Spectre, Meltdown and the Mill CPU

Will Edwards
will@millcomputing.com

Introduction

The Mill CPU is not vulnerable to the Spectre and Meltdown attacks.

The Mill is an in-order machine and Spectre and Meltdown as described take advantage of speculative execution on out-of-order machines. However, software and compiler speculation can occur on the Mill and this paper examines those cases and shows how design decisions have eliminated these attack vectors.

Overview of Meltdown and Spectre

Vulnerabilities webpage https://spectreattack.com/

Meltdown and Spectre are attacks that exploit processor vulnerabilities created through the use of *out-of-order* and *speculative* execution.

Out-of-order execution is when a processor executes instructions based on the availability of their inputs and resources rather than on their original order in the program. Conversely, in-order execution is when the processor executes instructions in the strict order that they occur in the original program, waiting for inputs and resources as necessary.

Speculative execution is when a processor executes an instruction based on the assumption that the instruction will likely be needed, before it has determined that it actually is needed.

A Meltdown attack targets a processor that makes memory accesses before, or in parallel with, protection checks. The attack speculatively loads a secret value from a location for which the attacker does not have appropriate permission, and then uses the loaded value in address calculations in a second speculative load. If the hardware speculatively executes both loads before resolving the permission check on the first, then the side effects of the speculated loads can leave traces in the cache from which the attacker can infer the secret value despite failure of the permission check.

A Spectre attack targets a processor that makes memory accesses before, or in parallel with, or without, a bounds check guard. The accessed data is used in address calculations for further loads and an attacker can infer the secret data in memory from the side-effects the second load has on the cache.

One key distinction between Meltdown and Spectre is that Meltdown is accessing memory that the attacker program has no access to and the protection check will fail, whereas in Spectre in the attacker is causing the victim to access memory that the victim has permission to access, and protection checks will succeed.

Meltdown has been used to read data at kernel addresses from user-space when the kernel is mapped into the user space, and Spectre has been used to read data at kernel addresses by

finding Spectre-vulnerable gadgets in the kernel. Spectre can also be used to attack within user-space when the attacker is restricted to a portion of user space by programmatic guard tests.

So far, almost all Intel and some ARM processors have been found to be vulnerable to Meltdown in a way that can be practically exploited. All mainstream out-of-order processors seem to be vulnerable to Spectre variants.

While it is a good generalization that in-order processors that do not engage in hardware speculative execution are immune to Meltdown and Spectre, programs and/or compilers may also engage in software speculation, leading to vulnerabilities that are analogous to Meltdown or Spectre. In our analysis we found and fixed one such bug in the Mill compiler tool chain.

Meltdown-type attacks

Relevant talk: https://millcomputing.com/docs/#memory

The Meltdown attack comprises two steps: a first load of a secret value sought by the attacker, located at an address for which the attacker lacks permissions; and a second load at an address in part determined by the result of the first load.

Architecturally the result of the first load is not available to be used to form the second load, because the failed permissions check will interrupt the program before the second load is issued. However, processors using out-of-order execution may speculatively execute both loads before the permissions exception occurs. The results of the speculative loads will be discarded eventually by the exception, but the side-effects on the cache can be used by the attacker to obtain the secret value that is the result of the first load.

The Mill is a statically scheduled in-order machine and does not speculate ahead, so the first load must complete execution fully and return a result before the second load, which uses that result as an address component, can begin execution. The Mill does not speculate on chains of operations as an out-of-order machine might.

Because the first load fails its permission checks, the Mill returns a special error value NaR (Not a Result) rather than the secret value from memory; for information about NaR value on the Mill see https://millcomputing.com/docs/#metadata. When used as an address in the second step, the NaR causes the second load to also fail with no changes to the cache.

So, on the Mill, a value that the current protection domain cannot access is never used in a following computation that can expose that value. As a result, the Mill is immune to Meltdown attacks.

