SVE Impact on Secure Firmware

Document number:	ARM DEN 0091
Release Quality:	EAC
Issue Number:	1.1
Confidentiality:	Non-Confidential
Date of Issue:	October 2019

Contents

About this document

This is a white paper which supports but does not form part of the SVE architecture specification [sve].

Scope

This document describes the impact of the Scalable Vector Extension [sve] for Armv8-A [armv8] on software that is executing in Secure states on the application processor. This document considers the impact of deploying existing Armv8-A TrustZone software on systems implementing SVE and provides recommendations for updating that software and the design of future Secure software for systems that implement SVE.

Release information

The change history table lists the changes that have been made to this document.

Date	Version	Confidentiality	Change
May 2016	1.0	Confidential	Initial release
October 2019	1.1	Non-Confidential	License changed to Non-Confidential

SVE Impact on Secure Firmware

Arm Non-Confidential Document Licence ("Licence")

Any breach of this Licence by a Subsidiary shall entitle Arm to terminate this Licence as if you were the party in breach. Any termination of this Licence shall be effective in respect of all Subsidiaries. Any rights granted to any Subsidiary hereunder shall automatically terminate upon such Subsidiary ceasing to be a Subsidiary.

The Document consists solely of commercial items. Licensee shall be responsible for ensuring that any use, duplication or disclosure of the Document complies fully with any relevant export laws and regulations to assure that the Document or any portion thereof is not exported, directly or indirectly, in violation of such export laws.

If any of the provisions contained in this Licence conflict with any of the provisions of any click-through or signed written agreement with Arm relating to the Document, then the click-through or signed written agreement prevails over and supersedes the conflicting provisions of this Licence. This Licence may be translated into other languages for convenience, and Licensee agrees that if there is any conflict between the English version of this Licence and any translation, the terms of the English version of this Licence shall prevail.

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

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: LES-PRE-21585

References

This document refers to the following documents.

Ref	Document Number	Author(s)	Title
[armv8]	ARM DDI 0487E.a	Arm Ltd	Arm® Architecture Reference Manual Armv8 for Armv8-A architecture profile
[sve]	ARM DDI 0584A.f	Arm Ltd	Arm® Architecture Reference Manual Supplement The Scalable Vector Extension (SVE), for Armv8-A
[tfa]	Trusted Firmware-A	Arm Ltd	https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/about/

Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
SVE	Scalable Vector Extension. An extension to the Armv8-A architecture that introduces a new set of vector registers and instructions.
GP	General-purpose (scalar integer)
FP	Floating-point
SIMD	Single Instruction Multiple Data
AdvSIMD	Advanced SIMD instruction set
SIMD and Floating-point	AdvSIMD and Floating-point instruction set
SIMD&FP	SIMD and Floating-point
SIMD&FP registers	Common register file shared by SIMD and Floating-point instructions
Normal world	The execution environment when the processor is in the Non-secure state.
Secure world	The execution environment when the processor is in the Secure state. When the processor is in Monitor mode it is in Secure state.
TEE	Trusted Execution Environment. An environment that runs alongside but isolated from other execution environments, and that, because of the rules of the environment, can be trusted in clearly-defined ways. To achieve this, a TEE has security capabilities and meets certain security-related requirements.
Trusted OS	This is the operating system running in the Secure world. It supports the TEE.
Rich OS	Application operating system such as Linux or Windows.

Conventions

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace`

Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the Glossary.

Red text

Indicates an open issue.

Blue text

Indicates a link. This can be

- A cross-reference to another location within the document
- A URL, for example http://infocenter.arm.com

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`.

In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode

is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

Assembler syntax descriptions

This book is not expected to contain assembler code or pseudo code examples.

Any code examples are shown in a `monospace` font.

Current status and anticipated changes

First draft, major changes and re-writes to be expected.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send an e-mail to errata@arm.com. Give:

- The title (SVE Impact on Secure Firmware).
- The number and issue (ARM DEN 0091 1.1 EAC).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

1 Introduction

SVE introduces an extension to the vector processing capabilities of Armv8-A, in particular, a new set of scalable vector and predication registers of implementation-defined size to the AArch64 Execution state.

In line with the architecture of the existing SIMD&FP registers, the state of the SVE registers is not affected by changes to Exception level or Security state. If required, providing separate virtual copies of this state to the Normal and Secure worlds must be managed by software.

The least-significant 128 bits of each SVE vector register overlays the correspondingly numbered SIMD&FP register, and any modification of the SIMD&FP register corrupts the SVE content. Software which currently switches the SIMD&FP register state must be updated or supplemented in order to preserve the additional SVE register state.

