arm

Whitepaper

Addressing Spectre Variant 1 (CVE-2017-5753) in Software

Date: October 2018

Version 1.0

Introduction

In January 2018, Google Project Zero disclosed several potential speculation side-channel attack techniques collectively termed Spectre and Meltdown. This white paper examines some mitigation techniques for protecting code against Spectre Variant 1 (CVE-2017-5753) attacks and considers the merits of each approach.

This paper should be read in conjunction with the other material published by Arm, which can be found at **https://www.arm.com/security-update**.

Note that although the examples used in this white paper use the AArch64 execution state and the A64 instruction set, the same principles can be applied when using AArch32 with the A32 and T32 instruction sets. The source code fragments are written in C but apply equally to any compiled high-level programming language.

Outline

We will begin by giving a brief overview of the original Spectre Variant 1 attack and shows how, at the assembly level, mitigations can be applied. We'll then show that it is not possible to use normal high-level language constructs to protect programs using these techniques.

We will then discuss some practical mitigation techniques that can be applied and show how, with the help of some compiler instrumentation of the control flow, we can then insert annotations into the source program that can lead to effective mitigation.

Finally, we will discuss some of the sub-variants of Variant 1 and show that the same mitigation techniques can be applied there as well.

Principles of Spectre Variant 1 and its mitigation

Spectre Variant 1 uses the principle of a speculative bounds-check violation to access a memory location (the secret) that would normally be inaccessible to the attacker. The secret may then be used to make controlled modifications to information held in the cache. Later, cache timing analysis can be used to deduce the secret even though it cannot be read directly.

The most basic form of this form of attack uses a code sequence that looks something like:

```
if (untrusted_value < limit)
  val = array[untrusted_value];
// Use val to access other memory locations
```

although it should be appreciated that the real situation in software may be considerably more complex. It is presumed that the attacker has control over `untrusted_value`.

To exploit the speculative bounds-check bypass, the attacker first calls the code sequence repeatedly with `untrusted_value` set to a value that is less than `limit`. This trains the branch predictor to predict that the test will always succeed, and that `array` will always be accessed. A final call uses an out-of-range value to access the real secret value of interest. The branch predictor, however, based on the training period will still predict that the array will be accessed. Although the processor will eventually detect the misprediction and unwind the processor's state, the value returned from the accessed memory location may be useable for several cycles following the misprediction.

When the above program fragment is compiled, it will typically produce machine instructions as follows:

```
      CMP   untrusted_value, limit
      B.HS  label
      LDRB  val, [array, untrusted_value]
label:
      // Use val to access other memory locations
```

This code can be protected against the misprediction of the conditional branch by inserting a `CSEL` (conditional select) instruction to clamp the range of `untrusted_value`:

```
      CMP   untrusted_value, limit
      B.HS  label
      CSEL  tmp, untrusted_value, WZr, LO
      CSDB
      LDRB  val, [array, tmp]
label:
      // Use val to access other memory locations
```

The `CSEL` instruction *clamps* the array index value to zero in the case where the conditional branch should have been taken but branch-prediction guessed otherwise. The additional `CSDB` (Consume Speculative Data Barrier) instruction is a new barrier that ensures that the result from the `CSEL` instruction is not based on a prediction as to which operand will be used. The result of this is that if a branch misprediction occurs, then `array[0]` will be returned rather than accessing a memory location under general control of an attacker. It will be appreciated that if loading any value in this situation is potentially unsafe, then performing an address calculation into a temporary register and clamping that to zero will lead to a NULL address which in most circumstances would then lead to a speculative address-bounds violation since the address zero is not normally mapped into a program's memory space.

Protecting against Spectre Variant 1 in source code

It is not generally possible to directly protect against Variant 1 in source code using standards-conforming statements. The problem is that the additional code would be redundant; an optimizing compiler would simply remove it as having no logical effect on the program. For example, changing the first source code example to:

```
if (untrusted_value < limit)
  val = array[untrusted_value < limit
              ? untrusted_value : 0];
// Use val to access other memory locations
```

It is obvious to the compiler's optimizers that the test inside the array access is simply a repeat of the immediately preceding test and can thus be simplified progressively to

```
if (untrusted_value < limit)
  val = array[true
              ? untrusted_value : 0];
// Use val to access other memory locations
```

and then back to the original code fragment.