Spectre-type attacks

Two general variants of Spectre have been described - one is to induce the processor to speculatively perform a load whilst the guard instructions have yet to be checked, and the second is to mistrain the branch predictor so that the processor jumps to and speculatively executes code accessing the memory before the branch predictor miss is noticed.

Variant 1 - speculative loads before guards

Relevant talk: https://millcomputing.com/docs/#execution

In this variant the attacker locates (or constructs) a gadget in the victim that it can cause the victim to execute. The gadget comprises a load pair that work in the same way as with the Meltdown attack: the first load accesses an address provided by the attacker to load a secret datum, and the second uses the secret in its address calculation to exfiltrate the secret to the attacker.

The pair are assumed to be guarded by a test that checks whether the attacker-provided address is in a permitted range and bypasses the loads if not. The exploit occurs when the loads are speculatively executed by the hardware before the guard has completed its checks, leaving side effects in the cache that can reveal the secret to the attacker.

The Mill is a statically-scheduled in-order machine and executes instructions in the strict order they occur in the instruction stream. Because the guard is always executed before the first load, and that load before the second, the Mill will not execute either load if the guard check fails, and the secret is never available to be exfiltrated. Consequently the Mill hardware is immune to Spectre variant 1 attacks.

Variant 2 - speculative execution of mispredicted branches

Relevant talk: https://millcomputing.com/docs/#prediction

In this variant, there is no guard. Instead, the victim speculatively executes the gadget because of a branch mispredict.

The Mill does not have a conventional branch predictor. Instead, it has a novel transfer predictor. However, it still predicts transfers and continues fetching and decoding code speculatively whilst it works out that a transfer was mispredicted. In addition, it may issue instructions speculatively down the predicted path. These speculated instructions are in all cases revocable; their entire effect will be undone by the hardware if the prediction was in fact in error.

In contrast to OOO machines which may speculate hundreds of instructions, the Mill speculation is limited to those that that the hardware can track for revocation. Current Mill configurations will issue, and revoke, a maximum of two instructions. Revocation includes all cache and other micro-architectural side effects.

The Mill misprediction penalty is very low - in the order of 5 cycles. All the same, the first load instruction of the Spectre pair may be speculatively issued. The processor notices that the branch mispredicted and starts abandoning these loads whilst it is loading the actual instructions that should have been taken instead. The second load is never issued because the misprediction is recognized before the response to the first load is received, and consequently the Spectre variant 2 attack is impossible on a Mill.

Software and compiler speculation

As explained above, the Mill is immune to attacks that exploit the hardware speculation performed by Out-of-Order execution. However, speculation may also be done by the software of a victim program, or be induced by the compiler in the process of translating and optimizing the victim code.

There is little that a vendor can do about victim code that deliberately loads from an attacker-supplied address without checking the validity of the address first; such code is a bug, plain and simple. Exploitable speculation introduced by the compiler as an optimization is also a bug of course, but more insidious because inspection of the victim source will not reveal the problem.

The Mill is a very wide-issue machine, which allows the compiler to improve performance by carefully interleaving speculative execution using meta-data and predicated (conditional) instructions. For example, it is common for the two legs of an if-statement to be interleaved and both sides executed unconditionally but with one side masked out via meta-data or not used by subsequent conditional instructions.

To illustrate this speculation, consider the code generated for the simple code listing 1 in the Spectre paper:

```
int foo(int* array1, int* array2, int array1_size, int x) {
    if (x < array1_size)
        return array2[array1[x] * 256];
    return -1;
}
```

In the Spectre paper the `array1_size` variable had to be loaded from memory which is how the attackers induced the out-of-order processor to speculate the load before the check was complete; this aspect is not required to illustrate speculation on the Mill so we pass it ‘on the belt’ (which is analogous to ‘via registers’ on a conventional machine.)

