



Armv8.1-M PACBTI Extensions

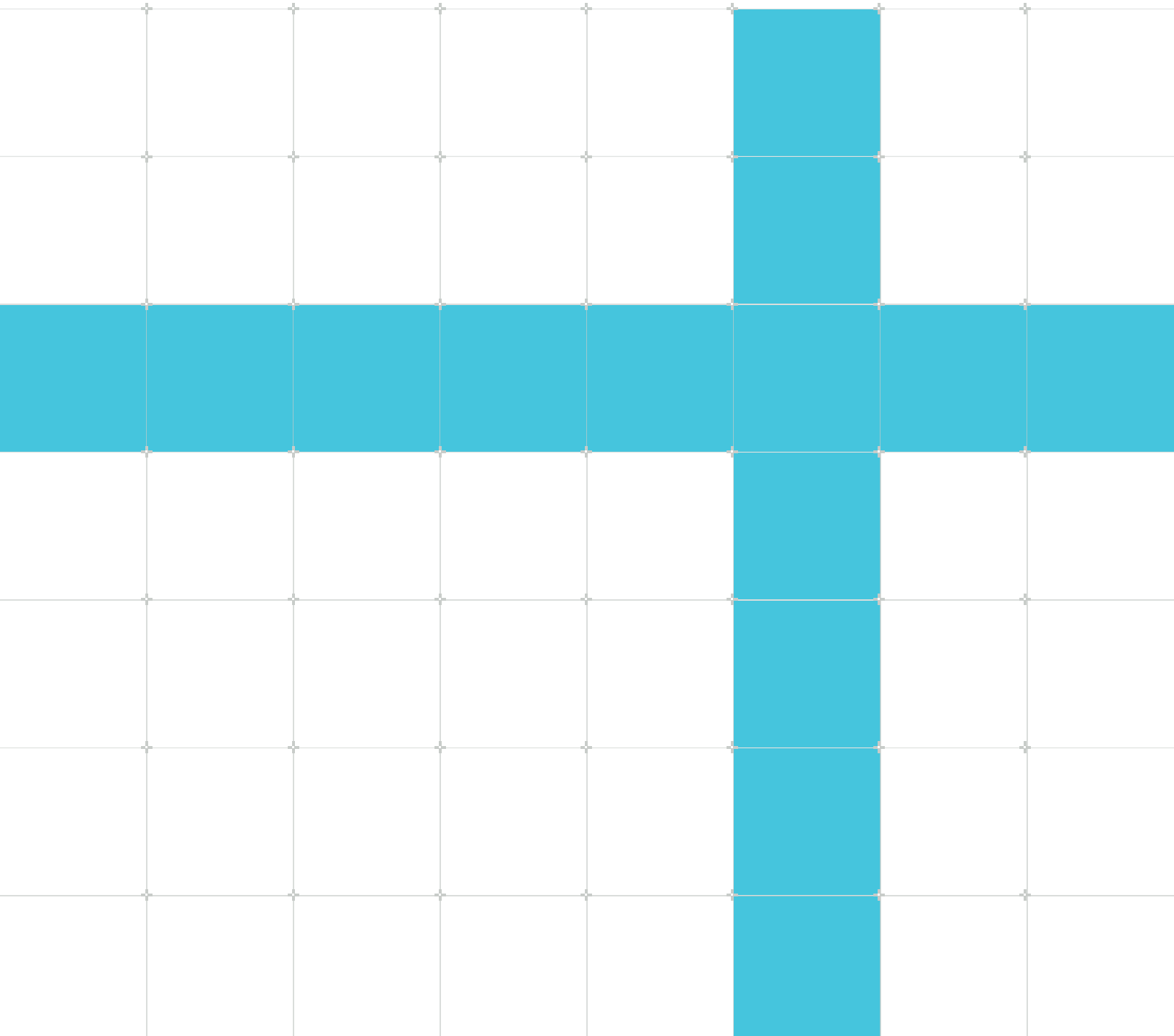
Version 1.0

Non-Confidential

Copyright © 2024 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

109576_0100_01_en



Armv8.1-M PACBTI Extensions

Copyright © 2024 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

| Issue | Date | Confidentiality | Change |
|---------|---------------|------------------|-----------------|
| 0100-01 | 12 March 2024 | Non-Confidential | Initial release |

Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm’s view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm's trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

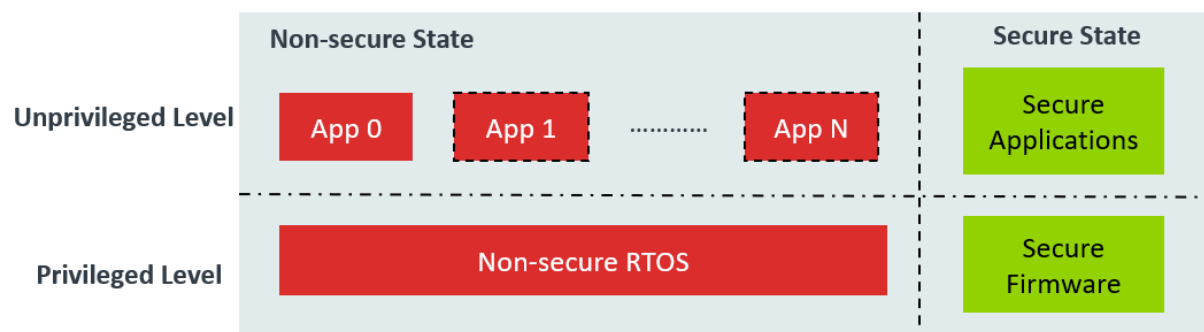
| | |
|---|-----------|
| 1. Introduction..... | 6 |
| 2. Pointer Authentication Code..... | 8 |
| 2.1 Introduction to PAC..... | 8 |
| 2.2 Cryptographic algorithm..... | 9 |
| 2.3 Key management..... | 9 |
| 2.4 Instructions..... | 12 |
| 2.5 Enabling pointer authentication..... | 14 |
| 3. Branch Target Identification..... | 16 |
| 3.1 Landing pads..... | 16 |
| 3.2 Enabling BTI..... | 17 |
| 3.3 How is BTI implemented?..... | 17 |
| 4. Using PAC and BTI features in applications..... | 20 |
| 4.1 Security transitions..... | 20 |
| 4.2 Examples..... | 21 |
| 4.2.1 Simple function..... | 21 |
| 4.2.2 Protecting memory copy functionality..... | 22 |
| 4.3 Debug feature interactions with the PACBTI extensions..... | 22 |
| 5. Tools and software support..... | 24 |
| 5.1 Compiler options..... | 24 |
| 5.2 Impact of PAC and BTI on software libraries..... | 25 |
| 6. Performance..... | 26 |
| 7. Differences between PAC/BTI in Cortex-M and Cortex-A..... | 27 |
| 8. References..... | 28 |
| 9. Appendix..... | 29 |
| 9.1 Return-Oriented Programming..... | 29 |
| 9.2 Jump-Oriented Programming..... | 30 |

