

CS 489 / 698: Software and Systems Security

**Module 9: Hardware Security**  
security features, enablers, and accelerators

Meng Xu (*University of Waterloo*)

Winter 2024

# Outline

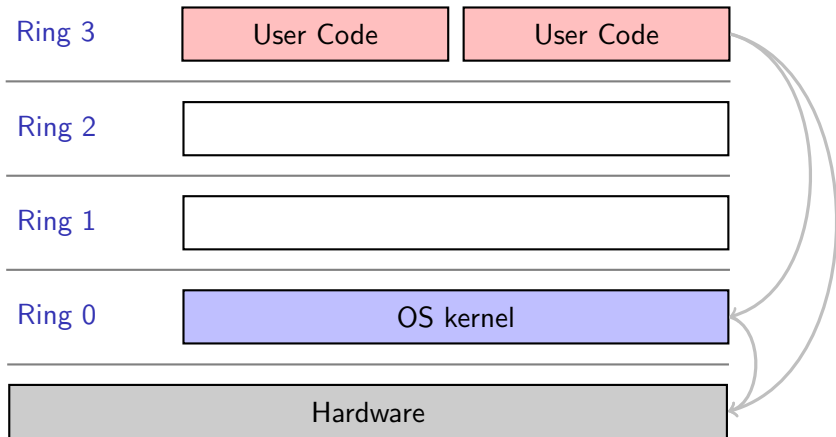
- 1 Introduction
- 2 Intel Control-flow Enforcement Technology (CET)
- 3 Arm Pointer Authentication (PA)
- 4 Intel Memory Protection Extensions (MPX)
- 5 Arm Memory Tagging Extension (MTE)
- 6 Capability Hardware Enhanced RISC Instructions (CHERI)
- 7 Authenticated boot and Root-of-Trust (RoT)

# Motivation

**Q:** What can hardware do for software and system security?

# Motivation

**Q:** What can hardware do for software and system security?



# Motivation

**Q:** What can hardware do for software and system security?

**A:** There are generally two views on hardware-assisted security:

# Motivation

**Q:** What can hardware do for software and system security?

**A:** There are generally two views on hardware-assisted security:

- Hardware runs at an even higher privilege level such that a malicious or compromised kernel cannot temper with — e.g., TPMs or TEEs (next lecture)

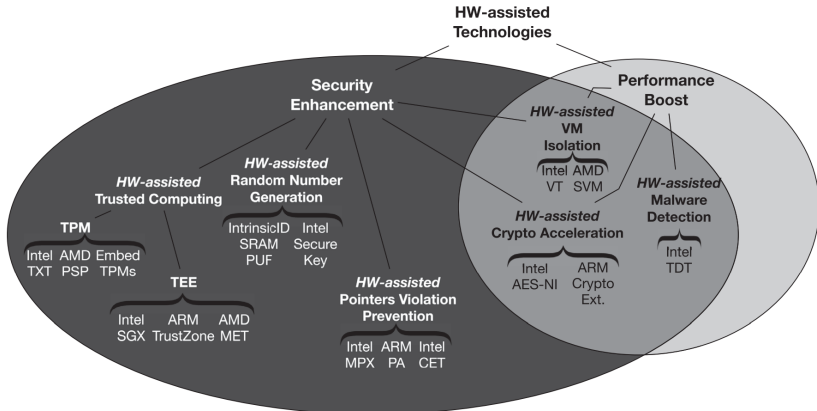
# Motivation

**Q:** What can hardware do for software and system security?

**A:** There are generally two views on hardware-assisted security:

- Hardware runs at an even higher privilege level such that a malicious or compromised kernel cannot temper with — e.g., TPMs or TEEs (next lecture)
- Hardware can accelerate security mechanisms that are conventionally enforced by kernel, compiler, or even the developers manually — e.g., CHERI (this lecture)

# Categorization of hardware-assisted security



Adapted from survey paper [A Comprehensive Survey of Hardware-Assisted Security: From The Edge to The Cloud](#)



# Outline

- 1 Introduction
- 2 Intel Control-flow Enforcement Technology (CET)**
- 3 Arm Pointer Authentication (PA)
- 4 Intel Memory Protection Extensions (MPX)
- 5 Arm Memory Tagging Extension (MTE)
- 6 Capability Hardware Enhanced RISC Instructions (CHERI)
- 7 Authenticated boot and Root-of-Trust (RoT)

# Recap on CFI

Control-Flow Integrity (CFI) is a classic example of **runtime reference monitor** in software security.

