Pointer Authentication (PAC) and Branch Target Identification (BTI) Extension

Course 4 Advanced Armv8-M Features

Agenda

Learning objectives

At the end of this module you will be able to:

- Define the terms Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP)
- Describe PAC features
- Summarize the impact of BTI landing pad on the program flow
- Outline the debug feature interactions with PACBTI extension

Introduction

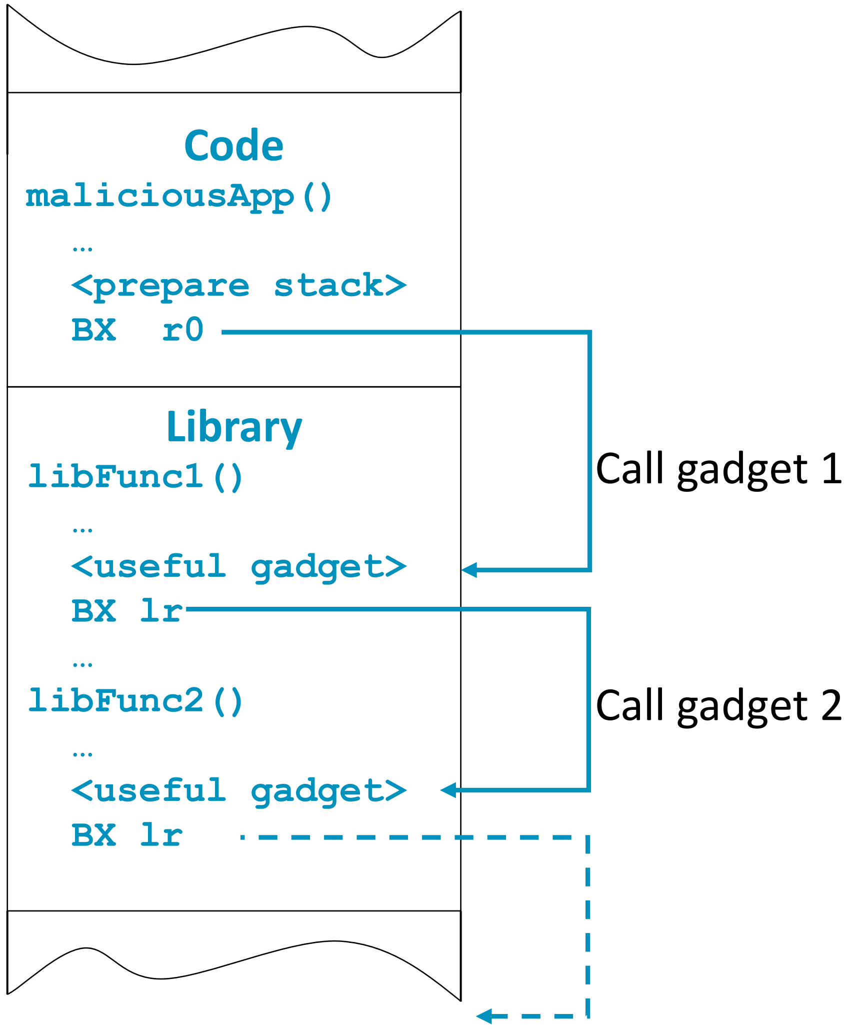
Return/Jump Orientated Programming (ROP/JOP)

Attack vector using return orientated programming, or ROP

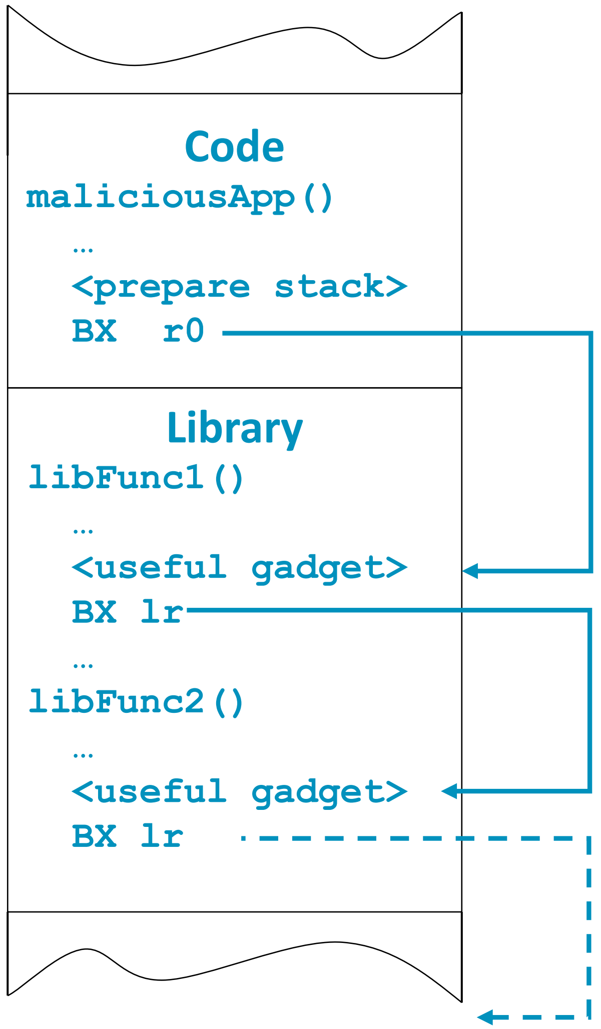
- Identify a set of *gadgets*: fragments of existing functions ending in a return
- Chain these *gadgets* together to form a new program by modifying the LR
- JOP is like ROP, but using indirect branches
  - More on JOP in the next section

ROP/JOP can be powerful attack vectors

- Re-uses existing legitimate code, so execution permissions do not help
- Modern rich software environments are so large that enough gadgets can be found to form just about any new program



Return/Jump Orientated Programming (ROP/JOP)



How do we find useful gadgets?

Will `BX lr` return back into `maliciousApp()`?

- Can we find a gadget where LR is popped from the stack?

Now we can execute gadgets; what can we do?

PACBTI extension

Added to Armv8.1-M as an optional extension

- Help address a range of stack attacks

Pointer Authentication

- Also known as Pointer Authentication Code (hence the “C” in the PAC)
- Used to verify function return addresses in stack
- Can also be used to verify other generic pointers

Branch Target Identification

- BTI instructions indicate valid landing pads for indirect branches
  - When BTI is enabled, the processor ensures indirect branches land on landing pads
- If a stack corruption taken place / the system is under ROP / JOP
  - Execution of return (e.g. “BX LR”) landing in middle of functions (with BTI) triggers fault exceptions
  - Significantly reduce number of gadgets

Pointer Authentication (PAC)

Pointer Authentication Code

Armv8.1-M introduces pointer authentication to mitigate ROP attacks

- Concept borrowed from AArch64 Armv8.3-A architecture
- Act as a set of NOP compatible instructions when executed on legacy processors that do not support PACBTI extension.