The most privileged Execution state in Armv8-A (EL3) is designed to provide an environment for switching execution between Normal and Secure worlds. This is implemented in a software program, sometimes called a Secure Monitor.

The obvious design for managing the SVE register state between Normal and Secure worlds is to save and restore the SVE registers state in the Secure Monitor. In practice, this solution may be undesirable due to the memory and performance impact. Existing Secure software for the EL3 and Secure-EL1 Execution states has taken various approaches to managing the SIMD&FP register state, depending on the design of the Monitor and the Trusted OS and on their use of the SIMD&FP registers.

The use cases for Secure software vary across market segments. This means that the optimal approach to managing the SVE register state between Secure and Non-secure software may depend on the full system and software design.

This document reviews the current approaches to SIMD&FP state management in Secure software today, evaluates the options when including systems with SVE, and makes recommendations for developing Secure software that will run on systems that implement SVE. We assume that software in the Normal world is required to have full access to SVE.

2 SVE impact on Secure software

2.1 Boot and runtime usage

It is valuable to distinguish between the use of SIMD&FP and SVE during Secure Boot of an SoC and subsequent runtime usage by Secure software.

2.1.1 Vector register usage during boot

There is typically a clear point of handover from a Secure boot phase, during which the Secure Runtime services are loaded and initialized, and the Non-secure boot phase when Non-secure firmware, hypervisors, and operating systems are loaded and initialized. This is usually the point at which the Secure Runtime firmware first hands over execution to the Non-secure bootloader.

SVE places simple requirements on the Secure boot software in order to:

- Configure the processor for use of SVE
- Ensure that any residual register state from using SVE during boot has been wiped prior to execution of Non-secure software

2.1.2 Vector register usage at runtime

After boot, the use of SIMD&FP or SVE registers by Secure software must be carefully managed. This is to ensure that the Non-secure SVE state is maintained and that any Secure register state is not leaked to Non-secure execution.

The simplest way to implement this requirement is for the Secure Monitor running at EL3 to swap the SIMD&FP and SVE register states. This software is already responsible for swapping the general purpose and system register states when switching between Secure and Non-secure execution.

However, this approach substantially increases the memory required to maintain the register contexts, as shown in Table 2-1.

GP and EL1 System registers	0.5KB
SIMD&FP registers	+ 0.5KB
SVE registers	+ 0.5KB to 8.5KB

Table 2-1: PE state save/restore memory requirements.

A typical Monitor would manage two such contexts for every CPU. Arm currently recommends that the EL3 runtime software memory is in internal SRAM for security reasons. However, the additional memory requirement for SIMD&FP and SVE register state may be undesirable.

2.2 SIMD&FP usage in current Secure runtime software

In current systems using SIMD&FP instructions, Secure software designers have developed various strategies to balance the Secure memory requirements against the benefits of utilizing these instructions. There are four main approaches that are based on the degree to which the Secure software at Secure-EL1 makes use of SIMD&FP. These approaches exploit the knowledge of the Trusted OS design and implementation to reduce the memory and performance impact of saving the SIMD&FP register state.

The approaches are summarized in the following table:

SIMD&FP at S-EL1	Non-secure state	Memory usage	Software complexity
Not used or disabled	Retained in CPU	Zero	Low
Limited use	Save/restore as needed	Up to 0.5KB at S-EL1 on stack or per-task	Medium
General use	Save/restore at S-EL1	1KB at S-EL1 per CPU	High
Unlimited use	Virtualise at EL3	1KB at EL3 per CPU	Medium

Many TEEs today use the Limited use or General use approaches.

2.2.1 No use of SIMD&FP Secure software

For systems where there is no use of SIMD&FP in the Secure software after boot, the Non-secure SIMD&FP register state can remain resident in the CPU registers and does not need to be managed by the Secure software.

For protection against software defects or misconfigured tools, the Secure software may enable the architected traps to prevent Secure software from accidentally touching these registers.

2.2.2 Limited use of SIMD&FP Secure software

Many Trusted OS implementations make limited use of the SIMD&FP registers, for example for software implementation of cryptographic algorithms. These implementations will only save and restore the Non-secure SIMD&FP register state before and after such usage.

This enables the Trusted OS to significantly optimize the memory usage and performance impact by only saving those registers that are used for the computation. In addition, where the Trusted OS only executes one Secure task at a time, only a single set of SIMD&FP registers must be saved.

This design depends on the Trusted OS retaining control of all exceptions to EL3 so that the Non-secure register state can be restored before any return to Non-secure execution. Interrupts must be first handled by the Trusted OS exception vectors instead of being trapped to EL3.

