

Introduction

The following guide offers a high-level overview of the software impact and mitigations for the exploits known as Spectre and Meltdown, more correctly identified as:

Variant 1: bounds check bypass (CVE-2017-5753)

Variant 2: branch target injection (CVE-2017-5715)

Variant 3: rogue data cache load (CVE-2017-5754)

Arm has additionally identified a variant 3a (CVE-2018-3640) and this is also discussed [here](#).

Variant 4: speculative bypassing of stores by younger loads despite the presence of a dependency (CVE-2018-3639)

Note: It is not intended as a definitive document but should be considered a supporting note to the whitepaper supplied by Arm and available [here](#).

Overview

All variants are based on speculative memory access causing cache allocation. Timing analysis of memory accesses can then be used to reveal data that would otherwise be kept secret. Each mechanism is slightly different, and some are microarchitecture dependent. The impact of each variant on Arm cores, plus the solutions and how they should be applied, are described below.

Variant 1 (CVE-2017-5753)

Variant 1 uses speculative execution to coerce a privileged piece of software into revealing secrets by providing it with illegal, untrusted values. This can occur even if the privileged software sanitises the values by performing explicit bounds-checking.

In this context, ‘privileged’ can mean a boundary between software running at different architectural ELs (Exception Levels). For example, between userspace and kernel or kernel and hypervisor. It also impacts any kind of ‘sandboxing’ software (JavaScript engine, for example) running at the same architectural Exception Level.

It is not possible for hardware to automatically determine where these exploitable code sequences are. Nor is it practical to implement generic fixes in software, by inserting existing barrier instructions at all entry points to privileged code, without incurring significant performance degradation.

The solution proposed by Arm is to introduce a performance optimized code sequence that mitigates speculation and enforces validation of the limits of the untrusted value. Such code sequences are based around specific data processing operations (conditional select, for example) and a new CSDB barrier instruction.

For CPUs capable of data value prediction, CSDB waits for any outstanding predictions to architecturally resolve before allowing speculative execution to continue. To be sure of defending against this exploit now and in the future, any mitigating sequence should include both a suitable data processing sequence and a CSDB.

This sequence needs to be inserted at specific locations where exploitable code is identified. Such sequences are thought to be uncommon, but deploying a fix is complex.

- Many types of software are affected. This is not limited to kernel and hypervisor code, but includes anything that offers to provide some isolation in the way of JIT, including, for example, web browsers and Java runtimes.
- There is as yet, no released tool that provides a reliable, automated way of identifying where to deploy this barrier in code. Arm is investigating/developing code scanners, however, thus far, they are impractical as they produce too many false positives. We will continue to pursue this avenue, but for now are unable to make a code scanner publicly available.

In Linux: Where vulnerable code sequences have been identified, impacted kernel/driver code should be reworked to use the new cross architectural 'nospec' accessors. These accessors implement suitable Arm CSDB sequences and ensure the compiler generates speculation safe code sequences for bounds checks.

The KPTI (4.15) integration branch includes an implementation of the nospec helpers:

<https://git.kernel.org/pub/scm/linux/kernel/git/arm64/linux.git/log/?h=kpti>

There is ongoing work to identify locations where speculative execution could leak information. The eBPF JIT has been identified as one area. Other areas of the kernel are also being investigated.

In Userspace: For applications which attempt to provide software isolation (Java runtimes, JITs, etc.) between components running in the same process. The vendor needs to determine if exploitable code sequences can be identified. Work is underway in Arm to add support for a builtin function, `__builtin_load_no_speculate`, to toolchains. This is being added to GNU and LLVM, and will implement the necessary barrier sequence.

Patches have been upstreamed for both AArch64 and AArch32. However, the implementation is under community review and subject to change. References:

- <https://gcc.gnu.org/ml/gcc-patches/2018-01/msg00205.html>
- <https://reviews.lvm.org/D41761>
- <https://reviews.lvm.org/D41760>

Until these patches are stable the advice is to use the generic header file and function approach detailed [here](#).

Variant 2: (CVE-2017-5715)

Variant 2 relies on training the branch predictor to take certain paths, and subsequently using speculation to read memory which may contain privileged data. In impacted Arm implementations branch predictor structures are not fully tagged by execution context, requiring a workaround to be applied in the following contexts.

Userspace program attacking the kernel

Involves userspace priming the branch predictor prior to a straight branch into the kernel. When this happens, the kernel will catch the fault, and inject a signal into the program. Adding a branch predictor invalidation at this stage will mitigate any impact from branch predictor training.

Userspace program attacking another userspace program

Is mitigated by invalidating the branch predictor on each context switch from one program to another.

