arm

Learn the architecture - Providing protection for complex software

Version 1.0

Learn the architecture - Providing protection for complex software

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	8 January 2020	Non-Confidential	Initial release
0100-02	30 June 2022	Non-Confidential	Minor bug fix in Return-oriented programming.

Proprietary Notice

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. Overview

This guide introduces some common forms of attacks that are used against complex software stacks. The guide also examines the features, including pointer authentication, branch target identification and memory tagging, that are provided in Armv8-A to help mitigate against such attacks. The guide is an overview of these features, and not a technical deep dive. You can use the Related information section to explore some topics in this guide in more detail.

At the end of this guide, you will be able to:

- Define the terms Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP).

- List the features in Armv8-A that help protect against ROP and JOP attacks.

- Describe how memory tagging can be used to detect memory safety violations, like buffer overruns or use-after-free.

Before you begin

We assume that you are familiar with the Arm memory model. If you are not, you might want to first read our Memory model and Memory management guides.

If you are not familiar with security, we also recommend that you read our Introduction to security guide before reading this guide.

2. Stack smashing and execution permissions

One of the oldest forms of attack is stack smashing. There are many types of stack smashing. The basic form of stack smashing involves malicious software writing new opcodes into memory and then attempting to execute the written memory. This process is illustrated here:

Figure 2-1: Stack smashing process
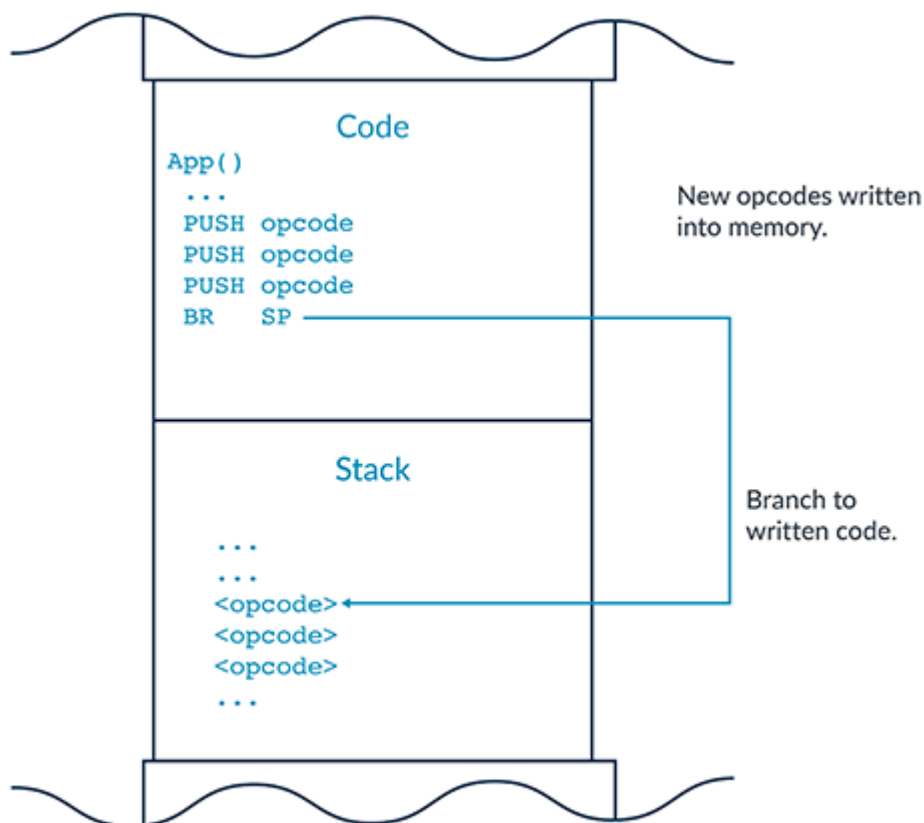
Typically, the memory that is used to launch the attack is stack memory. This is where the name stack smashing comes from. To protect against stack smashing, modern processor architectures, like the Arm architecture, have execution permissions. In Armv8-A, the main controls are execution permission bits in the translation tables. If we focus only on EL0 and EL1:

UXN

User (EL0) Execute-never

PXN

Privileged Execute-never

Setting one of these bits marks the page as not executable. This means that any attempt to branch to an address within that page triggers an exception, in the form of a Permission fault. There are separate Privileged and Unprivileged bits. This is because application code needs to be executable in user space (EL0) but should never be executed with kernel permissions (EL1/EL2). Another form of attack involves abusing system calls to try to get privileged code to call code from user memory.

The following diagram shows a simplified, but typical, virtual address space for an application that is running under an Operating System (OS), with the expected execution permissions:

Figure 2-2: Virtual address space
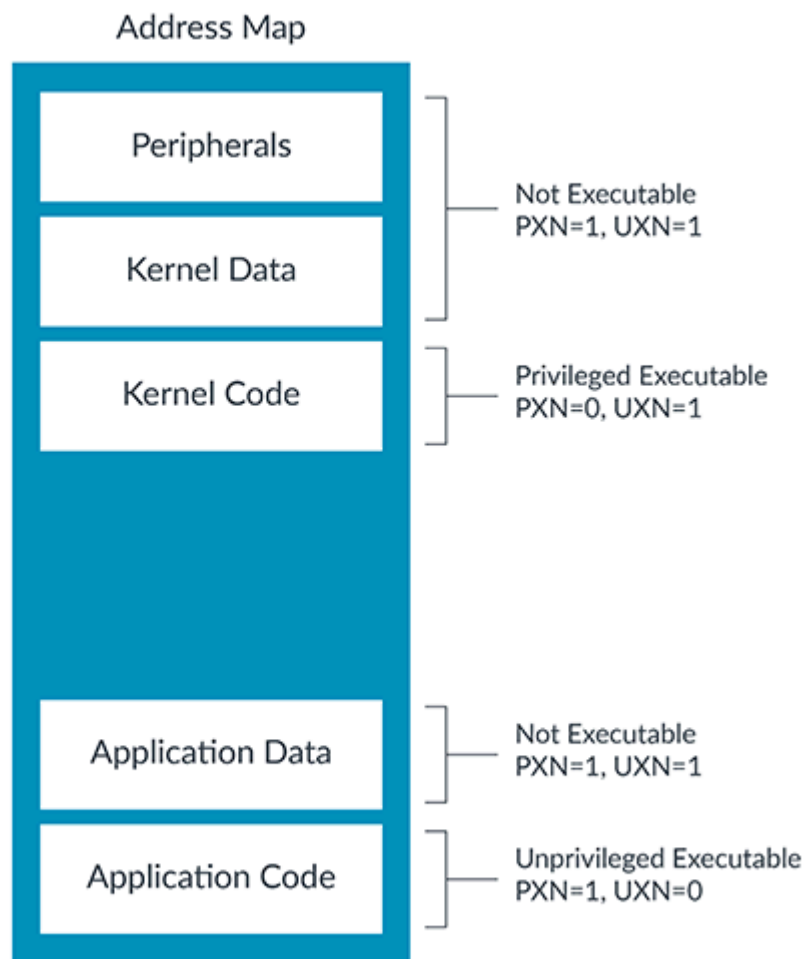
> **Note**
> By convention, kernel space is at the top of memory and user space is at the bottom of memory. Although this is not required by the architecture, it is the most common layout and the examples in this guide follow this convention.

The architecture also provides control bits in the system control register, SCTLR_ELx, to make all writable addresses non-executable. Enabling this control makes locations like the stack non-executable.