1. Introduction

The [Armv8-M architecture](#) provides various security features, including the following:

- TrustZone for Armv8-M isolates security-critical code and resources from normal applications.
- Privilege levels isolate the OS from application tasks.
- The Memory Protection Unit (MPU) provides isolation between application tasks by configuring different memory regions.

Figure 1-1: Security states and privilege levels



The isolation mechanism given in the figure [Figure 1-1: Security states and privilege levels](#) on page 6 ensures that software components can only access the resources that they are authorized to and reduces the impact of software security vulnerabilities. However, we want to prevent a software vulnerability being exploited in the first place. To achieve this, Arm has added other security features to the Arm architecture.

There are many ways of attacking a processor system through the stack memory system. If code processes a data array and the array index is not being tested properly, an attacker can compromise the system if the array index can be manipulated beyond the programmer's intent, for example by stack corruption. Because stack memory can contain data from an external environment, an attacker can combine multiple attack methods to inject and execute malicious code. The Armv8-M architecture provides the eXecute Never (XN) feature to help prevent code injection attacks, by marking the stack pointer marked as XN.

On systems where the attacker cannot directly inject code (for example, because the XN attribute has been used), attackers can use code re-use attacks such as Return-Oriented Programming. In these attacks, snippets of existing code, called gadgets, are pieced together to perform the operation the attacker wants. This often involves corrupting return addresses to change the control flow by jumping into the middle of functions.

To limit the scope of these security exploits, Armv8.1-M architectures introduces the following two new features:

1. Pointer Authentication Code (PAC)

2. Branch Target Identification (BTI)

We refer to these two new concepts as the PACBTI optional extension in the Armv8.1-M architecture. These features establish valid entry points for indirect branches and authenticate the contents of a register before that register is used as the target of an indirect branch. You can also use the generic variants of the PAC instructions to authenticate critical data values or addresses before they are used. Dedicated new instructions are introduced for computing and validating pointers along with branch target identification.



PAC and BTI are not completely new concepts. PAC was introduced in [Armv8.3-A](#) and BTI was introduced [Armv8.5-A](#). The PAC and BTI features in Armv8.1-M are based on the same concepts. However, the low-level details are different. See [Differences between PAC/BTI in Cortex-M and Cortex-A](#) for a comparison between PAC and BTI in Cortex-M and Cortex-A processors. For more information about PAC and BTI in Cortex-A, see [Providing protection for complex software](#).

2. Pointer Authentication Code

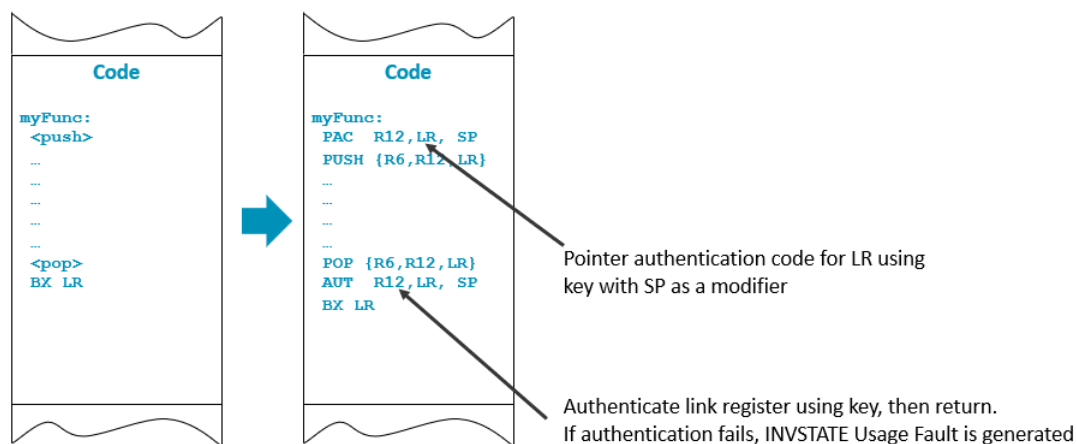
This chapter describes the Pointer Authentication Code (PAC) feature of the Armv8.1-M architecture.

2.1 Introduction to PAC

The pointer authentication instructions in the Armv8.3-A architecture were introduced as a software security countermeasure for Return-Oriented Programming (ROP) attacks. The Armv8.1-M architecture uses this concept for small 32-bit microcontrollers.

One of the key uses of the PAC feature is to detect corruption of return addresses in function calls. Because the return address is often stored on the stack, there is a risk that stack corruption can change the return address, allowing an attack to be carried out. The following figure shows a high-level overview of how you can use the PAC feature to protect systems:

Figure 2-1: Using PAC to sign and authenticate the pointer



The process is as follows:

1. In the function prologue, the `PAC` instruction creates a PAC for the LR using the SP as a modifier, and stores the PAC in register R12.
2. At the end of the function, the `AUT` instruction authenticates the LR using the PAC in register R12.

At the authentication stage, if the value of the return address (LR) or the SP do not match the original values used to generate the PAC, then the PAC calculated by the `AUT` instruction does not match the value in R12. This results in a fault exception. This pointer authentication mechanism also fails if the PAC value in R12 is corrupted.

The PAC is computed using a cryptographically strong algorithm. The cryptography key is protected and is unknown to the attacker. Also, the nature of the cryptographic algorithm makes it difficult for an attacker to decode the key by reading authenticated pointers and PACs from memory. Because the keys cannot be determined, the attacker cannot create new PAC codes to validate corrupted return addresses. As a result, PAC significantly reduces the likelihood of undetected return address corruption.