CFI is also sometimes referred to as **program shepherding**

*monitoring control flow transfers during program execution to enforce a security policy — from [a paper in USENIX Security'02](#).*

# Basic ideas of CFI

```
1 void f1();
2 void f2();
3 void f3();
4 void f4(int, int);
5
6 void foo(int usr) {
7     void (*func)();
8
9     if (usr == MAGIC)
10        func = f1;
11    else
12        func = f2;
13
14    // forward edge CFI check
15    CHECK_CFI_FORWARD(func);
16    func();
17
18    // backward edge CFI check
19    CHECK_CFI_BACKWARD();
20 }
```

Option 1: allow all functions

- f1, f2, f3, f4, foo, printf, system, ...

Option 2: allowed only functions defined in the current module

- f1, f2, f3, f4, foo

Option 3: allow functions with type signature `void (*)()`

- f1, f2, f3

Option 4: allow functions whose address are taken (e.g., assigned)

- f1, f2

# Example: Microsoft Control-flow Guard (CFG)

CFG implements **coarse-grained control-flow integrity** for indirect calls

## Compile time

```
void Foo(...) {
    // SomeFunc is address-taken
    // and may be called indirectly
    Object->FuncPtr = SomeFunc;
}
```

Metadata is automatically added to the image which identifies functions that may be called indirectly

```
void Bar(...) {
    // Compiler-inserted check to
    // verify call target is valid
    _guard_check_icall(Object->FuncPtr);
    Object->FuncPtr(xyz);
}
```

A lightweight check is inserted prior to indirect calls which will verify that the call target is valid at runtime

## Runtime

### Process Start

- Map valid call target data

### Image Load

- Update valid call target data with metadata from PE image

### Indirect Call

- Perform O(1) validity check
- Terminate process if invalid target
- Jump if target is valid

CFG is a deterministic mitigation, its security is not dependent on keeping secrets.

For C/C++ code, CFG requires no source code changes.

```
ntdll!LdrpDispatchUserCallTarget:
00007ffb`4e100e10 4c8b1d59e50d00  nov    r11,qword ptr
[ntdll!LdrSystemDllInitBlock+0xb0]
00007ffb`4e100e17 4c8b00      nov    r10,rax
00007ffb`4e100e1a 49c1ea09    shr    r10,9
00007ffb`4e100e1e 4f8b1cd3    nov    r11,qword ptr [r11+r10*8]
00007ffb`4e100e22 4c8b00      nov    r10,rax
00007ffb`4e100e25 49c1ea03    shr    r10,3
00007ffb`4e100e29 a80f      test  al,0fh
00007ffb`4e100e2b 7509      jne   ntdll!LdrpDispatchUserCallTarget+0x26
ntdll!LdrpDispatchUserCallTarget+0x1d:
00007ffb`4e100e2d 4d0fa3d3    bt    r11,r10
00007ffb`4e100e31 7303      jae   ntdll!LdrpDispatchUserCallTarget+0x26
ntdll!LdrpDispatchUserCallTarget+0x23:
00007ffb`4e100e33 48ffe0     jmp  rax
```

Illustration taken from [Microsoft Talk: The Evolution of CFI Attacks and Defenses](#)

# Example: Microsoft Return-flow Guard (RFG)

RFG was our compatible, ABI compliant, performant software shadow stack

## Compile Time

NOP's added to the prolog & epilog of all functions

Metadata added to the image to locate the prolog and epilog NOP bytes

## Runtime

### Process Start

- 1TB shadow stack region created
- Region cannot be queried
- AV's in region are fatal
- FS segment points to the shadow stack of the current thread

### Image Load

- If process enables RFG: patch NOP's with RFG prolog/epilog

### Function Calls

- Prolog: Push return address to shadow stack
- Epilog: Fast fail if return address on stack and shadow stack are mismatched

Parent Function	Child Function
[...] //Prior code	
call ChildFunction	
	mov rax, [rsp]
	mov fs:[rsp], rax
	[...] //Child code
	mov rcx, fs:[rsp]
	cmp rcx, [rsp]
	jne _fast_fail
	ret
0xABCD: [...] //Remainder of parent function	

If attacker changes the return address at these points RFG is defeated

RFG relies on a secret: the shadow stack's virtual address

Illustration taken from [Microsoft Talk: The Evolution of CFI Attacks and Defenses](#)

# RFG deployment experience

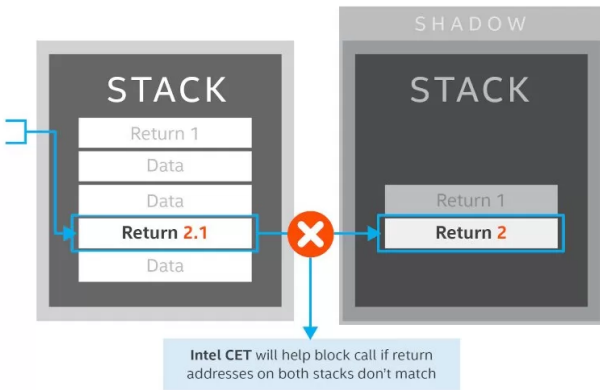
Secrets are bad!

