KVM Forum 2018

KVM/arm meets the villain

Mitigating Spectre at the architecture level

Marc Zyngier <marc.zyngier@arm.com>

October 26, 2018

References

A bit of interesting literature:

- The original Google Project Zero findings (Variants 1, 2 and 3)

  https://googleprojectzero.blogspot.co.uk/2018/01/reading-privileged-memory-with-side.html

  Also called Spectre-v1, Spectre-v2 and Meltdown

- Microsoft and GPZ's disclosure of Variant 4 (SSB)

  https://blogs.technet.microsoft.com/srd/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass-cve-2018-3639/

- ARM's white paper

  https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper

Glossary

- Architecture: The contract between software and hardware
  - Independent of any actual implementation
  - A set of rules describing what is permissible, and what isn't
- μ -architecture: A hardware implementation of the architecture
  - Anything is possible as long as it doesn't contravene the architecture
- Speculative execution: an optimisation where a CPU performs a task before it may be required
  - Allows parallellisation to occur in hardware
  - May have to rollback state when mispredicted

“Anything, Anytime, Anyplace, For No Reason At All”

Frank Zappa

arm

Threat model

- The hypervisor is at a higher priviledge than the guests
- We fundamentally assume that its own state is not visible to the guests.
- Side channel timing attacks allows potential disclosure of secrets
  - Passwords, keys and other sensitive information
- Making use of speculative execution
- Limited to information disclosure, no alteration of data

Timing attacks

The VM could attack the hypervisor by doing something like:

- Start with an array of guest memory
- Make sure none of it is cached at the moment
- Get the hypervisor to speculatively use one of its secrets as an offset in the array, and issue a load
- In the guest, measure the time it takes to access the corresponding cache lines in the array
  - If the access is “fast”, then the hypervisor has accessed the cacheline corresponding to its secret offset
  - Now possible to infer information about the hypervisor secret

The hard part is for the guest to be “convince” the hypervisor to use the array under speculation.

Classes of attacks

- Boundary check bypass due to branch mispredition: Spectre-v1 (variant 1)
- Branch re-steering: Spectre-v2 (variant 2)
- Privilege separation bypass: Meltdown and Spectre-v3a (variants 3 and 3a)
- Memory ordering misprediction: SSB (variant 4)

All these can be composed to create more complex attacks.

Spectre-v1

arm

Spectre-v1, the *very* simplified view

At the core of Spectre-v1 is this simple condition:

```
if (index < array_bound)
        value = array[index];
```

Where the condition check can be speculatively bypassed, letting the access happen.

Fun things happen if:

- The index is untrusted
- The array address is under control of a lower privilege level
- See full example in the GPZ blog post

Mitigation of Variant 1

```
unsigned long array_index_mask_nospec(unsigned long idx,
                                      unsigned long sz)
{
        unsigned long mask;
        asm volatile("cmp     %1, %2\n"
                     "sbc     %0, xzr, xzr\n"
        : "=r" (mask)
        : "r" (idx), "Ir" (sz) : "cc");

        csdb();
        return mask;
}

#define array_index_nospec(index, size)                                \
({                                                                     \
        typeof(index) _i = (index);                                    \
        typeof(size) _s = (size);                                      \
        unsigned long _mask = array_index_mask_nospec(_i, _s);         \
                                                                       \
        (typeof(_i)) (_i & _mask);                                     \
})
```

We need a way to sanitize untrusted values despite speculation

- We introduce a `array_index_nospec` accessor
- Uses a mask that is `~0` if the index is valid, and `0` if not

The `CSDB` barrier prevents use of speculated data after the barrier

- To be used with either a `csel` instruction or something that affects the flags

Mitigation of Variant 1: Are we done yet?

- Mostly affects the userspace/kernel interface, not so much the guest/host interface
- We now have a robust accessor to mitigate variant 1
- Toolchains are also aware of it
- The real problem is where to use it
- Identifying that kind of sequence is extremely hard
- We only mitigate a few known spots in the Linux kernel
- Static analysis is only starting to be useful
  - See Dan Carpenter's `smatch` tool
- All privileged software is affected, not only the kernel and hypervisors

Still a work in progress :-(

Spectre-v2

Variant 2 (Spectre v2)

What is it?

- This is about training the branch predictor
- Force the CPU to speculate along a predicted “taken” path
- Specially “interesting” if you can force speculation in a more privileged context
- Can be used to target a Variant 1 gadget

How is that possible?