A location that is writable at EL0 is never executable at EL1, regardless of how the PXN and SCTLR_ELx controls are configured.

Together, these controls can provide robust protection against the kinds of attack that we have described. The translation table attributes and write controls can block execution from any location that the malicious code could write to, as you can see in the following diagram:

Figure 2-3: Protection against malicious code stack
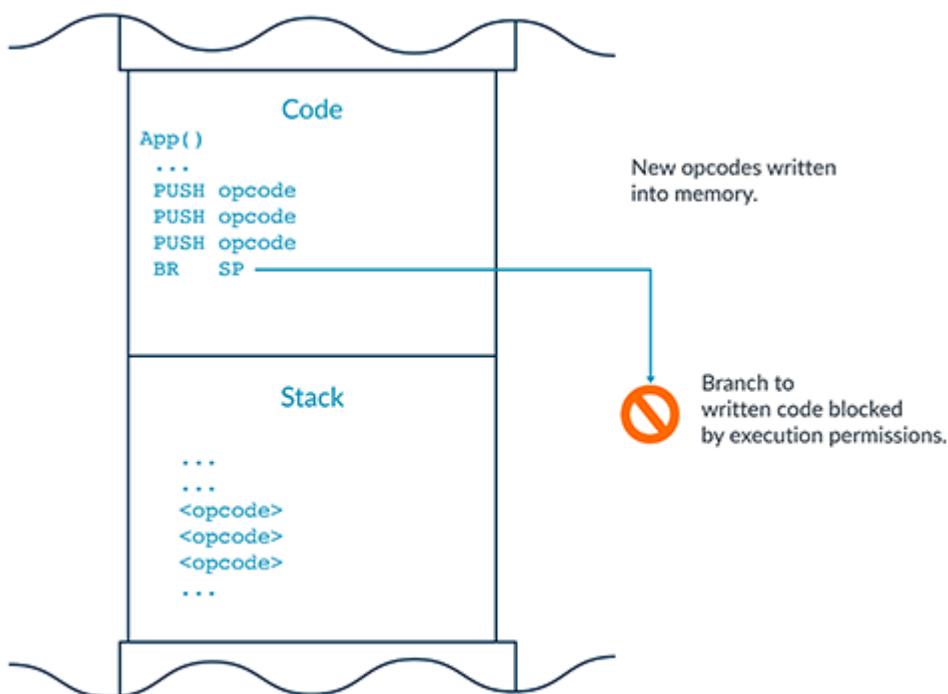
3. Return-oriented programming

Features like the execution permission that we described have made it increasingly difficult to execute arbitrary code. This means that attackers use other approaches like Return Oriented Programming (ROP). ROP takes advantage of the scale of the software stack in many modern systems. An attacker analyzes the software in a system, looking for gadgets. A gadget is a useful fragment of code, usually ending with a function return. for example:

```
...
ADD x0, x1, x2
RET
```

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 like execution permissions. The attacker strings together a chain of gadgets, forming what is effectively a new program, made up of existing code fragments. You can see an example in the following diagram:

Figure 3-1: Gadget attack code
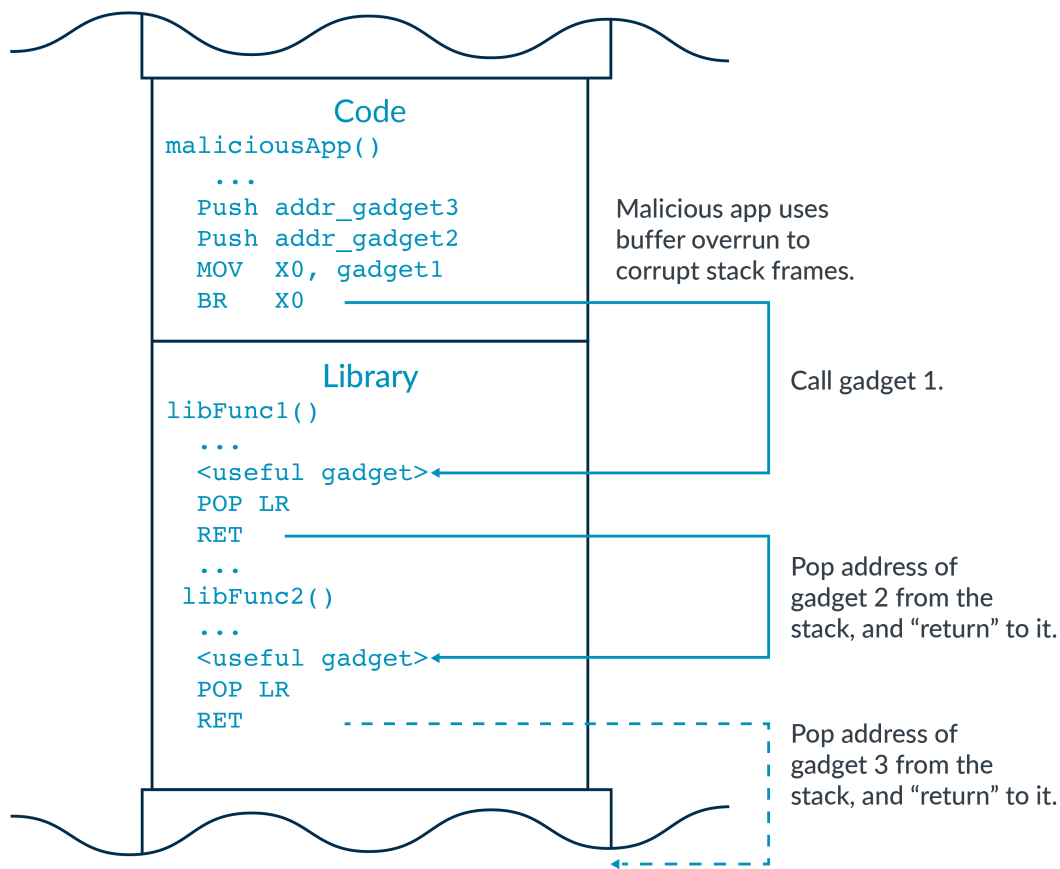
Any library that is available in the address space for the process is a potential source of gadgets. For example, the C library contains many functions, each offering potential gadgets. With so many gadgets available, statistically enough gadgets are available to form any arbitrary new program. Some compilers are even designed to compile to gadgets, rather than assembler. An 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. Address Space Randomization (ASLR) can help prevent the practice of automated and multiple attacks.

Pointer authentication

Armv8.3-A introduces the option of pointer authentication. Pointer authentication can mitigate against ROP attacks.