AnC attack (a side-channel attack) could successfully leak where shadow stacks are mapped.

# Back-edge protection: shadow stack

## SHADOW STACK (SS)

SS delivers return address protection to defend against return-oriented programming (ROP) attack methods.



## CET: shadow stack

- For every regular stack CET adds a shadow stack region, which is indexed via a new register `%ssp`.
- Regular memory stores (executed from any ring) are not allowed in shadow stack region

When enabled,

- Each time a `call` instruction gets executed, in addition to the return address being pushed onto the regular stack, a copy of it is also pushed (automatically) onto the shadow stack.
- Each time a `ret` instruction gets executed, the return addresses pointed by `%rsp` and `%ssp` are (automatically) popped from the two stacks, and their values are compared together.



# CET: Indirect Branch Tracking (IBT)

CET introduces a new (4-byte) instruction, i.e., `endbr`, which becomes the **only** allowed target of indirect `call/jmp` instructions.

In other words, forward-edge transfers via (indirect) `call` or `jmp` instructions are pinned to code locations that are “marked” with an `endbr`; else, an exception (`#CP`) is raised.

# IBT example

```
1 void main() {  
2     int (*f) {};  
3     f = foo;  
4     f();  
5 }  
6  
7 int foo() {  
8     return 0;  
9 }
```

```
1 <main>:  
2 movq    $0x4004fb, -8(%rbp)  
3 mov     -8(%rbp), %rdx  
4 call   *%rdx  
5 :  
6 retq  
7  
8 <foo>:  
9 endbr64  
10 :  
11 mov    rax, 0  
12 :  
13 retq
```

# IBT example

```
1 void main() {
2     int (*f) {};
3     int (*g) {};
4     f = foo;
5     g = bar;
6     f();
7     g();
8 }
9
10 int foo() {
11     return 0;
12 }
13
14 int bar() {
15     return 1;
16 }
```

```
1 <main>:
2 movq    $0x4004fb, -16(%rbp)
3 mov     -16(%rbp), %rdx
4 call   *%rdx
5 mov     -8(%rbp), %rdx
6 call   *%rdx
7 :
8 retq
9
10 <foo>:
11 endbr64
12 :
13 mov     rax, 0
14 :
15 retq
16
17 <bar>:
18 endbr64
19 :
20 mov     rax, 1
21 :
22 retq
```

# Outline

- 1 Introduction
- 2 Intel Control-flow Enforcement Technology (CET)
- 3 Arm Pointer Authentication (PA)**
- 4 Intel Memory Protection Extensions (MPX)
- 5 Arm Memory Tagging Extension (MTE)
- 6 Capability Hardware Enhanced RISC Instructions (CHERI)
- 7 Authenticated boot and Root-of-Trust (RoT)

# Motivation

**Goal:** ensures **pointers** in memory remain **unchanged**.

# Motivation

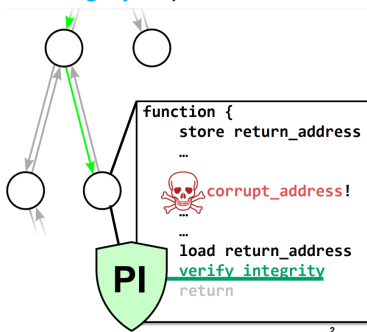
**Goal:** ensures **pointers** in memory remain **unchanged**.

- i.e., the **value of the pointer** remains unchanged, not the memory content referred to by this pointer.

# Motivation

**Goal:** ensures **pointers** in memory remain **unchanged**.

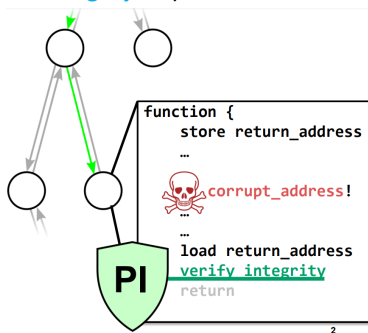
- i.e., the **value of the pointer** remains unchanged, not the memory content referred to by this pointer.
- Perfect **code pointer integrity** implies control-flow integrity (CFI).