2.2.3 General use of SIMD&FP in Secure software

Where the Trusted OS makes more extensive use of SIMD&FP, or might do so concurrently on multiple CPUs, the SIMD&FP register state can be managed by the code in the Trusted OS that controls entry and exit from/to EL3. This is like the EL3 solution, but places the logic and policy for the save and restore within the Trusted OS, which is the component that is using the functionality.

Also, Trusted OS implementations often have much larger memory requirements than the EL3 software. This means that the Trusted OS implementations can more easily accommodate any additional memory that is required for the SIMD&FP register state.

The Trusted OS software may implement the switch lazily, by using the CPACR_EL1 trap functionality. This would be a performance advantage if only a subset of Secure operations made use of the SIMD&FP registers.

2.2.4 Unlimited use of SIMD&FP in Secure software

The SIMD&FP register state is switched at EL3, and maintained alongside the GP and system register state for the Non-secure and Secure execution environments.

The EL3 software may implement the switch lazily, by using the CPTR_EL3.TFP trap functionality. This would be a performance advantage if only a subset of Secure operations made use of the SIMD&FP registers.

2.3 SIMD&FP or SVE usage in Secure runtime software

For systems that implement SVE and use SVE functionality in Non-secure software, some of the Secure software must be aware of the SVE implementation. This is to preserve the Normal world SVE state if the Secure software uses any of these register sets.

Because saving and restoring the SVE register state has larger memory and performance implications than saving and restoring SIMD&FP, we expect to find that a similar range of designs are appropriate for managing the SVE state.

SVE extends the control registers at EL3 and EL1 to enable the same control of SVE registers that is currently available for SIMD&FP register use. This means that existing approaches to deny access or provide lazy switching to SIMD&FP registers are readily extended to SVE register state. The only difference is that SVE is only accessible from AArch64.

2.3.1 No use of SIMD&FP or SVE in Secure software

For systems where floating point or vector functionality is not used in Secure runtime software, there is no change in the implementation requirements. The Normal world SIMD&FP and SVE state is always maintained in the CPU.

2.3.2 Use of AArch32 TEEs

TEEs that do not support execution in AArch64 will have no access to the SVE instructions or register state, although they might make use of SIMD&FP. If such a TEE is used in a system that implements SVE, then the Secure Monitor must save and restore the Normal world SVE state on entry and exit of the TEE.

This would have most of the memory and performance disadvantages of the Unlimited use design.

2.3.3 Use of unmodified third-party AArch64 TEEs

For market segments that are currently using third-party Trusted OSs or TEEs that do support AArch64, you might consider designs that avoid the need for the third-party code to be updated or made aware of SVE when they are deployed on SVE systems.

If the TEE is using SIMD&FP, then the existing preservation of the Normal world SIMD&FP state is inadequate to preserve the SVE state. The only approach that does not modify the TEE software is for the Secure Monitor to save and restore the Normal world SVE state on entry and exit of the TEE, in addition to the SIMD&FP state management that is carried out by the TEE.

Although correct, the approach mentioned in the previous paragraph would have most of the memory and performance disadvantages of the Unlimited use design.

2.3.4 Make all Secure software SVE-aware

Instead of maintaining the Non-secure SVE state in the Monitor, it is more efficient for most systems to update the Secure software that currently maintains the Non-secure SIMD&FP state to instead save and restore the Non-secure SVE state.

This approach requires that all vendors of Secure software that use SIMD&FP must update their software for SVE systems. Even if the software will not immediately exploit the SVE functionality, it must ensure that Normal world SVE state is preserved in the same way that SIMD&FP state is preserved today.

This approach maintains the existing benefits of the system-specific approaches that are currently deployed.

3 Recommendations

3.1 All systems implementing SVE

Secure Boot and CPU initialization software must be implemented and updated to configure the SVE functionality for use by Normal world software, and to ensure that any residual Secure register contents from Secure Boot usage is overwritten before Non-secure software executes.

3.2 Systems using existing Secure software

Arm recommends that all Secure software that makes use of SIMD&FP is updated to be able to execute in AArch64 and become SVE-aware.

Support for AArch64 ensures that the software is relevant for all Armv8-A processor designs, including those without support for AArch32 at EL1. Being SVE-aware enables optimization of memory and performance, by selecting a design that best matches the Secure software architecture.

Being SVE-aware may take the form of specific builds for systems implementing SVE, or use of runtime detection of the presence of SVE.