The Mill LLVM-based compiler generates this intermediate representation (IR) for this test function (edited for clarity):

```
define external function @foo w (%0, %1, %2, %3) {
label $0:
    %4 = con(-1)              // create constant -1
    %5 = lsssb(%3, %2)        // %3 < %2, compare signed
    br(%5, $1, $2)            // branch if %5 to $1 else to $2
label $1:
    %6 = load(%0, 0, %3, 4)   // load [%0 + 0 + %3 * 4]
    %7 = shiftlu(%6, 8)       // shift left unsigned
    %8 = load(%1, 0, %7, 4)   // load [%1 + 0 + %7 * 4]
    br($2)
label $2:
    %9 = phi(w, %8, $1, %4, $0)
    retn(%9)
}
```

The Mill specializer - the part of the compiler that generates member-specific machine code – formerly generated erroneous executable code from the above:

```
F("foo") %0 %1 %2 %3;
    load(%0, 0, %3, w, 3) %6;
    nop();
    nop();
    shiftl(%6, 8) %7;
    load(%1, 0, %7, w, 3) %8;
    nop();
    lsssb(%3, %2) %5,
       con(w(-1)) %4,
       pick(%5, %8, %4) %14,
       retn(%14);
```

Note that the control flow of the branch seen in the IR has been if-converted into data flow; there is only a single sequence of instructions and no branches. This generated code speculatively loads `array1[x]`, multiplies it by 256 (by shifting it 8 to the left) and then uses it as an index into `array2`. These loaded values are only actually used if the guard - which is scheduled at the end of the function where it returns - is true. This optimization causes the two load operations to issue, and be executed, before the guard is tested – exactly the same vulnerability as OOO hardware speculation induces.

The specializer code generator is specification driven; a machine specification defines the semantics of every operation, and the specializer uses those semantics to schedule operations in the generated machine code. In this case, the specializer executed the loads speculatively because the specification said that non-volatile loads have no side-effects. However, Spectre demonstrates that speculative loads do in fact have side effects, so we have corrected the bug in the specification. As a result, the specializer no longer hoists loads to before their guard branches. The generated code is now:

```
F("foo") %0 %1 %2 %3;
    lsssb(%3, %2) %5,
       loadtr(%5, %0, 0, %3, w, 3) %6;
    nop();
    nop();
    shiftl(%6, 8) %7;
    load(%5, %1, 0, %7, w, 3) %8;
    nop();
    con(w(-1)) %4,
        pick(%5, %8, %4) %14,
        retn(%14);
```

This does the guard check in the first instruction, and does the loads conditionally on the check succeeding. Although the loads are scheduled and execute, they will not actually load something if the guard fails: instead, they retire a special result marked with the ‘None’ NaR meta-data. When this NaR result is used in the second load it prevents that load from executing, and produces a ‘None’ NaR result itself.

Folding both legs of the if-test into one instruction sequence is only an optimization; it won’t be done if compilations flags specify –O0. Here the optimization exposed a software bug in the Mill tool chain. There are many alternate programs that could be generated e.g. a conditional return could be encoded early, and some of these alternates may contain bugs too. While Mill software is as vulnerable to bugs as that for any machine, the Mill architecture is fundamentally immune to Spectre-like vulnerabilities.

Spectre and protection failures

There is no requirement that the second address that a Spectre attack loads from is accessible to the current protection domain. If there are cache-visible side-effects from a load that fails a protection check then the attacker can still infer the secret being exfiltrated. (This may mean that Spectre gadgets are naturally more numerous on processors susceptible to Meltdown, although that is a moot point if attackers are able to hand-craft the exploit rather than search for accidental gadgets)

The code generated in the above section is immune to Spectre because the code contained a guard and the specializer was careful to predicate the loads on the guard expression. However, if loads that fail protection checks can have cache-visible side-effects, then an attacker can induce a victim program that omits the correct bounds checks to use a secret data value that the victim *can* access as part of an address calculation into memory that the victim *cannot* access - and the victim faults but the attacker can still infer the secret!

So we have to determine if a single Mill load can have cache side effects even if protection checks fail.