# Motivation

**Goal:** ensures **pointers** in memory remain **unchanged**.

- i.e., the **value of the pointer** remains unchanged, not the memory content referred to by this pointer.
- Perfect **code pointer integrity** implies control-flow integrity (CFI).

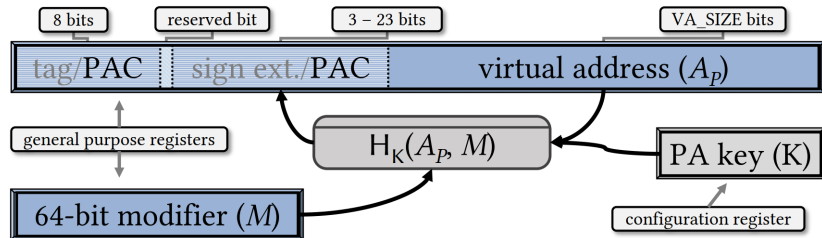


- Data pointer integrity is also important (e.g., against data-only attacks and data-oriented programming) and can be (partially) achieved via Pointer Authentication.



# Overview

Available since Armv8.3-A instruction set architecture (ISA) when the processor executes in 64-bit Arm state (AArch64)



PA consists of a set of instructions for creating and authenticating **pointer authentication codes (PACs)**.

# PAC details

- Each PAC is derived from
  - A pointer value
  - A 64-bit context value (modifier)
  - A 128-bit secret key

# PAC details

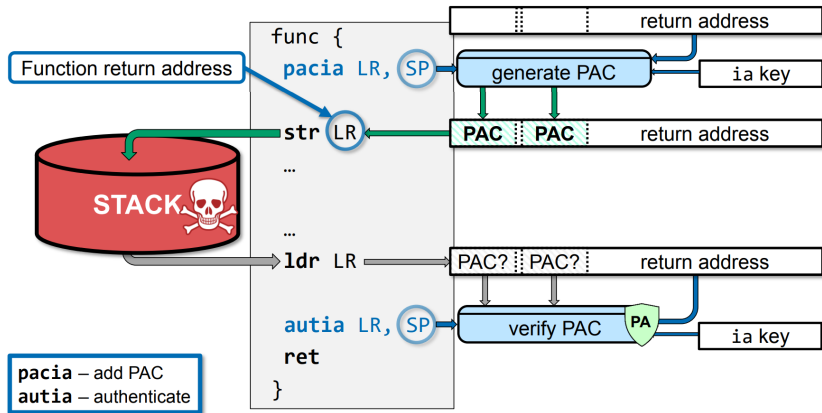
- Each PAC is derived from
  - A pointer value
    - \* an N-bit memory address
  - A 64-bit context value (modifier)
    - \* doesn't need to be secret, as long as it provides enough entropy
  - A 128-bit secret key
    - \* held in system registers, set by the kernel per each process,
    - \* can be used, but cannot be read/written by userspace

# PAC details

- Each PAC is derived from
  - A pointer value
    - \* an N-bit memory address
  - A 64-bit context value (modifier)
    - \* doesn't need to be secret, as long as it provides enough entropy
  - A 128-bit secret key
    - \* held in system registers, set by the kernel per each process,
    - \* can be used, but cannot be read/written by userspace
- PAC essentially a key-ed message authentication code (MAC) where the MAC algorithm can be implementation defined
  - by default, it is [QARMA](#)
- Instructions hide the algorithm details (sign + authenticate)

# Example: PA-based return address signing

Deployed as `-msign-return-address` in GCC and LLVM/Clang



# Outline

- 1 Introduction
- 2 Intel Control-flow Enforcement Technology (CET)
- 3 Arm Pointer Authentication (PA)
- 4 Intel Memory Protection Extensions (MPX)**
- 5 Arm Memory Tagging Extension (MTE)
- 6 Capability Hardware Enhanced RISC Instructions (CHERI)
- 7 Authenticated boot and Root-of-Trust (RoT)

## Brief history

Intel MPX (Memory Protection Extensions) was a set of extensions to the x86 instruction set architecture to perform bounds checking.

## Brief history

Intel MPX (Memory Protection Extensions) was a set of extensions to the x86 instruction set architecture to perform bounds checking.