In addition to protecting return addresses in function calls, you can also use the PAC feature for signing and authenticating other pointer values.

The PAC functionality has three key elements:

- [Cryptographic algorithm](#)
- [Key management](#)
- [Instructions](#)

2.2 Cryptographic algorithm

Cryptography is a vital part of Pointer Authentication functionality. To compute the encrypted PAC value using `PAC*` instructions, the Armv8.1-M PACBTI optional extension provides a choice of either the [QARMA algorithm](#) or an **IMPLEMENTATION DEFINED** algorithm. For more details, see the [Armv8-M Architecture Reference Manual](#).

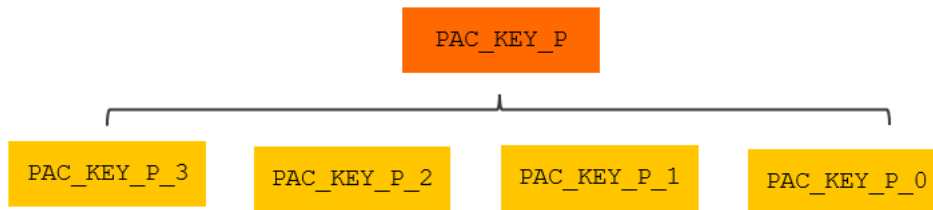
2.3 Key management

The following table shows the details where, the architecture provides four 128-bit keys for pointer authentication one for each security and privilege state:

Figure 2-2: PAC keys

| | Non-Secure State | Secure State | |
|--------------|------------------|--------------|--|
| Privileged | PAC_KEY_P_NS | PAC_KEY_P_S | } Both Privileged and Unprivileged keys are Privileged access only |
| Unprivileged | PAC_KEY_U_NS | PAC_KEY_U_S | |
| | | | Secure access only |

The following figure shows the details where four 32-bit register values are concatenated to form a single PAC_KEY. Each 32-bit value that forms a key value is part of the internal system registers, and can therefore be accessed using `MRS` and `MSR` instructions.

Figure 2-3: PAC_KEY_* register concatenation

When using the PAC feature, software developers need to consider the following issues:

- Where do the PAC keys come from?
- Is the key's confidentiality important?
- How many keys are needed?

To answer these questions, software developers need to define the following:

- What are the aims of deploying PAC for their projects
- The software architecture of their project, for example are there multiple security domains in the software?
- Threat model
- Value of the system
- Other technical constraints

The aims of deploying PAC usually fall into two common categories:

1. For security, detecting software-based attacks such as Return Oriented Programming (ROP)
2. For enhancing failure detection in systems that require a high level of robustness

For the second aim, failure detection, the requirements for PAC key management can be much simpler. For example:

- You can hard-code the PAC keys used in the application code. There is no need to keep the PAC key confidential
- You can share a PAC key between different types of software

However, when PAC is used as a security feature, the PAC keys are confidential and need to be protected. If an attacker is able to determine the PAC keys being used, then they can create fake return stack frames that can bypass the PAC protection feature.

As a result, the most basic recommendations of PAC keys are as follows:

- Do not hardcode the value of the PAC keys into the program image



If an attacker can read the key in the program image, for example by examining firmware update images or using the debug interface, they can then bypass the PAC protection in all devices that use the same firmware.

- The value of the keys should not be easily determined, for example from the product serial number
- PAC key values should be stored in memory that is privileged access only. This includes retention SRAM if it is used for holding PAC key values if the processor logic is powered down during sleep

You can use the random number generator to create keys. Also, the following general IoT security practices are also relevant to PAC keys:

- Create a new key for each thread and swap keys on every context switch
- If the software is split into processes with different permissions, each process can have its own key. Threads belonging to the same process can share keys. The keys are swapped on every context switch
- All unprivileged code shares the same key. There is no need for the keys to be swapped after initial setup
- Avoid reusing a key

The same key must be used for the entire lifetime of a thread so that all the return addresses on the stack can be validated.

To further reduce security risks, you must disable the debug access to the memories. This prevents attackers from modifying or examining the stack contents.

The next important question is how many PAC keys are needed. Generally, it is expected that:

- Software running in the Secure world and Non-secure world should use different PAC keys
- Baremetal applications with no embedded OS, or that have a single privilege domain with no process isolation within the OS, can use a single key

For systems running an RTOS with process isolation, we expect that:

- As a minimum, privileged and unprivileged software components need to have different keys
- For best security, each unprivileged software context should have its own key

However, there are disadvantages of using more keys:

- In an ideal world, each of the PAC keys would be random values. However, generating keys using a True Random Number Generator (TRNG) can take some time and lead to performance issues if all PAC keys are directly generated by a TRNG. Alternatively, you can use a TRNG (or an alternative source of randomness if no TRNG is available) to seed a Pseudo Random Number Generator (PRNG), and then use the PRNG to generate multiple PAC keys. In the extreme case where a device does not contain a TRNG or other source of random values at all, you can use a

non-volatile memory location to store a device-unique entropy value to seed the PRNG. Always avoid hardcoding the seeds in the program image

- When different application threads in an RTOS environment have different key values, the RTOS context switching operations must also update the PAC key registers according to the context. This adds to the context switching overhead. If all the unprivileged threads use the same key, the unprivileged PAC key can be configured at system initialization and can remain static, simplifying the operation

Keys need to be stored securely to prevent leakage. As a result, the keys need to be stored in memory that is privileged access only. In some applications, the processor might need to be powered down during sleep to reduce sleep power, while the stack memories need to be retained. In this case, the PAC keys might also need to be saved in retention SRAM and protection arrangements put in place to prevent the keys from being read by unprivileged threads.

To help reduce the delay caused by key generation, an application can generate random values for PAC keys as a background operation and store them in writeable non-volatile memory. Next time the device is booted up, it can take the random values in the non-volatile memory to speed up boot time.

