Secure software guidelines for Armv8-M

Non-Confidential

Issue 0300
100720_0300_00

Secure software guidelines for Armv8-M

Release information

Document history

Issue	Date	Confidentiality	Change
0100	23 August 2016	Non-Confidential	First release
0200	28 February 2017	Non-Confidential	Second release
0300	30 July 2020	Non-Confidential	Third release

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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.

Web Address

www.arm.com

Contents

1 Introduction

TrustZone technology for Armv8-M is an optional Security Extension that is designed to provide foundation for improved system security in a wide range of embedded applications.

Arm TrustZone technology enables the system and the software to be partitioned into Secure and Normal worlds. Secure software can access both Secure and Non-secure memories and resources, while Normal software can only access Non-secure memories and resources.

This document focusses on some of the secure software guidelines and recommendations for an Armv8-M based platform. If you want to learn more, you can refer to following documents as well.

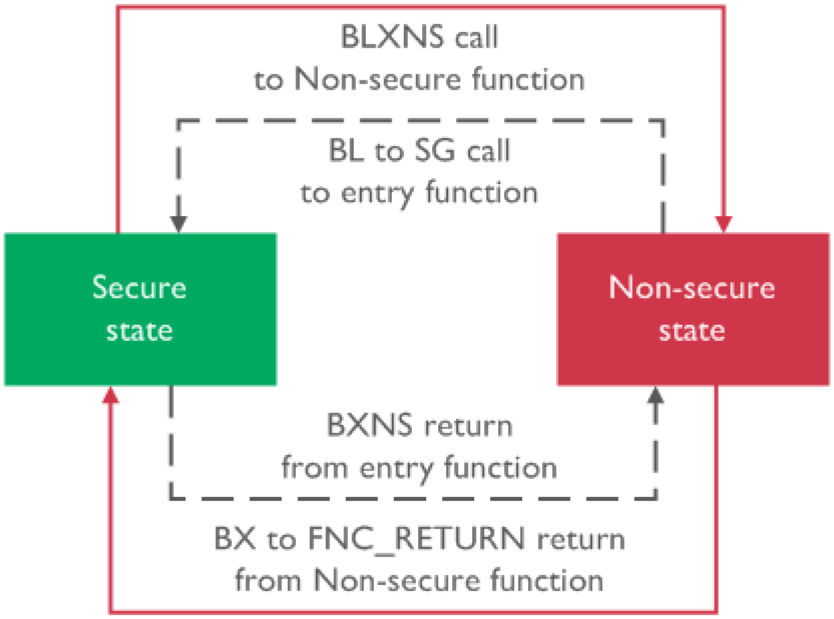
- Armv8-M Security Extensions: Requirements on Development Tools explains concepts implemented in compiler toolchains to support the Armv8-M architecture.
- The Arm C Language Extensions (ACLE) for Armv8-M enables the Armv8-M Security Extension to build a secure image, and to enable a non-secure image to call a secure image. This document includes details of a possible compiler implementation.
- The Armv8-M Architecture Reference Manual gives a complete overview of the Armv8-M architecture. The following sections put a spotlight on some of the aspects that are of special interest to the software developer

1.1 Switching between Secure and Non-secure states

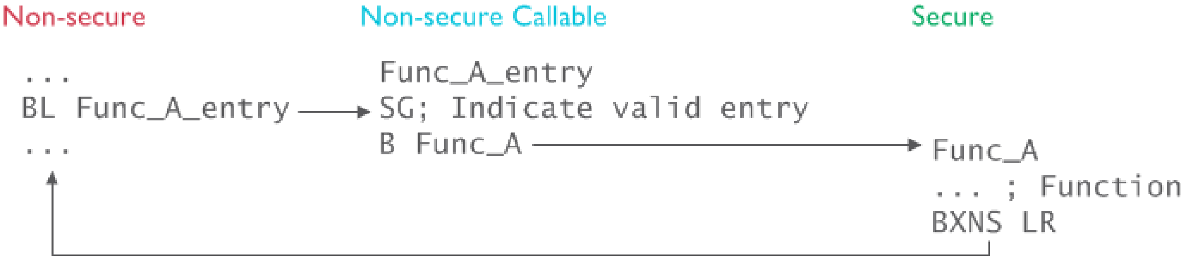
The Armv8-M Security Extensions allow direct calling between Secure and Non-secure software. Several instructions are available for state transition handling in Armv8-M processors:

`SG`	Secure gateway. Used for switching from Non-secure to Secure state at the first instruction of Secure entry point.
`BXNS`	Branch with exchange to Non-secure state. Used by Secure software to branch or return to Non- secure program.
`BLXNS`	Branch with link and exchange to Non-secure state. Used by Secure software to call Non-secure functions.

The following figure shows the security state transitions.



A direct API function call from Non-secure to Secure software entry points is allowed if the first instruction of the entry point is SG, and it is in a Non-secure callable memory location, as in the following diagram.



```
...
BL Func_A_entry
...
```

```
Func_A_entry
SG; Indicate valid entry
B Func_A
```

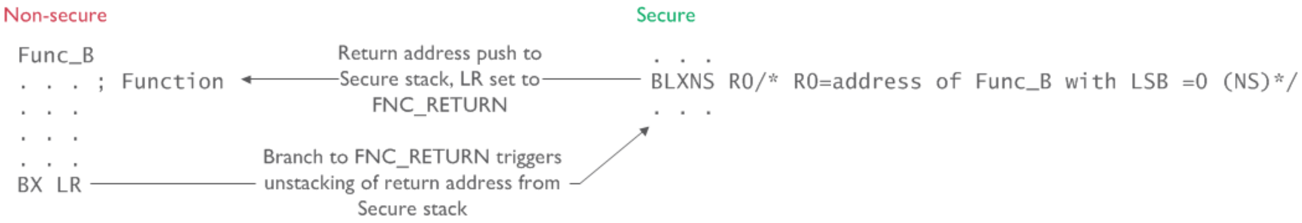
```
Func_A
... ; Function
BXNS LR
```

When a Non-secure program calls a Secure API, the API completes by returning to a Non-secure state using a BXNS instruction. If a Non-secure program attempts to branch, or call a Secure program address without using a valid entry point, a fault event is generated. In Armv8-M architecture the HardFault in Secure state handles the fault event. In Armv8-M architecture with Main Extension, the SecureFault exception type is used.

When a Non-secure program calls a Secure API, the API completes by returning to a Non-secure state using a BXNS instruction. If a Non-secure program attempts to branch, or call a Secure program address without using a valid entry point, a fault event is generated. In the Armv8-M architecture with Main extension, the SecureFault exception type is used. For the Armv8-M architecture, the HardFault in Secure state handles the fault event.