- 2013-07: Intel introduces MPX in its ISA manual
- 2015-02: Linux kernel adds support to MPX in its 3.19 release
- 2015-04: GCC adds support to MPX in its 5.0 release
- 2015-08: MPX becomes available in Skylake microarchitecture
- 2018-06: An important paper [Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack](#) was published.
- 2019-??: Intel removes MPX from its ISA manual
- 2019-05: GCC drops support for MPX in its 9.1 release
- 2020-03: Linux kernel drops support for MPX in its 5.6 release



# How does MPX work?

```
1 struct obj { char buf[100]; int len }
2 obj* a[10];    total = 0;
3 for (i=0; i<M; i++) { total += a[i]->len; }
```

---

# How does MPX work?

```
1 struct obj { char buf[100]; int len }
2 obj* a[10]; total = 0;
3 for (i=0; i<M; i++) { total += a[i]->len; }
```

---

```
1 for (i=0; i<M; i++):
2   ai = a + i           // Pointer arithmetic on a
3   objptr = load ai     // Pointer to obj at a[i]
4   lenptr = objptr + 100 // Pointer to obj.len
5   len = load lenptr
6   total += len        // Total length of all objs
```

---

# How does MPX work?

```

1 struct obj { char buf[100]; int len }
2 obj* a[10];      total = 0;
3 for (i=0; i<M; i++) { total += a[i]->len; }

```

---

```

1 for (i=0; i<M; i++):
2   ai = a + i           // Pointer arithmetic on a
3   objptr = load ai     // Pointer to obj at a[i]
4   lenptr = objptr + 100 // Pointer to obj.len
5   len = load lenptr
6   total += len        // Total length of all objs

```

---

```

1 a_b = bndmk a, a+79
2 for (i=0; i<M; i++):
3   ai = a + i
4   bndcl a_b, ai        // Lower-bound check of a[i]
5   bndcu a_b, ai+7     // Upper-bound check of a[i]
6   objptr = load ai
7   objptr_b = bndldx ai // Bounds for pointer at a[i]
8   lenptr = objptr + 100
9   bndcl objptr_b, lenptr // Lower-bound check of obj.len
10  bndcu objptr_b, lenptr+3 // Upper-bound check of obj.len
11  len = load lenptr
12  total += len

```

## Recap: spatial safety

At any point of time during the program execution,  
for any object in memory, we know its  
(object\_id, size [int], alive [bool])

At the same time, for each memory access, we know:

- Memory read: (object\_id, offset [int], length [int])
- Memory write: (object\_id, offset [int], length [int], \_)

It is a violation of spatial safety if:

- $\text{offset} + \text{length} \geq \text{size}$  or
- $\text{offset} < 0$

# Supporting MPX

Adopting Intel MPX requires modifications at each level of the hardware-software stack:

- At the hardware level,
- At the kernel level:
- At the compiler level,
- At the application level,

# Supporting MPX

Adopting Intel MPX requires modifications at each level of the hardware-software stack:

- At the hardware level,
  - new instructions
  - a set of 128-bit registers (why 128-bit?)
  - the #BR exception thrown by these new instructions
- At the kernel level:
  
- At the compiler level,
  
  
- At the application level,

# Supporting MPX

Adopting Intel MPX requires modifications at each level of the hardware-software stack:

- At the hardware level,
  - new instructions
  - a set of 128-bit registers (why 128-bit?)
  - the #BR exception thrown by these new instructions
- At the kernel level: a new #BR exception handler for
  - allocating storage for bounds on-demand, and
  - sending a signal to the program upon bound violation.
- At the compiler level,
  
  
- At the application level,

# Supporting MPX

Adopting Intel MPX requires modifications at each level of the hardware-software stack:

- At the hardware level,
  - new instructions
  - a set of 128-bit registers (why 128-bit?)
  - the #BR exception thrown by these new instructions
- At the kernel level: a new #BR exception handler for
  - allocating storage for bounds on-demand, and
  - sending a signal to the program upon bound violation.
- At the compiler level,
  - new MPX transformation passes
  - new runtime libraries for initialization/finalization routines, debug information, and bridges to other non-MPX-protected libraries.
- At the application level,



# Supporting MPX

Adopting Intel MPX requires modifications at each level of the hardware-software stack:

- At the hardware level,
  - new instructions
  - a set of 128-bit registers (why 128-bit?)
  - the #BR exception thrown by these new instructions
- At the kernel level: a new #BR exception handler for
  - allocating storage for bounds on-demand, and
  - sending a signal to the program upon bound violation.
- At the compiler level,
  - new MPX transformation passes
  - new runtime libraries for initialization/finalization routines, debug information, and bridges to other non-MPX-protected libraries.
- At the application level,
  - manual change of troublesome C coding patterns
  - multithreading issues
  - interaction with other ISA extensions (e.g., TSX and SGX).