2.4 Instructions

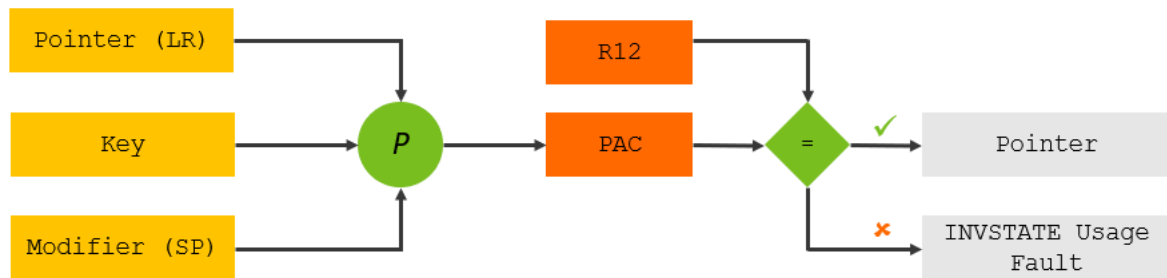
To perform pointer authentication, two types of instruction operate in pairs:

1. Obtain a PAC value using the signing process. The signing process is computed using a cryptographic algorithm. This is done using `PAC*` instructions, such as `PAC`, `PACG`, and `PACBTI`
2. Validate the pointer by authenticating the PAC. Validating the PAC and authenticating the pointer is done using `AUT*` instructions, such as `AUT`, `AUTG`, and `BXAUT`

The input arguments for `PAC*` instructions to compute the PAC are:

1. Cryptographic key register
2. Register that is used as a modifier
3. Register that contains the pointer

The `PAC*` instructions return a 32-bit value containing the pointer authentication code. The cryptographic key is selected based on the current security state and privilege level. On execution of an `AUT*` instruction, the 32-bit PAC value is validated and the pointer address is authenticated. If there is a failure during the validation process of the PAC, then an `INVSTATE` usage fault is triggered. The `INVSTATE` usage fault is also triggered when the input arguments used to compute the PAC are corrupted or if there is a change in the Security or privilege state of the PE before executing the corresponding `AUT` instruction. The following diagram shows an example of the authentication process using the instruction `AUT R12, LR, SP`:

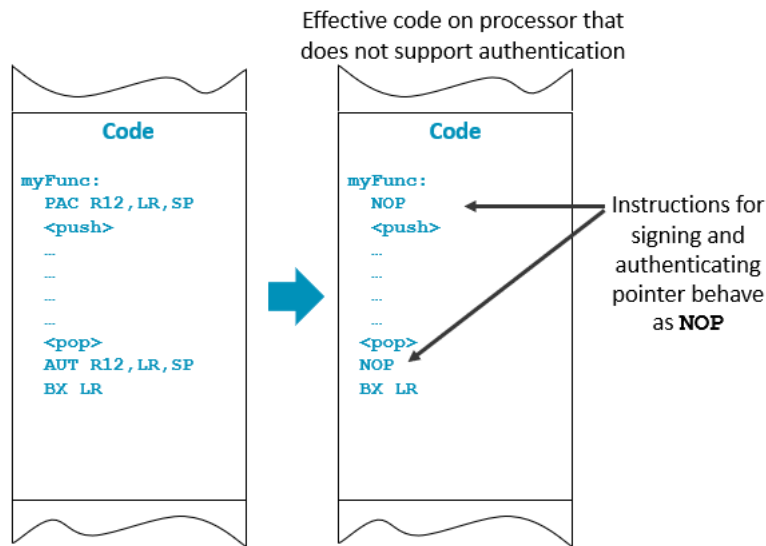
Figure 2-4: Authentication of pointer using AUT instruction**Example: AUT R12, LR, SP**

The following table lists the available pointer authentication instructions:

Table 2-1: Pointer authentication instructions

| Instructions | Description |
|--------------------|---|
| PAC R12, LR, SP | Compute PAC using LR as the address, SP as the modifier, and a key. The computed PAC is written to R12. |
| PACG Rd, Rn, Rn | Compute PAC using two input registers and a key. The computed PAC is written to the specified register Rd. |
| PACBTI R12, LR, SP | Compute PAC using LR as the address, SP as the modifier and a key, and also provide a validation BTI landing pad. |
| AUT R12, LR, SP | Authenticate LR using SP as the modifier. |
| AUT Ra, Rn, Rm | Authenticate Rn using Rm as the modifier. |
| BXAUT Ra, Rn, Rm | Branch and Exchange after authenticating Rn using Rm as the modifier. |

The pointer authentication instructions `PAC`, `PACBTI` and `AUT` are part of the **NOP**-compatible instruction encoding space. This means that code protected by these instructions still runs on older processors that do not support PAC features, treating the instructions as NOPs.

Figure 2-5: PAC backwards compatibility

There is no architectural restriction on the number of instructions that can be executed between the instruction that computes the PAC and the instruction that validates the PAC.



- In the general case, we expect that an `AUT` or `AUTG` instruction is executed as the penultimate instruction before function return. However, there might be some cases outside into this scenario. In these scenarios, Arm recommends that between the authentication of a pointer and any subsequent reference to that pointer, the unprotected pointer is not saved and restored from memory
- If the instruction that computes the PAC is executed as a part of an `IT` block, and there is a condition code fail within the `IT` block for it, then an `INVSTATE Usage` fault is triggered and the `IT` state is not advanced
- Instructions that access the PAC key registers such as `PAC*`, `AUT*`, `MRS`, and `MSR` must not have their timing dependent on the key values regardless of `AIRCR.DIT`

2.5 Enabling pointer authentication

When the PACBTI extension is implemented, the architecture provides individual control mechanisms to enable or disable PAC within a specific security or privilege level. The following table shows these control mechanisms:

Table 2-2: PAC control mechanisms

| Mode | Non-secure state | Secure state | Current Security state |
|--------------|---------------------------------|--------------------------------|------------------------------|
| Privileged | <code>CONTROL_NS.PAC_EN</code> | <code>CONTROL_S.PAC_EN</code> | <code>CONTROL.PAC_EN</code> |
| Unprivileged | <code>CONTROL_NS.UPAC_EN</code> | <code>CONTROL_S.UPAC_EN</code> | <code>CONTROL.UPAC_EN</code> |

The following diagram shows the CONTROL register assignment bits:

Figure 2-6: Extension of CONTROL register to support PACBTI extensions



CONTROL.UPAC_EN:

This bit acts as an unprivileged pointer authentication enable. When this feature is enabled the pointer authentication instructions can create and validate the PAC in unprivileged mode. This bit is banked between Security states. The possible values of this bit are:

- 0: Pointer authentication is disabled for unprivileged accesses
- 1: Pointer authentication is enabled for unprivileged accesses

CONTROL.PAC_EN:

This bit acts as a privileged pointer authentication enable. When this feature is enabled the pointer authentication instructions can create and validate the PAC in privileged mode. This bit is banked between Security states. The possible values of this bit are:

- 0: Pointer authentication is disabled for privileged accesses
- 1: Pointer authentication is enabled for privileged accesses

3. Branch Target Identification

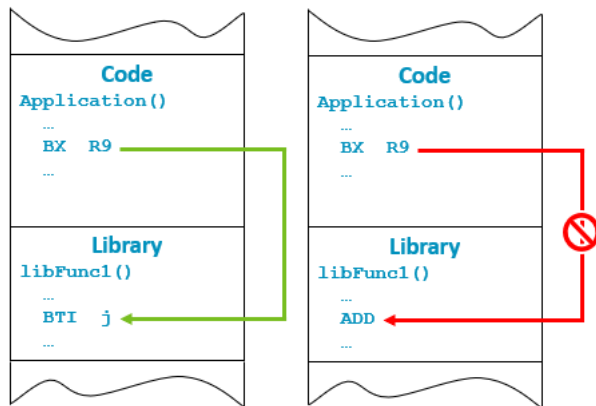
The pointer authentication instructions are a countermeasure against Return-oriented programming (ROP) attacks. Another code-reuse attack that is commonly used by attackers is [Jump-Oriented Programming](#).

To protect against JOP attacks, the architecture includes a new feature called Branch Target Identification (BTI). Branch Target Identification instructions (BTIs) are also called landing pads.

3.1 Landing pads

The mechanism used to create and identify valid branch landing pads is called Branch Target Identification (BTI). The processor can be configured in such a way that when BTI is enabled, all indirect branches must target landing pads. If the target address of the branch instruction does not have a landing pad, then the processor triggers an exception. This reduces the number of possible target addresses, and therefore reduces the number of possible gadgets that can be created using JOP, as shown in the following diagram:

Figure 3-1: When BTI is enabled, indirect branches must target a landing pad instruction



Valid landing pad instructions include `BTI`, `PACBTI`, and `sg`. For more information, see [How is BTI implemented?](#).

3.2 Enabling BTI

You can enable or disable Branch Target Identification (BTI) using the following settings in the CONTROL register, depending on state and mode of instruction execution:

Table 3-1: BTI settings in the CONTROL register

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

Figure [Figure 2-6: Extension of CONTROL register to support PACBTI extensions](#) on page 15 shows the CONTROL bit assignments.

CONTROL.UBTI_EN:

This bit acts as an unprivileged branch target identification enable. This bit is banked between Security states. The possible values of this bit are:

- 0: Branch target identification disabled for unprivileged accesses
- 1: Branch target identification enabled for unprivileged accesses

CONTROL.BTI_EN:

This bit acts as a privileged branch target identification enable. This bit is banked between Security states. The possible values of this bit are:

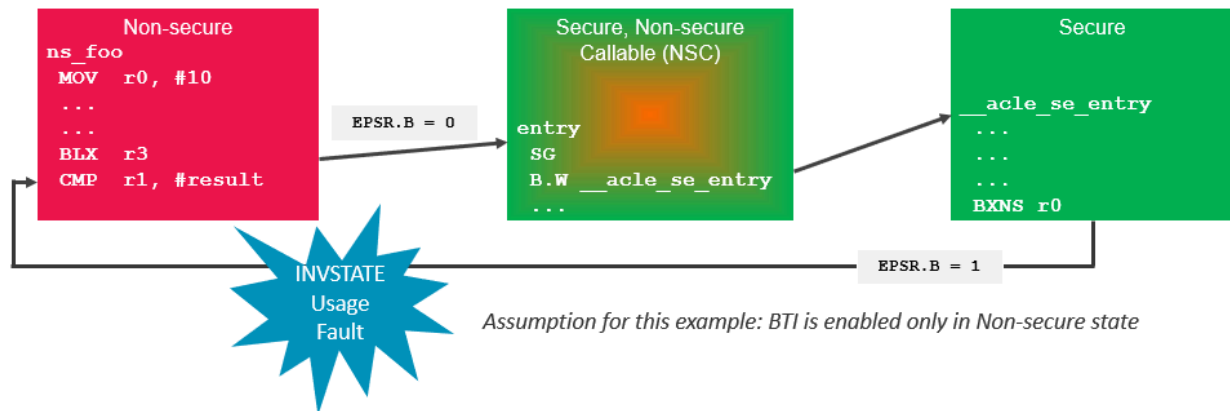
- 0: Branch target identification disabled for privileged accesses
- 1: Branch target identification enabled for privileged accesses

3.3 How is BTI implemented?

When BTI is enabled for the target state, then BTI becomes active when the EPSR.B bit is set to 1 by any one of the following branch instructions. These instructions are called BTI setting instructions:

- BLX and BLXNS instructions.
- BX and BXNS instructions, where the register holding the branch address is not LR
- LDR <register>, and LDR <literal> instructions, when the address is loaded into PC
- LDMIA, LDMDB, and LDR <immediate> instructions, when the address is loaded into PC without SP as base register, or SP as base register and without writeback operation

When the EPSR.B bit is set to 1 by a BTI setting instruction, it can be only cleared by a landing pad also known as a BTI clearing instruction. When the EPSR.B bit is set to 1 by a BTI setting instruction, then the next executed instruction must be a BTI clearing instruction or a landing pad. If this is violated, then the processor triggers an INVSTATE usage fault.

Figure 3-2: BTI settings across security states

The BTI clearing instructions are as follows:

- `BTI`
- `SG`
- `PACBTI`

After executing a BTI clearing instruction or landing pad, the `EPSR.B` bit is set to zero to indicate that Branch Target Identification is inactive.