The Armv8-M Security Extensions also allow a Secure program to call Non-secure software. In such a case, the Secure program uses a BLXNS instruction to call a Non-secure program. During the state transition, the return address and some processor state information are pushed onto the Secure stack, while the return address on the Link Register (LR) is set to a special value called FNC_RETURN. The Least Significant Bit (LSB) of the function address must be 0.

The following figure shows the software flow when a secure program calls a Non-secure function:


Non-secure
Secure
Func_B
. . . ; Function
. . .
. . .
. . .
BX LR
Return address push to Secure stack, LR set to FNC_RETURN
Branch to FNC_RETURN triggers unstacking of return address from Secure stack
. . .
BLXNS R0/* R0=address of Func_B with LSB =0 (NS)*/
. . .

The Non-secure function completes by performing a branch to the FNC_RETURN address. This automatically triggers the unstacking of the true return address from the Secure stack and returns to the calling function. The state transition mechanism automatically hides the return address of the Secure software. Secure software can choose to transfer some of the register values to the Non-secure side as parameters, and clears other Secure data from the register banks before the function call.

1.2 Security state changes

Transitions from Secure to Non-secure state can be initiated by software by using either a BXN or BLXNS instruction that has the Least Significant Bit (LSB) of the target address unset. This enables the LSB of an address to denote the security state.

Note

Transitions from Non-secure to Secure state can be initiated by software in two ways:

- A branch to a Secure gateway.
- A branch to the reserved value FNC_RETURN.

A Secure gateway is an occurrence of the Secure Gateway instruction (`SG`) in a special type of Secure region, named a Non-secure Callable (NSC) region. When branching to a Secure gateway from Non-secure state, the `SG` instruction switches to the Secure state and clears the LSB of the return address in the LR. In any other situation, the `SG` instruction does not change the security state or modify the return address. The `SG` instruction must be fetched from NSC memory.

A branch to the reserved value FNC_RETURN causes the hardware to switch to Secure state, read an address from the top of the Secure stack, and branch to that address. The reserved value FNC_RETURN is written to the `LR` when executing the `BLXNS` instruction.

Security state transitions can be caused by hardware through the handling of interrupts. Those transitions are transparent to software.

1.3 Secure gateway veneers

A toolchain must support generating a Secure gateway veneer (extra code to reset the program counter) for each entry function with external linkage. It consists of an SG instruction followed by a `B.W` instruction that targets the entry function it veneers.

Secure gateway veneers decouple the addresses of Secure gateways (in NSC regions) from the rest of the Secure code. By maintaining a vector of Secure gateway veneers at a forever-fixed address, the rest of the Secure code can be updated independently of Non-secure code. This also limits the amount of code in NSC regions that potentially can be called by the Non-secure state.

Vectors of Secure gateway veneers are expected to be placed in NSC memory. All other code in the Secure executable is expected to be placed in Secure memory regions. This placement is under the control of the developer.

Preventing inadvertent Secure gateways is a responsibility that is shared between a developer and their toolchain. A toolchain must make it possible for a developer to avoid creating inadvertent Secure gateways.

Excluding the first instruction of a Secure gateway veneer, a veneer must not contain the bit pattern of the SG instruction on a 2-byte boundary.

A vector of Secure gateway veneers must be aligned to a 32-byte boundary, and must be zero padded to a 32-byte boundary.

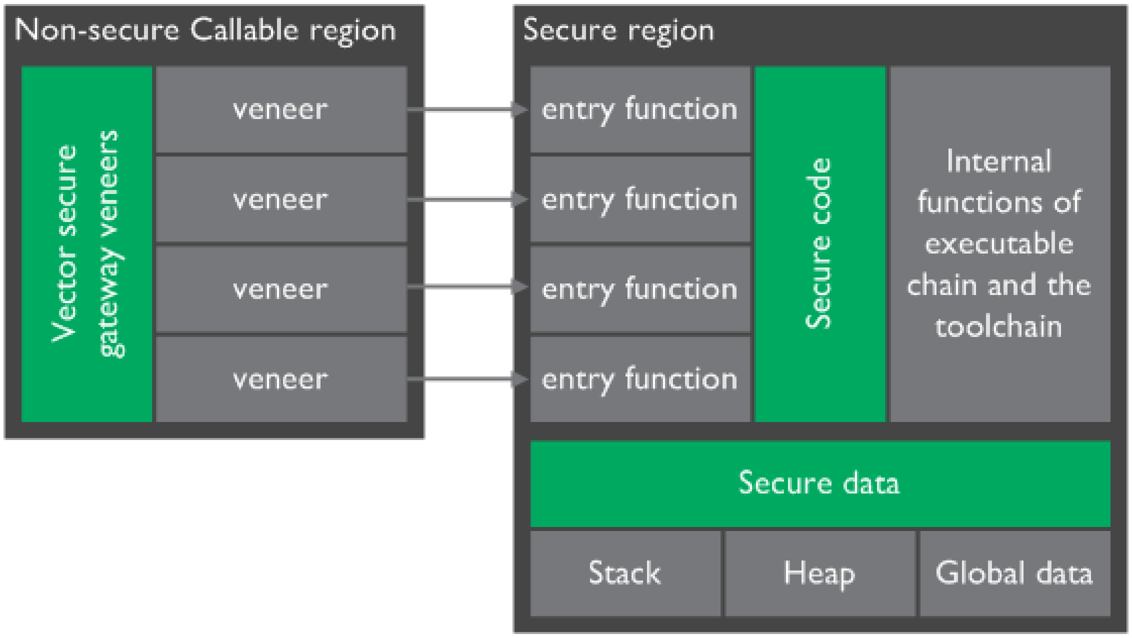
The developer must take care that the code or data before the vector of Secure gateway veneers does not create an inadvertent Secure gateway with the first Secure gateway veneer in the vector. ARM recommends placing the vector of Secure gateway veneers at the start of an NSC region.

The position of Secure gateway veneers in a vector must be controllable by the developer.

This last requirement gives the developer complete control over the address of a Secure gateway veneer.

It allows the developer to fix the addresses of the Secure gateway veneers so that Secure code can be updated independently of Non-secure code.

