Introduction

As observed by many security researchers, speculative execution of certain instruction sequences may cause changes in caches and similar structures where the location allocated is indicative of secrets, and timing analyses of other instruction sequences can then reveal those secrets to an attacker. Such attacks are well documented and given names such as Spectre, Meltdown, and others. Arm has published a **whitepaper** giving an overview of these cache speculation side-channels while also outlining possible mitigations. Arm also has an **FAQ** page on the subject that is regularly updated with the latest guidance.

In general, the Arm architecture does not stipulate requirements for most instructions regarding speculation, to permit implementations the freedom to leverage speculation for improved performance. However, following research from the **Google SafeSide project**, Arm is systematically documenting the possibilities for a processor to speculatively execute the instructions immediately following what should be a change in control flow, including:

- Exception generating instructions (SVC, HVC, SMC, UNDEF, BRK)
- Exception returns (ERET)
- Unconditional direct branches (B, BL)
- Unconditional indirect branches (BR, BLR)
- Function returns (RET)

Arm refers to this as "straight-line speculation" past an unconditional change in control flow and has allocated CVE-2020-13844 accordingly.

For example, given the following code sequence:

```
1    ...
2    ...
3    ADRP  x3, __foo
4    ADD   x3, x3, :lo12:__foo
5    BLR   x3                        // x0 = __foo(x0);
6    LDR   x1, [x0]                  // Use result to access memory
7    ...
```

Straight-line speculation would involve the processor speculatively executing the next instructions linearly in memory past the unconditional change in control flow, i.e. speculatively executing the LDR past the BLR.

Where the speculative path contains a suitable code sequence, often described by researchers as a "Spectre Revelation Gadget", such straight-line speculation could lead to changes in the caches and similar structures that are indicative of secrets, making those secrets vulnerable to revelation through timing analysis.

Take for example an operating system kernel which may include a function that operates on user data, such as a user memory copy routine, where the general purpose registers contain data retrieved from user memory (EL0 controlled) at the time of the function return, despite the fact that the function is executed in kernel space (e.g. at EL1).

If, immediately following the function return there is a code sequence of the form:

```
1   ...
2   ...
3   RET
4   LDR  Xa, [Xb]
5   AND  Xa, Xa, #0xFFFF
6   LDR  Xc, [Xd, Xa]
7   ...
```

where `Xb` and `Xd` are registers that contain data retrieved from user memory, then an attacker at EL0 could select the contents of `Xb` and `Xd` to be such that the speculative execution of this sequence causes changes in the caches and other structures that are indicative of the contents of a memory location in kernel space; the attacker at EL0 could then perform a timing analysis to reveal that kernel space data.

It is expected that such gadgets are rarely found (and in some cases may have been eliminated as part of mitigations for Spectre v1) and that this is difficult to exploit in practice. Nevertheless, the possibility cannot be dismissed and for this reason Arm is recommending that speculation barrier sequences be inserted immediately following certain unconditional changes in control flow where threat modelling shows that this vulnerability needs to be mitigated in that project.

Speculation barrier sequences

The following instruction(s)** prevent speculative execution past the instruction(s) from altering any state of the system in a way that could be observed through side-channels:

- `SB`
- `DSB` followed by `ISB` (`DSB+ISB`)

Of these, the `SB` instruction is only available on processors that implement the ARMv8.0-SB extension; processors not implementing ARMv8.0-SB will therefore need to fall back on a `DSB+ISB` sequence.

While a `DSB+ISB` sequence is expected to have a significantly greater impact on performance than an `SB` if architecturally executed, in many of the cases outlined in this document the `DSB+ISB` sequence is not architecturally executed and should have no direct impact on performance save for a reduction in code density. Secondary effects may include marginally increased pressures on the instruction caches and branch predictors.

> **While the `CSDB` instruction can also be used to limit the scope of speculative execution and data value prediction, it is not discussed here.

B, BL

Arm has been unable to demonstrate straight-line speculation past these instructions and believes no action is required.

RET, BR

Where threat modelling shows that this vulnerability needs to be mitigated in a project, Arm recommends placing an SB/DSB+ISB sequence immediately following any instance of these instructions.