Virtual machine attacking the hypervisor

Is mitigated by invalidating the branch predictor on each guest exit.

These mitigations are complicated by the fact that the architecture does not specify a guaranteed architectural way of invalidating the branch predictor. This means the sequence required can vary significantly between CPUs of different types.

For example:

- Cortex-A8, A9 and A17 cores implement BPIALL as an invalidating instruction
- Cortex-A15 implements BPIALL as a NOP and a more complex solution is required
- Cortex-A57 and A72 require turning the MMU off and on to perform the BPI

Systems which have big.LITTLE implementations offer both opportunities and challenges – the LITTLE cores are not vulnerable to any variant, but a solution must support having multiple micro-architectures in the same system.

Overall, variant 2 must be mitigated across a range of CPU microarchitectures. Branch Predictor Invalidation sequences require a mixture of firmware and kernel support.

The KPTI (4.15) integration branch includes the relevant Arm64 kernel mitigations:

<https://git.kernel.org/pub/scm/linux/kernel/git/arm64/linux.git/log/?h=kpti>

An SMC (Secure Monitor Call) has been defined in the Arm architectural range, to allow clients (such as Linux) to request invalidation of the branch predictor. See the relevant [Firmware Interfaces](#) document.

Arm's reference firmware implementation Arm Trusted Firmware (ATF) provides support for mitigations in EL3 runtime. Guidance on the status of these mitigations is available here:

<https://github.com/ARM-software/arm-trusted-firmware/wiki/ARM-Trusted-Firmware-Security-Advisory-TFV-6>

Variant 3: (CVE-2017-5754)

Variant 3 creates a bypass of the protection offered by page table permissions using speculative execution. This only affects Exception Levels that share a translation regime with another exception regime.

For example: Code running at EL1 (either secure or non-secure) can potentially leak data to less privileged code running at EL0 (with the same security mode). The mitigation is preventing EL0 from having access to EL1 memory.

In Linux: Historically a userspace application process has included both Global kernel mappings alongside the Non Global mappings specific to the application.

A new mechanism called kernel Page Table Isolation (KPTI) is used to unmap the kernel from a process address space when that process is running. Only a small trampoline is still mapped, containing a set of kernel vectors. When an exception occurs (interrupt, fault...), it is taken by that trampoline, which in turn maps the kernel and jumps to it. On exception return, the kernel is unmapped again.

The net effect of this is that it becomes impossible for userspace to speculatively read the kernel memory, as that memory is not mapped. This is the mitigation for variant 3.

With this approach:

- Taking an exception from EL0 is slightly slower (no changes for an exception taken while in kernel space)
- The removal of Global kernel mappings leads to a slightly higher TLB pressure. This is minimised by the use of block mappings covering larger areas of memory.

Benchmarking has shown the performance overhead is minimal compared to non KPTI kernel workloads.

The only Arm Cortex core affected by variant 3 is **Cortex-A75**. However, the KPTI patches have wider applicability as they lead to an increase in security against unrelated more generic software exploits such as [KAISER](#).

The KPTI patches complement the kernel layout randomisation (KASLR) implementation, leading to an increase in security that applies to all 64bit CPUs.

There is no known 32bit Arm core susceptible to variant3 and no 32bit implementation of KASLR. At time of writing there are no plans for a 32bit variant of KPTI.

Variant 3a: (CVE-2018-3640)

This is a different form of variant 3 identified by Arm. Instead of accessing memory, it uses speculation to access a privileged system register. Arm's current assessment is that this is not a major threat in most use cases.

System registers can be split into configuration and addresses information. Generally, configuration settings can be inferred from source code. However, it has been determined that variant 3a could potentially be used to obtain the vector base address (VBAR_EL1, VBAR_EL2...).

In Linux this is mitigated by KPTI, where VBAR_EL1 has a fixed value and doesn't reflect the location of the kernel.

For hypervisors variant 3a can potentially be used to infer details of the memory layout. Work is under way for KVM to decouple the vector base address from the memory layout.

The only Arm Cortex cores affected are Cortex-A72, Cortex-A57 and Cortex-A15.

Variant 4: (CVE-2018-3639)

Variant 4 derives a Spectre-type attack using a CPU technology known as memory disambiguation. This technology is used in high-end CPUs to enable greater 'out of order execution' and improve performance.

In some circumstances memory disambiguation can result in a speculative load overtaking a store, to the same memory location, resulting in the load returning stale data. The load will not be architecturally committed, but in a similar way to the earlier Spectre variants - if a chain of suitable instructions can be constructed, this stale data can be used to construct an address that drives cache allocation. This can potentially be used to leak data to an attacker across a privilege boundary.