The following figure shows the memory layout of a Secure executable:



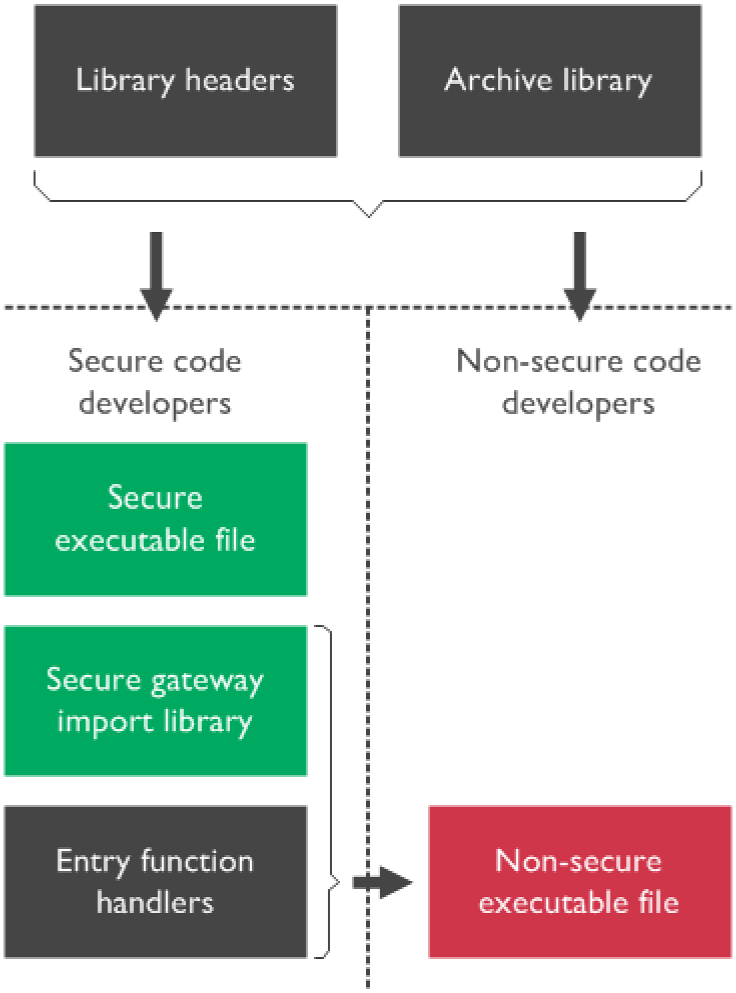
1.4 Executable files

There are two different types of executable files, one for each security state. The Secure state executes Secure code from a Secure executable file. The Non-secure state executes Non-secure code from a Non-secure executable file. The Secure and Non-secure executable files are developed independently of each other.

From the point of view of the Non-secure state, a call to a Secure gateway is a regular function call, as is the return from a Non-secure function call. You can develop Non-secure code with a toolchain that is not CMSE aware, that is, you do not require new tools when you are only building Non-secure code.

Developing a Secure executable file requires toolchain support whenever a function is called from, calls, or returns to Non-secure state and whenever memory is accessed through an address that is provided by the Non-secure state. Calling a Secure API is no different from calling a normal library function from Non-secure software point of view. The Secure code ABI is otherwise identical to the Non-secure code ABI.

The following figure shows the interaction between developers of Secure code, Non-secure code, and (optional) security agnostic library code:



The Secure gateway import library contains the addresses of the Secure gateways of the Secure code. This import library consists of or contains a relocatable file that defines symbols for all the Secure gateways. The Non-secure code links against this import library to use the functionality that is provided by the Secure code.

A relocatable file containing only copies of the (absolute) symbols of the Secure gateways in the Secure executable must be available to link Non-secure code against.

Linking against this import library is the only requirement on the toolchain that is used to develop the Non-secure code. This functionality is similar to calling ROM functions, and is expected to be available in existing toolchains.

1.5 Development tools

Development tools are expected to provide C and assembly language support for interacting between the security states. Code that is written in C++ must use the extern C linkage for any inter-state interaction.

Security state changes must be expressed through function calls and returns.

This use of the extern C linkage provides an interface that fits naturally with the C language.

A function in Secure code that can be called from the Non-secure state through its Secure gateway is called an entry function. A function call from Secure state to the Non-secure state is called a Non-secure function call.

1.6 CMSE Support

CMSE is an extension to the C language that can be implemented by tool vendors to provide toolchain support for Secure executable files that are written in the C language. Non-secure executable files do not require any additional toolchain support.

The `<arm_cmse.h>` header must be included before using CMSE support, except for using the

ARM_FEATURE_CMSE macro.

Bits 0 and 1 of feature macro `ARM_FEATURE_CMSE` are set if CMSE support for Secure executable files is available.

Availability of CMSE implies availability of the `TT` instruction.

A compiler might provide a switch to enable support for creating CMSE Secure executable files. ARM recommends such a switch to be named `-mcmse`.

1.7 Non-secure memory usage

Secure code must only use Secure memory except when communicating with the Non-secure state.

The security implications of accessing Non-secure memory through a pointer are the responsibility of the developer.

1.7.1 Arguments and return value

A caller from the Non-secure state is not aware it is calling an entry function. If it must use the stack to write arguments or read a result value that uses the Non-secure stack.

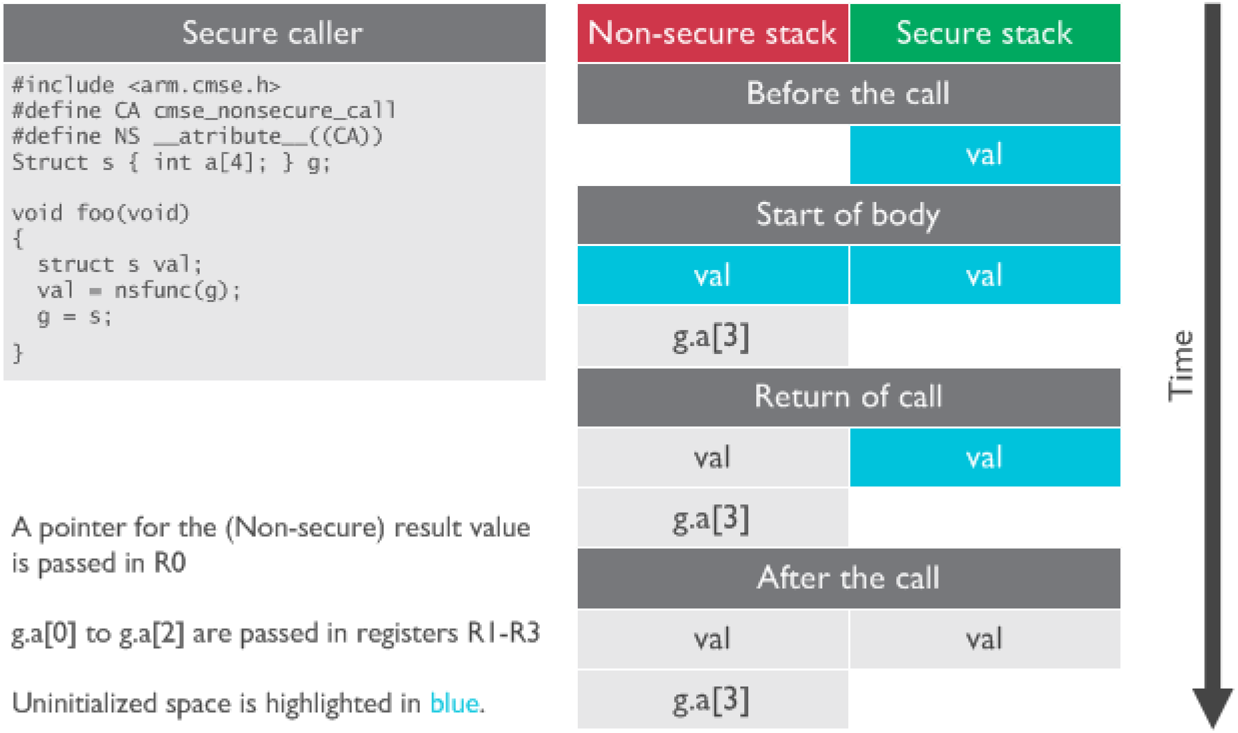
If a toolchain supports stack-based arguments, it must be aware of the volatile behavior of Non-secure memory and the requirements of using Non-secure memory.