If an interrupt or exception occurs between the BTI setting and BTI clearing instructions, then the `EPSR.B` bit is automatically stacked as a part of exception entry. When it enters the handler, the `EPSR.B` bit is cleared to zero to indicate a new context executing in handler mode.

- `BX LR` and `BXNS LR` instructions do not require a BTI clearing instruction at the branch target

This is because `BX LR` and `BXNS LR` are typically used for function returns. In a PACBTI system, if BTI is implemented then PAC is also expected to be implemented. This means that functions can be protected using PAC authentication. That is, a `BX LR` or `BXNS LR` uses an authenticated pointer and therefore a BTI clearing instruction or landing pad is not needed. Some instructions that compute the PAC also act as a BTI clearing instruction. This is beneficial for a function entry and return because the number of landing pads can be reduced, which improves code density and reduces the number of available gadgets.



Note

- The target of direct branch instructions does not need to be a landing pad

This is because the branch addresses are fixed and cannot be manipulated using stack attacks.

- The target of a `BXAUTH` instruction does not need to be a landing pad

This is because the address has already been authenticated as a part of the `BXAUT` operation.

4. Using PAC and BTI features in applications

The main objective of the Armv8.1-M PACBTI extension is to enable software developers to write software which reduces exposure to typical ROP and JOP attacks. All the features included in this extension are designed to limit the number of available gadgets where the compiler plays a key role in providing precise support.



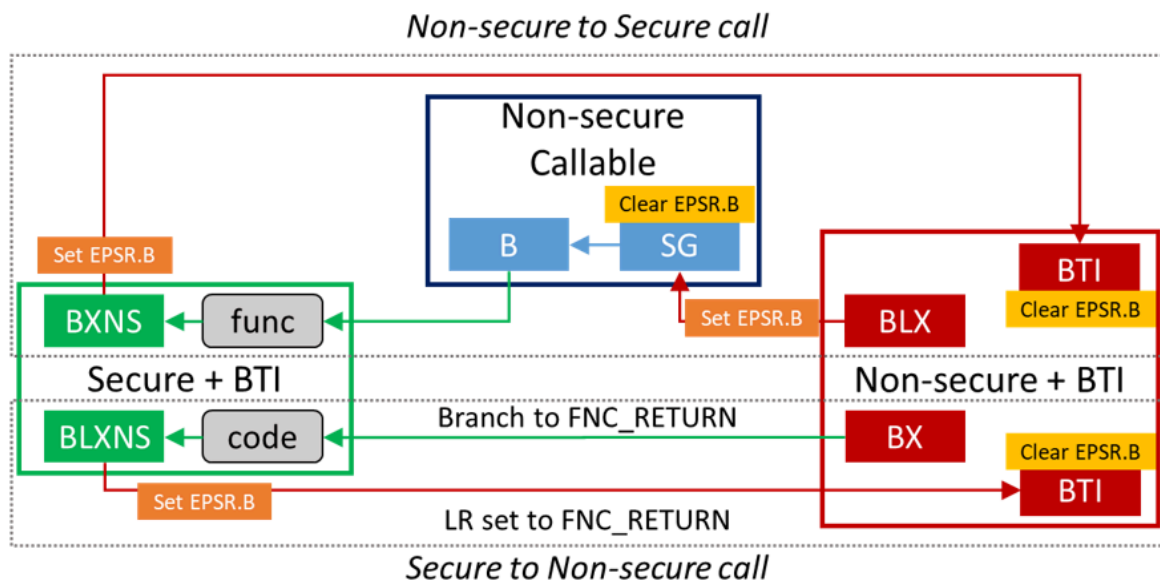
If you require a handwritten assembly code, then it is programmer's responsibility to adhere to the architecture rules.

4.1 Security transitions

The PACBTI extension introduces separate controls for enabling BTI in every Security and privilege domain. For example, user code compiled without BTI can call into a Secure library which is protected by BTI.

The following figure shows the security state transition model when BTI is implemented:

Figure 4-1: Security state transitions with PACBTI extensions



As a part of security state transitions, here are some more possible scenarios:

- Non-secure software supports BTI, but the Secure software does not
 - If the Non-secure code calls a Secure function with a `BLX` instruction, which is BTI setting, then the `SG` instruction will automatically clear `EPSR.B`, even though Secure software does not support BTI
- Secure software supports BTI, but the Non-secure software does not
 - If the Secure code calls a Non-secure function using a `BLXNS` instruction, which is BTI setting, then the `BLXNS` instruction checks the BTI enables for the Non-secure side. Because `CONTROL_NS.BTI_EN` reads as zero, the Secure `BLXNS` does not set `EPSR.B`, and the target Non-secure instruction does not need to be BTI clearing

4.2 Examples

The examples used for demonstrating pointer authentication can be further strengthened by including BTI protection.

4.2.1 Simple function

Adding a BTI at the start of the function ensures that even if an attacker can manipulate a pointer, then jumping into the middle of `func` fails because the rules for BTI are violated and the PE generates an exception.

Figure 4-2: Function with and without PACBTI extensions

| Original | Protected using PAC |
|--|---|
| <pre>func: PUSH {R4-R6, LR} ... // Function body POP {R4-R6, LR} BX LR</pre> | <pre>func: BTI PAC R12, LR, SP PUSH {R4-R6, R12, LR} ... // Function body POP {R4-R6, R12, LR} AUT R12, LR, SP BX LR</pre> |

Simple function with and without PACBTI extension

You can improve this code significantly by substituting the individual `BTI` and `PAC` instructions with the combined `PACBTI` instruction. You can also use the combined `BXAUT` instruction instead of individual `AUT` and `BX` instructions. This is shown in following figure:

Figure 4-3: Function call with PACBTI and BXAUT instructions

| Original | Protected using PAC |
|--|---|
| <pre>func: PUSH {R4-R6, LR} ... // Function body POP {R4-R6, LR} BX LR</pre> | <pre>func: PACBTI R12, LR, SP PUSH {R4-R6, R12, LR} ... // Function body POP {R4-R6, R12, LR} BXAUT R12, LR, SP</pre> |

Function call with PACBTI and BXAUT instructions

4.2.2 Protecting memory copy functionality

Similar to the example in section [Simple function](#), you can boost the robustness of memcpy by ensuring a single valid entry point into the function.