- Affected CPUs do not fully tag their branch prediction data by context (EL, ASID, VMID)
- Only consider the virtual addresses (PC and target)
- If two exception levels can have aliasing VAs, we’re in trouble
- As it turns out, host and guest do alias

How to mitigate Spectre v2

The obvious solution is to invalidate the branch predictor (aka BTB) in specific places

- In the kernel, when context-switching tasks
  - plus a couple of other places to protect the kernel itself
- In KVM, when exiting the guest
  - So that we can safely run the host
- The latter needs to be done without executing a single branch instruction
  - Otherwise the guest might prime the branch predictor to skip the invalidation

So, let’s invalidate the branch predictor...

Branch prediction invalidation on the ARM architecture

On AArch32, things are pretty easy:

- We have a dedicated instruction for that
- `BPIALL`, aka `mcr p15, 0, rx, c7, c5, 6`
- Works nicely on Cortex-A12 and A17
- But...

Branch prediction invalidation on the ARM architecture

On AArch32, things are pretty easy:

- We have a dedicated instruction for that
- `BPIALL`, aka `mcr p15, 0, rx, c7, c5, 6`
- Works nicely on Cortex-A12 and A17
- But...
- The architecture doesn't mandate that the BTB is visible to SW
- Which means that implementing `BPIALL` as `NOP` is valid!
- Duh!

arm

Branch prediction invalidation on the ARM architecture

On AArch32, things are pretty easy:

- We have a dedicated instruction for that
- `BPIALL`, aka `mcr p15, 0, rx, c7, c5, 6`
- Works nicely on Cortex-A12 and A17
- But...
- The architecture doesn't mandate that the BTB is visible to SW
- Which means that implementing `BPIALL` as `NOP` is valid!
- Duh!

On AArch64, the situation is much clearer:

Branch prediction invalidation on the ARM architecture

On AArch32, things are pretty easy:

- We have a dedicated instruction for that
- `BPIALL`, aka `mcr p15, 0, rx, c7, c5, 6`
- Works nicely on Cortex-A12 and A17
- But...
- The architecture doesn't mandate that the BTB is visible to SW
- Which means that implementing `BPIALL` as `NOP` is valid!
- Duh!

On AArch64, the situation is much clearer:

- There is no architectural way of invalidating the branch predictor
- I said clearer, I didn't say good!

arm

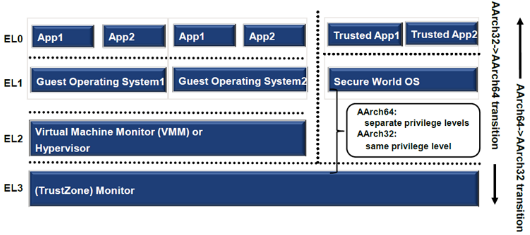
Branch prediction invalidation, the non-architected way

- We need to find per-implementations ways to invalidate the BP
- Requires intimate knowledge of the μ -architecture:
  - Cortex-A15: Set a chicken bit, and invalidate the I$-cache
  - Cortex-A57: Turn the MMU off, turn it back on
  - Cortex-A73: Switch to AArch32, issue a `BPIALL`
  - ...
- And that's just for a few ARM Ltd CPUs
- What about CPUs designed by others implementers?
- We could litter the kernel with multiple ways of invalidating the BP
- But this doesn't scale

Or we could start abstracting thing...

Firmware to the rescue

- Almost all implementations of ARMv8 have a secure mode
  - Most privileged execution level known as `EL3`
- Usually used for things like power-management, secure services
  - Things you don't want to see in the Linux kernel
  - Perfect for abstracting things that are very different from one implementation to another



- The hypervisor can easily trap into EL3 to execute a service on its behalf
  - Using the `SMC` instruction
- Only one implementation (XGene-1) doesn't have this feature
  - Let's ignore it (users can turn the BP off altogether)
- Let's implement our branch predictor invalidation at EL3!