Aside from the optimization issue, there's also nothing in the original high-level programming language that guarantees that the second comparison and value selection would be implemented using a CSEL instruction: a compiler could legitimately convert this into a conditional branch sequence that implements the same logical behavior.

Since we cannot rely on the compilers not optimizing away simple code modifications, more radical approaches are needed. It may, in some cases be possible to restructure the program entirely so that any vulnerable data is simply not mapped into memory at a time when an attack might occur; but such changes are not always feasible and would certainly be costly to implement. We will focus here on techniques that can be used with more local changes.

Fast masking to limit overrun

In the examples above, limit will often be a simple constant value. A quick technique to limit the extent of any bounds check bypass is to use a logical mask operation to clear any high bits in `untrusted_value`. If `ceil_log2(x)` is the smallest integer such that $2^{ceil\_log2(x)}$ is greater than, or equal to, x, we can use a sequence equivalent to

```
if (untrusted_value < limit)
  val = array[untrusted_value
              & ((1 << ceil_log2(limit)) - 1)];
// Use val for memory accesses.
```

Since the expression used for the mask is itself constant, this will simplify down to a simple logical AND with the appropriate constant (for example, if limit were 57, a mask value of 63 would be used). To be complete, a CSDB instruction is needed after the mask operation to ensure that the mask operation, which is technically redundant, cannot be optimized away inside the processor.

If `array` is small, or can be arranged to be a size of a power of 2, then very little, if any, data beyond the end of the array can be leaked.

Since the operation is often entirely redundant given an earlier bounds check, care must be taken to ensure that the mask operation cannot be removed by the compiler. The need for the additional CSDB instruction in this sequence also means that such an operation will normally use some form of inline assembly to apply the mask.

The masking technique is only practical when `limit` is a constant and is either small or an exact power of two. For larger limits the amount of data beyond the end of the array can become significant, or wasteful. In these cases, a more precise bounds check may be needed.

Using inline assembly

Most compilers provide an extension mechanism that permits specific machine instructions to be embedded into the program. The examples here are based on the GNU C inline assembler syntax which is also supported by many other compilers, including LLVM. If using this approach, wrapping the assembly statements inside a C macro can make the programming API much easier to deploy. Arm has published an example of such an approach which can be downloaded from **https://github.com/ARM-software/speculation-barrier**. This header file defines three macros that can be used to protect against incorrect speculation in several different scenarios.

```
load_no_speculate (ptr, lo, high)

load_no_speculate_fail (ptr, lo, high, failval)

load_no_speculate_cmp (ptr, lo, high, failval, cmpptr)
```

Starting with the first of these macros, which implements the following program fragment:

```
(ptr >= lo && ptr < high) ? *ptr : 0;
```

except that the code expansion guarantees that it will expand to a sequence that is protected as described above if the comparisons are mispredicted. Because the macro can handle more than one type of pointer and it needs to return a value from dereferencing that pointer, the implementation is quite complex, so for simplicity we will consider just the case where the pointer returns a single character (byte) from memory. The assembly expansion in this case becomes:

```
__asm__ ("CMP    %[ptr], %[lo]\n\t"
         "CCMP   %[ptr], %[hi], %[cs]\n\t"
         "BCS    1f\n\t"
         "LDRB   %[result], [%[ptr]]\n"
         "1:\tCSEL    %[result], %[result], Wzr, CC\n\t"
         "CSDB"
         : [result] "=r" (result)
         : [ptr] "r" (ptr), [lo] "r" (lo), [hi] "r" (hi)
         : "cc");
```

Note that in this case we have chosen to allow the load to execute even if it might speculatively fetch from a secret location. We can do this because we will then overwrite the result with zero if the comparison should have failed: this ensures that the result cannot be used for further speculative execution that might leave a detectable imprint on the cache.

This macro could then be used to rewrite our original example as

```
if (untrusted_value < limit)
  val = load_no_speculate (array + untrusted_value
                           array, array + limit);
// Use val to access other memory locations
```

It will be apparent from this case that with this specific macro, the limit check will effectively be repeated, once for the original program check that the array offset is less than the permitted limit and a second in the assembly expansion itself. Unfortunately, we can't avoid this given the semantics of the original program: if the test fails then `val` should not be updated at all, but the macro cannot provide a way to do this. The most that we can do in this case is to always continue executing with the zero result, which cannot reveal what was stored at the secret location.

