SPECTREPPCWP

Speculative Execution Vulnerabilities and Mitigations on NXP PowerPC Processors

Rev. 0 — 08/2019 White Paper

Contents

1 Introduction

On January 3, 2018, a new class of side-channel attacks was disclosed. These side-channel attacks are novel in their use of speculative execution and exploitation of various caches in high-performance processors. These attacks use the fact that the speculative execution has side effects and these side effects allow malicious code to find out the contents of memory regions that the malicious code is not authorized to access.

The vulnerabilities were registered with National Institute of Standards and Technology (NIST), a non-regulatory agency within the U.S. Department of Commerce, as CVE-2017-5753 (Spectre variant 1), CVE-2017-5715 (Spectre variant 2), and CVE-5754 (Meltdown). Additional derivations of Spectre and Meltdown, also based on speculative execution side effects, have been discovered since the major January 3, 2018 disclosure.

In this paper, we present the mitigations for Spectre variant 1 and Spectre variant 2 for NXP PowerPC processors. The NXP PowerPC processors are not vulnerable to the Meltdown attack, Spectre variant 1.1/1.2, or Spectre variant 4. However, the new class of side-channel attacks has opened the door for more research and it is expected that new vulnerabilities or variations of the existing ones will be disclosed in the future.

2 Spectre vulnerability

To exploit Spectre, it is required that specially-crafted software be executed on the target system to influence the victim behavior and to extract the secrets. If the system is closed and does not allow running any untrusted code, then the vulnerability is low. Other layers of security should protect against malware in this case.

Also, in case of a system where all the tasks are sharing the same address space, Spectre does not apply because any task can already read the data from any other task.

The Spectre vulnerability comprises a series of vulnerabilities, some of them being just variations of the original ones: Spectre variant 1, Spectre variant 2, Spectre variant 4, Spectre variant 1.1/1.2, Spectre RSB. In certain cases, mitigations for one variant apply to other variants as well. The NXP Power Architecture processors are vulnerable to Spectre variant 1 and Spectre variant 2. The sections that follow describe the proposed mitigations for the affected processors.

3 Spectre variant 1 (CVE-2017-5753)

The Spectre variant 1 vulnerability, known as *bounds check bypass*, takes advantage of speculative execution of the instructions that follow a conditional branch.

Let us take the following code block as an example:

```
if (untrusted_array_index < array_size)
{
   value = array1[untrusted_array_index]     //value is used for other memory accesses
}
```

At first glance, there is nothing wrong with the above code block; the program is written correctly. In theory, there is no way for the array to be accessed out of bounds.

To speed up processing, modern CPUs include a feature, called *branch prediction*, which detects in advance whether the branch is taken or not.

The Spectre variant 1 attack exploits the branch prediction and speculative execution of instructions after a branch. When **array_size** is not in the cache (while it is getting loaded), the processor can execute instructions speculatively. Functionally, this is correct. When the outcome of the branch is successfully predicted, the speculatively-executed instructions will be committed.

If the branch was mispredicted, then the speculatively-executed instructions will be discarded; however, some indirect side effects, such as CPU cache changes, are not reverted. In the example above, the processor may speculatively execute the instructions after the branch. If the code uses **value** to create an address to access a memory location, then the memory location will be loaded in cache and the attack code can use a cache side channel to learn a few bits of **value**, though the attack code is not able to access **value** directly.

3.1 Software mitigations

Let us extend the example to index a second array using **value**.

```
if (untrusted_array_index < array_size)
{
   value = array1[untrusted_array_index];
   data = array2[value * 64];
}
```

The way the branch predictors work is to learn from the branch history. Therefore, at first, an attacker can use some sane values for **untrusted_array_index** to train the branch predictor. The second step will be to use an out of bounds value for **untrusted_array_index**. The branch predictor predicts that the branch will be taken (as it was previously trained) and the instructions after the branch will be speculatively executed.

If **array2** was not in the cache, then the attacker can probe all the entries and measure the response time. If the access is quick, it means that the data was accessed, and **value** can be inferred.