Figure 4-4: Memcpy function with and without PACBTI extensions

| Original | Protected using PAC |
|---|--|
| <pre>memcpy: PUSH {R0, LR} WLS LR, R2, loopEnd loopStart: LDRSB R3, [R1], #1 STRB R3, [R0], #1 LE LR, loopStart loopEnd: POP {LR} BX LR</pre> | <pre>memcpy: PACBTI R12, LR, SP PUSH {R0, LR} WLS LR, R2, loopEnd loopStart: LDRSB R3, [R1], #1 STRB R3, [R0], #1 LE LR, loopStart loopEnd: POP {LR} BXAUT R12, LR, SP</pre> |

Memcpy function with and without PACBTI extension

4.3 Debug feature interactions with the PACBTI extensions

You need to update the debug tools to support the PACBTI extension:

- Additional special registers are added in the processor core. This affects the core registers debug access mechanism, for example the definition of the Debug Core Register Select Register
- During single stepping, if the software developer changes the program counter value immediately after a BTI setting instruction so that the program counter is not pointing to a landing pad, a fault is triggered. A debug tool might optionally offer clearing of the BTI bit in the EPSR when changing the program counter while the BTI bit is set

Debug authentication settings can affect access permission to PAC keys. For example, an external debugger can access PAC keys depending on the DAUTHCTRL and DHCSR register control settings as per following table.

Table 4-1: DAUTHCTRL and DHCSR register control settings

| Current Security | S_SDE | S_SUIDE | S_NSUIDE | Access Current PAC_KEY | Access Secure PAC_KEY | Access Non-secure PAC_KEY |
|------------------|-------|---------|----------|------------------------|-----------------------|---------------------------|
| NS | - | - | 0 | Y | N | Y |
| NS | - | - | 1 | N | N | N |
| NS | - | - | 0 | Y | N | Y |
| NS | - | - | 1 | N | N | N |
| NS | - | - | 0 | Y | N | Y |
| NS | - | - | 1 | N | N | N |
| NS | - | - | 0 | Y | N | Y |
| NS | - | - | 1 | N | N | N |
| S | 0 | 0 | 0 | Y | N | Y |
| S | 0 | 0 | 1 | Y | N | N |
| S | 0 | 1 | 0 | N | N | Y |
| S | 0 | 1 | 1 | N | N | N |
| S | 1 | 0 | 0 | Y | Y | Y |
| S | 1 | 0 | 1 | Y | Y | N |
| S | 1 | 1 | 0 | N | N | Y |
| S | 1 | 1 | 1 | N | N | N |

5. Tools and software support

This chapter gives a brief overview of compiler options and software support for the Armv8.1-M PACBTI extensions.

5.1 Compiler options

Arm Compiler 6 supports the PACBTI extension in Armv8.1-M with two control-flow integrity approaches:

- Return address signing and authentication as a mitigation for ROP-style attacks
- BTI instruction placement as a mitigation for JOP-style attacks

The following command-line options have been added for PAC and BTI for Armv8-A and will work for Armv8.1-M:

1. `-mbranch-protection=<opt>`

Enables branch protection, depending on `<opt>`:

- `none`: disable all forms of branch protection. This is the default.
- `pac-ret`: enable return address sign and authentication.
- `bti`: enable placement of BTI instructions.
- `standard`: equivalent to `pac-ret+bti`.

This is the same option as for AArch64 PAuth/BTI, with the same syntax.

2. `-march=...+pacbti`

Enables the PACBTI extension for Armv8.1-M. This affects the choice of instructions, emitted when branch protection is enabled using the `-mbranch-protection=...` option. If the PACBTI extension for Armv8.1-M is enabled, the compiler is allowed to use non-**NOP**-space instructions. Otherwise the compiler uses only **NOP**-space instruction. Currently the compiler always uses **NOP**-space instructions. For example, it does not use `bxaut`. In addition to the compilation option, C runtime libraries are also be updated and software developers need to ensure that correct runtime libraries are used for linkage. For more information on using PACBTI with Arm Compiler 6, see [Armv8.1-M PACBTI extension mitigations against ROP and JOP style attacks](#)



IAR Arm Compiler supports PACBTI from version 9.40. GCC supports PACBTI from GCC 13.

5.2 Impact of PAC and BTI on software libraries

If you use PAC or BTI features, you should update the initialization code for system startup for both Secure and Non-secure boot code. You should also update the CONTROL register configuration.

If you use BTI feature, software developers must ensure that legacy libraries are recompiled to support BTI. While you can execute legacy libraries that was compiled without PAC in a PAC-enabled system, for best security we recommend to recompile the code.

If you use PAC or BTI in the Secure firmware and multiple Secure software partitions are presented, the context switching code sequence of Secure software partitions requires the Secure CONTROL register to be updated. Each unprivileged Secure partition can use its own cryptography keys for PAC operations.

Similarly, RTOS code that handles context switching also needs to be modified to update PAC and BTI settings for each new context it is switching into, unless all application threads use the same settings.

6. Performance

Using the PAC and BTI protection mechanism impacts processor's performance. When PAC is enabled, instead of using a `POP { ..., PC}` instruction to retrieve the return address from the stack and return, you need a multi-step procedure:

1. Stack pop to restore registers
2. Authentication
3. Function return



If the software does not need to be backward compatible to processors that do not support PACBTI, you can combine steps 2 and 3 into a single instruction using `BXAUT`.

This function return overhead, combined with the and pointer signing required at the beginning of a function call, can cause each function call to take several additional clock cycles.

You can reduce the performance penalty using the following strategies:

- Function inlining. This reduces the number of function calls and function returns during program operations
- Using high optimization levels when compiling. Some of the Link Time Optimization (LTO) techniques such as function tail-chaining reduce the number of function returns

7. Differences between PAC/BTI in Cortex-M and Cortex-A

The concept of Pointer Authentication (PAC) and Branch Target Identification (BTI) was originally introduced in the Armv8.3-A and Armv8.5-A architectures for Cortex-A. When incorporating these concepts into the Armv8.1-M architecture for Cortex-M, there were a considerable number of changes to make it suitable for the microcontroller market.