Pointer authentication takes advantage of the fact that pointers are stored in a 64-bit format, but not all those bits are needed to represent the address. The following diagram shows the virtual address space layout:

Figure 3-2: Virtual address space
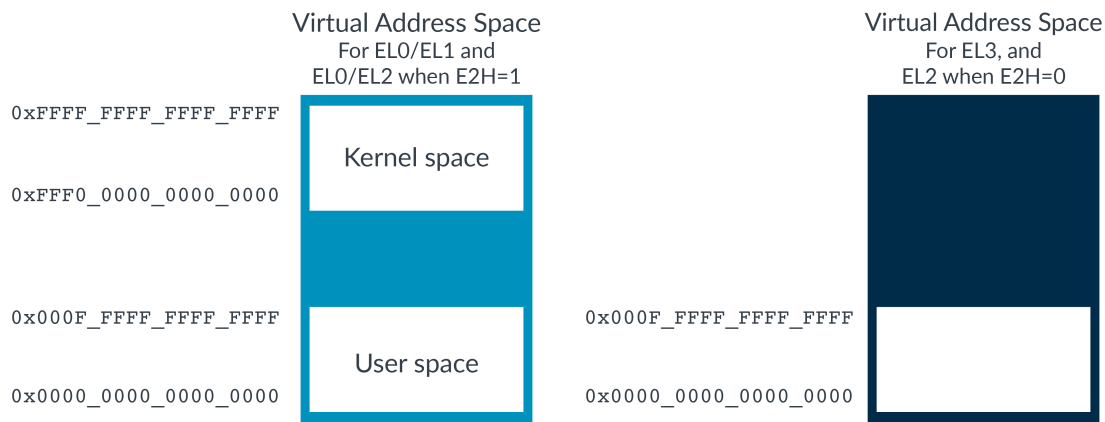
You can see that there are potentially two 2^{52} byte address ranges, one at the top of the address space, and one at the bottom of the address space:

Bottom Range: `0x0000_0000_0000_0000` – `0x000F_FFFF_FFFF_FFFF`

Top Range: `0xFFF0_0000_0000_0000` – `0xFFFF_FFFF_FFFF_FFFF`

Any address that falls outside of both ranges is always invalid and results in a fault if accessed.

> **Note**
>
> Before the release of Armv8.1-A, the maximum size of each range was 2^{48}.

You can see that any valid virtual address will have its top 12 bits as `0x000` or `0xFFF`. When pointer authentication is enabled, the upper bits are used to store a signature and are not treated as part of the address. This signature is referred to as a Pointer Authentication Code (PAC).

The PAC uses the top bits of the pointer. Bit[55] is reserved to indicate whether the top or bottom region is being accessed. This is illustrated here:

Figure 3-3: Pointer Authentication Code

The exact number of bits that are available for the PAC depends on the configured size of the virtual address space, and on whether tagged pointers are enabled. The smaller the virtual address space, the more bits that are available.

To protect against ROP attacks, at the start of a function the return address in the `LR` is signed. This means that a PAC is added in the upper order bits of the register. Before returning, the return address is authenticated using the PAC. If the check fails, an exception is generated when the address is used for a branch. The following diagram shows an example:

Figure 3-4: Protection against ROP attacks
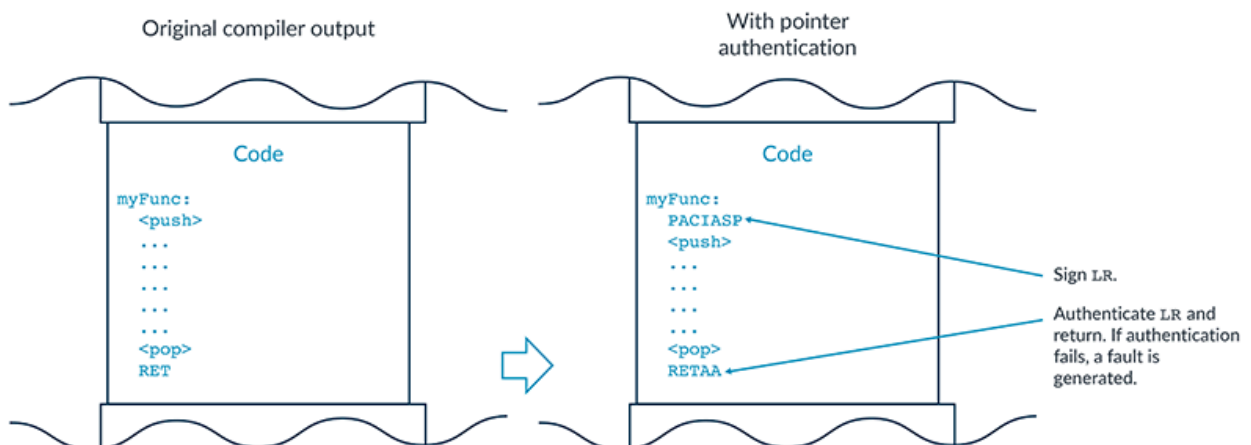
This change makes ROP attacks much harder to launch. This is because, to form the chain of gadgets, the attacker needs to know the location of those gadgets, and correctly signed pointers to those locations. To get a signed pointer it would need access to signing gadget.

How is the PAC formed?

The architecture provides five 128-bit keys. Each key is stored in a pair of 64-bit System registers:

- Two keys, A and B, for instruction pointers
- Two keys, A and B, for data pointers
- One key for general use

The registers that store these keys are only accessible at EL1 and above.

For data and instruction addresses, the instruction used to create and check the PAC specifies whether the A key or the B key is used. For a particular pointer, the instruction that generates the PAC and the instruction that authenticates the PAC must agree on which key to use.

The signature is formed from the address itself, the key, and a modifier, as you can see here:

Figure 3-5: Key and modifier authentication
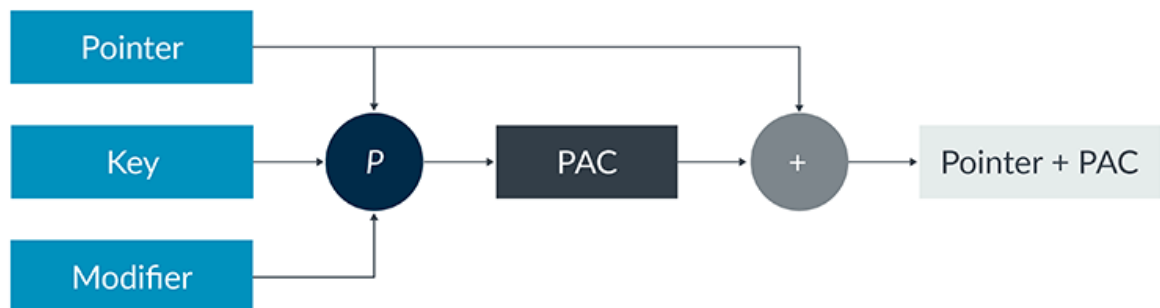
The architecture allows different implementations, for example from different vendors, to use different encryption algorithms. The recommended algorithm is QARMA, which is required by SBSA level 5. `ID_AA64ISAR1_EL1` reports which algorithm is supported on a specific processor.

The instructions that generate and authenticate the PAC specify whether the modifier is another processor register or is 0. The modifier needs to be a value which will be the same on entry and exit if the function is called correctly. For example, the Stack Pointer (SP) can have a different value every time that a function is called but will have the same value at the start and at the end of a given call. Using the SP as a modifier gives you a PAC that is only valid for that call of the function. This is because the SP will probably be in a different location on future calls.

The limited size of the PAC means that the strength of the signature is potentially low, depending on the size of the configured virtual address size. However, the keys are typically of limited life span. Each running application can use different keys, and a given application can be given different

keys each time that it is launched. When forming a chain of gadgets, the attacker must get every pointer correct, otherwise an exception will be raised.

How is the PAC checked?