Given that execution is not expected to return to the next instruction in program order, the SB/DSB+ISB sequence will not be architecturally executed and should have no direct impact on performance save for a reduction in code density. Secondary effects may therefore include marginally increased pressure on the instruction caches.

Arm is working to develop toolchain solutions for gcc and llvm to allow for the automatic addition of these sequences. Links to the patches are available on the **FAQ page**.

ERET

The same guidance applies as for RET and BR.

That said, given that a typical codebase is expected to have only a small number of ERET instructions, it may be practical to assess these on a case-by-case basis and only apply an SB/DSB+ISB sequence immediately following the ERET if it may be executed while the GPRs contain adversary-controlled values.

SVC/HVC/SMC that are *not* expected to return

The same guidance applies as for ERET.

SVC/HVC/SMC that *are* expected to return

Arm believes these instructions only present an issue in cases where *all* the following conditions are met:

- The SVC/HVC/SMC call is followed linearly in memory by an instruction sequence that consumes memory locations or GPR values modified by the SVC/HVC/SMC call.
- That instruction sequence forms a revelation gadget.
- Those memory locations or GPR values are adversary controlled prior to modification by the SVC/HVC/SMC call.

Where all these conditions are met, Arm recommends placing an SB/DSB+ISB sequence between the SVC/HVC/SMC call and the revelation gadget.

It should be noted that the instructions that appear after the SVC/HVC/SMC are expected to be executed on return from the SVC/HVC/SMC and so any register state that is not modified by the SVC/HVC/SMC call will be used non-speculatively by the code after the return and not be a security issue. Correspondingly, it is expected to be extremely uncommon that a revelation gadget will exist after such a call, particularly as SVC/HVC/SMC calls typically make minimal alterations to GPR state, such as a single return value in X0 with all other GPRs restored to their pre-call state.

For example, in the following sequence:

```
1  ...
2  ...
3  SVC  #0
4  LDR  Xa, [Xb]
5  AND  Xa, Xa, #0xFFFF
6  LDR  Xc, [Xd, Xa]
7  ...
```

Here, if Xb and Xd are not modified by the SVC call then they are immaterial as executing this sequence would have the same effect regardless of whether it was speculatively executed before the SVC or after it; this sequence is only vulnerable when Xb and/or Xd are modified by the SVC call but the sequence is speculatively executed using the pre-call values and those pre-call values are adversary controlled.

UNDEF, BRK

These instructions are not typically used to cause an expected change in control flow such that straight-line speculation past these instructions poses any practical speculative exploit. As such Arm believes no action is required for these instructions.

BLR

Similar guidance applies to BLR as for SVC/HVC/SMC calls that are expected to return, with the additional caveat that there may be function-local workarounds that are not available for SVC/HVC/SMC.

That said, manually inspecting each instance of BLR is likely to be deemed impractical given that:

- a typical codebase is expected to have orders of magnitude more BLR instructions than SVC/HVC/SMC calls.
- The majority of BLR instructions are generated by tooling rather than being handwritten.

With that in mind, Arm is currently investigating a proposal for tooling support to replace all instances of BLR with a BL+BR sequence, for example:

```
1  ...
2  BL __call_indirect_x<N>      // where x<N> could be e.g. x5
3  ...
4  ...
5
6  __call_indirect_x<N>:
7  BR x<N>
8  <SB/DSB+ISB>
```

Arm is working to develop toolchain solutions for gcc and llvm to allow for the automatic addition of these sequences. Links to the patches are available on the **FAQ page**.

Conclusions

The mitigations described in this document require tooling support that is still in the early stages of architecting, planning, and development.

Arm believes it may take some time for this tooling support to mature and is recommending an incremental approach. For example, a command-line compiler switch for applying the mitigations everywhere would at least help to prevent the exploitation of existing gadgets.

If there is suitable drive from the ecosystem, this could be followed by per-function overrides e.g. using function attributes, allowing developers to investigate security vs performance trade-offs for their projects and use-cases.

Document history

Version/Issue	Date	Confidentiality	Change
1.0	8 June 2020	Non-Confidential	First release.