The second macro defined by this header provides a small extension over always returning zero in the event of misprediction of the branch. Instead, a specific value can be specified for use in this case. There are times when zero might be less safe than some other value: consider for example if we were to subsequently subtract 1 from the result returned and then use that as a mask for some subsequent operation. Subtracting 1 from zero yields -1, which in two's complement form is all bits set to one; we'd rather that our fail-safe value was 1 in that case, so that once we subtract one we end up with zero and that is safe to use as a mask.

The third macro defined by this header goes one step further still and permits a fourth pointer to be used for the comparisons whilst using the first pointer only for the final load operation. It will be apparent that the first and second macros are simplified version of the third, for example, the first could be written as:

```
load_no_speculate_cmp (ptr, lo, high, 0, ptr)
```

It will also be apparent that the macro we have defined can only be used to compare pointers. The need to hide the comparisons from the compiler's optimizers means that we must choose when defining the macro what types of object can be compared.

A further limitation of the macro approach is that sometimes the structure of the program can make it difficult to use. For example, consider this case:

```
if (untrusted_value >= limit)
  return;
// do something …
val = array[untrusted_value];
// Use val for memory accesses.
```

In this scenario the check is separated from the unsafe access by an unspecified amount and it might not be possible to repeat the check immediately before dereferencing the array. For example, if the array dereference were in a subroutine and limit was not passed to that routine, then protecting this access would imply changing the function prototype to include an additional parameter.

Finally, even `limit` might be a speculatively calculated result from an earlier conditional branch. In that case we cannot trust either the conditional branch or the result of the comparison itself, making the condition flags used by the `CSEL` instruction potentially incorrect. In this scenario, the `CSEL`+`CSDB` sequence is insufficient for any localized protection and either a hard speculation barrier or more sophisticated speculation tracking is required.

Tracking speculation state

A posting on the LLVM developers list[1] describes how a compiler may automatically insert code to track the speculation state through a sequence of conditional branches if a machine has instructions that can recalculate the expected flow-control state without themselves speculatively predicting the result. In AArch64 the `CSEL` instruction, in conjunction with `CSDB`, can perform this calculation.

For an implementation on AArch64, we start with a register, `tracker`, that is initialized to all-bits set, then at each conditional branch location the code is annotated as shown in the following example:

```
    B.EQ  taken
    …

taken:
    …
```

```
    B.EQ  taken
    CSEL  tracker, tracker, Xzr, NE
    …
taken:
    CSEL  tracker, tracker, Xzr, EQ
    …
```

Since the condition on each `CSEL` instruction exactly matches the architectural condition for the branch outcome, the defined behavior of this sequence is to always preserve the original contents of the tracker variable. However, on a machine with speculative execution, if the branch predictor predicts the branch incorrectly, the `CSEL` instruction will detect this and set the contents of the tracker register to zero. The result is sticky, in that once cleared due to an incorrect prediction, it can never become non-zero again until the incorrect speculation has been unwound and the code correctly re-executed. This property allows us to chain multiple conditional branches into a single tracking state register which may contain one of two values: all bits set, when all previous conditional branches were correctly predicted, and all bits unset if any of the earlier branches were incorrectly predicted.

With the tracker value established, protection of a vulnerable memory access can now by protected by using the tracker register to mask the vulnerable address or offset. Going back to our original example, the assembly code generated for that would typically be:

```
    CMP   untrusted_value, limit
    B.HS  skip_load
    LDR   val, [array, untrusted_value]
skip_load:
    …
```

And with the tracking and protection this can be transformed to:

```
    CMP   untrusted_value, limit
    B.HS  skip_load2
    CSEL  tracker, tracker, Xzr, LO
    AND   tmp, untrusted_value, tracker
    CSDB
    LDR   val, [array, tmp]
skip_load:
    …
skip_load2:
    CSEL  tracker, tracker, Xzr, HS
    B     skip_load
```

There are a couple of points to note about this code. Firstly, the `CSDB` instruction is only needed at the point that we need to use the tracker variable to protect a potentially vulnerable access; this not only saves on code size, but it also significantly reduces the number of times when the processor might to stall to resolve any outstanding speculative `CSEL` operations. Secondly, because the first block of code falls directly through into the code that follows the conditional block, it is necessary to redirect the original branch to a temporary location where the other `CSEL` operation can be safely executed; once this has been done we can then jump back to the original code sequence (if we were to leave it at the original target, then it could be executed in a context where the condition is not guaranteed to be true by construction and it would permanently corrupt our tracker variable).