In practice, a compiler might generate code that:

- Copies stack-based arguments from the Non-secure stack to the parameter on the Secure stack in the prologue of the entry function.
- Copies the stack-based return value from the Secure stack to the Non-secure stack in the epilogue.

A possible optimization would be to access the Non-secure stack directly for arguments that read at most once, but accessibility checks are still required.

The following figure shows the stack use of an entry function.



Secure caller

```
#include <arm.cmse.h>
#define CA cmse_nonsecure_call
#define NS __atribute__((CA))
Struct s { int a[4]; } g;

void foo(void)
{
  struct s val;
  val = nsfunc(g);
  g = s;
}
```

A pointer for the (Non-secure) result value is passed in R0

g.a[0] to g.a[2] are passed in registers R1-R3

Uninitialized space is highlighted in blue.

1.8 Return from an entry function

An entry function must use the BXNS instruction to return to its Non-secure caller.

This instruction switches to Non-secure state if the target address has its LSB unset. The LSB of the return address in the LR is automatically cleared by the SG instruction when it switches the state from Non-secure to Secure.

Note

To prevent information leakage when an entry function returns, the registers that contain secret information must be cleared.

The code sequence directly preceding the BXNS instruction that transitions to Non-secure code must:

- Clear all caller-saved registers except:
  - Registers that hold the result value and the return address of the entry function.
  - Registers that do not contain secret information.
- Clear all registers and flags that have UNDEFINED values at the return of a procedure, according to the Procedure Call Standard for the ARM Architecture (AAPCS).

- Restore all callee-saved registers as required by the Procedure Call Standard for the ARM Architecture (AAPCS).

Floating-point registers can be cleared conditionally by checking the SFPA bit of the special-purpose CONTROL register.

A toolchain can provide the developer with the means to specify that some types of variables never hold secret information. For example, by setting the TS bit of FPCCR, The Armv8-M Security Extension assumes that floating-point registers never hold secret information.

Because of these requirements, performing tail-calls from an entry function is difficult.

Security state of the caller

An entry function can be called from Secure or Non-secure state. Software must distinguish between these cases. To enable this, the Armv8-M Security Extensions define the following intrinsic:

`int cmse_nonsecure_caller(void)`	Returns non-zero if entry function is called from Non- secure state and zero otherwise.

1.9 Non-secure function call

A call to a function that switches state from Secure to Non-secure is called a Non-secure function call. A Non-secure function call must use function pointers. This is a consequence of separating Secure and Non-secure code into separate executable files.

A Non-secure function type must be declared using the function attribute `_attribute ((cmse_nonsecure_call))`.

A Non-secure function type must only be used as a base type of a pointer. This restriction disallows function definitions with this attribute and ensures that a Secure executable file only contains Secure function definitions.

1.10 Performing a call

A function call through a pointer with a Non-secure function type as its base type must switch to the Non-secure state. To create a function call that switches to the Non-secure state, an implementation must emit code that clears the LSB of the function address and branches using the `BLXNS` instruction.

Note

A Non-secure function call to an entry function is possible. This call to an entry function behaves like any other Non-secure function call.

All registers that contain secret information must be cleared to prevent information leakage when performing a Non-secure function call. Registers that contain values that are used after the Non-secure function call must be restored after the call returns. Secure code cannot depend on the Non-secure state to restore these registers.

The code sequence directly preceding the BLXNS instruction that transitions to Non-secure code must:

- Save all callee- and live caller-saved registers by copying them to Secure memory.
- Clear all callee- and caller-saved registers except:
  - The LR.
  - The registers that hold the arguments of the call.
  - Registers that do not hold secret information.
- Clear all registers and flags that have UNDEFINED values at the entry to a procedure according to the AAPCS.

A toolchain could provide the developer with the means to specify that some types of variables never hold secret information.

When the Non-secure function call returns, caller and callee that are saved registers that are saved before the call must be restored.

An implementation need does not have to save and restore a register if its value is not live across the call. However, callee-saved registers are live across the call in almost all situations. These requirements specify behavior that is similar to a regular function call, except that:

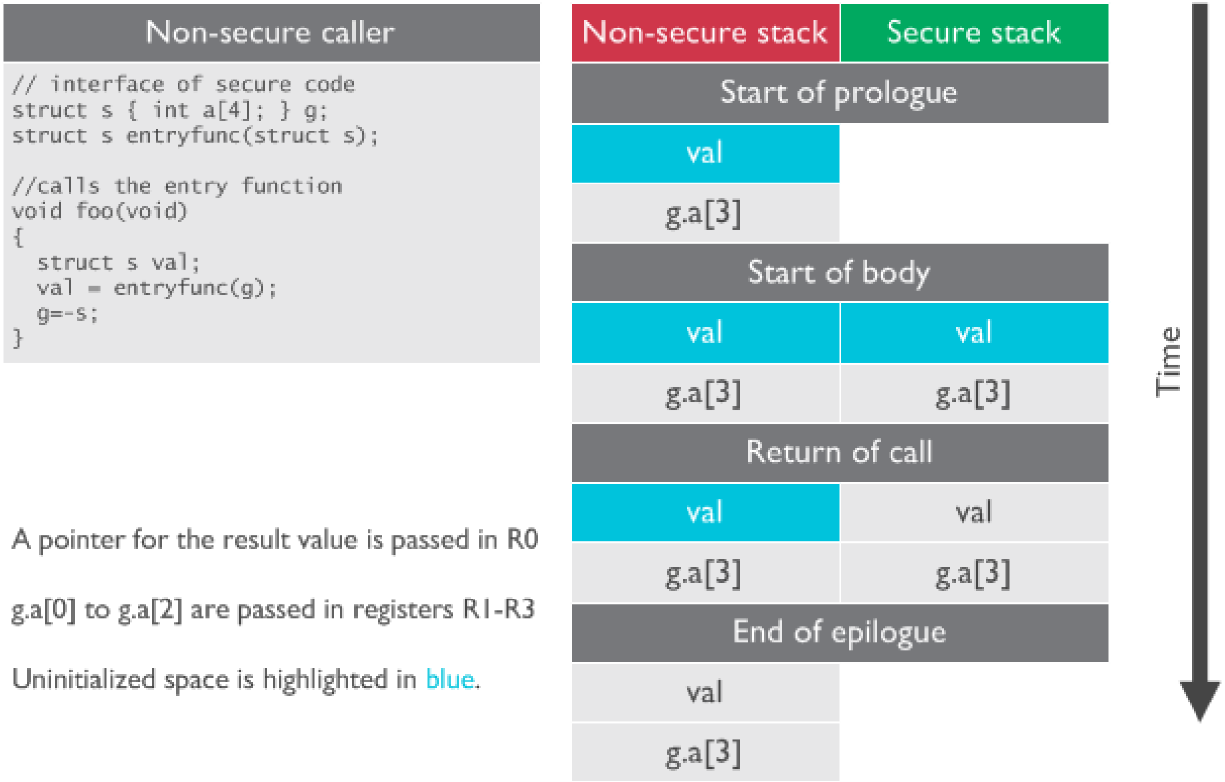
- Callee-saved registers must be saved as if they are caller-saved registers.
- Registers that are not banked and potentially contain secret information must be cleared.

The floating-point registers can efficiently be saved and cleared using the VLSTM instruction, and restored using VLLDM instruction.

1.11 Arguments and return value

The callee of a Non-secure function call is called in Non-secure state. If stack use is required according to the AAPCS, the Non-secure state expects to find the arguments on the Non-secure stack and writes the return value to Non-secure memory.

The stack usage during a Non-secure function call is shown in the following figure:



Non-secure caller

```
// interface of secure code
struct s { int a[4]; } g;
struct s entryfunc(struct s);

//calls the entry function
void foo(void)
{
  struct s val;
  val = entryfunc(g);
  g=-S;
}
```

A pointer for the result value is passed in R0

g.a[0] to g.a[2] are passed in registers R1-R3

Uninitialized space is highlighted in blue.

Non-secure stack	Secure stack
Start of prologue	
val	
g.a[3]	
Start of body	
val	val
g.a[3]	g.a[3]
Return of call	
val	val
g.a[3]	g.a[3]
End of epilogue	
val	
g.a[3]	

Intrinsic	Description
cmse_nsfptr_create(p)	Returns the value of p with its LSB cleared. The argument p can be any function pointer type.
cmse_is_nsfptr(p)	Returns non-zero if p has LSB unset, zero otherwise. The argument p can be any function pointer type.

Note

The exact type signatures of these intrinsics are implementation-defined because there is no type defined by the C programming language that can hold all function pointers. ARM recommends implementing these intrinsics as macros and recommends that the return type of cmse_nsfptr_create() is identical to the type of its argument.

A Non-secure returning function must be declared by using the attribute `_attribute((cmse_nonsecure_return))` on a function declaration.

A Non-secure returning function has a special epilogue, identical to that of an entry function.

1.12 Calling Non-secure functions

Calling a Non-secure function from Secure code requires special code generation to be architecturally correct.

- BLXNS must be used instead of BLX.
- The LSB of the calling address must be zeroed.
- Registers must be sanitized to prevent leaking of Secure data.
- Non-secure functions are prototyped as normal in an interface header. For example:

  ```
  secure_interface.h

  int entry1(int x);

  int entry2(int x);
  ```

1.13 Calling a Non-secure function using CMSE

- Define a Non-secure function pointer using:

  ```
  _attribute_((cmse_nonsecure_call))
  ```

- Create a function pointer using:

  ```
  cmse_nsfptr_create()
  ```

- Validate the function pointer before calling using

  ```
  cmse_is_nsfptr()
  ```

For example

```
Typedef void _attribute_((cmse_nonsecure_call)) nsfunc(void);

Nsfunc *FunctionPointer;

FunctionPointer = cmse_nsfptr_create((nsfunc *) (0x21000248u)); If
(cmse_is_nsfptr(FunctionPointer))

FunctionPointer();
```

2 Data validation and prevention of information leakage

2.1 Non-secure memory access

When Secure code has to access Non-secure memory using an address that is calculated by the Non-secure state, it cannot trust that the address lies in a Non-secure memory region.

Furthermore, the Memory Protection Unit (MPU) is banked between the security states. Secure and Non-secure code might have different access rights to Non-secure memory.

Secure code that accesses Non-secure memory on behalf of the Non-secure state must only do so if the Non-secure state has permission to perform the same access itself.

The Secure code can use the TT instruction to check Non-secure memory permissions.

Take care when using Secure code to access Non-secure memory unless it does so on behalf of the Non-secure state. Data belonging to Secure code must reside in Secure memory.

2.2 The Test Target instruction

To allow software to determine the security attribute of a memory location, the TT instruction (Test Target) is used.

Test Target (TT) queries the security state and access permissions of a memory location.

Test Target Unprivileged (TTT) queries the security state and access permissions of a memory location for an unprivileged access to that location.

Test Target Alternate Domain (TTA) and Test Target Alternate Domain Unprivileged (TTAT) query the security state and access permissions of a memory location for a Non-secure access to that location. These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.

When executed in the Secure state the result of this instruction is extended to return the Security Attribution Unit (SAU) and Implementation Defined Attribution Unit (IDAU) configurations at the specific address.

For each memory region defined by the SAU and IDAU, there is an associated region number that is generated by the SAU or by the IDAU. This region number is used by software to determine if a contiguous range of memory shares common security attributes.

The TT instruction returns the security attributes and region number, and the MPU region number, from an address value. By using a TT instruction on the start and end addresses of the memory range, and identifying that both reside in the same region number, software can quickly determine that the memory range, for example, for data array or data structure, is located entirely in Non- secure space.

The TT instruction is useful for determining the security state of the MPU at that address. Although the instruction cannot be accessed in C/C++ code, there are several intrinsics which make this functionality available to the developer.