# What do we gain?

Approach	Detects	RIPE bugs	Other bugs	Broken	Perf (x)
Native: no protection	—	64 (34)	6 (3)	0 (0)	1.00 (1.00)
MPX security levels:					
L1: only-writes and no narrowing of bounds	inter-object overwrites	14 (14)	3 (0)	3 (5)	1.29 (1.18)
L2: no narrowing of bounds	+ inter-object overreads	14 (14)	3 (0)	2 (8)	2.39 (1.46)
L3: only-writes and narrowing of bounds	all overwrites*	14 (0)	2 (0)	4 (7)	1.30 (1.19)
L4: narrowing of bounds (default)	+ all overreads*	14 (0)	0 (0)	4 (9)	2.52 (1.47)
L5: + <code>fchkp-first-field-has-own-bounds</code> *	+ all overreads	0 (-)	0 (-)	6 (-)	2.52 (-)
L6: + <code>BNDPRESERVE=1</code> (protect all code)	all overflows in all code	0 (0)	0 (0)	34 (29)	-
AddressSanitizer	inter-object overflows	12	3	0	1.55

\* except intra-object overwrites & overreads through the first field of struct, level 5 removes this limitation (only relevant for GCC version)

Evaluation results available on [this website](#)

## Lessons learned

- New MPX instructions are not as fast as expected
  - The average overhead of 20-50% is not significantly better than ASan
- The supporting infrastructure is not mature enough
  - MPX transformation in compilers might be buggy
  - Other libraries needs to have MPX-enabled
- MPX provides no temporal protection
  - ASan has partial support
- MPX does not support multithreading transparently
  - Both false positives and false negatives if the application does not conform to C11 memory model or if the compiler does not update bounds in atomic primitives
- MPX is not compatible with some C idioms
  - e.g., using a struct field (usually the first field of struct) to access other fields of the struct
  - custom memory management, e.g., arbitrary type casts and in-pointer bit twiddling

# Outline

- 1 Introduction
- 2 Intel Control-flow Enforcement Technology (CET)
- 3 Arm Pointer Authentication (PA)
- 4 Intel Memory Protection Extensions (MPX)
- 5 Arm Memory Tagging Extension (MTE)**
- 6 Capability Hardware Enhanced RISC Instructions (CHERI)
- 7 Authenticated boot and Root-of-Trust (RoT)

# Overview

Introduced into the Armv8.5-A instruction set architecture (ISA) as Memory Tagging Extension (MTE) in 2018.

- 64-bit architecture only (AArch64)
- As a hardware accelerator for detecting memory errors

# Overview

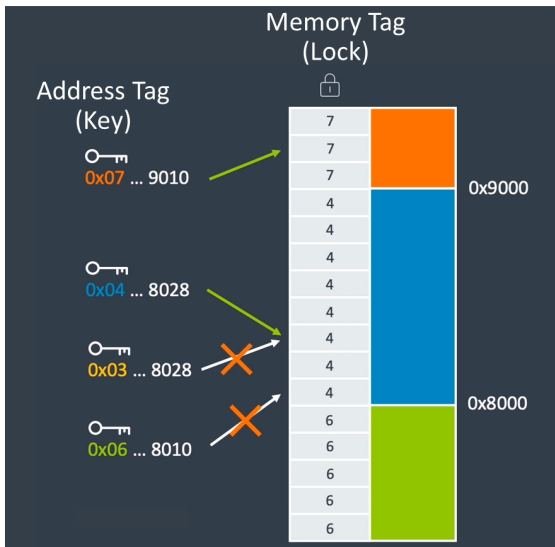
Introduced into the Armv8.5-A instruction set architecture (ISA) as Memory Tagging Extension (MTE) in 2018.