Using Pointer Authentication is a multiple step process

- Step 1: Generate a PAC from a pointer (and other inputs including crypto key)
- … (Normal function operations)
- Step 2: Authenticate pointer to make sure it has not been changed (e.g. due to a stack memory corruption)
- Pointer is considered safe to use if authentication passed

Authentication code is generated using cryptography

- Without knowing the crypto key, very hard to create a valid pair of fake pointer + PAC to pass authentication
- PAC collision is possible but still significantly improve security

Pointer Authentication Code – Contd.

Individual control for Pointer Authentication Code (PAC) in each: (via CONTROL.PAC_EN or CONTROL.UPAC_EN bits)

- Security state
- Privilege level

Operates with a pair of instructions for signing (PAC*) and validating (AUT*) return pointers

- PAC* – PAC, PACBTI, PACG
- AUT* – AUT, AUTG, BXAUT
- **Note:** PAC, PACBTI, PACG operations can interoperate with any of AUT, BXAUT, AUTG instructions; provided that the same input arguments are used for creating and authenticating PAC

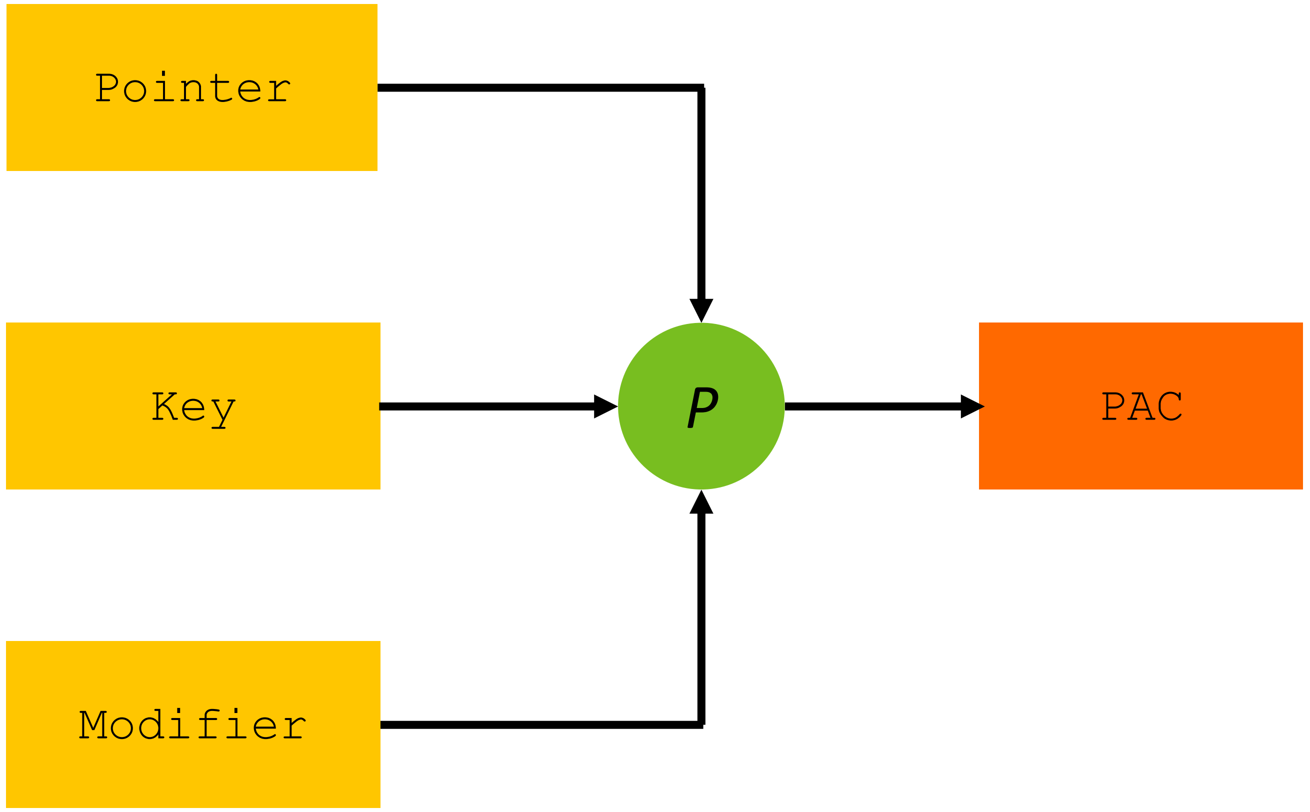
PAC Generation

Each PAC is derived from

- A 32-bit **pointer** value
  - Zero extended to 64 bits
- A 32-bit **modifier** value
  - Zero extended to 64 bits
- A 128-bit secret **key**

PAC algorithm *P* can be

- QARMA
- **`IMPLEMENTATION DEFINED`**



Cryptographic Keys

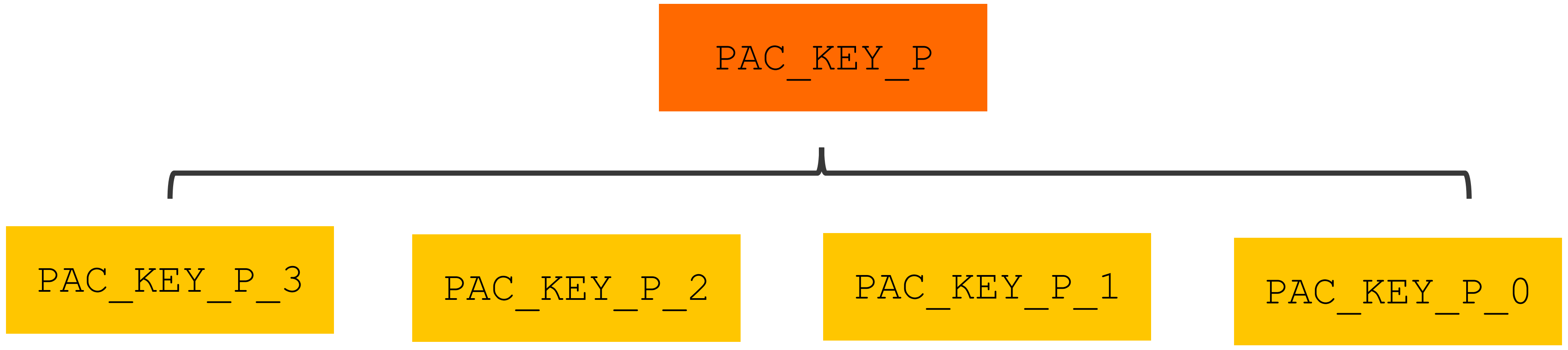
Four 128-bit keys are available for PAC generation