The `<arm_cmse.h>` header must be included before using the TT intrinsics.

2.2.1 TT intrinsics

The result of the TT instruction is described by a C type containing bit-fields. This type is used as the return type of the TT intrinsics.

Intrinsic	Semantics
`cmse_address_info_t cmse_TT(void *p)`	Generates a TT instruction.
`cmse_address_info_t cmse_TT_fptr(p)`	Generates a TT instruction. The argument p can be any function pointer type.
`cmse_address_info_t cmse_TTT(void *p)`	Generates a TT instruction with the T flag.
`cmse_address_info_t cmse_TTT_fptr(p)`	Generates a TT instruction with the T flag. The argument p can be any function pointer type.

Note

Arm recommends that a toolchain behaves as if these intrinsics would write the pointed-to memory. That prevents subsequent accesses to this memory being scheduled before this intrinsic.

The exact type signatures for `cmse_TT_fptr()` and `cmse_TTT_fptr()` are IMPLEMENTATION DEFINED because there is no type that is defined by the C programming language that can hold all function pointers.

Note

Arm recommends implementing these intrinsics as macros.

2.2.2 Address range check intrinsic

Checking the result of the TT instruction on an address range is essential for programming in C. It is used to check permissions on objects larger than a byte. The address range check intrinsic defined in this section can be used to perform permission checks on C objects.

Some Secure Attribution Unit (SAU), Implementation Defined Attribution Unit (IDAU), and Memory Protection Unit (MPU) configurations block the efficient implementation of an address range check. This intrinsic operates under the assumption that the configuration of the SAU, IDAU, and MPU is constrained as follows:

- An object is allocated in a single region.

- A stack is allocated in a single region.

These points imply that a region does not overlap other regions.

The TT instruction returns an SAU, IDAU, and MPU region number. When the region numbers of the start and end of the address range match, the complete range is contained in one SAU, IDAU, and MPU region. In this case two TT instructions are executed to check the address range.

Regions are aligned at 32-byte boundaries. If the address range fits in one 32-byte address line, a single TT instruction suffices.

Arm recommends that programmers use the returned pointer to access the checked memory range. This generates a data dependency between the checked memory and all its subsequent accesses and prevents these accesses from being scheduled before the check.

2.2.3 Validation of Non-Secure Pointers

Data pointers passing from the NS world should be validated using TTA/TTAT instructions.

Function pointers passing from the NS world should be:

1. Processed by "cmse_nsfptr_create" (which clears bit 0 of the address value) to indicate that it is Non-secure.
2. Checked by TTA/TTAT e.g. using cmse_TTA_fptr (to test address is indeed Non-secure).

Note

Method (1.) is quicker. If the function pointer is pointing to a Secure address, a Security violation would be triggered when using the pointer with BLXNS/BXNS (because bit 0 of the pointer is 0 but the call/branch is not targeting a Secure address).

Method (2.) has the advantage that it allows the Secure API used for transferring the pointer to return an error status immediately.

2.3 Information leakage

Information leakage from the Secure state to the Non-secure state can occur through parts of the system that are not banked between the security states. The unbanked registers that are accessible by software are:

- General purpose registers except for the stack pointer (R0-R12, R14-R15).
- Floating-point registers (S0-S31, D0-D15).
- The N, Z, C, V, Q, and GE bits of the XPSR register.
- The FPSCR register.

Secure code must clear secret information from unbanked registers before initiating a transition from Secure to Non-secure state.

3 Secure software guidelines

To prevent Secure code and data from being accessed from Non-secure state, Secure code must meet several requirements. The responsibility for meeting these security requirements is shared between hardware, toolchain, and software developer.

There are requirements to use special instructions (`BXNS` and `BLXNS`) to branch to Non-secure code and the requirement to preserve and protect Secure register values before calling Secure functions.

CMSE is an extension to the C language that can be implemented by tool vendors to provide a standard way to generate this code.

3.1 Security with MVE and Floating-point Extension

The Armv8-M architecture includes functionality to clear the floating-point caller saved registers (S0 – S15, FPSCR) to zero on return from exception (including tail-chaining) when FPCCR.CLRONRET is set. FPCCR.CLRONRETS bit controls whether FPCCR.CLRONRET bit is writable from Non-secure state.

The Armv8-M architecture also includes a functionality wherein floating point registers can be treated as Secure Enable or not. This functionality is controlled by FPCCR.TS bit. If FPCCR.TS bit is set to 0, the Floating-point registers are treated as Non-secure even when the PE is in Secure state and, therefore, the callee saved registers are never pushed to the stack on an exception. If the Floating-point registers never contain data that needs to be protected, clearing this bit can reduce interrupt latency. As this field changes on how secure stack frames are interpreted, firmware must take care when modifying this value.

If the Secure software will ever use the floating point and MVE registers for sensitive data, then it should set FPCCR.TS, FPCCR.CLRONRET and FPCCR.CLRONRETS at boot time, and not change it again afterwards.

If the Secure software does not use the floating point and MVE registers for sensitive data, then FPCCR.TS and FPCCR.CLRONRETS can stay at zero and Non-secure privileged software can set FPCCR.CLRONRET to prevent privileged data in the FPU from being visible to unprivileged software.

3.2 Volatility of Non-secure memory

Non-secure memory can be changed asynchronously to the execution of Secure code. There are two possible causes:

- Interrupts that are handled in Non-secure state can change Non-secure memory.
- The debug interface can be used to change Non-secure memory.

There can be unexpected consequences when Secure code accesses Non-secure memory. For example:

```
int array[N]
```

```
void foo(int *p) {
if (*p >= 0 && *p < N) {
// Non-secure memory (*p) is changed at this point array[*p] = 0;
       }
}
```

3.3 Inadvertent Secure gateway

An SG instruction can occur inadvertently. This can happen in the following cases:

Uninitialized memory.

- General data in executable memory, for example jump tables.
- A 32-bit wide instruction that contains the bit pattern `0b1110100101111111` in its first halfword that follows an SG instruction, for example two successive SG instructions.
- A 32-bit wide instruction that contains the bit pattern `0b1110100101111111` in its last halfword that is followed by an SG instruction, for example an `SG` instruction that follows an `LDR` (immediate) instruction.

If an inadvertent `SG` instruction occurs in an NSC region, the result is an inadvertent Secure gateway.

Memory in an NSC region must not contain an inadvertent `SG` instruction.

The Secure gateway veneers limit the instructions that must be placed in NSC regions. If the NSC regions contain only these veneers, an inadvertent Secure gateway cannot occur.