Given that all the state needed to track the speculation condition is now inserted automatically by the compiler, it is possible to define a new compiler built-in function that can be used to describe which accesses (or more precisely, which values) may be vulnerable if accessed speculatively. The new built-in function has the prototype:

```
T __builtin_speculation_safe_value (T unsafe_value);
```

Where `T` may be any integer type, or a pointer to any type. The type of the result returned by the built-in function is always the same type as the argument to it. Given this new built-in function it is possible to rewrite our original example as

```
if (untrusted_value < limit)
  val = array[__builtin_speculation_safe_value (untrusted_value)];
// Use val to access other memory locations
```

This example presumes that it is always safe to return the contents of `array[0]` (that is, doing so will not leak useful information to an attacker). If that is not the case, then it is possible to rewrite the example as:

```
if (untrusted_value < limit)
  val = *__builtin_speculation_safe_value (array
                                           + untrusted_value);
// Use val to access other memory locations
```

With this version the built-in function will return a pointer to address zero if the speculation state indicates that misspeculation has occurred which will normally result in an invalid memory access which cannot return any data.

Tracking speculation through function calls

[This section can be skipped unless you want a deeper understanding about how speculation is tracked across function boundaries.]

Tracking speculative execution within the scope of a single function is relatively straightforward: at the beginning of each function a tracking register is selected from amongst the callee-saved registers and initialized to -1 (i.e. all bits set) and at each conditional branch it is updated as described above. Unfortunately, the lack of knowledge about whether the function was entered speculatively due to an incorrect branch prediction, or whether a callee of the current function returned speculatively for the similar reason, means that a local tracking register is of very limited use and might give the illusion of mitigating against a vulnerable access when in practice it has not.

In principle it would be possible to make the register used for tracking the speculation state a global register. It would then be available at every point during program execution to recover the speculative execution state of the machine, regardless of how large a speculative execution window exists on the processor. Unfortunately, declaring the register to be global in this way would be a change to the procedure call standard for the machine since the AAPCS64[2] makes no provision for such a register. Furthermore, implementing such a change would imply incompatibility with all existing binaries and force a recompilation of all programs and any dynamically-linked libraries that they use.

Fortunately, the stack pointer register, SP, must always contain a valid address and it is impossible for this to be zero[1]; we can exploit this property to propagate the speculation tracking state between function calls. Before each function call we perform a logical AND between the stack pointer and the tracker register, storing the result back in the stack pointer; on entry to each function we test for the stack pointer being NULL and use that to initialize a new tracker register before making any allocations to the stack. Identical sequences can be used before returning from a function to communicate the speculation state back to the caller.

The code sequences needed to encode the speculation state in the stack pointer are complicated slightly due to the restrictions on using SP in normal data-processing operations. In practice it is necessary to copy SP into a temporary register that can then be modified before copying the result back into SP. We can use the sequence:

```
MOV   tmp, SP
AND   tmp, tmp, tracker
MOV   SP, tmp
```

The sequence used to establish a new tracker register, on the other hand can use SP directly:

```
CMP   SP, #0
CSETM tracker, NE  // tracker = (sp != 0) ? ~0 : 0
```

A significant advantage of these sequences is that they can safely be used when mixed with code that was built without speculation tracking (though obviously the level of mitigation in this case will be reduced). In particular, at no point in the program do we need to explicitly initialize the tracker to a constant value; we can always rely on the property of SP never being 0. If a function without speculation tracking calls a function with tracking, then SP will always be none-zero and thus the tracker will always initialize to all bits set. Similarly, if a function with speculation tracking calls a function that does not, then the worst that can happen is that on return the speculation tracker will be reinitialized to indicate no misspeculation; if the called function is a very small leaf function the encoded state from before the call may still be sufficient to fully restore the tracking state after the return.

Several optimizations can be applied by the compiler to remove redundant transfers of the speculation state between the tracking register and the stack pointer:

[1] The stack is full-descending and so the stack pointer points immediately above the next allocatable slot. A program starting with a stack allocated at the very top of virtual memory might conceivably start with SP = 0 so that after the first block of stack space is allocated SP will wrap around into the correct space. In practice, however, this would only affect start-up code and that code is very unlikely to need to mitigate against speculative side channel attacks; by the time the main body of code for the program starts to execute the stack pointer can never be zero.

