Module 5: Hardware Security
security features, enablers, and accelerators

Meng Xu (University of Waterloo)
Spring 2023
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?

Ring 3
- User Code
- User Code

Ring 2
- 

Ring 1
- 

Ring 0
- OS kernel

Hardware
Motivation

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

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

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

```c
void f1();
void f2();
void f3();
void f4(int, int);

void foo(int usr) {
    void (*func)();
    if (usr == MAGIC)
        func = f1;
    else
        func = f2;

    // forward edge CFI check
    CHECK_CFI_FORWARD(func);
    func();

    // backward edge CFI check
    CHECK_CFI_BACKWARD();
}
```

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**

```c
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.

**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
- Jmp 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.

---

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

<table>
<thead>
<tr>
<th>Compile Time</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP's added to the prolog &amp; epilog of all functions</td>
<td>Process Start</td>
</tr>
<tr>
<td>Metadata added to the image to locate the prolog and epilog NOP bytes</td>
<td>Image Load</td>
</tr>
<tr>
<td></td>
<td>Function Calls</td>
</tr>
<tr>
<td></td>
<td>• 1TB shadow stack region created</td>
</tr>
<tr>
<td></td>
<td>• Region cannot be queried</td>
</tr>
<tr>
<td></td>
<td>• AV's in region are fatal</td>
</tr>
<tr>
<td></td>
<td>• FS segment points to the shadow stack of the current thread</td>
</tr>
<tr>
<td></td>
<td>• If process enables RFG: patch NOP's with RFG prolog/epilog</td>
</tr>
<tr>
<td></td>
<td>• Prolog: Push return address to shadow stack</td>
</tr>
<tr>
<td></td>
<td>• Epilog: Fast fail if return address on stack and shadow stack are mismatched</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parent Function</th>
<th>Child Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[...] //Prior code</td>
<td>call ChildFunction</td>
</tr>
<tr>
<td>mov rax, [rsp]</td>
<td>mov fs:[rsp], rax</td>
</tr>
<tr>
<td>[...] //Child code</td>
<td>mov rcx, fs:[rsp]</td>
</tr>
<tr>
<td>cmp rcx, [rsp]</td>
<td>jne _fast_fail</td>
</tr>
<tr>
<td>ret</td>
<td>0xABCD: [...] //Remainder of parent function</td>
</tr>
</tbody>
</table>

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

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.

*Intel CET will help block call if return addresses on both stacks don't match.*
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 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

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

```assembly
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

```c
void main() {
    int (*f) {};
    int (*g) {};
    f = foo;
    g = bar;
    f();
    g();
}

int foo() {
    return 0;
}

int bar() {
    return 1;
}
```

```asm
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
Each PAC is derived from
- A pointer value
  * an N-bit memory address
- A 64-bit context value (modifier)
  * doesn’t need to 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
Each PAC is derived from
- A pointer value
  * an N-bit memory address
- A 64-bit context value (modifier)
  * doesn’t need to 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)
Intel MPX (Memory Protection Extensions) was a set of extensions to the x86 instruction set architecture to perform bounds checking.
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?

```c
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; }

```c
for (i=0; i<M; i++):
    ai = a + i          // Pointer arithmetic on a
    objptr = load ai    // Pointer to obj at a[i]
    lenptr = objptr + 100 // Pointer to obj.len
    len = load lenptr
    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; }

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

```c
a_b = bndmk a, a+79
for (i=0; i<M; i++):
    ai = a + i
    bndcl a_b, ai  // Lower-bound check of a[i]
    bndcu a_b, ai+7  // Upper-bound check of a[i]
    objptr = load ai
    objptr_b = bndldx ai  // Bounds for pointer at a[i]
    lenptr = objptr + 100
    bndcl objptr_b, lenptr  // Lower-bound check of obj.len
    bndcu objptr_b, lenptr+3  // Upper-bound check of obj.len
    len = load lenptr
    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:

- offset + length >= size or
- offset < 0
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).
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,
  - 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,**
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?

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

* 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

Memory Tag (Lock)

Address Tag (Key)

- 0x07 ... 9010
- 0x04 ... 8028
- 0x03 ... 8028
- 0x06 ... 8010

Source: article Delivering enhanced security through Memory Tagging Extension
Detecting heap overflow

```c
char *p = new char[20]; // 0xa007ffffffffff1240
```

-32:-17  -16:-1  0:15  16:31  32:47  48:64
Detecting heap overflow

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

```
    -32:-17 -16:-1  0:15  16:31  32:47  48:64
p[32] = ... // heap-buffer-overflow ≠
```
Detecting use-after-free

char *p = new char[20]; // 0xa007ffffff1240
Detecting use-after-free

```c
char *p = new char[20]; // 0xa007fffffff1240
```

```c
delete [] p; // Memory is retagged
```

Detecting use-after-free

```c
char *p = new char[20]; // 0xa007fffffff1240
```

delete [] p; // Memory is retagged ⇒

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>128</th>
<th>127</th>
<th>109</th>
<th>94</th>
<th>63</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Permissions</td>
<td>Object type</td>
<td>Bounds</td>
<td>Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18-bit:</td>
<td>limits usage</td>
<td>of the capability</td>
<td>87-bit bound, limits</td>
<td>the scope of the capability</td>
<td>(31 + 56 bits)</td>
<td></td>
</tr>
<tr>
<td>64-bit:</td>
<td>56-bit bounds and 8-bit flag. This is offset from the Bounds field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A “pointer”, or rather, a memory capability, in the view of the CHERI Morello architecture (source of image: Pawel Zalewski’s blog post).
#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);
}
#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);
}
#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?
#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);
}
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.
**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 〉