Whitepaper

Spectre-BHB: Speculative Target Reuse Attacks

Version 1.8-r0p2

Overview

Branch prediction allows modern CPUs, including those from Arm, to speculatively select the instruction stream based on multiple mechanisms that consider the branch instruction information including the history of previously executed branches. On Arm CPUs, that information is stored in an internal structure sometimes referred to as the Branch History Buffer (BHB). The architecture does not describe such mechanisms, and implementations can make use of different techniques to speculatively change the instruction stream. This particular attack, known as Spectre-BHB [1], has been assigned the CVE number CVE-2022-23960.

While Spectre-BHB is similar to Spectre v2 [2], the CSV2 hardware features introduced to mitigate against Spectre v2 do not work against Spectre-BHB. This whitepaper discusses the differences between the two attacks and describes the mitigations necessary to protect against Spectre-BHB.

Spectre v2

Description

Code running in a Variant 2 vulnerable CPU in one security domain or context (i.e., security state, exception level, VM, or process) could train the branch predictor to induce another context to speculate over incorrect instruction streams. See Figure 1. Transient execution attacks have shown that this mis-speculation in one context can leave traces (e.g., cache allocations) to be later measured from another context to infer secret information.

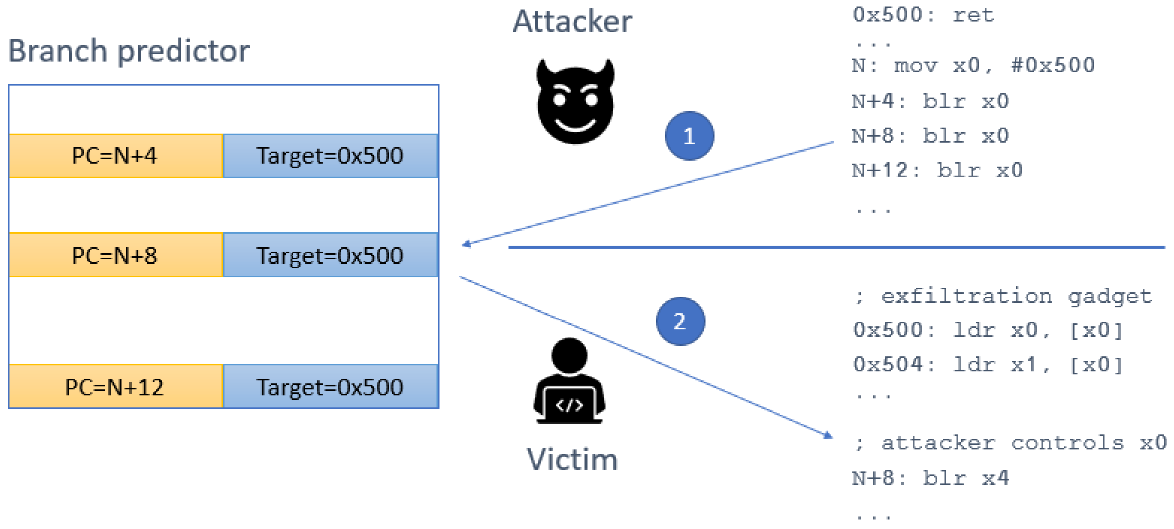
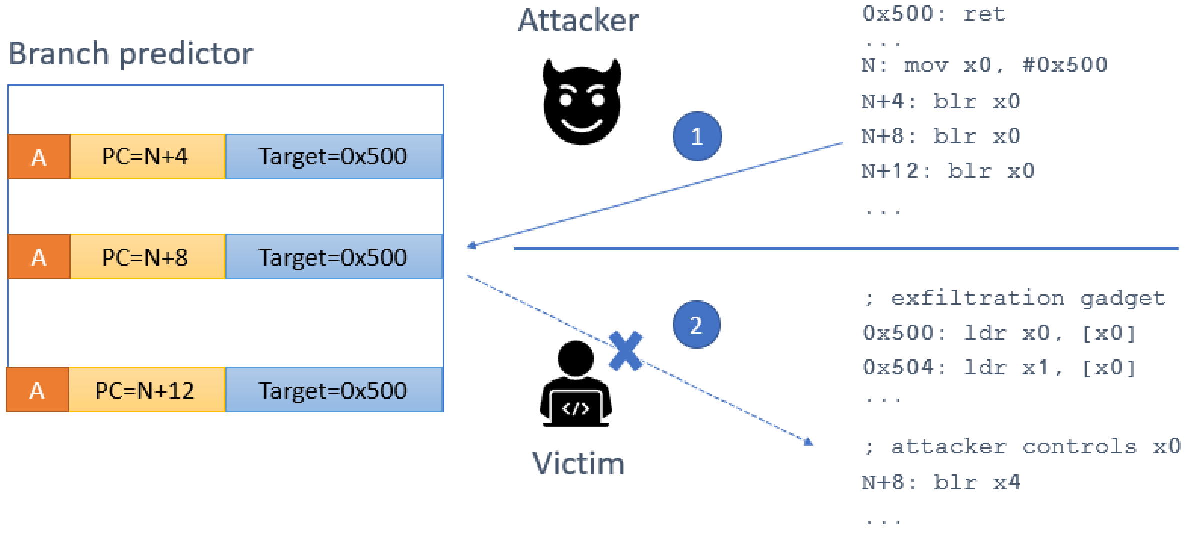


Figure 1. Spectre v2 attack. (1) Adversary trains branch predictor from the adversary's own context to branch to address 0x500, where the victim VA space contains an exfiltration gadget. (2) Victim executes a branch instruction and consumes the prediction generated for the adversary, leading to speculative leakage.

FEAT_CSV2

Arm added FEAT_CSV2 which provides additional restrictions on the architecture to filter branch prediction by the hardware described context (e.g., security state + exception level + VMID + ASID, and optionally SCXTNUM[1]) that the processor is in. The purpose of these restrictions is to prevent code running in one context from training the branch predictors in an adversary-controlled way, that could induce other contexts to speculatively leak secret data. With FEAT_CSV2, code running in one context cannot inject the targets consumed by branch predictions in another context. See Figure 2.



```
0x500: ret
...
N: mov x0, #0x500
N+4: blr x0
N+8: blr x0
N+12: blr x0
...
```

```
; exfiltration gadget
0x500: ldr x0, [x0]
0x504: ldr x1, [x0]
...
; attacker controls x0
N+8: blr x4
...
```

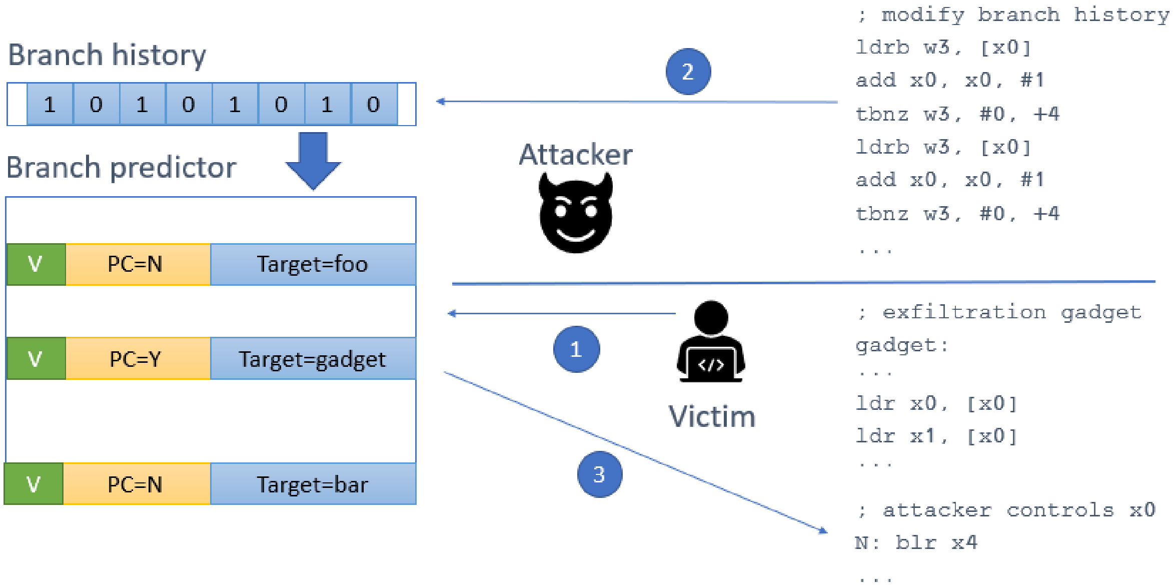
Figure 2. FEAT_CSV2 separates predictions by context, in such a way that predictions made for the adversary cannot be consumed by the victim. (1) Adversary trains branch predictor from her own context to branch to address 0x500, where the victim VA space contains an exfiltration gadget. (2) Victim executes a branch instruction, but because the entry's context differs no prediction is made.

[1] Software Context Numbers allow to define finer grained contexts to separate security domains running in the same process. For instance, to separate an eBPF program from the rest of the kernel or isolate the JavaScript code in a web browser.

Spectre-BHB

Description

Since FEAT_CSV2 stops an adversary from controlling predictions in another context, Spectre-BHB instead forces and exploits the mis-prediction of the victim's own predictions. For that, it relies on the fact that many implementations use, along with the branch information, a globally shared branch history to inform branch predictions. In such implementations, the adversary could tamper with the branch history from one context and force mis-predictions in another context, leading to speculative execution of incorrect instruction streams. See Figure 3.



```
; modify branch history
ldrb w3, [x0]
add x0, x0, #1
tbnz w3, #0, +4
ldrb w3, [x0]
add x0, x0, #1
tbnz w3, #0, +4
...
```

```
; exfiltration gadget
gadget:
...
ldr x0, [x0]
ldr x1, [x0]
...

; attacker controls x0
N: blr x4
...
```

Figure 3. (1) Victim runs normally, executing an indirect branch with different safe targets ("foo" and "bar"), and some other branch to a potential (given the right register values) exfiltration "gadget". This makes several branch predictions in the victim's context. (2) The adversary tampers the global branch history, forcing the combination of PC=N and history to alias with the "gadget" entry. (3) The victim executes an allegedly safe branch that is mis-predicted, redirecting the control flow to a gadget that, with adversary-controlled registers, causing speculative leakage.

Practicality of target reuse attacks

The complexity of this attack is higher than conventional Spectre v2, as it requires:

1. The existence of an exfiltration primitive in the victim's domain that is predicted as a valid branch target as the result of the victim's normal execution.
2. Adversary's control over the mis-prediction (e.g., via the branch history) right before the branch; the more instructions executed in between—the adversary's control sequence and the victim's branch—the less control.

This contrasts with traditional Spectre v2, where the adversary could directly train the predictor across contexts with arbitrary branch targets. Furthermore, since a valid target will usually point to a valid function entry point rather than right to the exfiltration gadget, a longer speculation window is required. Whilst this alone is not sufficient to stop an attack, it adds to the list of constraints.

Branch target misprediction is an inherent problem of any efficient predictor implementation. For instance, in the best case, consider an indirect branch with multiple targets. It is unlikely that the predictor will not mis-predict from time to time. This case is similar to the conditional branch misprediction of Spectre v1, where the adversary invokes the victim several times to train the prediction into a certain direction before triggering the mis-speculation.

However, the problem is exacerbated when the predictor can mix targets used by branches at different locations (among an aliased subset or among all branches), and even more when the adversary has active control over the misprediction (e.g., via the branch history). This control implies that the adversary can reliably repeat a specific misprediction: deterministically (i.e., occurs each time), or probabilistically (i.e., occurs after enough repetitions).

Mitigations

On CPUs affected by Spectre-BHB, to protect against attacks across exception levels or security states, Arm recommends adding a loop to discard the branch history on exception entry to a higher exception level. That loop will execute some core specific number ("K") of iterations.

If the core implements the Speculation Barrier instruction (SB), then the following sequence should be used:

```
   MOV x0, #K // core specific number
loop:
   B PC+4
   SUBS x0, x0, #1
   BNE loop
   SB
```

Otherwise, the following sequence should be used:

```
   MOV x0, #K // core specific number
loop:
   B PC+4
   SUBS x0, x0, #1
   BNE loop
   DSB
   ISB
```

Table 1 contains the list of affected cores and their required K values:

Core	K value	CSV2	ECBHB
Cortex-A15	8	N/A	no
Cortex-A57	8	0000	no
Cortex-A65	(see note 4 below)	0001	no
Cortex-A65AE	(see note 4 below)	0000	no
Cortex-A72 prior to r1p0	8	0000	no
Cortex-A72 from r1p0	8	0001	no
Cortex-A73	(see note 2 below)	N/A	no
Cortex-A75	(see note 2 below)	N/A	no
Cortex-A76	24	0001	no
Cortex-A76AE	24	0001	no
Cortex-A77	24	0001	no
Cortex-A78	32	0001	no
Cortex-A78AE	32	0001	no
Cortex-A78C	32	0001	no
Cortex-X1	32	0001	no
Cortex-X2	32	0010	no
Cortex-X3 prior to r1p1	132	0010	no
Cortex-X3 from r1p1	132	0010	yes
Cortex-A710	32	0010	no
Cortex-A715 prior to r1p1	38	0010	no
Cortex-A715 from r1p1	38	0010	yes
Cortex-A720	38	0010	yes
Neoverse E1	(see note 4 below)	0001	no
Neoverse N1	24	0001	no
Neoverse N2	32	0010	no
Neoverse V1	32	0001	no
Neoverse V2 prior to r0p1	132	0010	no
Neoverse V2 from r0p1	132	0010	yes

Table 1. Number of iterations required to override the branch history. CSV2 values: 0b0000=not disclosed; 0b0001=Spectre-v2 mitigation w/o SCXTNUM; 0b0010=Spectre-v2 mitigation and SCXTNUM. Support for ECBHB.

Notes:

1. Even though the Cortex-A15, the Cortex-A57, and the Cortex-A72 prior to r1p0 do not implement FEAT_CSV2, the mitigation with the loop iterations specified for the Cortex-A72 from r1p0 will work to mitigate against Spectre-BHB.
2. Mitigating Spectre-BHB on the Cortex-A73 and Cortex-A75 requires the entire branch predictor to be invalidated, regardless of whether the revision implements FEAT_CSV2. In Aarch64, functionality can only be implemented in firmware.
   Accordingly, a new Secure Monitor Call SMCCC_ARCH_WORKAROUND_3 is specified to implement the mitigation on these (and similarly affected) cores.
3. While the Cortex-A510 does implement FEAT_CSV2, there is sufficiently limited load speculation that it should not be possible to create a practical attack using Spectre-BHB.
4. The mitigation for these cores consists of disabling and re-enabling the MMU at the highest implemented exception level.

Cores without FEAT_CSV2

The mitigations for Spectre v2, which involve flushing all branch predictions via an implementation specific route on every context switch, will also mitigate against Spectre-BHB. Accordingly, this is the recommended mitigation for cores like the A72 (prior to r1p0) and A57 which do not implement FEAT_CSV2.

Same context attacks

Environments like eBPF augment the risk of this class of attacks. eBPF programs can run in the same context as the rest of the kernel, allowing adversariess 1) to insert exfiltration gadgets (removing the need to find a suitable one in the kernel), and 2) to control the misprediction from the same context (rendering mitigations triggered on context switch, such as FEAT_ECBHB, insufficient).

Given the broad range of attack vectors for eBPF, and the high-performance requirements, Arm strongly recommends that systems ensure that only eBPF code supplied by trusted parties is used. Please note that since eBPF is not supplied or developed by Arm, we cannot guarantee security for different instances of eBPF.

Please also note that some environments provide mechanisms for signing eBPF code to ensure the trustworthy nature of this code.

Hardware Mitigations

CLRBHB

A new instruction, CLRBHB will be added in HINT space. This instruction is implemented as part of FEAT_CLRBHB, which is optional in all versions of the architecture from Armv8.0 to Armv8.8, and from Armv9.0 to Armv9.3.

CLRBHB clears the branch history for the current context to the extent that branch history information created before the CLRBHB instruction cannot be used by code before the CLRBHB instruction to exploitatively control the execution of any code in the current context appearing in program order after the instruction.

Allocation:

- AArch64: the CLRBHB instruction is allocated in Hint space, using HINT #22.
- AArch32, T32: HINT 001, with option 0110 is allocated as CLRBHB
- AArch32, A32: Move Special Register and Hints (immediate) instructions with R:imm4 ==00000 and imm12== (0000)00010110 is allocated as CLRBHB

Current implementations are protected against Spectre-BHB with the current loop value, while future implementations that might need a larger loop value would be built with the CLRBHB instruction so this sequence would be generically useful.

For future implementations, code that knows that CLRBHB has been implemented could omit the loop.

The CLRBHB instruction is completed by a subsequent ISB instruction executed by the same core.

An ID field ID_AA64ISAR2_EL1<31:28> is allocated as the CLRBHB field for the identification of CLRBHB in AArch64 as follows:

- 0000 - Hint #22 is a NOP
- 0001 - Hint #22 is implemented as CLRBHB

All other values reserved.

The ID_ISAR6/ID_ISAR6_EL1<31:28> field is allocated as the CLRBHB field for the identification of CLRBHB in AArch32 as follows:

- 0000 - CLRBHB is a NOP
- 0001 - The CLRBHB instruction is implemented

All other values reserved.

FEAT_ECBHB

The Arm architecture introduces a new feature FEAT_ECBHB, which requires that the branch history information created in a context before an exception to a higher exception level using AArch64 cannot be used by code before that exception to exploitatively control the execution of any code in a different context after the exception.

FEAT_ECBHB is optional in all versions of the architecture from Armv8.0 to Armv8.8, and from Armv9.0 to Armv9.3

An ID field ID_AA64MMFR1<63:60>, is allocated as the ECBHB field to allow the identification of FEAT_ECBHB:

- 0000 – The implementation does not disclose whether the branch history information created in a context before an exception to a higher exception level using AArch64 can be used by code before that exception to exploitatively control the execution of any code in a different context after the exception.
- 0001 – The branch history information created in a context before an exception to a higher exception level using AArch64 cannot be used by code before that exception to exploitatively control the execution of any code in a different context after the exception.

All other values reserved.

CSV2 Update

In the ID field ID_AA64PFR0_EL1.CSV2, a new encoding is added:

- 0b0011 Branch targets and branch history trained in one hardware-described context can exploitatively control speculative execution in a different hardware-described context only in a hard-to-determine way. The SCXTNUM_ELx registers are supported, and the contexts include the SCXTNUM_ELx register contexts.

This feature is described as FEAT_CSV2_3 and is optional to all versions of the architecture from Armv8.0 and Armv9.0.

References

1. “Branch History Injection”
   https://www.vusec.net/projects/bhi-spectre-bhb/
2. “Spectre attacks: Exploiting speculative execution”. Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. (2019)