	Non-Secure State	Secure State
Privileged	PAC_KEY_P_NS	PAC_KEY_P_S
Unprivileged	PAC_KEY_U_NS	PAC_KEY_U_S

Both Privileged and Unprivileged keys are Privileged access only

Secure access only

Four 32 bit register values are concatenated to form a key PAC_KEY_P

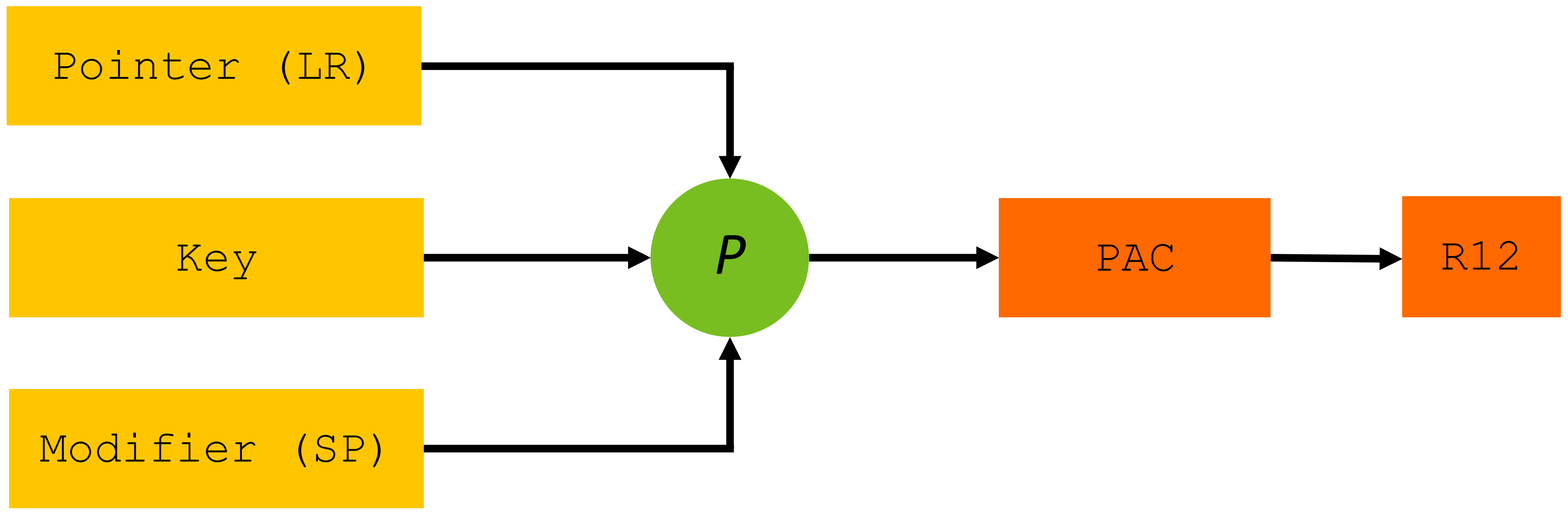


These registers are special registers and accessible via MRS/MSR instructions.

Operations: Signing

`PAC*` instructions sign pointers with PACs

Example: PAC   R12, LR, SP



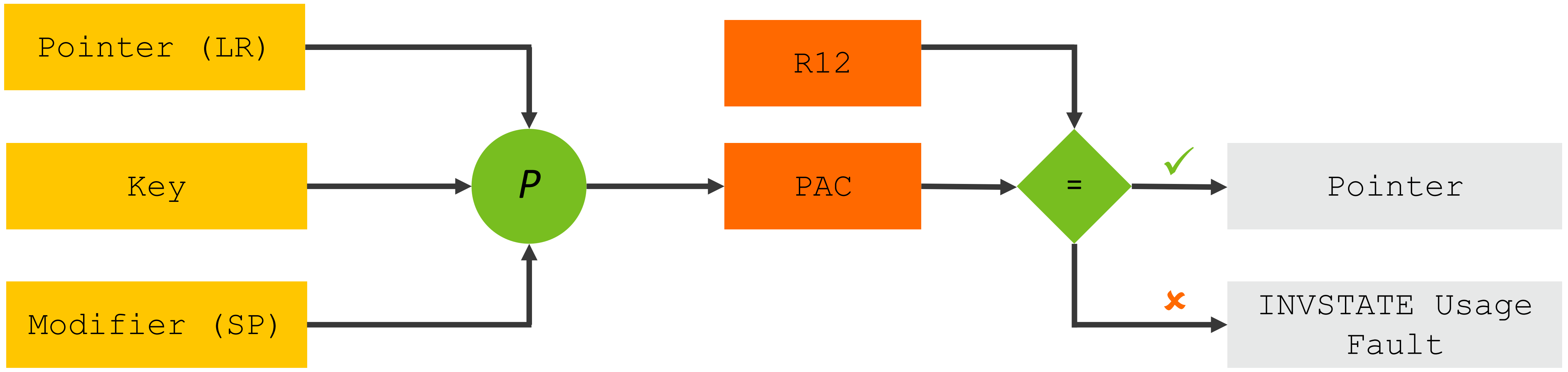
Result is stored in R12 for PAC/PACBTI and in Rd for PACG instruction