Before use, the pointer must be authenticated. The authentication process is shown in this diagram:

Figure 3-6: PAC check
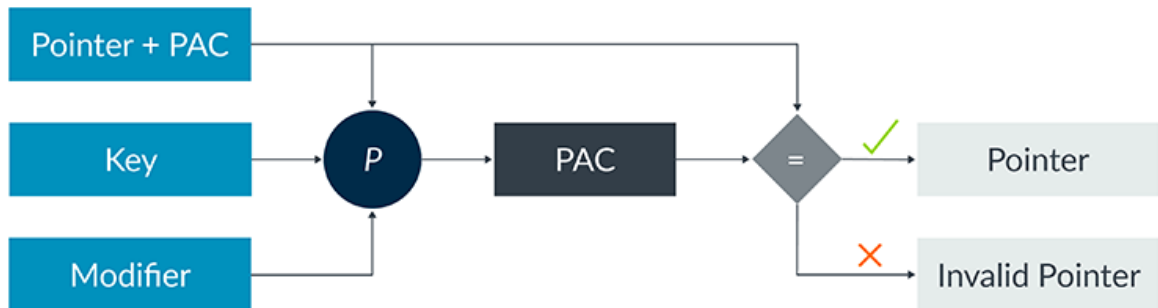
The authentication operation regenerates the PAC and compares it with the value that is stored in the pointer. If authentication succeeds, a pointer without the PAC is returned. If authentication fails, an invalid pointer is returned. This means that an exception is raised if the pointer is used.

New instructions

To support pointer authentication, new instructions are added to A64. Let's look at some examples of the operations that are related to the instruction pointers:

PACIxSP - Sign LR using SP as the modifier.

PACIxZ - Sign LR using 0 as the modifier.

PACIx - Sign Xn using a general-purpose register as modifier.

AUTIxSP - Authenticate LR using SP as the modifier.

AUTIxZ - Authenticate LR using 0 as the modifier.

AUTIx - Authenticate Xn using a general-purpose register as modifier.

BRAx - Indirect branch with pointer authentication.

BLRAx - Indirect branch with link, with pointer authentication.

RETAx - Function return with pointer authentication.

ERETAx - Exception return with pointer authentication.

In each case, replace x with A or B to select the wanted key.

The preceding list is not complete, but it shows the type of operations that are available. You can refer to the Arm ARM for a complete list and detailed descriptions.

Use of the NOP space

Some of the new authentication instructions are in the **NOP** space. Applications or libraries that protect themselves with these **NOP**-space instructions can run on older processors without pointer authentication support. Although the older processors will not benefit from the protections, this can be very useful in heterogeneous systems, as you can see in the following diagram:

Figure 3-7: Use of the NOP space
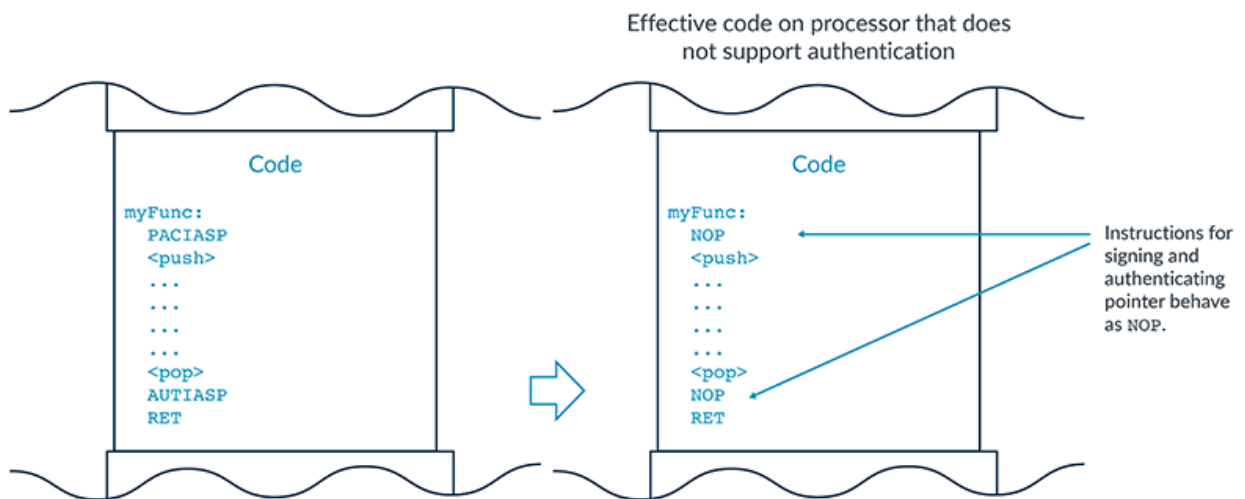
> **Note**
>
> To provide backwards compatibility, this program uses separate instructions to authenticate the LR and return. Ideally the combined authenticate and return instructions, RETAx, would be used. However, the RETAx instruction does not use the **NOP** instruction space. This means that it is not compatible with a processor that does not support authentication.

Enabling pointer authentication

Pointer authentication is controlled by Exception level using SCTLR_ELx. SCTLR_ELx uses separate controls for instruction checking and for data checking:

* EnIx - Enables instruction pointer authentication using key x.

* EnDx - Enables data pointer authentication using key x.

4. Jump-oriented programming

Jump-Oriented Programming (JOP), is similar to Return-Oriented Programming (ROP). In an ROP attack, the software stack is scanned for gadgets that can be strung together to form a new program. ROP attacks look for sequences that end in a function return (RET). In contrast, JOP attacks target sequences that end in other forms of indirect (absolute) branches, like function pointers or case statements. You can see an example here:

Figure 4-1: Jump oriented programming
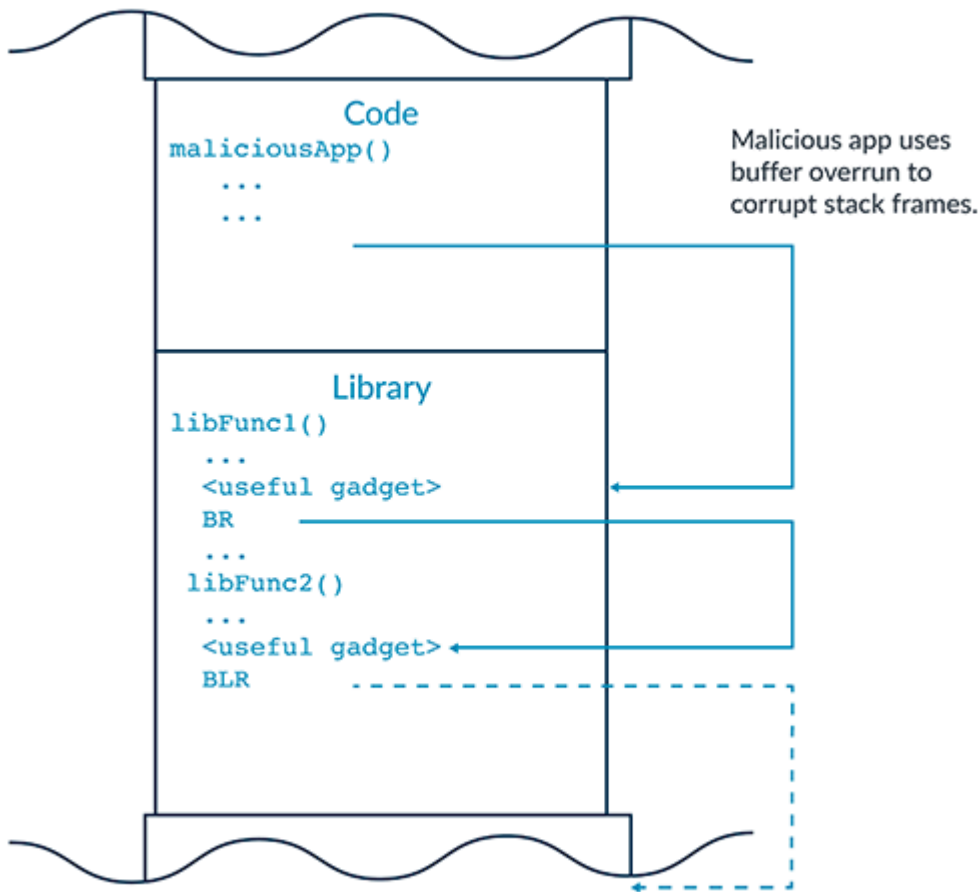
The attacker exploits the fact that BLR or BR instructions can target any executable address, and not just the addresses that are entry points defined by the compiler or developer. This means that the instructions can be hijacked to string gadgets together.