- 64-bit architecture only (AArch64)
- As a hardware accelerator for detecting memory errors

MTE implements a “lock-and-key” scheme for memory access:

- Two types of tags:
  - Every aligned 16 bytes of memory have a 4-bit tag **stored separately, i.e., not addressable** (the “lock”)
  - Every pointer has a 4-bit tag **stored in the top byte** (the “key”)
- LD/ST instructions check both tags, raise exception on mismatch
- New instructions are introduced to manipulate the tags

# MTE illustration



# Detecting heap overflow

```
char *p = new char[20]; // 0xa007ffffffff1240
```





# Detecting heap overflow

```
char *p = new char[20]; // 0xa007ffffffff1240
```



```
p[32] = ... // heap-buffer-overflow █ ≠ █
```

# Detecting use-after-free

```
char *p = new char[20]; // 0xa007ffffffff1240
```



# Detecting use-after-free

```
char *p = new char[20]; // 0xa007ffffffff1240
```



```
delete [] p; // Memory is retagged green ⇒ magenta
```



# Detecting use-after-free

```
char *p = new char[20]; // 0xa007ffffffff1240
```



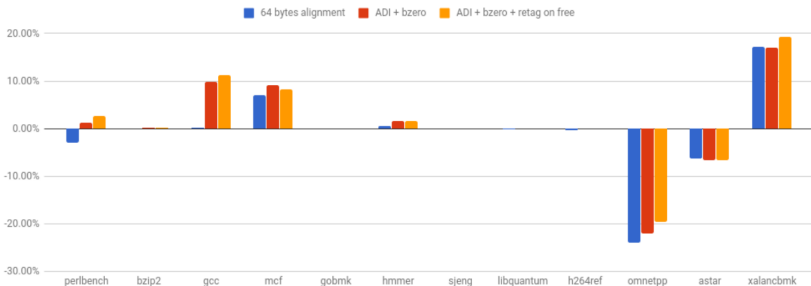
```
delete [] p; // Memory is retagged green ⇒ magenta
```



```
p[0] = ... // heap-use-after-free green ≠ magenta
```

# Adoption in practice

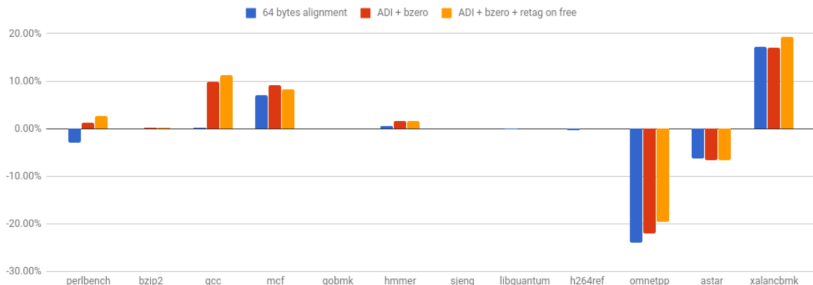
- LLVM [MemTagSanitizer](#) detects a similar class of errors as [AddressSanitizer](#) or [HardwareAssistedAddressSanitizer](#), but with **much** lower overhead.



Source of numbers: [LLVM whitepaper on memory tagging](#)

# Adoption in practice

- LLVM [MemTagSanitizer](#) detects a similar class of errors as [AddressSanitizer](#) or [HardwareAssistedAddressSanitizer](#), but with **much** lower overhead.



Source of numbers: [LLVM whitepaper on memory tagging](#)

- In Android 12, the kernel and userspace heap memory allocator can augment each allocation with metadata, based on [this article](#).

# Outline

- 1 Introduction
- 2 Intel Control-flow Enforcement Technology (CET)
- 3 Arm Pointer Authentication (PA)
- 4 Intel Memory Protection Extensions (MPX)
- 5 Arm Memory Tagging Extension (MTE)
- 6 Capability Hardware Enhanced RISC Instructions (CHERI)**
- 7 Authenticated boot and Root-of-Trust (RoT)

# Re-defining pointers

A pointer is not only an  $N$ -bit value representing a memory address, rather, it is a **capability** granting **certain permissions** to access a **restrictive range** in the memory address space.



# CHERI memory capability

1 bit capability tag:

1 - valid

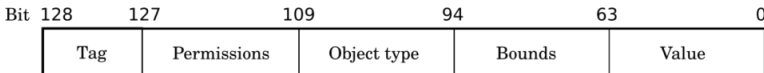
0 - invalid

15-bit:

defines if and how  
the capability is sealed

64-bit:

56-bit bounds and 8-bit  
flag. This is offset from  
the Bounds field



18-bit:  
limits usage  
of the capability

87-bit bound, limits  
the scope of the capability  
(31 + 56 bits)

A “pointer”, or rather, a **memory capability**, in the view of the CHERI [Morello](#) architecture (source of image: [Pawel Zalewski's blog post](#)).