3.4 Dealing with Fault Exceptions targeting Secure State

After a fault exception is triggered in the Secure world, it is recommended to prevent further execution of Non-secure code that could trigger operations (e.g. Non-secure to Secure function calls, exception returns, etc) on the secure context associated with the fault. In some security attack scenarios, while a Secure fault exception (e.g. MemManage or HardFault) is triggered, the Secure stack for that context could be corrupted.

If the corrupted Secure stack is a process stack (i.e. PSP_S was used) and the Secure thread associated with that stack can be terminated, it is safe to resume normal execution.

Otherwise, the system should be restarted. E.g. If the Secure main stack is corrupted there is no safe way to resume operation.

In order to ensure that fault exceptions targeting the Secure state have higher exception priority than Non-secure exceptions, though there are several ways to achieve, below are some of the recommendations:

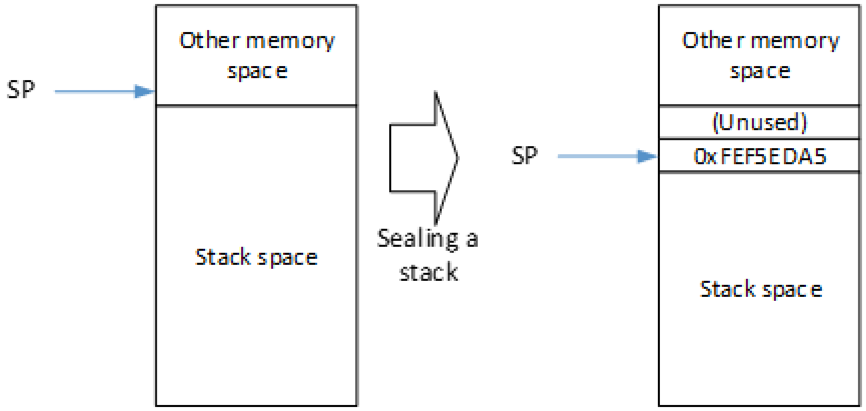
- Set AIRCR.PRIS to 1 and ensure that Secure fault exceptions i.e. the BusFault, UsageFault, SecureFault and MemManage fault for the Secure world are in exception priority range of 0 to 0x7F.

- Alternatively, do not enable BusFault, UsageFault, SecureFault and MemManage faults in the Secure world so that fault events targeting the Secure state escalate to a Secure HardFault.

3.5 Sealing a Stack

If a Secure stack is empty (e.g. when a new thread is created), potentially an attacker could use a fake EXC_RETURN or FNC_RETURN operation to trigger a stack underflow scenario. Because the memory contents above the stack memory could be unpredictable, there is a possibility that the data word above the stack to match a stack frame integrity signature or matching a Secure executable address value.

To ensure such attack to be detected and stopped, Secure software developer can reserve two word of stack memory and place a special value 0xFEF5EDA5 just above the real stack space. Two words of stack space is needed to keep the stack double word aligned. The special value used is never going to match the stack frame integrity signature, and, cannot be used as program address because address range 0xE0000000 to 0xFFFFFFFF is non-executable. This technique is referred as sealing a stack.



To ensure future compatibility Arm recommends that value used to seal the top of stack is 0xFEF5EDA5.

This value has the following properties:

- It is not a valid FNC_RETURN or EXC_RETURN value.
- It is not the integrity signature used to secure the bottom of the stack frame and cannot be used to inadvertently mark the stack as containing a valid exception stack frame.
- The value starts with 0xF and is therefore not a valid instruction address and will result in a fault if interpreted as a FNC_RETURN stack frame.

Note

Sealing of Secure main stack is needed when deprivileging a Secure exception, Refer section: Interrupt Deprivileging.

3.6 Restricting Secure library to be Unprivileged

If a Secure software library should be restricted to unprivileged level only, then:

3. The Secure CONTROL register must setup up CONTROL_S so that Secure thread is unprivileged and use PSP_S as the stack pointer.
4. If the processor being used is based on Armv8.1-M (e.g. Cortex-M55 processor) and if the MPU is available, then the library code memory region can be configured with the PXN MPU attribute to ensure that the code in the region can only execute in the unprivileged state. Under this situation, and the library code region could have its own NSC region(s). Alternatively, a system level protection measure could be implemented to achieve a similar feature. Otherwise, if the processor being used is an Armv8.0-M processor and no system level mechanism is deployed to prevent privileged execution, then the library code memory region must not have the Non-secure Callable (NSC) attribute.
5. The Secure MPU should be set up so that the Secure unprivileged software is not allowed to access privileged memory

3.7 Reentrancy to Secure world

Some Secure APIs might not support re-entrant and if it is the case, additional measures need to be taken to prevent such occurrence. Re-entrant of a Secure API is possible when the following scenario occurs:

1. A Non-secure software calls a Secure API, then
2. The processor takes a Non-secure interrupt, and then
3. Non-secure software then calls a Secure API, result in a re-entrant scenario.

If a Secure API need to be executable in Handler mode, it should be designed to support re-entrant.

If the software API cannot handle re-entrant, it should execute in unprivileged state in Thread mode and the following arrangements could be used to prevent potential software issues:

- If using an Armv8.0-M processor, a Secure API should include addition steps to detect if a previous API call is still in progress before executing the API function code. One approach to implement this is to use a simple "API busy" flag, however care must be taken to ensure that sequence of checking and setting the flag is a thread safe step. This can be achieved by using the LDREX / STREX instructions
- If using an Armv8.1-M processor (e.g. Cortex-M55), then CCR_S.TRD can be set to 1 to detect when a re-entrant occurred and trigger a fault exception. (Note: the CCR_S.TRD is used for protecting against thread mode APIs only).