Improving the SMC calling convention and discovery mechanisms

- A trap to EL3 itself is pretty cheap
- But the SMC Calling Convention (SMCCC) follows the PCC
- Imposes saving/restoring 18 GPRs (on guest exit, all the registers are live)!
- This becomes an overhead for exit-heavy workloads

Improving the SMC calling convention and discovery mechanisms

- A trap to EL3 itself is pretty cheap
- But the SMC Calling Convention (SMCCC) follows the PCC
- Imposes saving/restoring 18 GPRs (on guest exit, all the registers are live)!
- This becomes an overhead for exit-heavy workloads

Let's design a new variant of SMCCC that reduce this overhead: `SMCCC-1.1`

- Guarantees that at most 4 registers are clobbered
- Provides a discovery mechanism coupled with PSCI
  - Allows the SMCCC version to be queried
- Defines official "architecture workarounds"
- Implemented by ATF (ARM's reference firmware implementation)
- Each vendor to provide their back-end
  - No implementation? Potentially no mitigation. Pick your vendor carefully...

Meet SMCCC_ARCH_WORKAROUND_1

- An abstracted, firmware provided way to invalidate the branch predictor
- Implemented on top of SMCCC-1.1
- system-wide service
- Allow the caller to find out whether this particular CPU requires it...

Meet SMCCC_ARCH_WORKAROUND_1

- An abstracted, firmware provided way to invalidate the branch predictor
- Implemented on top of SMCCC-1.1
- system-wide service
- Allow the caller to find out whether this particular CPU requires it...
- ... to support asymmetric configurations
  - Yes, big-little strikes back, and it's not happy...
- It is always safe to call this service on any CPU
- Even those that doesn't require it
- May come at the expense of performance...

Spectre-v2: KVM implementation details

- Remember this “invalidate the BP before any branch”?
- This is a big constraint:
  - Must the first thing happening on guest exit
  - Has to fit in the exception vectors: 32 instructions on 64bit systems
  - Must not impact non-affected CPUs, including big-little systems
- The trick is to introduce per-CPU vector tables
- One set of vectors that just do what they are supposed to do
  - Let’s call them “canonical” vectors
- One set of vectors for affected CPUs
  - First calling `SMCCC_ARCH_WORKAROUND_1`
  - Then branch to the canonical set of vectors

And what about AArch32?

Do you really want to know?

And what about AArch32?

Do you really want to know?

- No good firmware story on 32bit
- Fortunately, there is exactly two affected implementations
  - Cortex-A15: can issue a full I$ invalidation
  - Cortex-A12/A17: can directly use BPIALL
- We can use the same per-CPU vector trick
- Only issue is that we need to fit both BP invalidation and branch in

And what about AArch32?

Do you really want to know?

- No good firmware story on 32bit
- Fortunately, there is exactly two affected implementations
  - Cortex-A15: can issue a full I$ invalidation
  - Cortex-A12/A17: can directly use BPIALL
- We can use the same per-CPU vector trick
- Only issue is that we need to fit both BP invalidation and branch in
- Exactly One Single Instruction
- Not happening

arm

And what about AArch32?

Do you really want to know?

- No good firmware story on 32bit
- Fortunately, there is exactly two affected implementations
  - Cortex-A15: can issue a full I$ invalidation
  - Cortex-A12/A17: can directly use BPIALL
- We can use the same per-CPU vector trick
- Only issue is that we need to fit both BP invalidation and branch in
- Exactly One Single Instruction
- Not happening
- Warning, ugliest hack follows

AArch32 hack, simplified version

```
	.align 5
__kvm_hyp_vector:
	.global __kvm_hyp_vector

	b	hyp_reset
	b	hyp_undef
	b	hyp_svc
	b	hyp_pabt
	b	hyp_dabt
	b	hyp_hvc
	b	hyp_irq
	b	hyp_fiq

__kvm_hyp_vector_bp_inv:
	.global __kvm_hyp_vector_bp_inv

	add	sp, sp, #1	/* Reset          7 */
	add	sp, sp, #1	/* Undef          6 */
	add	sp, sp, #1	/* Syscall        5 */
	add	sp, sp, #1	/* Prefetch abort 4 */
	add	sp, sp, #1	/* Data abort     3 */
	add	sp, sp, #1	/* HVC            2 */
	add	sp, sp, #1	/* IRQ            1 */
	nop			/* FIQ            0 */

	mcr	p15, 0, r0, c7, c5, 6	/* BPIALL */
	isb
```

```
	.macro vect_br val, targ
	eor	sp, sp, #\val
	tst	sp, #7
	eorne	sp, sp, #\val
	beq	\targ
	.endm

	vect_br	0, hyp_fiq
	vect_br	1, hyp_irq
	vect_br	2, hyp_hvc
	vect_br	3, hyp_dabt
	vect_br	4, hyp_pabt
	vect_br	5, hyp_svc
	vect_br	6, hyp_undef
	vect_br	7, hyp_reset
```

- Start with a stack 64bit aligned
- Increment SP based on the entry vector
- Perform some voodoo to test and restore SP to its original value
- Branch to the right vector
- More complicated with Thumb2 ISA

arm

Fixing Variant-2 for good

- Variant-2 only exists because of a lack of tagging in the branch predictor
- The architecture now outlaws the lack of branch predictor tagging
- This property is exposed to system software via a system register
- A number of current implementations have been updated to address this

arm

Meltdown

Meltdown, aka variant-3

What is it?

