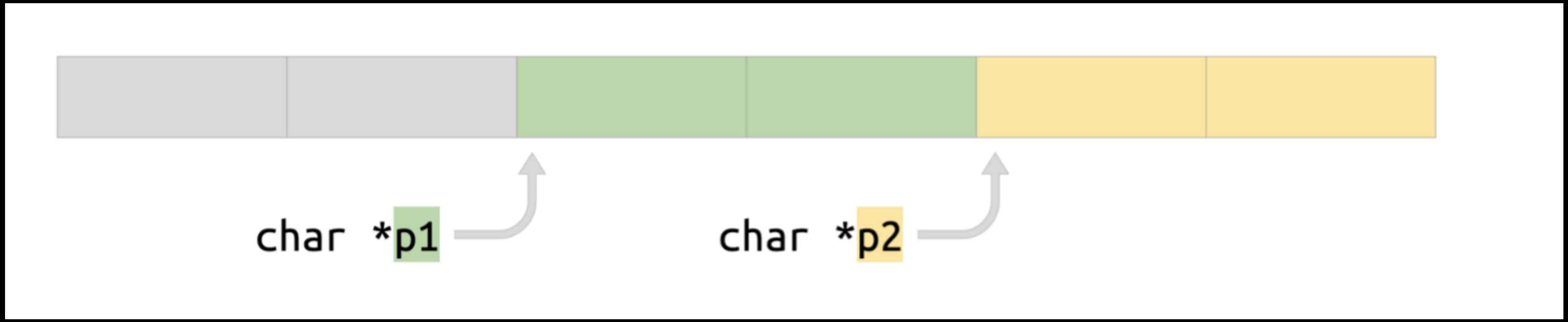
- Hardware enforced memory error detection tool.
- Can be used in production due to minimal performance impact.
- Used ARM addresses (Top byte ignore) to store tags.



DEEPSEC

MTE implementation

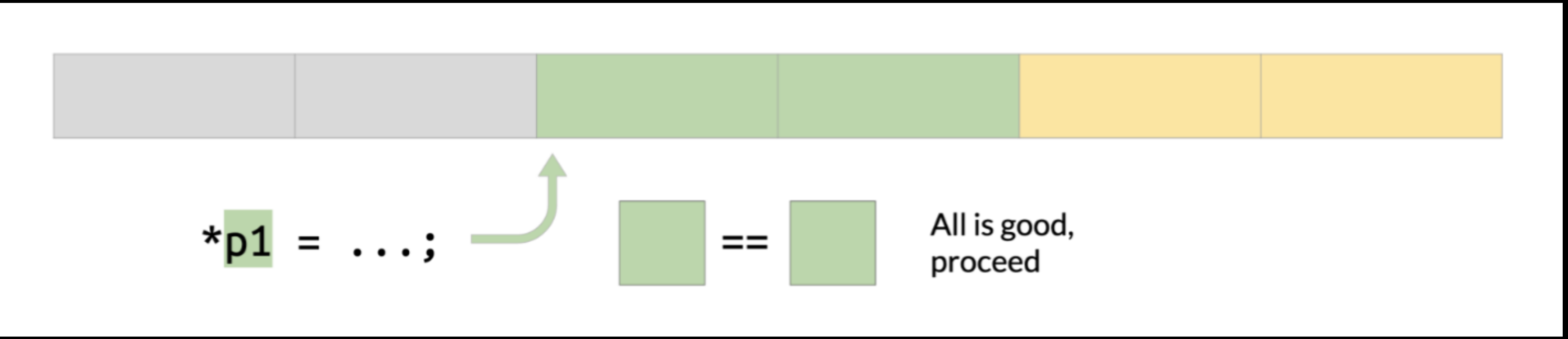
- Each memory granule has a tag (aka color)
- Every pointer has a tag
- On allocation, both memory and pointer get a matching random tag



DEEPSEC

MTE implementation

- Each memory granule has a tag (aka color)
- Every pointer has a tag
- On allocation, both memory and pointer get a matching random tag
- On pointer dereference, pointer tag must match memory tag



MTE implementation

- Each memory granule has a tag (aka color)
- Every pointer has a tag
- On allocation, both memory and pointer get a matching random tag
- On pointer dereference, pointer tag must match memory tag

DEEPSEC

Securing future using Rust

- Linux and windows (kernel) developers are moving toward rust lang due to absence of memory corruption.
- Has concept of ownership.
- Equivalent performance for low level usage.

DEEPSEC

Conclusion

- Memory corruption are there to stay but exploitation became harder and harder.
- Application developer need to identify what mitigations need to be added during compile time.
- Full research : https://nixhacker.com

Thank you

Any questions?