Branch target instructions

To help protect against JOP attacks, Armv8.5-A introduced Branch Target Instructions (BTIs). BTIs are also called landing pads. The processor can be configured so that indirect branches (BR and

BLR) can only allow target landing pad instructions. If the target of an indirect branch is not a landing pad, a Branch Target Exception is generated as you can see here:

Figure 4-2: Branch targer exception
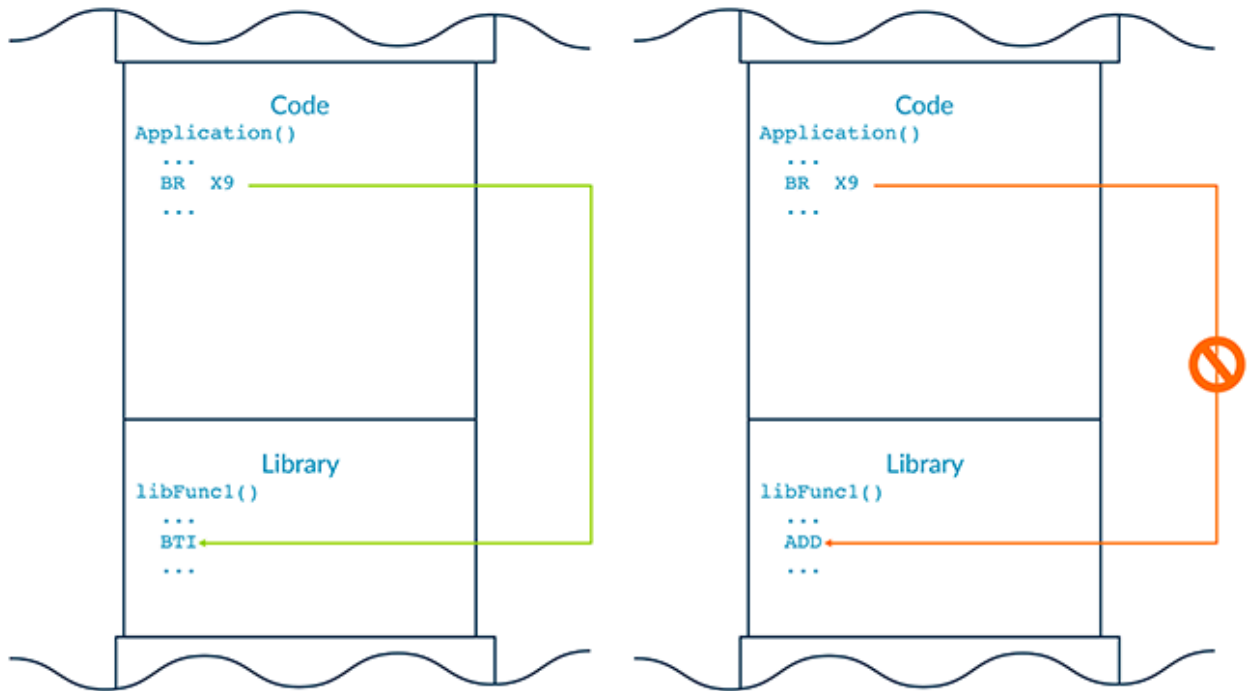
The use of landing pads significantly reduces the number of possible targets for an indirect branch and makes it harder to string chains of gadgets together to form a new program.

Enabling branch target checking

Support for landing pads is enabled for each page, using a new bit (GP bit) in the translation tables. Per-page controls allows a filesystem to contain a mixture of landing pad-protected code and legacy code, which is illustrated here:

Figure 4-3: Landing pad
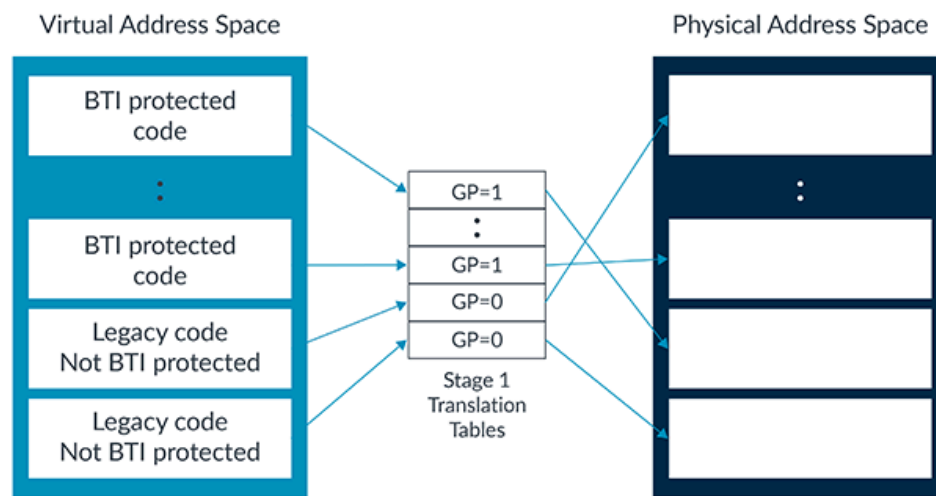
The encoding for BTI instructions, like the pointer-authentication instructions, is allocated within the **NOP** space. BTI-protected code can still function when run on older processors that do not support BTI, or when GP=0, although without the additional protection.

How BTI is implemented

PSTATE includes a field, BTYPE, that records the branch type. On executing an indirect branch, the type of indirect branch is recorded in PSTATE.BTYPE. The following list shows the value BTYPE takes for different branch instructions:

- BTYPE=11: BR, BRAA, BRAB, BRAAZ, BRABZ with any register other than X16 or X17
- BTYPE=10: BLR, BLRAA, BLRAB, BLRAAZ, BLRABZ
- BTYPE=01: BR, BRAA, BRAB, BRAAZ, BRABZ with X16 or X17

Executing any other type of instruction, including direct branches, causes BTYPE to be set to b00.

Why store two bits? A simple implementation could record whether an indirect branch was in process or not. However, recording the type of indirect branches further limits the possibilities of finding gadgets. The syntax of the BTI instruction includes an argument, specifying which types of indirect branch it can be targeted by:

Argument	Accepted PSTATE.BTYPE	Use case
BTI c	0b10 and 0b01	Function calls
BTI j	0b11 and 0b01	Non-function call branches, like case-statements
BTI jc	All	All

When BTYPE!=00, the processor checks whether the instruction being targeted is a landing. If it is not a landing, or if it is the wrong type of indirect branch, an exception is generated.

X16 and X17

Why does the architecture distinguish between indirect branches that use `x16` or `x17` and those
that do not?