- A speculative memory access can bypass permission checks

How is that possible?

- Fetching the data and the permission checks are done in parallel (oops...)
- Early Cortex-A75 is affected

Mitigating Meltdown in KVM

- The guest runs at EL0/EL1, and KVM at EL2
- Different translation regimes
- No possibility of using the EL2 translation regime
- No need to mitigate anything in KVM
- The userspace side is protected by KPTI
- The only known affected implementation has been fixed

Variant-3a

Variant 3a

What is it?

- Somehow similar to Variant 3, aka Meltdown
- Allows a privileged system register to be speculatively read
- It sounds worse than it is, really
  - Most system registers have static values
  - The only interesting thing is VBAR (vector base address), which is a VA
  - VBAR discloses information about the VA layout
  - Pretty annoying, as we're introducing HYP VA randomisation
  - This can be used together with other variants...
- Only affects two implementations: Cortex-A57 and A72

How is that possible?

- Probably similar to what happens on Variant 3
- We're speculating, so let's just return the data!
  - Resolving the speculation later will sort it out

Interlude: lessons learned from KPTI

- The mitigation for Meltdown already gives us a good start
- The kernel’s own vector base register (VBAR_EL1) is at a well known location
- Doesn’t disclose anything about the kernel layout
- We can use the same trick!

Interlude: lessons learned from KPTI

- The mitigation for Meltdown already gives us a good start
- The kernel's own vector base register (VBAR_EL1) is at a well known location
- Doesn't disclose anything about the kernel layout
- We can use the same trick!
- Let's identity-map the vectors
- VBAR_EL2 won't disclose much about the hypervisor VA layout
- Branch back to the original, non id-mapped vectors

Interlude: lessons learned from KPTI

- The mitigation for Meltdown already gives us a good start
- The kernel’s own vector base register (VBAR_EL1) is at a well known location
- Doesn’t disclose anything about the kernel layout
- We can use the same trick!
- Let’s identity-map the vectors
- VBAR_EL2 won’t disclose much about the hypervisor VA layout
- Branch back to the original, non id-mapped vectors

Did you say “branch from the vectors”?

Interlude: branching "far" on AArch64

- We need to branch from an id-mapped address (a PA)...
- ... to a random VA location
- Both are in a 52bit address space (though in practice 48bits)
- Yes, this is large
- Direct branches on AArch64 are PC-relative
- Max displacement is 4GB
- We must perform an indirect branch...
- ... After having applied the v2 mitigation

arm

Interlude: branching “far” on AArch64

- We need to branch from an id-mapped address (a PA)...
- ... to a random VA location
- Both are in a 52bit address space (though in practice 48bits)
- Yes, this is large
- Direct branches on AArch64 are PC-relative
- Max displacement is 4GB
- We must perform an indirect branch...
- ... After having applied the v2 mitigation

Two possibilities:

- We load the target VA from memory: easy, but also costly
- We patch the target VA at runtime with a series of constant loads

Interlude: branching “far” on AArch64

- We need to branch from an id-mapped address (a PA)...
- ... to a random VA location
- Both are in a 52bit address space (though in practice 48bits)
- Yes, this is large
- Direct branches on AArch64 are PC-relative
- Max displacement is 4GB
- We must perform an indirect branch...
- ... After having applied the v2 mitigation

Two possibilities:

- We load the target VA from memory: easy, but also costly
- We patch the target VA at runtime with a series of constant loads

Of course the second choice is much harder, let’s do that!