Operations: Authentication

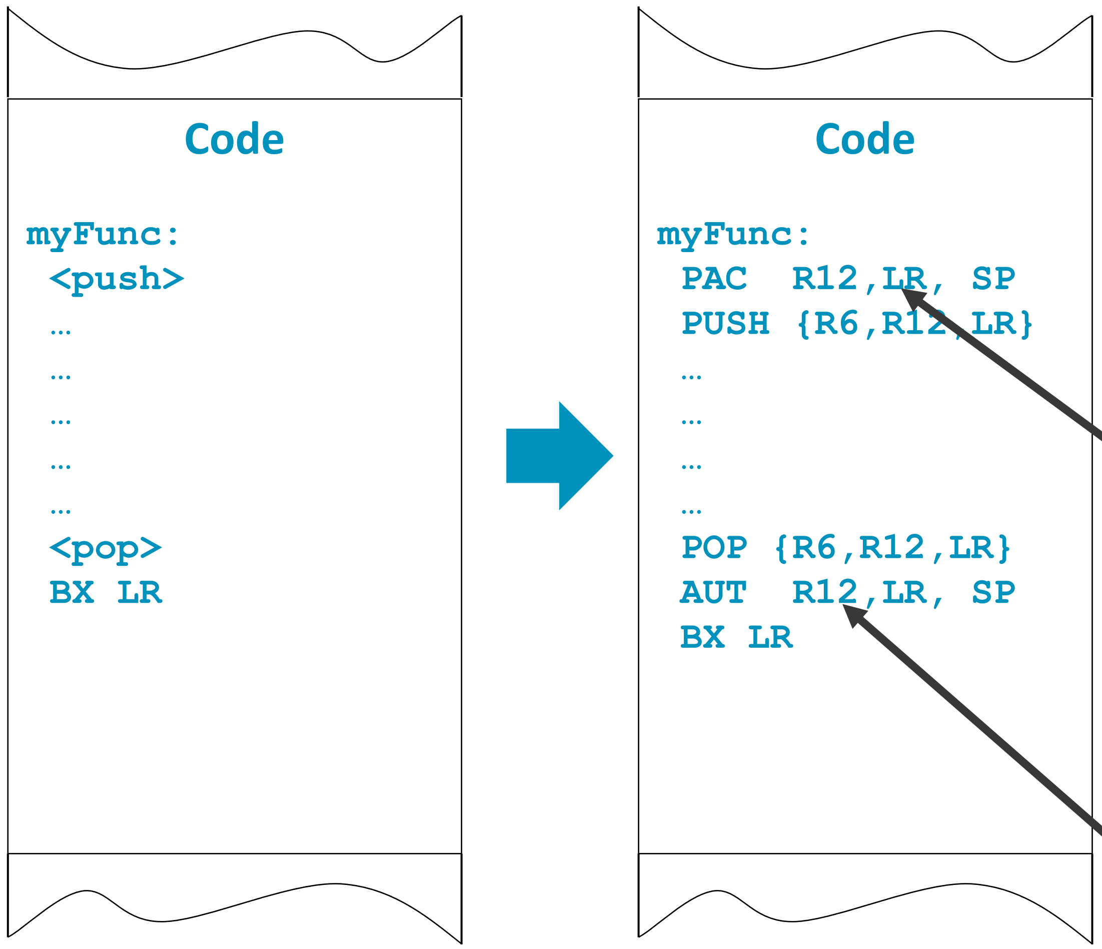
AUT* instructions authenticate PACs

- If PAC matches, the result is the original pointer
- If PAC fails, AUT instruction triggers an INVSTATE Usage Fault

Example: AUT R12, LR, SP



Pointer Authentication in use



```
myFunc:
 <push>
 …
 …
 …
 …
 …
 <pop>
 BX LR
```

```
myFunc:
 PAC  R12,LR, SP
 PUSH {R6,R12,LR}
 …
 …
 …
 …
 POP {R6,R12,LR}
 AUT  R12,LR, SP
 BX LR
```

Pointer authentication instructions can be added to all function prologues and epilogues

- The compiler can do this automatically
  - `armclang --target=arm-arm-none-eabi -march=armv8.1-m.main+pacbti -mbranch-protection=standard`
  - Linker options: `--library_security=pacbti-m`

Pointer authentication code for LR using key with SP as a modifier

Authenticate link register using key, then return.
If authentication fails, INVSTATE Usage Fault is generated

Instructions

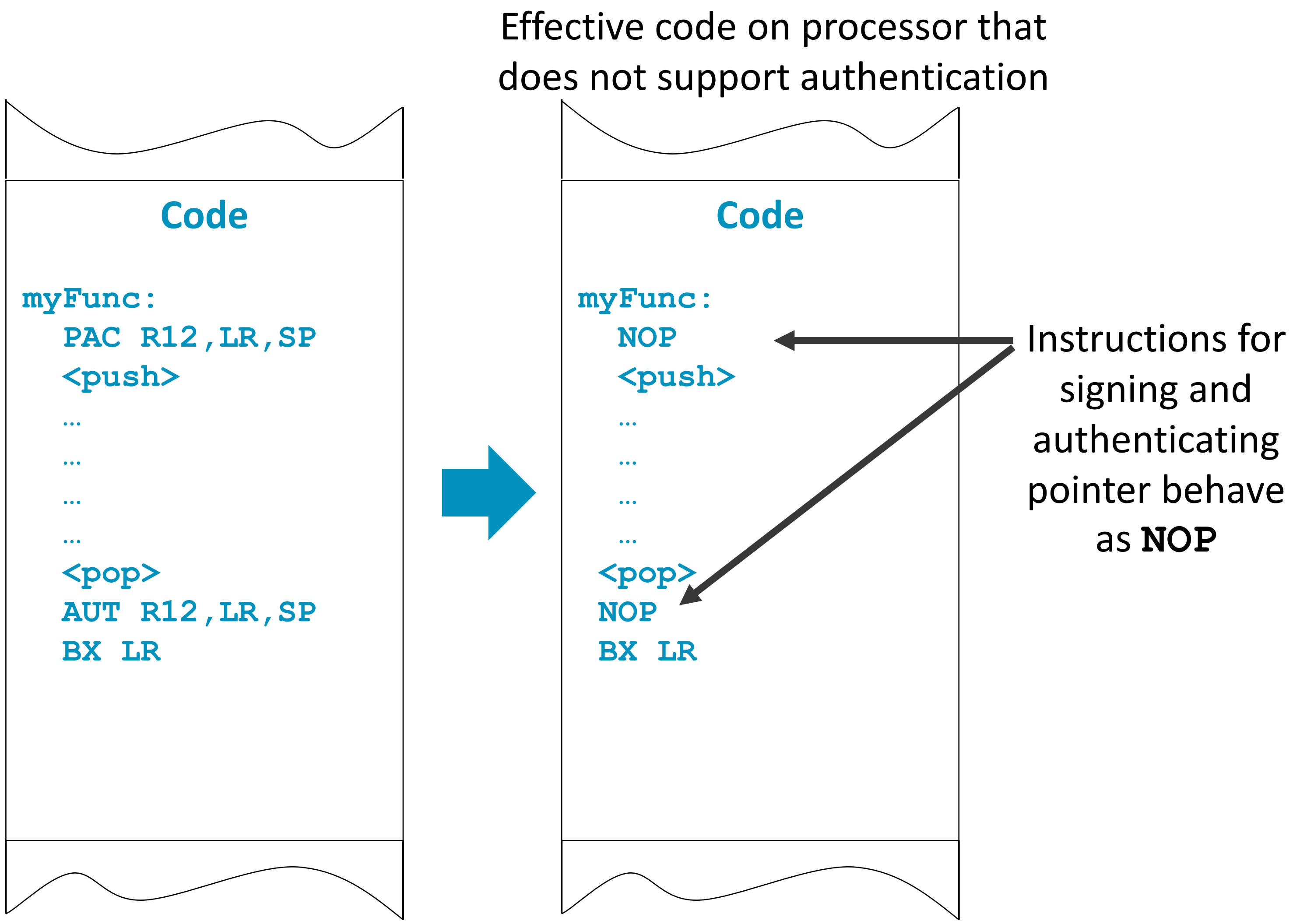
The following instructions are introduced