3.3 Systems implementing new Secure software

Secure software designers and developers should consider the costs and benefits of using SIMD&FP or SVE functionality in Secure software. Designers should choose a design that will preserve the Normal world SVE state, based on the overall Secure software architecture in their system. The four designs outlined in Section 2.2 provide some of the possible approaches.

Arm expects that for some of the markets in which SVE is initially deployed, the Secure software will not need to use SIMD&FP or SVE functionality and that the simplest design – No use – is appropriate.

4 Appendix - Example implementation for Limited Use design

This section provides pseudocode examples of the changes required in EL3 and Secure-EL1 software to implement a Limited Use design, as described in section 2.2.2.

In this design, the Secure Monitor at EL3 does not save or restore any SIMD&FP or SVE state, but switches the access configuration register CPACR_EL1. This design requires that the Secure-EL1 software executes in AArch64 state, which removes any need to manage FPEXC.

The changes illustrated in the example assume that the software is running on an implementation with SVE. Additional compile-time or run-time logic is required for the software to support use on both SVE and non-SVE systems.

4.1 Changes needed in EL3 software

The software at EL3, for example an instance of Arm Trusted Firmware [tfa], will need to initialize the SVE control state for EL3 on every CPU reset.

In addition, if the Secure firmware is configured to run the Non-secure software at EL1, then the EL3 firmware must also initialize the SVE control state for EL2 before entering EL1.

This design does not require EL3 software to manage any Non-secure SVE state that may be held in the CPU registers. This responsibility is delegated to the software running at Secure-EL1.

On every CPU reset, the following register fields must be updated:

```
ZCR_EL3.LEN    = 0xf
CPTR_EL3.EZ    = 1
```

Before a CPU first enters a Non-secure state, if the initial target Exception level is EL1 instead of EL2, then the following register fields must also be updated:

```
ZCR_EL2.LEN    = 0xf
CPTR_EL2.TZ    = 0
```

In Arm Trusted Firmware, those changes would be implemented in the `el3_arch_init_common` macro and `cm_prepare_el3_exit()` functions respectively.

4.2 Changes needed in Secure-EL1 software

The software at Secure-EL1 must ensure that the Non-secure SVE state is saved prior to any use of FP, AdvSIMD or SVE functionality and restored again before returning to Non-secure execution. This would replace any existing code that saves and restores the SIMD&FP state in the pre-SVE software.

In this example design, the CPU's Secure-EL1 copy of CPACR_EL1 is initialized with FPEN=0 and ZEN=0 to trap accidental use of SIMD&FP or SVE registers. These fields are set to 0b11 when the Non-secure state is saved and are returned to zero after the Non-secure state is restored.

It is not possible to determine whether the CPU contains any valid Non-secure SVE state based on the [Non-secure] EL1 and EL2 SVE configuration registers – as these may indicate trapped or disabled functionality purely to provide lazy switching behavior. In addition, the current [Non-secure] value of the ZCR_EL1.LEN register cannot be relied on to indicate the size of the SVE state that needs to be preserved. Secure-EL1 must assume that the full hardware register contents must be saved and restored. To do this, it must update ZCR_EL1.LEN before storing the SVE registers (Z0-Z31, P0-P15 and FFR).

```
int MAX_VLEN = 0xf

struct sve_state
{
     zreg    z[32]           // zreg sized for this implementation
     preg    p[16]           // preg sized for this implementation
     preg    ffr
     reg64   zcr
     reg32   fpsr
     reg32   fpcr
}

save_ns_sve_state(sve_state)
{
     assert(CPACR_EL1.FPEN == 0)

     CPACR_EL1.FPEN = 0x3
     CPACR_EL1.ZEN = 0x3
     isb()
     sve_state.zcr = ZCR_EL1
     sve_state.fpsr = FPSR
     sve_state.fpcr = FPCR
     ZCR_EL1.LEN = MAX_VLEN          // no isb as self-synchronizing
     sve_state.z[0] = Z0
     …
     sve_state.z[31] = Z31
     sve_state.p[0] = P0
     …
     sve_state.p[15] = P15
     sve_state.ffr = FFR
}

restore_ns_sve_state(sve_state)
{
     assert(CPACR_EL1.FPEN == 0x3)

     Z0 = sve_state.z[0]
     …
     Z31 = sve_state.z[31]
     P0 = sve_state.p[0]
     …
     P15 = sve_state.p[15]
     FFR = sve_state.ffr
```

```
      ZCR_EL1 = sve_state.zcr
      FPSR = sve_state.fpsr
      FPCR = sve_state.fpcr
      CPACR_EL1.FPEN = 0
      CPACR_EL1.ZEN = 0
      isb()
}
```