1. Simple leaf functions with no conditional branches and with no use of the tracking register can leave the tracking state entirely in SP; this applies even if they end with a tail-continuation jump to another function;
2. Back-to-back function calls that have no change of control flow between the calls and no use of the tracker can similarly avoid copying the state back into the tracking register;
3. Inter-procedural register allocation may permit the tracking register to be passed directly to called function when the address of that routine cannot escape and the call cannot be pre-empted later in the compilation or loading process.

Fully automated mitigation

In theory it would be possible to make a compiler emit a mitigation sequence before every single memory access to ensure that addressing bounds were not exceeded. In practice, however, a program built this way could well be too slow to be acceptable. The vast majority of memory accesses could never be used to form a Spectre style attack and protecting them would inhibit the CPU from using its pipeline resources effectively. To be practical, an automatic mitigation mode in a compiler would need to identify all the accesses that might be vulnerable without mis-identifying too many that cannot. Accurately identifying such sequences is still a matter of ongoing research.

Limitations of speculation tracking

It is not possible to track every form of speculative branch prediction encountered during the execution of a program. Indirect branches of all forms are generally very hard to protect; these include indirect jumps, indirect function calls and return instructions[2]. Some limited forms of indirect branch prediction can be protected in specific cases, but these have not yet been implemented in the compilers.

The built-in compiler function, `__builtin_speculation_safe_value`, can be implemented in such a way that, if speculation tracking has not been enabled, it will expand to a full speculation barrier, such as `ISB+DSB SY`.

The performance impact of tracking speculation may be significant; only benchmarking of specific cases will show whether the overhead of tracking speculation at every conditional branch is tolerable; in some circumstances it may be preferable to turn off tracking and rely on the full speculation barriers that will then be generated.

Speculation tracking has not yet been implemented for AArch32 (arm and thumb code generation) due to restrictions on the number of available registers. A full speculation barrier will always be emitted for AArch32.

[2] The architecture does not require that a return instruction use the link register, LR, to contain the return address, so it is not possible to insert code after each function call point to enforce the constraint that LR == current address.

Spectre Variant 1.1: Bounds Check Bypass Store (CVE 2018-3693)

The bounds check bypass store variant of Spectre, sometimes termed Spectre Variant 1.1 can also be mitigated against by using the `__builtin_speculation_safe_value` compiler built-in function. The principle of operation for this variant is that a store to a location protected by a bounds check may be mispredicted and the store may overwrite a location which is subsequently used for an indirect jump. The overwritten jump may then be used by an attacker to execute any arbitrary amount of code in the speculation window.

A typical code sequence might look like:

```
if (untrusted_value < limit) {
  array[untrusted_value] = unsafe_pointer;
  indirect_call ();
}
```

And a mitigated sequence might look like:

```
if (untrusted_value < limit) {
  *__builtin_speculation_safe_value (array + untrusted_value)
    = unsafe_pointer;
  indirect_call ();
}
```

The modified sequence will ensure that the address used for `indirect_call` can never be modified due to an incorrect speculative branch prediction.

Spectre variant 1.2 is similar to variant 1.1, except that the on some systems a store might bypass a write permission check and temporarily update a location that is read-only. Mitigation, by protecting against the speculated store address, is the same as for variant 1.1.

Summary and Conclusions

We have shown how simple source code modifications are insufficient to prevent speculative memory accesses from exceeding the safe bounds of an array, but that a combination of either inline assembler macros or a compiler built-in function, in conjunction with a modified compiler, can effectively limit the extent to which information can be leaked. We have also shown how these techniques can also be used to provide mitigation against variant 1.1 and 1.2 style attacks.

Arm has published patches to implement the built-in compiler function described in this paper (`__builtin_speculation_safe_value`) for both GCC and LLVM. At the time of writing, the GCC patches have been merged into the development version of the compiler, which will eventually become GCC-9; the LLVM patches are being reviewed by the developer community.

Arm continues to investigate more automated approaches for detecting and patching vulnerable sequences but currently, a combination of manual annotation and compiler assistance appears to deliver the most effective compromise between effective mitigation and overall performance impact.

References

1. **https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html**
2. **https://developer.arm.com/docs/ihi0055/latest/procedure-call-standard-for-the-arm-64-bit-architecture-aarch64**

Document history

Version/Issue	Date	Confidentiality	Change
1.0	12 October 2018	Non-Confidential	First release.