The Mill uses a novel load instruction that tolerates load misses as well as hardware out-of-order approaches can do, while avoiding the need for expensive load buffers and completely avoiding false aliasing. The load instruction is issued in one cycle and retires in a specified subsequent cycle; the number of cycles between the issue and retire is configurable and referred to as the deferral. Loads are typically issued as soon as the address is known and retired as long after that as possible, as close as possible to the time the result is needed. This is to hide latency accessing lower-level caches if the load misses in the L1 data cache. The value that retires is the value in memory when the value retires; the load units snoop stores to spot aliasing, and reissue loads that have been invalidated by store instructions.

The Mill cache is virtual so there is no conventional TLB between the processor and the top-level cache. Instead, there is a Protection Lookaside Buffer (PLB) containing recently-used protection entries. The addresses being loaded can be checked against the bounds of each entry in the PLB in parallel with the fetch from L1.

When a load is issued, the top-level data cache and the top-level PLB are accessed in parallel. The minimum load deferral is specified to be the time these two accesses take, and the size of the PLB is generally bounded only by the latency of the L1D access it is in parallel with. This latency is typically 3 or 4 cycles, which allows a reasonable size L1D and a very large PLB.

A load with a large deferral allows the processor to ask lower-level caches for data in the event of a cache miss. If the data or protection is not available by the time the load should retire then the processor stalls.

When a load misses in the L1D, the behavior of the processor between that time and the retirement may open it up to cache attacks. For this reason, we carefully define the behavior to avoid all cache-visible side-effects when protection misses.

After the latency of the L1D has occurred, the load unit knows if the L1D hit or missed and if the PLB hit or missed. This has four possible outcomes, and this table shows the behaviors we specify to prevent the side-channel:

Table 1		L1D access	
		hit	miss
PLB access	hit	Value can be retired	Request for data can go to lower-level caches
	miss	Value cannot be used until the protection entry has been searched for. There must be no cache-visible side-effects such as MRU promotion	Request for data can go to lower-level caches but there must be no cache-visible side-effects such as MRU promotion or hoisting or evictions until the protection entry has been found

When there is a PLB miss, the search for the protection entry has no cache-visible side-effects either. If the search is implemented as a software trap, the handler must be written with this in mind.

The cache side channel

Both Meldown and Spectre use the cache as a side channel to exfiltrate captured secrets. The Mill is immune to the secret capture of these attacks, but as in all other architectures the cache channel itself remains for exfiltration of secrets obtained by other sorts of attacks.

The Mill cache has many unusual properties, as described in the https://millcomputing.com/docs/#memory talk. But it is still a cache, and when memory lines are loaded into cache other lines are evicted. An attacker can use these side effects as a side-channel, no different from the caches on any other architecture. The Mill team has worked hard to make the Mill architecturally secure, but the general cache side-channel and timing side-channels remain.

"Low-latency memory access, low-latency conditionals, secure isolation ... choose two." (Marsh Ray)

We have internally discussed mitigations to - and elimination of - cache attacks, even before we became aware of the Spectre and Meltdown attacks. A cache that is still an effective cache, but where eviction tells you less or nothing about what evicted it, would be generally immune to all cache attacks. However, we do not have any security enhancements to the general caching problem for disclosure at this time.

Closing Thoughts

The Mill is not vulnerable to Meltdown.

The Mill is not vulnerable to Spectre variant 1 because the hardware does not perform speculative out-of-order execution.

A code generator bug exposing a software vulnerability akin to Spectre variant 1 has been resolved with a bug fix, without hardware modifications. Because the Mill is built around an intermediate distribution format that is not directly executed, we are able to update how existing

programs execute on a Mill CPU when software bugs and vulnerabilities become known, even without changing vulnerable programs.

The Mill has not been vulnerable to Spectre variant 2 because the Mill has a very short pipeline and low mispredict penalty, so loads erroneously issued will be revoked in time before they have any side effects. This was an unanticipated side effect of the Mill design; we had not explicitly considered Spectre-type attacks when designing the branch predictor. We have now codified this requirement to ensure all future Mills have this protection for this reason.