`x16` and `x17` have special significance in the Procedure Call Standard used by Arm. They are
referred to as the intra-procedure call corruptible registers, or IP0 or IP1. They can be used by
static linkers for inserting branch-range extending veneers, or by dynamic linkers for handling jump
tables.

This is relevant to us because it means that a function might be entered directly from the caller
using `BL` or `BLR` or indirectly via linker generated code using `x16` or `x17`. Therefore, the landing pad
for a function entry needs to be able to accept both.

Function entry and return

The function return instructions, `RET`, `RETAA` and `RETAB`, are also a form of indirect branch. If these
instructions were required to target a `BTI`, every function call would need to be followed by a `BTI`.
This would cause undesirable code bloat. Also, the pointer authentication feature already provides
a way to protect function returns.

For function entry, the pointer signing instructions `PACIxSP` and `PACIxZ` act like landing pads. These
instructions are like `BTI` instructions. This means that when the landing pad feature is used pointer
authentication, there is no need to start every function with a `BTI`. This also avoids code bloat.

5. Applying these techniques to real code

In Return-oriented programming (ROP) and Jump-oriented programming (JOP), we explored features that Arm introduced to the Arm architecture to mitigate against JOP-style and ROP-style attacks. Now we will look at the compiler support for these features, and how enabling these protections affects the number of that are gadgets available to attackers.

In this section, we refer to these versions of Arm Compiler 6 and Gnu C Compiler (GCC):

- Arm Compiler 6.11
- GCC 9.1

Compiler support for these features continues to evolve. Precise figures will vary based on the versions that you use.

Build an image with pointer authentication and branch target identification

For Arm Compiler 6, GCC and LLVM generation of pointer authentication and BTI-enabled code is controlled by:

- `mbranch-protection=<protection>`

Where `<protection>` can be any combination of:

- `pac-ret{+leaf+b-key}`
  - `pac-ret` enables return address signing for non-leaf functions using the A-key.
  - `+leaf` increases the scope of return address signing to include leaf functions.
  - `+b-key` uses B-key instructions to sign addresses instead of A-key instructions.
- `bti` protects code using Branch Target Identification.
- `standard` turns on all types of branch protection.
  - Currently standard implies `pac-ret+bti`.
- `none` turns off all types of branch protection.
  - This is the default if the `-mbranch-protection` flag is not provided.

Whether the combined or **NOP**-compatible instructions are generated depends on the architecture version that the code is built for. When building for Armv8.3-A, or later, the compiler will use the combined operations. When building for Armv8.2-A, or earlier, it will use the **NOP** compatible instructions. For example:

-march=armv8.2-a -mbranch-protection=standard	-march=armv8.3-a -mbranch-protection=standard
enableInt	enableInt
0x00000000: d503233f PACIASP	0x00000000: d503233f PACIASP
...	...
...	...
0x00000350: d50323bf AUTIASP	0x00000350: d65f0bff RETAA

-march=armv8.2-a -mbranch-protection=standard	-march=armv8.3-a -mbranch-protection=standard
0x00000354: 65f03c0 RET	

Note The function used in this example was taken from the example that accompanies our guide Arm CoreLink Generic Interrupt Controller v3 and v4 Overview and built with Arm Compiler 6.

The compiler generates the instructions that are required to perform signing and authentication. Generating and configuring keys is the responsibility of supervising software, typically an operating system.

Reduction in available gadgets

GLIBC is a large library that is used in C or C++ applications. This means that it is a good target for attackers, and a good place for us to see the effect of applying the measures to mitigate attacks. Arm used this tool to measure the number of available gadgets and modified the tool to fit our requirements.

The following graph shows the number of gadgets before and after the compiler options were enabled:

Figure 5-1: ROP and JOP gadgets gets in GLIBC graph
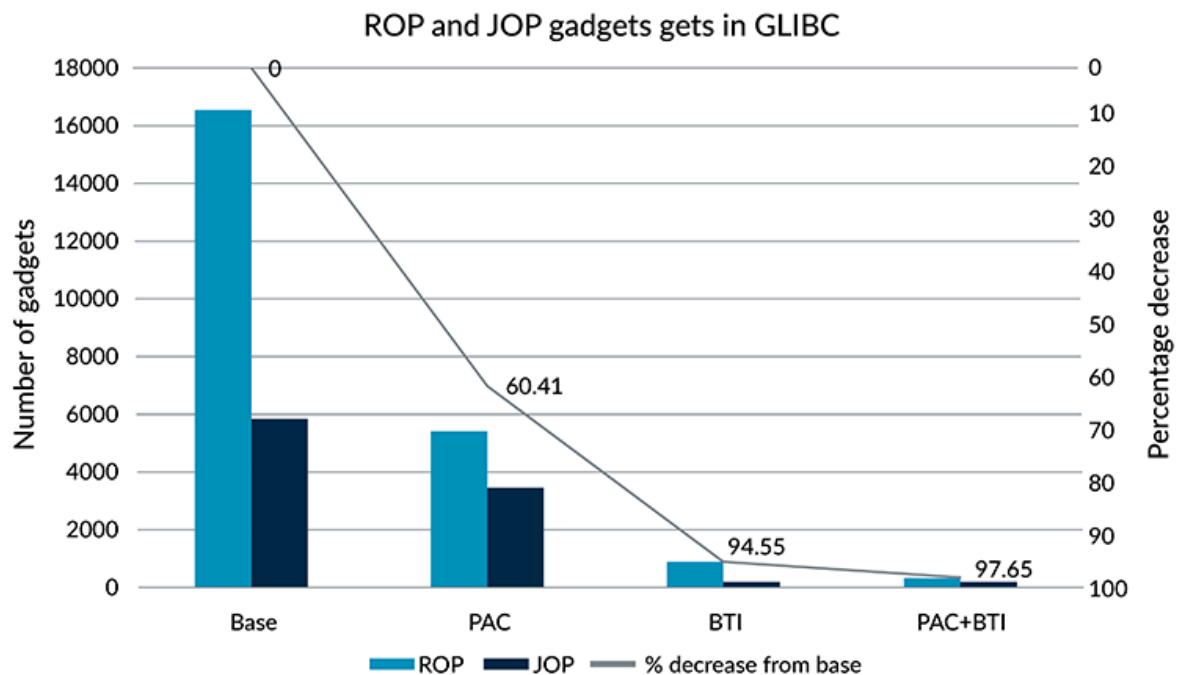
By enabling both pointer authentication and branch target identification, the number of gadgets that are available reduces by 97.65%.