Instruction	NOP hint space?	Description
PAC R12, LR, SP	Y	Sign return address in LR (Modifier must be SP and destination register must be R12)
PACBTI R12, LR, SP	Y	Same as PAC, but is also an indirect branch target
PACG<cond> <Rd>, <Rn>, <Rm>	N	Sign generic pointer (Rn). Rm is the modifier and Rd is the destination.
AUT R12, LR, SP	Y	Authenticate Link Register (return address)
BXAUT	N	Authenticate Link Register and return Note: Because the return address is checked, the branch target does not need to be a landing pad (e.g. BTI)
AUTG<cond> <Ra>, <Rn>, <Rm>	N	Authenticate generic pointer (Rn). Rm is the modifier and Ra is the expected value (Rd from corresponding PACG instruction).

Running on Older Hardware

Pointer authentication instructions use part of the **NOP** instruction space

- At the cost of flexibility: typically authenticate LR with SP as modifier
- Benefit is that code protected by these instructions runs on hardware that does not support them
  - Older processors will just treat the instructions as **NOP**s



Effective code on processor that does not support authentication

Code

```
myFunc:
   PAC R12,LR,SP
   <push>
   …
   …
   …
   …
   <pop>
   AUT R12,LR,SP
   BX LR
```

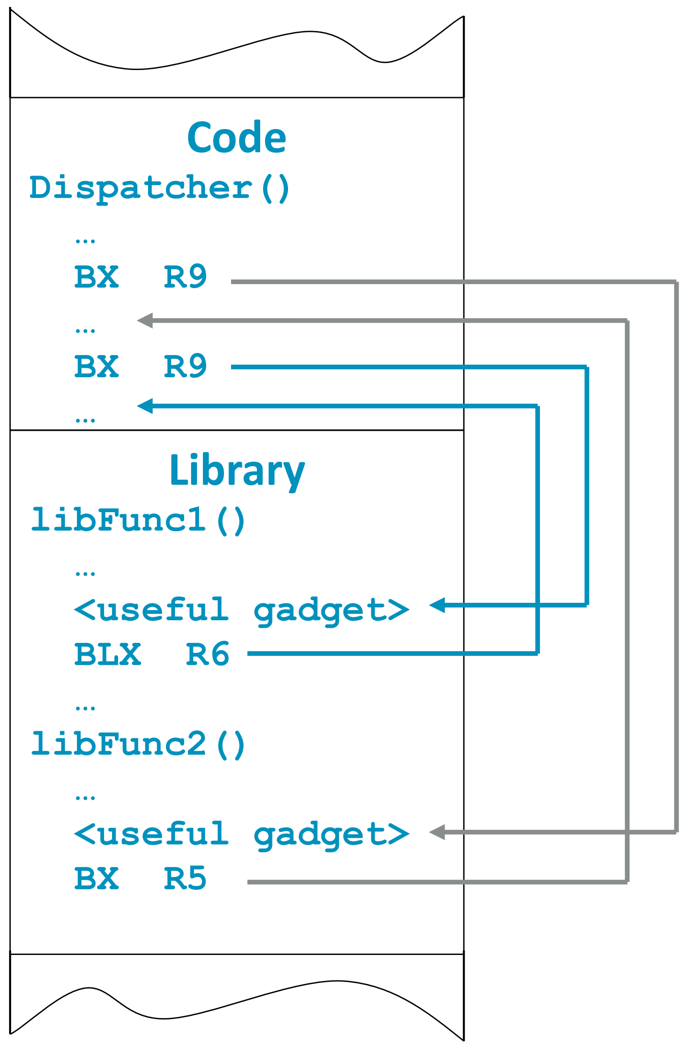
Code

```
myFunc:
   NOP
   <push>
   …
   …
   …
   …
  <pop>
  NOP
  BX LR
```

Instructions for signing and authenticating pointer behave as **NOP**

Branch Target Identification

Jump Orientated Programming (JOP)



Jump orientated programming is similar to ROP

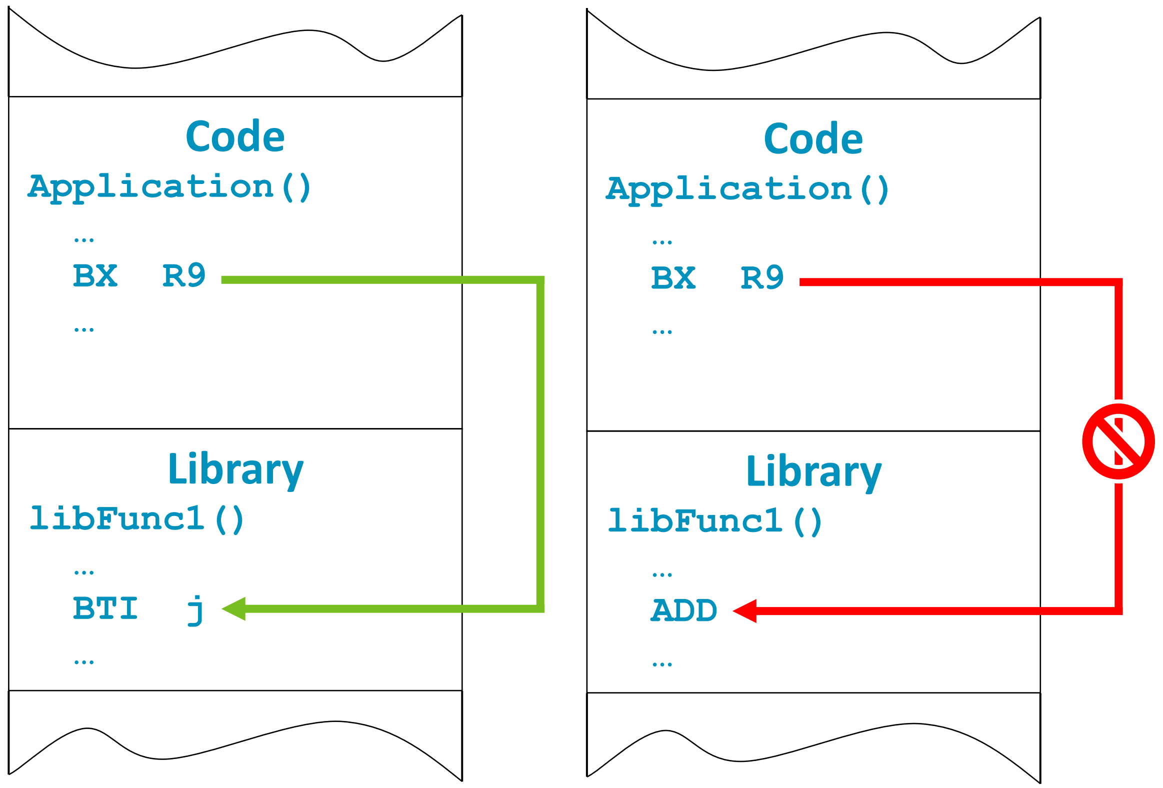
- Looks for gadgets ending in an indirect (absolute) branch, rather than a function return