Variant 4 allows an attacker to target boundaries within software managed privilege environments (such as JITs) or to attack software executed in EL1 or above from a lower Execution Level. Arm has not been able to establish a Proof of Concept in Linux for Variant 4. However, one theoretical vulnerability relates to stack accesses driven by system calls from userspace to kernel. These call paths are heavily optimized in the kernel and calls can occur in rapid succession. It is very hard for un-privileged code to establish the conditions necessary to enable this attack. However, it is not possible to rule out that this mechanism might be exploitable.

A general mitigation is available on all impacted Arm Cortex implementations via an IMP DEF control register to disable memory disambiguation (re-ordering of stores and loads). This is the approach used by Linux and Arm Trusted Firmware below.

Arm has also defined new barrier types (SSBB/PSSBB) which enforce ordering. These barriers can be added to vulnerable sites in code (where the general mitigation is not applied). In a similar way to Variant 1 it is hard to identify all vulnerabilities across large software stacks.

Arm has not yet found a viable way to produce a code scanner for Variant 4. In source code it is not possible to reason about how objects will overlap on the stack, or elsewhere, when they have different life times. In binary code there is insufficient information to determine whether loads will access memory produced by earlier stores that might have stalled. Arm will continue to explore the options available.

It is not possible to automatically detect and fix problematic load store sequences in hardware designs. However, Arm is considering hardware mitigations for future high-end cores to limit the performance degradation when disambiguation is disabled.

Software Mitigations

There are two approaches for disabling memory disambiguation to mitigate this variant. Arm is planning support in software for both approaches.

- Static: The mitigation is permanently enabled in EL3 (Trusted Firmware) at boot.
- Dynamic: The mitigation can be disabled dynamically for EL0 workloads (Arm is continuing to investigate).

Static Mitigation

This is the recommended approach for the following CPUs impacted by this variant: Cortex-A75, Cortex-A73, Cortex-A72, Cortex-A57. Mitigation requires an update to Trusted Firmware such that the mitigation is enabled by EL3 at boot time (via a write to an IMP DEF control register). The mitigation is applied to the entire software stack. No additional mitigation code is required in the Linux kernel.

Dynamic Mitigation

Dynamic mitigation is a more complex solution Arm is continuing to investigate. At time of writing, this approach is being developed for Arm's Cortex-A76 CPU.

Later performance Cortex CPU designs make more extensive use of memory disambiguation. Consequently, the cost of permanently disabling this hardware feature at boot is greater. A dynamic approach allows the IMP DEF control register mitigation to be disabled for EL0 workloads and may give performance benefits that outweigh the cost of the associated system calls.

Implementing this functionality involves updates to Trusted Firmware, Linux and Hypervisors:

- Trusted Firmware will enable the mitigation in EL3 at boot time. The mitigation will be enabled on each entry to EL3 and the caller's mitigation state restored on exit.
- Trusted Firmware will implement a defined SMC in the Arm architectural range to allow clients (such as Linux) to dynamically enable/disable mitigation for variant 4. An update to the relevant [Firmware interfaces](#) document will be available describing this new SMC.
- In Linux (and KVM), the mitigation will be enabled by default in kernel space, but (by default) run userspace workloads with the mitigation disabled.

It is possible for the kernel to query whether dynamic mitigation is required, for a given platform, via an SMC call. The query interface does not distinguish between statically mitigated and unaffected.

Patches for this approach are still under active development. For Linux it should be noted that initial patches are likely to be subject to significant change as the community is expected to adopt a cross architecture solution, in a similar way to Variant 1.

When available, Linux patches will be posted publicly on the kernel mailing list. The associated Trusted Firmware security advisory is available here <https://github.com/ARM-software/arm-trusted-firmware/wiki/Trusted-Firmware-A-Security-Advisory-TFV-7>.

Summary

Arm is continuing to co-ordinate with other industry companies to update and improve the software in the OSS community in order to secure and defend processors from these novel exploits, which, in effect, represent a new class of attack.

The solutions being upstreamed for the Linux kernel, Arm Trusted Firmware, and GCC are part of a co-ordinated effort to defend against these exploits and minimize performance impact on consumer devices powered by Arm-based processors.

Note that these patches are subject to the normal Open Source Software review process and may evolve significantly. Arm are working with Linaro to discuss backports for the Linux kernel and the latest update can be found [here](#).

The information provided in this document is based on Arm's present understanding of the relevant vulnerabilities and mitigations, and is subject to change as additional information is revealed and as additional analyses are performed.