Effect on code size

The protection described in the preceding section is helpful but comes at a cost. One obvious cost is the increase in code size. Here is an analysis of this cost:

Figure 5-2: Change in image size from enabling ROP and JOP mitigations

Change in image size from enabling ROP and JOP mitigations
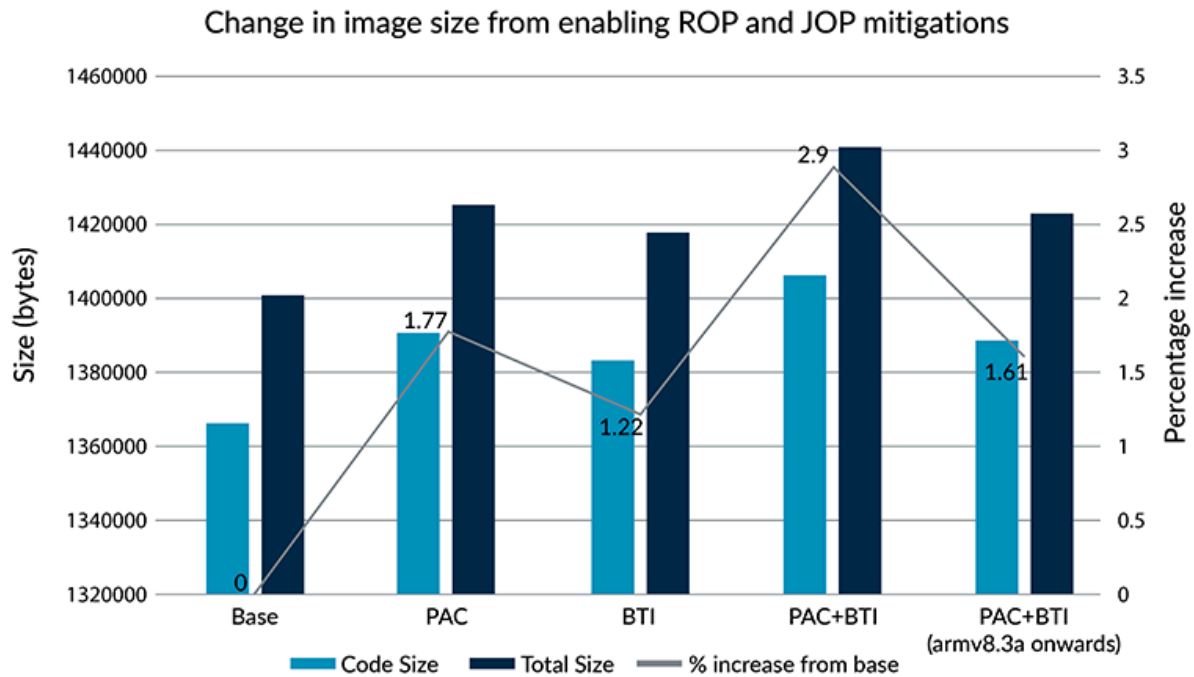
The graph shows that the code size effect on GLIBC is minimal. Even though turning on both the mitigations leads to a 2.9% code size increase, this increase is smaller when compiling with `-march=armv8.3-a.` Compiling for Armv8.3-A allows the compiler to use fused authenticate and return instructions. This means that, for Armv8.3-A, the code size increase is only 1.6%.

6. Detecting memory safety violations

Some classes of vulnerability that are related to memory usage can be difficult to detect and test for. Two examples of this are:

- Use after free - Applications continue to use allocated memory after releasing it, or after it is out of scope. This is a violation of temporal memory safety.

- Buffer overrun, or overflow - Going beyond the bounds of an allocated structure or buffer, usually because of insufficient bounds checking. This is a violation of spatial memory safety.

Armv8.5-A introduces the Memory Tagging Extension (MTE), also called memory coloring. Memory tagging makes detecting memory safety violations easier and more efficient.

Note   One of the first Internet-spread computer worms was the Internet Worm in 1988, which exploited a buffer overrun. More than thirty years later, we are still seeing attacks that exploit this type of programming bug.

Memory tagging

Regions of address space are allocated a tag, or lock. The upper bits of a virtual address are also used to store a tag, or key. On a memory access, the processor compares the key in the issued address with the lock that is assigned to that physical location. Here is an example:

Figure 6-1: Memory tagging
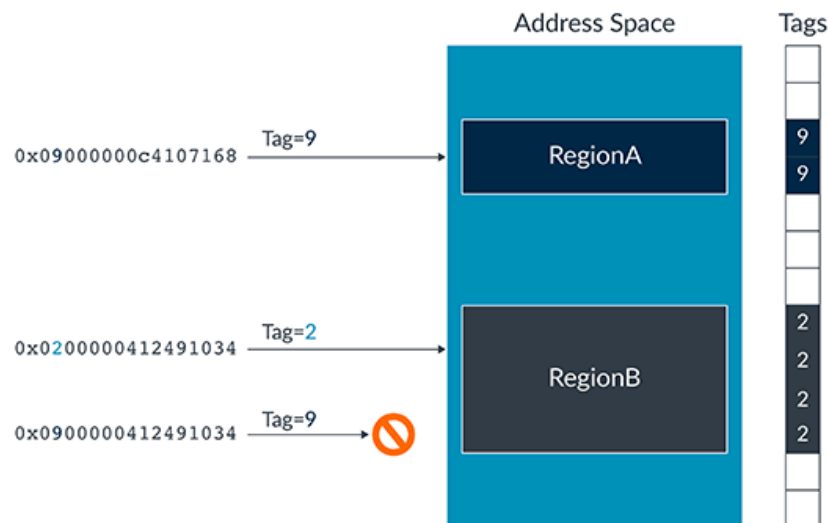
In the preceding diagram, two regions have been allocated, using tags 9 and 2.

For the first two pointers, the tag matches that of the accessed location. You can think of this as
the key fitting the lock. Accesses using these pointers would succeed as normal.

However, for the final pointer the tag does not match that of the accessed location. This will be
captured as a tag check failure. We will look at what happens in the case later.

Let's apply this mechanism to the problems that we identified earlier, starting with buffer overruns,
as you can see in this diagram:

Figure 6-2: Buffer overruns memory tagging

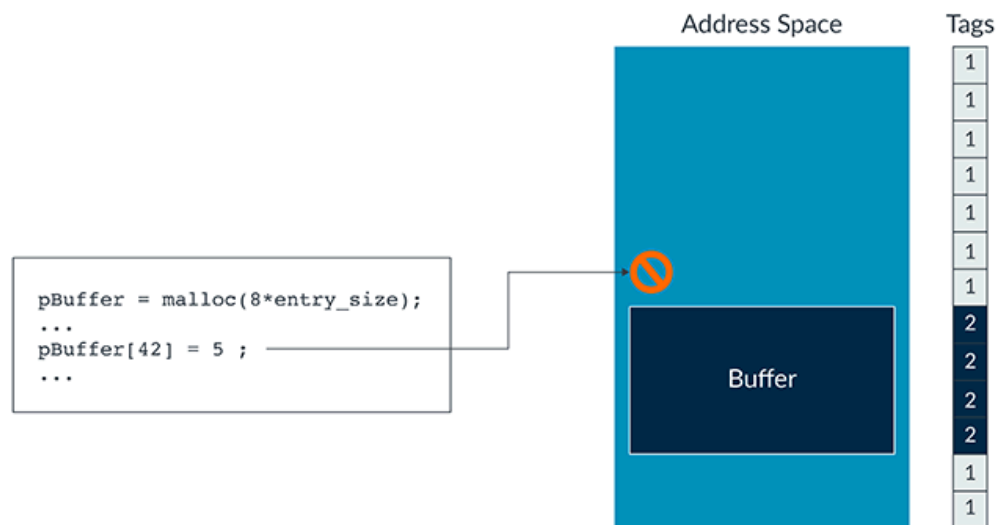On the call to `malloc()` the C library will allocate the memory and assign a tag for the buffer. The
returned pointer will include the allocated tag. If software using the pointer goes beyond the limits
of the buffer, the tag comparison check will fail. This failure will allow us to detect the overrun.

Similarly, for use-after-free, on the call to `malloc()` the buffer gets allocated in memory and
assigned a tag value. The pointer that is returned by `malloc()` includes this tag. Later the buffer is
released. The C library might change the tag when the memory is released or might wait until the
memory is reused for some other purpose. If software continues to use the old pointer, it will have
the old tag value and the tag check will catch it.