# CHERI basic idea

```
#include <stdio.h>
int x=1;
int secret_key = 4091;
int main() {
    int *p = &x;
    p = p+1;
    int y = *p;
    printf("%d\n",y);
}
```

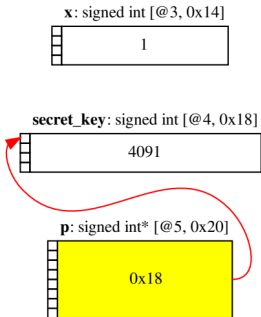
# CHERI basic idea

```
#include <stdio.h>
int x=1;
int secret_key = 4091;
int main() {
    int *p = &x;
    p = p+1;
    int y = *p;
    printf("%d\n",y);
}
```

**Q:** What will happen?

# CHERI basic idea

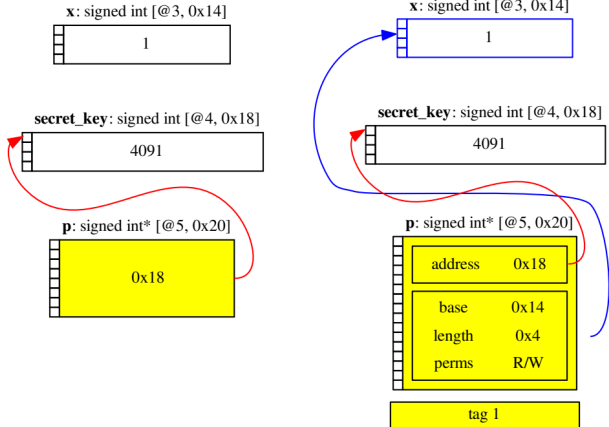
```
#include <stdio.h>
int x=1;
int secret_key = 4091;
int main() {
    int *p = &x;
    p = p+1;
    int y = *p;
    printf("%d\n",y);
}
```



**Q:** What will happen?

# CHERI basic idea

```
#include <stdio.h>
int x=1;
int secret_key = 4091;
int main() {
    int *p = &x;
    p = p+1;
    int y = *p;
    printf("%d\n",y);
}
```



**Q:** What will happen?

# CHERI software stack

Completely re-vamped software stack:

- Compilers: *custom-made* Clang/LLVM
- Operating systems: *hand-tuned* FreeBSD, FreeRTOS
- Applications: *ported* WebKit, OpenSSH, and PostgreSQL

# Outline

- 1 Introduction
- 2 Intel Control-flow Enforcement Technology (CET)
- 3 Arm Pointer Authentication (PA)
- 4 Intel Memory Protection Extensions (MPX)
- 5 Arm Memory Tagging Extension (MTE)
- 6 Capability Hardware Enhanced RISC Instructions (CHERI)
- 7 Authenticated boot and Root-of-Trust (RoT)**

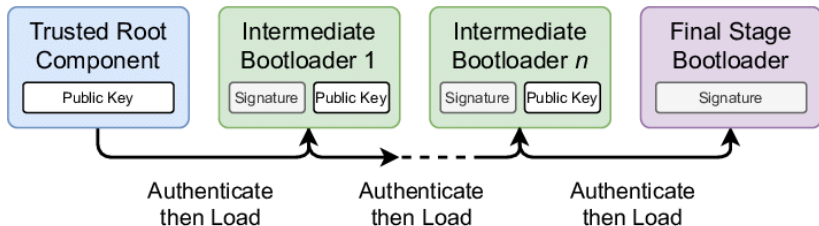
# Overview

**Goal:** ensures only **trusted** and **authenticated** software (e.g., firmware, kernel, application) runs on a computing system.



# Overview

**Goal:** ensures only **trusted** and **authenticated** software (e.g., firmware, kernel, application) runs on a computing system.



An abstract view of the authenticated boot process

# Requirements for the root-of-trust (RoT) component

- Boot process is guaranteed to start from the RoT component
- The cryptographic key is non-readable, non-writable at any privilege level
  - The only way to use the key is to verify the signature via special hardware instructions.
- The RoT component, upon booting, must first measure the code content of the first stage bootloader and validate the measurement with the signature.

# Requirements for the root-of-trust (RoT) component

- Boot process is guaranteed to start from the RoT component
- The cryptographic key is non-readable, non-writable at any privilege level
  - The only way to use the key is to verify the signature via special hardware instructions.
- The RoT component, upon booting, must first measure the code content of the first stage bootloader and validate the measurement with the signature.

Usually, the RoT component is encapsulated in a hardware module named [Hardware Security Module \(HSM\)](#).

〈 End 〉