More restrictive than ROP, but harder to defend against using pointer authentication

Branch Target Identification

To protect against JOP attacks, Armv8.1-M introduced Branch Target Identification (BTI) Instructions

- `BTI`s, or “landing pads”
- This drastically reduces the number of target addresses, and hence the number of possible gadgets
- Branch Target identification, provides landing pads, to harden code paths by restricting the processor from jumping into unexpected parts of a function



Branch Target Identification (BTI)

BTI can be enabled or disabled as follows

	Current Security State	Secure state	Non-Secure State
Privileged Mode	CONTROL.BTI_EN	CONTROL_S.BTI_EN	CONTROL_NS.BTI_EN
Unprivileged Mode	CONTROL.UBTI_EN	CONTROL_S.UBTI_EN	CONTROL_NS.UBTI_EN

An instruction setting EPSR.B bit to one is referred to as a **BTI setting** instruction

- EPSR.B = 1 ; indicates that Branch Target identification is active
- BTI setting instructions include
    - BLX, BLXNS
    - When the register holding the branch address is not the LR (BX, BXNS)
    - When the address is loaded onto PC (LDR <register>, LDR <literal>)
    - When the address is loaded onto PC without SP as base register OR SP as base register and without writeback operation (LDR <immediate>, LDMIA, LDMDB)

Branch Target Identification (BTI) - Contd

An instruction clearing EPSR.B bit to zero is referred as **BTI clearing** instruction

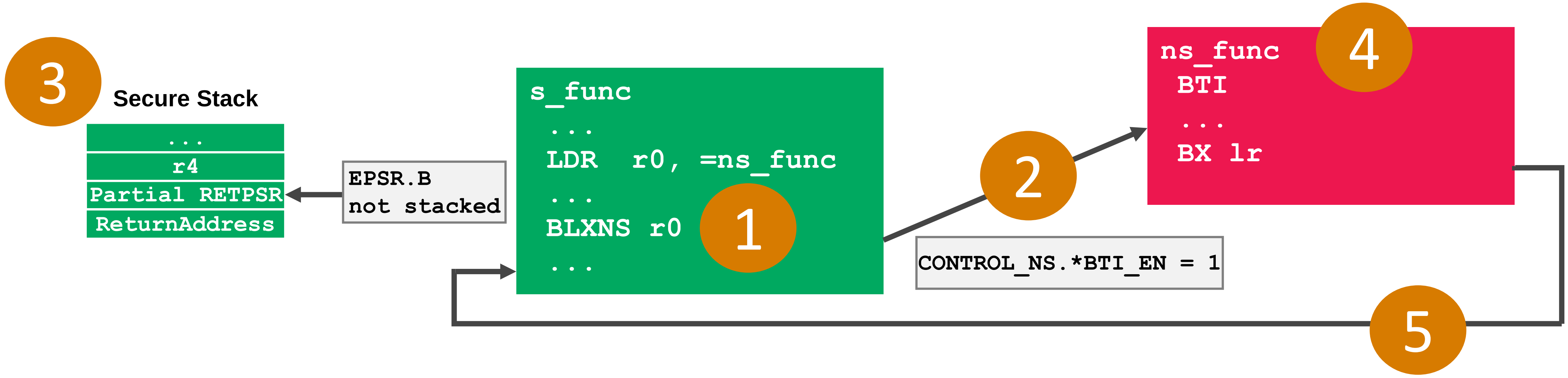
- EPSR.B = 0 ; indicates that Branch Target identification is inactive
- **BTI clearing instructions include**
    - **BTI**
    - **SG (Secure Gateway)**
    - **PACBTI (Combining Return Address Signing and Landing Pad)**

When EPSR.B bit is set to one via BTI setting instruction, then next executed instruction **must** be a BTI clearing instruction

- INVSTATE Usage Fault is triggered otherwise

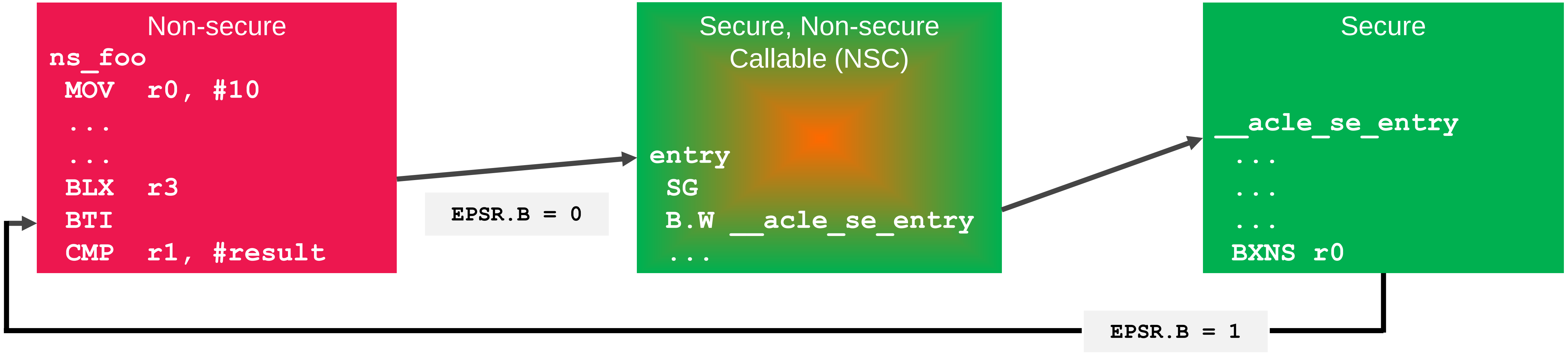
EPSR.B is automatically stacked and cleared to zero on exception entry and restored on exception return.