Figure 6-3: Old tag value check
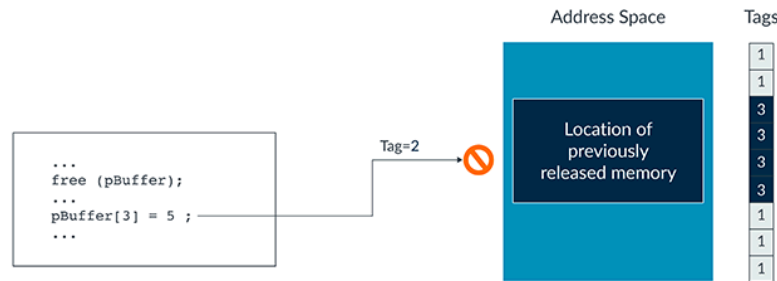
> **Note**
>
> The total number of possible tags is small. Therefore, the same tag value might be used for several different regions over time, or at the same time. However, with careful tag allocation, sequential overruns or underruns can be detected. Wild accesses are statistically likely to be caught.

Tags

To work with tags, the architecture gains several new instructions, including:

*   IRG - Generates a random tag value and inserts it to a pointer
*   STG - Sets the tag value for a block of memory
*   STZG - Sets the tag value for a block of memory, and zeros corresponding memory location
    *   If the allocator is going to zero the allocated memory, STZG offers better performance than separate zeroing and tagging.
*   LDG - Reads the tag value for a block of memory

Tags are four bits and are stored in two places:

*   Key - Stored in bits [59:56] of a pointer
    *   This requires pointer tagging to be enabled. We will discuss this later in the guide.
*   Lock - A new address space, the tag address space, is added. The tag address space records the tag to a memory region.

On allocating a block of memory, software allocates a tag either randomly, using IRG, or using a custom algorithm. Each tag covers 16 bytes. This means that software needs to execute STZG or STG multiple times to cover all the 16-byte blocks within the allocated memory.

Tagged and untagged addresses

Not all memory accesses require tag checking. We describe an access as Checked or Unchecked, depending on whether tag checking is carried out.

The following accesses are always Unchecked:

*   Instruction fetches

- Translation table walks, including hardware updates of the Access Flag or Dirty state

- Data cache maintenance operations

- Accesses to the Allocation tags

For data accesses, a new memory attribute is added to indicate that accesses to this region should be Checked:

- `MemAttr[] == 0xF0:` Inner+Outer Write-Back Cacheable, Read or Write-Allocate, Tagged

Data accesses to a region that is marked as Tagged are classed as Checked, unless one of the following applies:

- `TCR_ELx.TBI==0`

- The Logical tag (bits [59:56] of the virtual address) are `b0000` or `b1111`.

- The load or store uses the SP as a base register with an immediate offset, or no offset

- It is a PC relative load.

- `PSTATE.TCO==1`

Data accesses to any region without the Tagged attribute are Unchecked.

Note

Loads or stores using the stack pointer with an immediate offset can be statically checked at build time. This means that there is less benefit to checking with MTE. The same principle applies to PC-relative loads.

What happens when a comparison fails?

Let's discuss what happens when the tag comparison fails. The architecture makes the behavior of tag comparison failure configurable, controlled by `SCTLR_ELx.TCF`, or `SCTLR_ELx.TCF0` for EL0:

- `TCF==00` - Tag comparison failures are ignored.

- `TCF==01` - Tag comparison failures are reported as a synchronous Data Abort. The address that caused the failure is reported in `FAR_ELx`.

- `TCF==10` - Tag comparison failures are reported asynchronously by updating bits in `TFSR_ELx`, or `TFSR0_EL1` for EL0. Optionally, checks can be synchronized on exception entry, to allow check failures to be attributed to a specific process.

The architecture provides both synchronous and asynchronous mechanisms to report tag comparison failures. Synchronous checking makes debugging simpler, because it allows you to identify the precise instruction and address that caused the failure. However, synchronous checking typically has a significant performance impact. This performance impact might be acceptable in a development environment but is too high for deployment.

Asynchronous checking is less costly. This means that asynchronous checking is potentially acceptable even on production systems. Although asynchronous checking provides less precise information on where the tag comparison failure occurred, it can provide some mitigation and be

used for profiling. Profiling allows problem areas to be identified, narrowing down the search area for bugs.

Combining memory tagging and pointer authentication

Memory tagging and pointer authentication both use the upper bits of an address to store additional information about the pointer: a tag for memory tagging, and a PAC for pointer authentication.

Both technologies can be enabled at the same time. The size of the PAC is variable, depending on the size of the virtual address space. When memory tagging is enabled at the same time, there are fewer bits available for the PAC.

7. Check your knowledge

The following questions will help you test you knowledge.

What is a gadget in Return oriented-programming (ROP) and Jump-oriented programming (JOP) attacks?

A gadget is a piece of existing code which ends in either a function return or an indirect (absolute) branch. In ROP and JOP attacks, these gadgets are chained together to form new programs.

Describe how Branch Target Identification (BTI) limits the scope of JOP attacks.

BTI restricts indirect branches to only target-ing- BTI instructions, or PACIxSP and PACIxZ instructions. This greatly reduces the number of possible targets and makes it difficult to form chains of gadgets.

When using pointer authentication, where is the signature of an address stored?

In the upper bits of the virtual address.

In the Arm Memory Tagging Extension (MTE), what happens when the tag issued alongside a memory access does not match the allocation tag?

This situation is known as a tag checking failure. The behavior is configurable, via `SCTLR_ELx.TCF`. The failure can be ignored, reported synchronously, or reported asynchronously.

How many bits are used to store the logical tag in the Arm memory tagging extension?

4 bits, but the values `0b0000` and `0b1111` are reserved.

8. Related information

Here are some resources related to information in this guide:

- **Arm architecture and reference manuals:** Find technical manuals and documentation relating to this guide and other similar topics
- **Arm Community:** Ask development questions, and find articles and blogs on specific topics from Arm experts
- **Armv8-A Instruction Set Architecture:** More information on the Procedure Call Standard
- **The QARMA Block Cipher Family:** Information on the QARMA cipher from the International Association for Cryptologic Research
- **Control-Flow-Integrity:** NSA paper on control flow protection. While not specific to the Arm architecture, the paper provides good background reading on the topic

Detecting memory safety violoations

- **Adopting the Arm Memory Tagging Extension in Android:** Google blog about their use of memory tagging techniques to locate memory safety bugs
- **Armv8.5-A Memory Tagging Extension:** Arm white paper with a detailed description of the memory tagging technology

Pointer authentication

- **Armv8.3-A pointer authentication support:** The patches that added support for pointer authentication to the Linux kernel give information on how the technology is used in practice
- **Code reuse attacks:** the compiler story: Arm blog discussing the use of Pointer authentication and Branch Target

9. Next steps

This guide introduced features available in the Arm architecture which can provide robust defenses for complex software stacks. We have looked at the pointer authentication and branch target identification extensions, which can be used to defend against ROP and JOP attacks. We also looked at how memory tagging can be used to detect and locate potential vulnerabilities before they are exploited.

Next you might want to learn about Arm's TrustZone technology, another feature available in the Arm architecture.

The knowledge in this guide, and in the TrustZone guide, will be useful to you as you design your own complex systems. Enabling you to decide which combination of technologies you should deploy to protect different assets in the system.