Interlude: Alternative sequences, the dynamic way

The arm64 port so far has used a simple way of patching the kernel

- Each “canonical” sequence can only have a single possible alternative

```
alternative_if ARM64_HAS_PAN
        b        1f
alternative_else_nop_endif
```

- It worked so far, but we we now need to make it more dynamic
- We already have a way to generate kernel code thanks to BPF...
- We can reuse the code generation part to our advantage

arm

Interlude: Alternative sequences, the dynamic way

The arm64 port so far has used a simple way of patching the kernel

- Each “canonical” sequence can only have a single possible alternative

```
alternative_if ARM64_HAS_PAN
        b        1f
alternative_else_nop_endif
```

- It worked so far, but we we now need to make it more dynamic
- We already have a way to generate kernel code thanks to BPF...
- We can reuse the code generation part to our advantage

```
alternative_cb kvm_patch_vector_branch
        b        __kvm_hyp_vector + (1b - 0b)
        nop
        nop
        nop
        nop
alternative_cb_end
```

→

```
stp     x0, x1, [sp, #-16]!
movz    x0, #(addr & 0xffff)
movk    x0, #((addr >> 16) & 0xffff), lsl #16
movk    x0, #((addr >> 32) & 0xffff), lsl #32
br      x0
```

- Where `kvm_patch_vector_branch` is a C function that generates the sequence,
- and `addr` is the vectors VA computed at runtime

Life after Variant-3a

- Quite a lot of effort to hide one single system register
- But worth the effort to allow randomisation of the HYP VA space
- Thankfully the number of affected implementation is limited

SSB

Variant-4, aka Speculative Store Bypass

What is it?

- A load from an address may, under speculation, observe the result of a store that is not the latest store to that address
- For the Linux kernel, this impacts the way the stack is used:
  - User space performs a syscall, passing some parameters
  - Kernel copies user data on the stack
  - Syscall executed, stack frame discarded, and control returns to userspace
  - Later on, the kernel uses that same stack region for its own purpose
  - Writes something to the stack, reads it back, and uses it to index an array

How is that possible?

- Speculation may ignore the dependency
- Due to a write buffer
- Or the use of a different VA

Mitigation of Variant 4

At the time of discovery:

- No provision for preventing this behaviour in the architecture
- Luckily, all affected implementations have a “Load Bypass Store Disable” chicken bit!
  - Guarantees that Variant 4 cannot occur (affects all loads)
  - Configured from EL3
- Can either be statically set from boot (if the overhead is minimal)
- Or switched on demand using a secure service: `SMCCC_ARCH_WORKAROUND_2`
  - Enabled on entry from userspace to the kernel
  - Disabled on exit to userspace
  - Userspace can query the mitigation status, and ask to be mitigated
  - Slight departure from the x86 mitigation (kernel always mitigated)

My new best friend: SMCCC_ARCH_WORKAROUND_2

- Implemented similarly to SMCCC_ARCH_WORKAROUND_1
  - Except it has a state!
- Implemented system wide, per-CPU vulnerability status
- Mitigation is also exposed to virtual machines
  - Guest can change mitigation state by using the same SMCCC_ARCH_WORKAROUND_2 call
  - KVM forwards the call to EL3
  - Requires context tracking
  - Guest state part of the migration state

Moving on *with* variant 4

The ARMv8.5 architecture has added some new features to deal with more efficiently with variant-4.

- A `PSTATE` bit that serves the same purpose as `SMCCC_ARCH_WORKAROUND_2`
- Defaults to “safe”
- Avoids traps to EL2/EL3
- Patches merged into 4.20!

Conclusion

- The Linux kernel and KVM on ARM have been heavily modified to cope with Spectre
- We've managed to do it using a number of abstractions
- And as the result of an intense collaborative effort
  - Architecture
  - Implementation
  - Firmware
  - Kernel
- The architecture itself has evolved to limit these problems in the future

arm

Thank you

Backup

Variant 1 (Spectre v1) canonical example

```
1  struct array {
2          unsigned long length;
3          unsigned char data[];
4  };
5
6  struct array *arr1 = ...; /* small array */
7  struct array *arr2 = ...; /* array of size 0x400 */
8  unsigned long untrusted_offset_from_user = ...;
9
10 if (untrusted_offset_from_user < arr1->length) {
11         unsigned char value;
12
13         value = arr1->data[untrusted_offset_from_user];
14         unsigned long index2 = ((value & 1)*0x100)+0x200;
15
16         if (index2 < arr2->length) {
17                 unsigned char value2 = arr2->data[index2];
18         }
19 }
```

10 Slow to resolve condition, speculating it as valid

13 `value` is now speculated using an untrusted offset

- This gives the attacker a control address

17 `index2`, which is the secret is now used to perform a speculative access

We can now to do a timing measurement on the `arr2` array to extract one bit of `value`.