The following table highlights the key differences between the PAC and BTI features in the Cortex-M and Cortex-A architectures.

Table 7-1: PAC and BTI differences between the Cortex-M and Cortex-A architectures

| Cortex-A | Cortex-M |
|---|---|
| Computed PAC is saved to the upper 12 bits of a 64-bit virtual address | Computed PAC is saved to a register (R12 or general-purpose register), not added to the pointer address |
| Five 128-bit keys are provided. Each key is formed by concatenating two 64-bit system registers | Four 128-bit keys are provided. Each key is formed by concatenating four 32-bit system registers |
| Two keys (A and B) for instructions, two keys (A and B) for data pointers, and one key for general purpose are provided separately. | Since the register can contain either data or instruction pointer, PAC key registers are not separated by instruction and data pointers. Instead, PAC key registers are segregated based on security state and privilege level. |
| Instruction and data pointer authentication is enabled using SCTL | Pointer authentication is enabled using the CONTROL register, with individual bit settings for each security state and privilege level combination. |
| On executing an indirect branch, the type of indirect branch is stored in PSTATE.BTYPE. | Only a specific set of branch instructions are termed as BTI setting instructions that set the EPSR.B bit. |
| The architecture distinguishes between indirect branches that use X16 or X17 | Indirect branches are not distinguished. |
| Support for landing pads is enabled for each page, using a new bit (GP bit) in the translation tables | Because there is no concept of MMU, there is no specific architecture update related to how memory is partitioned. |

8. References

Here are some resources related to material in this guide:

- [Armv8-M Architecture Reference Manual](#)
- [Armv8-A architecture: 2016 additions](#)
- [Arm A-Profile Architecture Developments 2018: Armv8.5-A](#)
- [Providing protection for complex software](#)
- [Pointer Authentication on ARM](#)
- [The QARMA Block Cipher Family](#)
- [PACBTI Blog](#)
- [Better Security at the Flick of a \(Compiler\) Switch: Enabling Pointer Authentication and Branch Target Identification](#)

9. Appendix

This section provides additional reading about Return-oriented Programming and Jump-oriented Programming techniques.

9.1 Return-Oriented Programming

To protect against stack buffer overflow attacks and stack smashing attacks, the architecture provides Memory Protection Unit (MPU) features including the following:

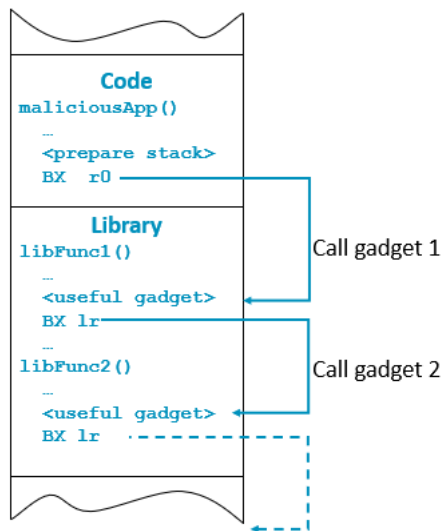
- eXecute Never (XN) attribute
- Privilege eXecute Never (PXN)

Restricting memory execution permissions makes it more difficult for an attacker to inject and execute their own arbitrary piece of code. In such cases, attackers might try to use an exploitation technique called Return-Oriented Programming (ROP).

In a ROP attack, instead of executing an arbitrary piece of code inserted by the attacker, the attacker tries to exploit small instruction sequences that are already present in the binary or linked libraries to form a chain of instructions dictated by the attacker. These small instruction sequences that are already present in the application are called gadgets. An attacker analyzes the software in a system, looking for useful gadgets, usually ending with a function return, for example:

```
...  
ADD r0, r1, r2  
BX LR
```

This code provides a gadget for adding two registers together. By scanning all the available libraries, an attacker can build a library of gadgets. These gadgets are existing legal code, within executable regions. This means that they are not affected by protections such as execution permissions. The attacker strings together a chain of gadgets, forming what is effectively a new program made up of existing code fragments. The following figure shows an example of how this can be achieved:

Figure 9-1: Return-Oriented Programming (ROP)

Any library that is available in the address space for the process is a potential source of gadgets. For example, the C library contains many functions, each offering potential gadgets. With so many gadgets available, statistically enough gadgets are available to form any arbitrary new program. Some compilers are even designed to compile to gadgets, rather than an assembler. A ROP attack is effective because it is made up of existing legal code, so it is not trapped by execution permissions or checks on executing from writable memory.

It is time-consuming for an attacker to find gadgets and create the sequence that is necessary to produce a new program. However, this process can be automated and can be reused to attack multiple systems.

9.2 Jump-Oriented Programming

To understand why the Armv8.1-M architecture introduced the BTI feature, it is useful to understand a brief overview of Jump-Oriented Programming (JOP).

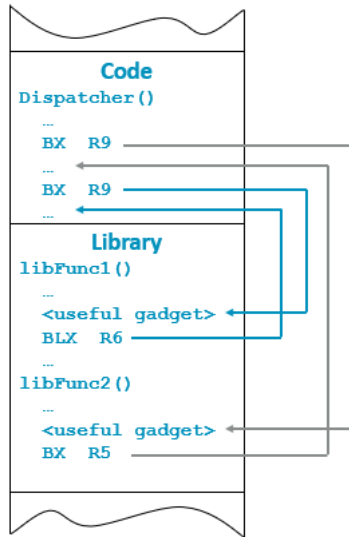
Like ROP, JOP is also a code-reuse attack that uses a chain of gadgets. However, JOP uses gadgets that end with an indirect branch instruction, for example `BLX <reg>`, rather than a function return `BX LR` instruction.

A typical JOP attack involves the following three elements

- Functional gadgets: These are blocks of code that have some effect on state
- Dispatcher table: Holds the addresses of functional gadgets to be executed
- Dispatcher gadget: Iterates through the dispatcher table and runs the functional gadget that the current entry points to

An example JOP attack is shown in the below figure:

Figure 9-2: Jump-Oriented Programming (JOP)



The attacker exploits the fact that **BLX** and **BX** instructions can target any executable address, not just addresses that are entry points defined by the compiler or developer. This means that the instructions can be hijacked to string gadgets together to perform the attack.