Note

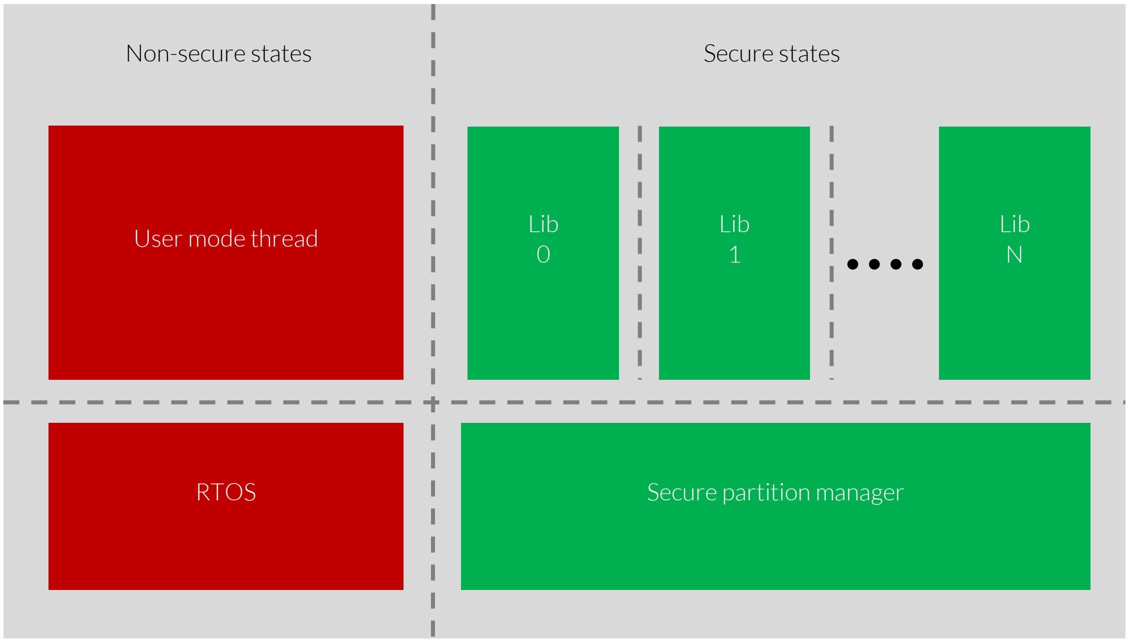
If BFHFNMINS is used, then re-entry to the Secure state must be forbidden. This means:

- all NSC region attributes should be disabled.
- Secure interrupts should be disabled.

- Secure stacks are sealed (Refer: Sealing a Stack)

3.8 Interrupt Deprivileging

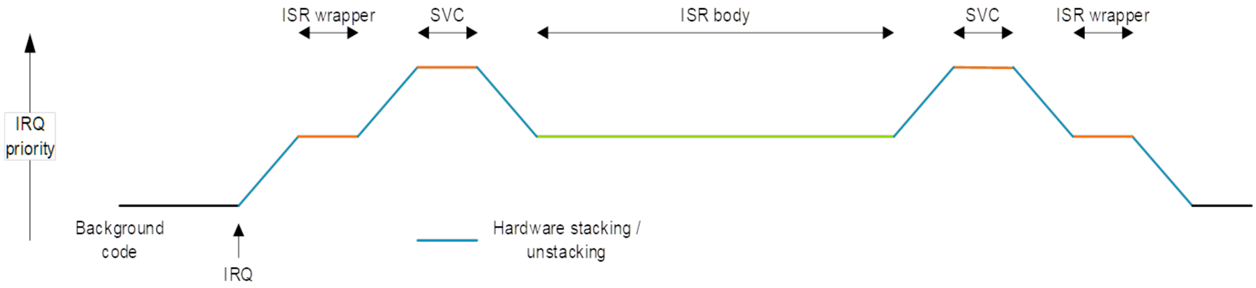
Platform Security Architecture (PSA) comprises multiple building blocks to meet security objectives. The foundation of PSA is a separation of the system into a secure processing environment (SPE) for the sensitive assets and the code that manages them. The SPE is isolated from the Non-secure Processing Environment (NSPE), in which the main application and communication firmware executes. The secure partition manager (SPM) is the Trusted component within the SPE that is responsible for the isolation of the SPE and providing communication between the SPE and NSPE. Secure Partition Manager (SPM) that creates independent secure domains and provides hardware-enforced compartments, called partitions, for individual code blocks by limiting access to memories and peripherals.



As one of the major functionality of secure partition manager (SPM), it forwards and deprivileges interrupts to the unprivileged handler that has been registered for them. For example, if there is an interrupt is targeted for secure partition (Lib-1 from above diagram), then secure partition manager registers this interrupt request before giving the control to Secure-Partition (Lib-1) to execute in unprivileged mode.

3.8.1 Concept of Deprivileging Interrupts:

To enable secure services of peripherals where some of the peripherals communicate via interrupts, it is important to provide secure partitions as a mean to handle interrupts along with isolation levels. In order to keep the PSA isolation levels in-tact, interrupts are isolated using concept of deprivileging interrupt as depicted in below diagram.



Here are a set of brief architectural steps involved in deprivileging an interrupt.

1. In a privileged background code sequence
2. Program priority for both IRQ and SVC : IRQ with a priority level lower than SVC
3. Trigger IRQ
4. Enter IRQ handler (acts as a wrapper to create an unprivileged code container)
    a. Save Callee registers and clear contents of registers in IRQ handler. You also need to context switch any other state required to run the unprivileged context. Usually this is just swapping Process stack pointer (PSP) and the MPU configuration.
    b. Execute SVC instruction to deprivilege the execution
        i. Enter SVC Handler and do all context save required
        ii. Modify CONTROL.npriv to be 1 (if not already programmed during your boot-up process)
        iii. The SVC then fakes up an exception return stack frame to enter an unprivileged thread mode. PSP should be used to prevent codes in a partition from affecting privileged software, which uses MSP.
    c. On SVC exception return, you are in 'Unprivileged thread mode', but still running with the execution priority of the IRQ, so only a higher priority interrupt can pre-empt. This can also be considered as a one of the sandbox executing in unprivileged mode.
    d. Execute SVC instruction to wind back to IRQ handler (wrapper code)
        i. Retrieve back all context switch performed in point:b
        ii. Remove fake stack created in point: b)
        iii. and return to the IRQ handler (wrapper code).
    e. Execute BX LR in IRQ handler to return back to privileged background thread mode. You also need to restore the callee registers saved in step: a), and swap back any other bits of context (eg. Process Stack Pointer and MPU configuration)
5. Get back to privileged background thread mode.

When Secure interrupt handler need to be executed in unprivileged level, then in that case, the Secure interrupt(s) should be intercept by Secure firmware and a deprivileging process is needed to switch the processor to unprivileged level. In addition to creating the fake stack frame on the process stack, the SVC handler should also seal the main stack. (Refer: Sealing a Stack)

Key questions that may arise:

- Why I need two SVC instruction? Why can't I change my execution privilege in IRQ handler?
    - The important thing to note is that if we change privilege in IRQ handler and return back to thread mode, then the execution priority would drop which means that an unprivileged secure software can be pre-empted by non-secure interrupts and secure partition manager (SPM) shall loose the control of its partitions. Hence it is required to have one SVC exception to save the context and the other SVC to restore the context. It is worth noting that IPSR cannot be written by software and hence we need to fake the exception return stacked value in order to get into unprivileged thread mode.
- Does this method of deprivileging interrupt is valid only when Security extensions are implemented?
    - No, deprivileging interrupt concept can be used in processors when Security extensions are not implemented as well (Eg. Cortex-M3)