Branch Target Identification – Across security states



Assumption for this example: BTI is enabled in both Secure and Non-secure state

Branch Target Identification – Across security states



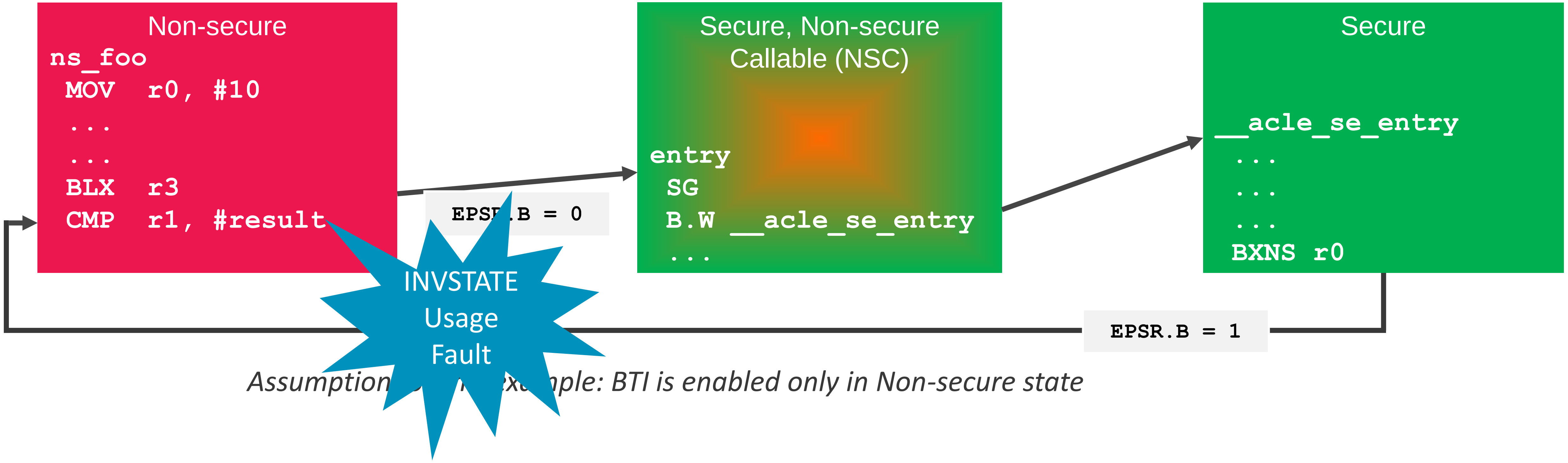
```
ns_foo
 MOV  r0, #10
 ...
 ...
 BLX  r3
 BTI
 CMP  r1, #result
```

```
entry
 SG
 B.W __acle_se_entry
 ...
```

```
__acle_se_entry
 ...
 ...
 ...
 BXNS r0
```

Assumption for this example: BTI is enabled only in Non-secure state

Branch Target Identification – Across security states

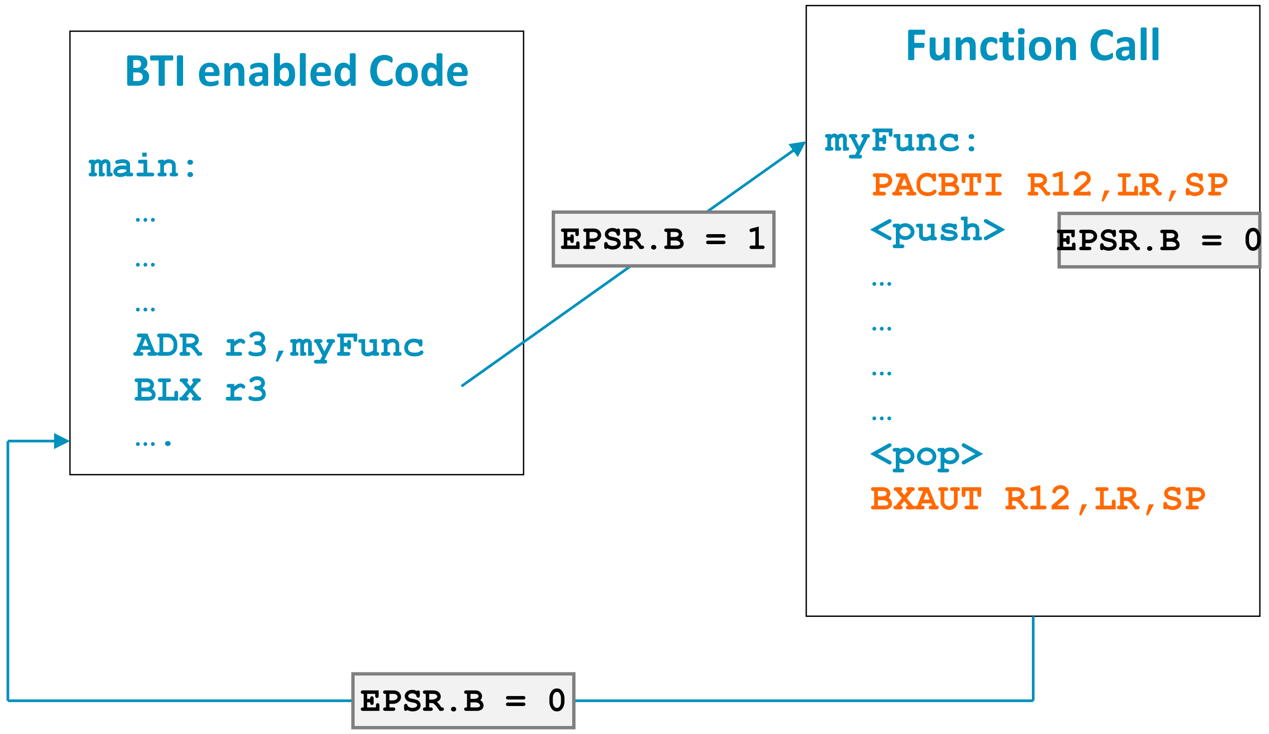


Assumption [illegible] example: BTI is enabled only in Non-secure state

Branch Target Identification – with PAC

- PACBTI instruction acts as a BTI clearing instruction along with computing PAC

- BXAUT instruction should not be used, if a program image need to be run on a system that does not support PACBTI extension