The above code can be translated into assembly code, as follows:

```
lis      %r9,array_size@h          //r9 – contains the array_size
lbz      %r9,array_size@l(%r9)
cmplw    cr7,%r9,%r3               //if (untrusted_array_index < array_size)
blelr    cr7
lis      %r9,array1@h              //the code can be executed speculatively
addi     %r9,%r9,array1@l
lbzx     %r10,%r9,%r3              //value loaded
lis      %r8,array2@h
addi     %r8,%r8,array2@l
rlwinm   %r10,%r10,6,0,25
lbzx     %r10,%r8,%r10             //data loaded
```

To prevent speculation, we insert a barrier after the branch, as follows:

```
lis      %r9,array_size@h
lbz      %r9,array_size@l(%r9)
cmplw    cr7,%r9,%r3               //if (array_index < array_size)
blelr    cr7
isync                              //barrier
sync                               //barrier
lis      %r9,array1@h              //the code can be executed speculatively
addi     %r9,%r9,array1@l
lbzx     %r10,%r9,%r3
lis      %r8,array2@h
addi     %r8,%r8,array2@l
```

```
rlwinm   %r10,%r10,6,0,25
lbzx     %r10,%r8,%r10             //data loaded
```

The inserted barrier prevents the CPU from speculatively executing the instructions after the branch. The barrier is forcing the result of the bounds check to resolve in the instruction pipeline.

Adding a barrier in assembly code will prevent speculative execution, but in practice, we will be dealing with C code. In this case, we can define a barrier function that can be called in the C code:

```
static inline void barrier_nospec(void)
{
   asm("isync;sync" : : : "memory");
}
```

The vulnerable example should be rewritten as follows:

```
if (untrusted_array_index < array_size)
{
   barrier_nospec()
   value = array1[untrusted_array_index];
   data = array2[value * 64];
}
```

3.2 Mitigations in Linux kernel

Similar to any other software component, Linux kernel may have code that is vulnerable to Spectre variant 1. Currently, the vulnerable pieces of code in the Linux kernel can be sanitized using one of the following macros:

- **`array_index_nospec()`**
- **`barrier_nospec()`**

array_index_nospec()

The Linux community has introduced a generic mechanism, a macro, **`array_index_nospec()`**, to prevent speculatively accessing the array out of bounds. The macro always returns a sane value for the **`untrusted_array_index`** to be used in the speculation. The macro needs to be manually inserted in places where the vulnerability exists. Currently, the macro is being used at several places in the Linux kernel; however, it is an ongoing process to update other vulnerable pieces of code as they are discovered by the Linux community.

Let us take a short example from the Linux kernel that uses the **`array_index_nospec`** mechanism:

```
 @@ -1397,6 +1398,7 @@ long vhost_vring_ioctl(struct vhost_dev *d, unsigned int ioctl, void __user
*arg
      if (idx >= d->nvqs)
      return -ENOBUFS;
+     idx = array_index_nospec(idx, d->nvqs);
      vq = d->vqs[idx];

***[drivers/vhost/vhost.c – kernel 5.0]***
```

The **`idx`** in the above example is controlled by user space; therefore, the code has a potential of getting exploited using Spectre variant 1 attack. The code after the bounds check might be executed speculatively, and the solution is to sanitize **`idx`** to always have a value that is within the bounds.

The **`array_index_nospec()`** macro is used in situations similar to above to force the array index value to be within the bounds.

barrier_nospec()

The **barrier_nospec()** macro is a barrier that prevents instructions following it from being executed speculatively. In some situations, the complexity of the code in Linux kernel is higher than just a simple array access. In such a situation, **barrier_nospec()** is used instead of **array_index_nospec()**. Examples in the PowerPC Linux kernel include **syscall** table sanitization and **copy_from_user()** function.

The kernel patches that add support for **barrier_nospec** implementation were included in kernel 4.19 release (see below). They were also backported on LTS kernels, 4.14 and 4.9.

```
Diana Craciun (6):
powerpc/64: Disable the speculation barrier from the command line
powerpc/64: Make stf barrier PPC_BOOK3S_64 specific
powerpc/64: Make meltdown reporting Book3S 64 specific
powerpc/fsl: Add barrier_nospec implementation for NXP PowerPC Book3E
powerpc/fsl: Sanitize the syscall table for NXP PowerPC 32-bit platforms
Documentation: Add nospectre_v1 parameter

Michael Ellerman (2):
powerpc/64: Add CONFIG_PPC_BARRIER_NOSPEC
powerpc/64: Call setup_barrier_nospec() from setup_arch()
```

The mitigations are enabled by default; however, the user can disable **barrier_nospec** using the kernel boot parameter, **nospectre_v1**.

In addition, the user can determine if the barrier is enabled by checking the following sysfs entry:

```
cat /sys/devices/system/cpu/vulnerabilities/spectre_v1
```

4 Spectre variant 2 (CVE-2017-5715)

The Spectre variant 2 vulnerability, known as *branch target injection*, takes advantage of speculative execution of instructions following an indirect branch. By influencing the indirect branch target, an attacker can make malicious code to be executed speculatively. Then, the attacker can use a cache side channel to retrieve the data.

To support high-performance execution, branch predictors use the history of the previous branches to predict the branch target. However, the history is not filtered by the privilege level or process ID, and therefore, an attacker from a different privilege level/process can mistrain a branch predictor to speculatively execute a gadget.

Let us take an example with two processes:

1. The attacker finds a gadget in the victim address space. The attacker finds the virtual address of the gadget.
2. The attacker performs indirect branches to the gadget address. The indirect branches are performed in the attacker address space. Because the branch predictor is shared between processes, the attacker can influence the behavior of the branch predictor in the victim.
3. The victim speculatively executes code at an unintended branch target (the branch predictor was previously trained to predict this target).
4. The effects of the incorrect speculative execution are eventually reverted; however, caches have side effects that can be used to leak sensitive data.

4.1 Mitigations in Linux kernel

Mitigations for Spectre variant 2 were implemented in the Linux kernel that protect against the following situations:

- User space process attacking another user space process
- User space process attacking the kernel
- Guest attacking the host (in case of KVM hypervisor)

The kernel patches add support for flushing the branch predictor whenever the privilege level changes. This way, any history of mistraining will be lost.

The patches were included in kernel 5.0 release (see below). They were also backported on LTS kernels, 4.19, 4.14, and 4.9.

```
Diana Craciun (11):
powerpc/fsl: Add infrastructure to fix-up branch predictor flush
powerpc/fsl: Add macro toflush the branch predictor
powerpc/fsl: Fix spectre_v2 mitigations reporting
powerpc/fsl: Emulate SPRN_BUCSR register
powerpc/fsl: Add nospectre_v2 command line argument
powerpc/fsl: Flush the branch predictor at each kernel entry (64bit)
powerpc/fsl: Flush the branch predictor at each kernel entry (32it)
powerpc/fsl: Flush branch predictor when entering KVM
powerpc/fsl: Enable runtime patching if nospectre_v2 boot argument is used
powerpc/fsl: Update Spectre V2 reporting
powerpc/fsl: Add FSL_PPC_BOOK3E as supported arch for nospectre_v2 boot argument

Diana Craciun (1):
powerpc/fsl: Fix warning: orphan section `__btb_flush_fixup'
```

The mitigations are enabled by default; however, the user can disable the branch predictor flush using the kernel boot parameter, **`nospectre_v2`**.

In addition, the user can determine if the branch predictor flush is enabled by checking the following sysfs entry:

```
cat /sys/devices/system/cpu/vulnerabilities/spectre_v2
```

4.2 Mitigations in other operating systems

Software running under the Linux kernel is protected by the mitigations discussed earlier. However, software running on a different operating system might be vulnerable. As a possible solution, a mitigation similar to one for Linux kernel can be implemented that clears the history of the branch predictor when switching between privilege levels.

The branch predictor history can be invalidated by setting BBFI bit in BUCSR register. The branch predictor can be invalidated and enabled using the following code sequence:

```
// Branch prediction
xor r4,r4,r4          // set r4 to 0
ori r5,r4,0x0201      // set BBFI and BPEN
oris r5,r5,0x0140     // set STAC_EN and LS_EN
mtspr BUCSR,r5        // flash invalidate and enable branch prediction
isync                 // synchronize setting of BUCSR
```

5 Related resources

Following are links to some resources related to the material presented in this paper:

- Spectre and Meltdown Updates for Power ISA Cores
- Spectre Attacks: Exploiting Speculative Execution
- https://meltdownattack.com/
- https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

6 Revision history

The table below summarizes the changes made to this document.

Table 1. Revision history

Revision	Date	Topic cross-reference	Change description
Rev. 0	08/2019		Initial public release

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 08/2019

Document identifier: SPECTREPPCWP