BTI enabled Code

```
main:
   …
   …
   …
   ADR r3,myFunc
   BLX r3
   ….
```

Function Call

```
myFunc:
   PACBTI R12,LR,SP
   <push>
   …
   …
   …
   …
   <pop>
   BXAUT R12,LR,SP
```

Branch Target Identification – with implied branches

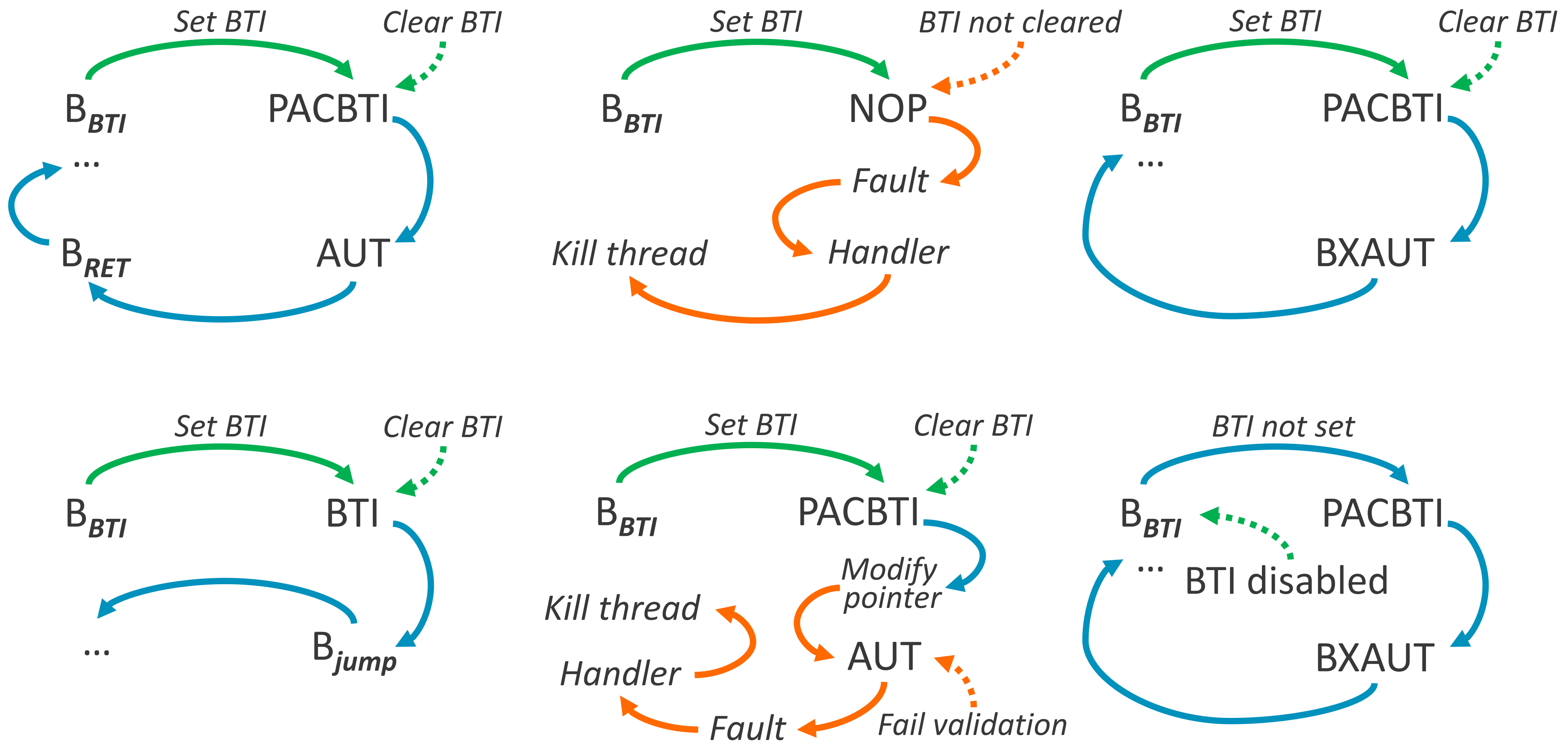
Branch Future (BF<c> <b_label>, <label>) notifies the PE of an upcoming branch to <label>

BTI			Upcoming Branch	BF	Jump addr	Valid

- BF initializes LO_BRANCH_INFO with a label (BF branch point)
- LO_BRANCH_INFO.BTI is set to 1 when BFLX and BFX (BTI setting) instructions are executed
- EPSR.B = 1 when LO_BRANCH_INFO.BTI, LO_BRANCH_INFO.VALID bits are set to 1 and BF Branch is taken
- Any instruction operating on LO_BRANCH_INFO which isn't a BTI setting instruction (Eg: WLS/LE) will clear BTI bit in LO_BRANCH_INFO

```
main:
    LDR   R6,=Func
    BFX return, R6  ← Set BF
                      branch
    …                 point
return:  ←
    BX R6                       BF branch point
    …
Func:
    PACBTI R12,LR,SP
    …
    BXAUT R12,LR,SP
```

PACBTI Quick Summary



PACBTI - Creates pointer code and is a BTI landing pad

AUT - Validates pointer code and detects tampering

BXAUT - Validate pointer and return

BTI - BTI landing pad instruction

B_{BTI} - BTI setting branch

B_{RET} - Return from function

B_{jump} - Branch to code

arm

Debugging PAC and BTI

Debugging PAC and BTI

A debugger can access PAC keys using Debug Core Register Data Register(DCRDR) and Debug Core Register Selector Register (DCRSR) register mechanism

- Based on DHCSR.S_SUIDE and DHCSR.S_NSUIDE bit configurations, an unprivileged debugger access has restrictions on accessing any of:
  - PAC keys registers
  - CONTROL.{PAC_EN,UPAC_EN,BTI_EN, UBTI_EN}

Single stepping BTI setting instructions will cause EPSR.B bit to be set

- On a Halting Debug single step, EPSR.B bit can be read by external debugger
- On a Monitor Debug single step, EPSR.B bit is stored on to stacked XPSR

Hardware Breakpoint or a BKPT instruction at the branch target address of a BTI setting instructions will not clear EPSR.B bit

Additional Reading

Providing protection for complex software

https://developer.arm.com/architectures/learn-the-architecture/providing-protection-for-complex-software

Pointer Authentication on ARM

https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה