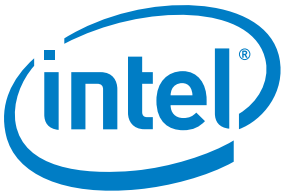


Retpoline: A Branch Target Injection Mitigation

White Paper

Revision 1.1
February, 2018

Document Number: 337131-001

Contents

Revision History

Date	Revision	Description
February 14, 2018	1.0	Initial release.
February 21, 2018	1.1	Added information about IA32_ARCH_CAPABILITIES MSR.

§

1.0 Introduction

As detailed by Google Project Zero and security researchers, three new side-channel analysis methods were discovered that potentially facilitate access to unauthorized information. All of the methods take advantage of speculative execution, a common technique in processors used to achieve high performance. Speculative execution is based on predictions, and differs from the normal execution visible to programmers. These predictions can be imprecise and can result in speculative execution that cannot possibly occur in "normal" execution.

For additional background information, refer to the overview in Intel's Analysis of Speculative Execution Side Channels. You can find more detailed explanations of Speculative Execution Side-Channel Mitigations and Intel's Mitigation Overview for Potential Side-Channel Cache Exploits in Linux* on our Side-Channel Security Support website.

This article discusses the details, exploit conditions, and mitigations for the exploit known as branch target injection (Spectre variant 2). There are a number of possible mitigation techniques for this method, the mitigation technique described in this document is known as retpoline. We discuss how it functions in detail below, along with the limitations and its preconditions for effective deployment.

2.0 Branch Target Injection (Spectre variant 2)

2.1 Background

The branch target injection (Spectre variant 2) exploit targets a processor's *indirect branch predictor*. Direct branches occur when the destination of the branch is known from the instruction alone. Indirect branches[1], on the other hand, occur when the destination of the branch is not contained in the instruction itself, such as when the destination is read from a register or a memory location. The indirect branch predictor uses information about previously-executed branches to predict the destinations of future indirect branches.

Programmers' use of function pointers in compiled languages, like C and C++, can result in indirect calls. For instance, sort functions are frequently passed a comparison function. Each call from inside `sort()` to `compare()` in the example below is likely to be an indirect call.

```
int compare(int a, int b)
{
    return a < b;
}
sort(array, &compare);
```

In C++, calls to object functions are frequently implemented with indirect calls, especially when inheritance is being used.

```
Vehicle *car = new Car();
car->drive();
```

In addition to indirect branches that are performed explicitly by programmers, the compiler itself might insert indirect branches without the programmer ever being aware of them.

2.2 Exploit Composition

An exploit using branch target injection (Spectre variant 2) is composed of five specific elements, all of which are required for successful exploitation. Traditional application software which is not security-sensitive needs to be carefully evaluated for all five elements before applying mitigation.

1. The target of the exploit (the victim) must have some secret data that an exploit wants to obtain. In the case of an OS kernel, this includes any data

[1] A full list vulnerable indirect branch instructions is listed in Table 2.1 of Speculative Execution Side Channels.

outside of the user's permissions, such as memory in the kernel memory map.

2. The exploit needs to have some method of referring to the secret. Typically, this is a pointer within the victim's address space that can be made to reference the memory location of the secret data. Passing a pointer of an overt communication channel[2] between the exploit and victim is a straightforward way to satisfy this condition.

3. The exploit's reference must be usable during execution of a portion of the victim's code which contains an indirect branch that is vulnerable to exploitation. For example, if the exploit pointer value is stored in a register, the attacker's goal is for speculation to jump to a code sequence where that register is used as a source address for a move operation.

4. The exploit must successfully influence this indirect branch to speculatively mispredict and execute a *gadget*. This gadget, chosen by the exploit, leaks the secret data via a side channel, typically by cache-timing.

5. The gadget must execute during the "speculation window," which closes when the processor determines that the gadget execution was mispredicted.

The retpoline mitigation is applied to mitigate the vulnerable indirect branches in element 4 and has no effect on the other elements. But because the exploit depends on satisfying all five elements, removing element 4 is sufficient to stop the branch target injection (Spectre variant 2) exploit.

[2] An example overt channel is the system call interface between an OS kernel and an application.

3.0 Retpoline Concept

Mitigations for speculation-based, side-channel security issues fall into two categories: directly manipulating speculation hardware, or indirectly controlling speculation behavior. Direct manipulation of the hardware is generally performed by microcode updates or manipulation of hardware registers. Indirect control is accomplished via software constructs that limit or constrain speculation. Retpoline is a hybrid approach since it requires updated microcode to make the speculation hardware behavior more predictable on some processor models. However, retpoline is primarily a software construct that leverages specific knowledge of the underlying hardware to mitigate branch target injection (Spectre variant 2).

As discussed earlier, the branch target injection (Spectre variant 2) exploit relies on influencing the speculated targets of indirect branches. Indirect JMP and CALL instructions consult the indirect branch predictor to direct speculative execution to the most likely target of the branch. The indirect branch predictor is a relatively large hardware structure which cannot be easily managed by the operating system. Instead of attempting to manage or predict its behavior, a *retpoline* is a method to *bypass* the indirect branch predictor. Refer to Figure 1 and Figure 2 for the flow of indirect-branch prediction before and after retpoline is implemented.

Prediction of RET instructions differs from JMP and CALL instructions because RET first relies on the Return Stack Buffer (RSB). In contrast to the indirect branch predictors RSB is a last-in-first-out (LIFO) stack where CALL instructions “push” entries and RET instructions “pop” entries. This mechanism is amenable to predictable software control.

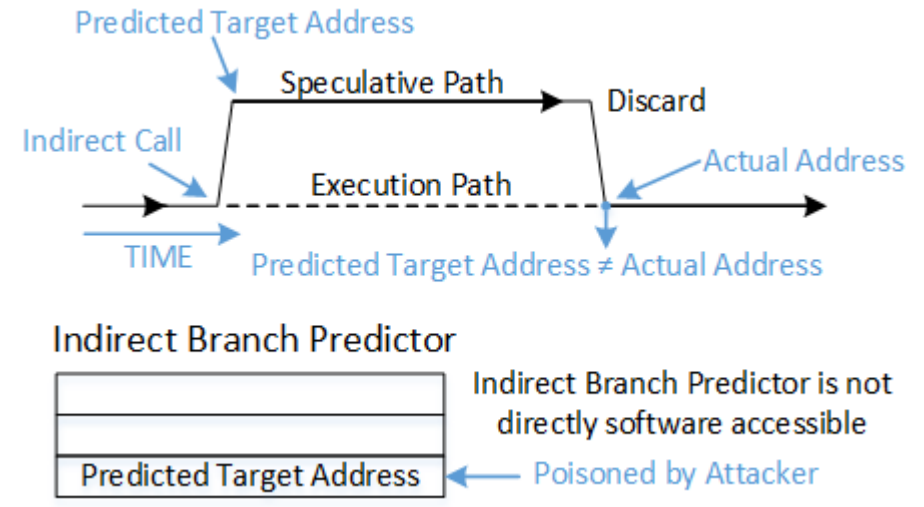


Figure 1: Speculative Execution without retpoline

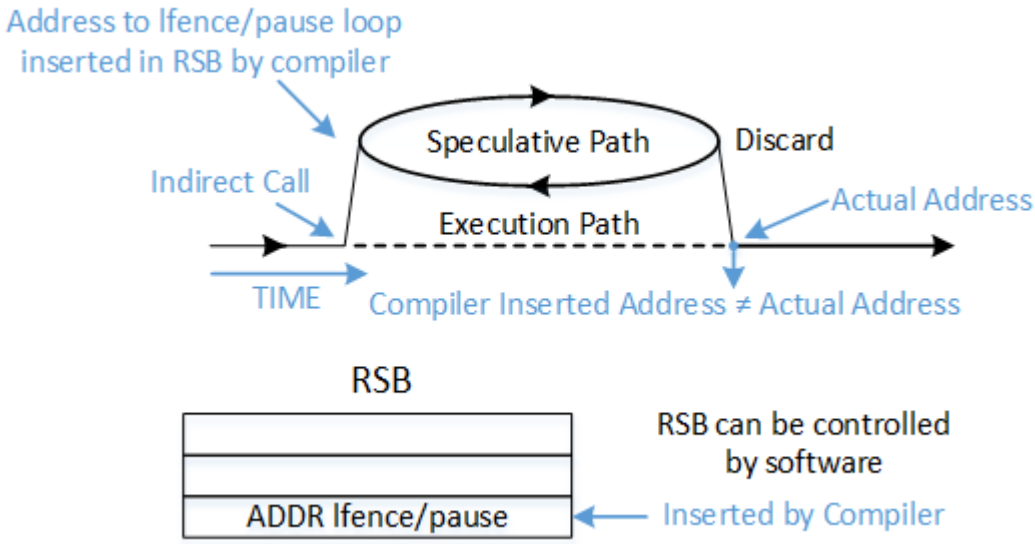


Figure 2: Speculative execution with retpoline

4.0 Retpoline Implementation

Deploying retpoline requires replacing vulnerable indirect branches with non-vulnerable retpoline sequences. The simplest retpoline sequence is a replacement for a single indirect JMP instruction.

Table 1: Indirect jump replacement with retpoline (gas syntax)

Before retpoline	`jmp *%rax`

After retpoline	`1: call load_label`
	`capture_ret_spec:`
	`2: pause ; lfence`
	`3: jmp capture_ret_spec`
	`load_label:`
	`4: mov %rax, (%rsp)`
	`5: ret`

In this example, a jump is performed to an instruction address stored in the `%rax` register. Without retpoline, the processor's speculative execution typically consults the indirect branch predictor and may speculate to an address controlled by an exploit (satisfying element 4 of the five elements of branch target injection (Spectre variant 2) exploit composition listed above).

The retpoline sequence is more complicated and works in several stages to separate the speculative execution from the non-speculative execution:

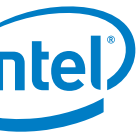
1. "`1: call load_label`" pushes the address of "`2: pause ; lfence`" on the stack and the RSB, and then jumps to:
2. "`4: mov %rax, (%rsp)`" takes the target of the indirect jump (in `%rax`) and writes it over the return address stored on the stack. At this point the in-memory stack and the RSB differ.
3. If speculating, the CPU uses the RSB entry created in step 1 and jumps to "`2: pause ; lfence`". It is "trapped" in an infinite loop. The Speculation Barriers section has more details about the importance of this sequence.
4. Eventually, the CPU realizes that the speculative `ret` does not agree with the in-memory stack value, and the speculative execution is stopped. Execution jumps to `*%rax`.

An indirect CALL is more complicated, but uses the same approach, as shown below:

Table 2: Indirect call replacement with retpoline (GNU Assembler syntax)

Before retpoline	`call *%rax`

After retpoline	`1: jmp label2`
	`label0:`
	`2: call label1`
	`capture_ret_spec:`



```
3:        pause ; lfence
4:        jmp capture_ret_spec
  label1:
5:        mov %rax, (%rsp)
6:        ret
  label2:
7:        call label0
8:        … continue execution
```

1. "`1: jmp label2`", jumps to "`7: call label0`".
2. "`7: call label0`" pushes the address of "`8: … continue execution`" on the stack and the RSB, then jumps to:
3. "`2: call label1`" which pushes the address of "`3: pause ; lfence`" on the stack and the RSB, then jumps to:

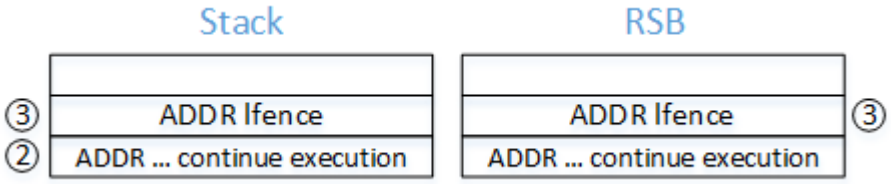


Figure 3: Stack and RSB with retpoline enabled (steps 1 to 3)

4. "`5: mov %rax, (%rsp)`" which takes the target of the indirect call (in `%rax`) and writes it over the return address stored on the stack. At this point the in-memory stack and the RSB differ.
5. "`6: ret`". If speculating, the CPU consumes the RSB entry created in step 3 and jumps to "`3: pause ; lfence`". It is "trapped" in an infinite loop. The speculation barriers section has more details about the importance of this sequence.
6. Eventually, the CPU realizes that the speculative `ret` does not agree with the in-memory stack value, and that speculative execution is stopped. Execution jumps to the target of the indirect call: `*%rax`, which was placed on the stack in step 4.

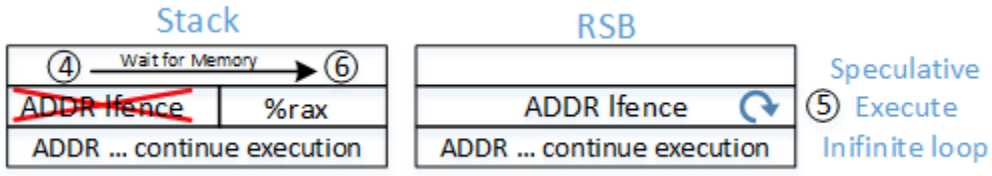


Figure 4: Stack and RSB with retpoline enabled (steps 4 to 6)

7. The target of the indirect call returns, consuming the RSB and in-memory stack entry placed there in step 2.

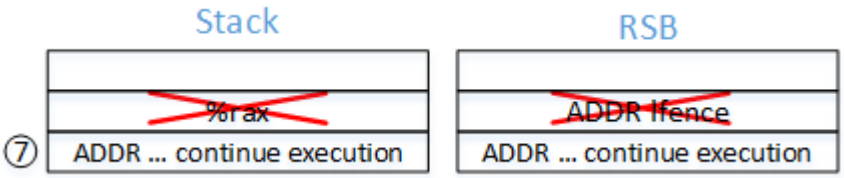


Figure 5: Stack and RSB with retpoline enabled (step 7)

4.1 Deploying Retpoline - Compilers

Since most indirect branches are generated by compilers when building a binary, deploying retpoline requires recompiling the software that needs mitigation. A retpoline-capable compiler can avoid generating *any* vulnerable indirect CALL or indirect JMP instructions and instead uses retpoline sequences. Of course, for code not generated by the compiler (such as inline assembly) programmers must insert retpoline sequences manually.

4.2 Deploying Retpoline – Runtime Patching

One option when deploying retpoline is to have the compiler insert a full retpoline sequence at each indirect branch that needs mitigation. However this makes the code larger than needed, so the preferred option is to have the program itself provide the retpoline sequences in one central place, and then have the compiler refer to these sequences. For example, the program might provide the sequence shown in Table 2 above at a location called `retpoline_target_in_rax`.

Table 3: Example of central retpoline sequence[3]

Before retpoline	`call *%rax`
After retpoline	`call retpoline_target_in_rax`

The program might also provide retpoline sequences for many possible call instruction possibilities, such as for making calls to targets stored in each of the general purpose registers.

This approach provides a more compact instruction sequence at each indirect call site, and also concentrates the retpoline implementations into a controlled set of locations. Programs supporting runtime patching (such as the Linux kernel) can evaluate systems for vulnerability to branch target injection (Spectre variant 2). If

[3] Runtime patching in this manner requires an out-of-line retpoline sequence which differs from the sequence in Table 2.

the system is not vulnerable (such as on systems with older processors, or with future processors implementing enhanced IBRS mitigations), the program-provided retpoline sequence can be replaced with a non-mitigated sequence.

Table 4: Runtime patch example for mitigated CPUs

Mitigated Code	Runtime Patch for Mitigated CPU

`retpoline_target_in_rax:`
`1: jmp label3`
`label0:`
`2: call label1`
`Capture_ret_spec:`
`3: lfence`
`...`
	`retpoline_target_in_rax`
	`1: call *%rax`
	`2: ret`
	
	`// never reached:`
	`3: lfence`
	`...`

4.3 Interaction with Control-flow Enforcement Technology (CET)

Control-flow Enforcement Technology (CET) is a future CPU technology which provides capabilities to defend against Return-Oriented-Programming (ROP) control-flow subversion attacks. However, the retpoline technique closely resembles the approaches used in ROP attacks. If used in conjunction with CET, retpoline might trigger false positives in the CET defenses.

To avoid this conflict, future Intel processors implementing CET will also contain hardware mitigations for branch target injection (Spectre variant 2) (enhanced IBRS), that obviate the need for retpoline. On these processors, runtime patching can be used both to remove conflicts with CET and regain use of the indirect branch predictor for hardened indirect branch speculation.

4.4 Speculation Barriers

The retpoline sequence contains instructions for which there can be performance concerns (LFENCE and PAUSE). Despite this, retpoline can still have attractive performance characteristics.

The architectural specification for LFENCE defines that it does not execute until all prior instructions have completed, and no later instructions begin execution until LFENCE completes. This specification limits the speculative execution that a processor implementation can perform around the LFENCE, possibly impacting processor performance, but also creating a tool with which to mitigate speculative-execution side-channel attacks.

However, this architecturally-defined speculation control behavior is only required when the processor actually executes (retires) the LFENCE. A speculatively-executed

LFENCE that never *actually* executes (retires) may have a smaller performance impact because the speculative behavior is not architecturally-defined. The LFENCE (and other instructions impacting speculation that are part of the *retpoline* construct) is only speculatively executed and thus may not exhibit the same performance impact typically associated with speculation barriers. This allows retpoline to impact speculative execution without the overhead traditionally associated with instructions that directly impact speculation.

5.0 Retpoline Preconditions

5.1 Processor Models

Retpoline is known to be an effective branch target injection (Spectre variant 2) mitigation on Intel processors belonging to family 6 (enumerated by the CPUID instruction) that do not have support for enhanced IBRS. On processors that support enhanced IBRS, it should be used for mitigation instead of, or in combination with, retpoline.

5.2 Empty RSB Mitigation on Skylake-generation

As described in the Retpoline Concept section, the RSB is a fixed-size stack implemented in hardware. As with any stack, it can underflow in certain conditions causing undesirable behavior. "RSB stuffing" is a technique to reduce the likelihood of an underflow from occurring.

The predictable speculative behavior of the RET instruction is the key to retpoline being a robust mitigation. RET has this behavior on all processors which are based on the Intel® microarchitecture codename Broadwell and earlier when updated with the latest microcode. Processors based on the Intel® microarchitecture codename Skylake and its close derivatives have different RSB behavior than other processors when the RSB is empty. Processors with this RSB behavior can be identified using the following DisplayFamily/DisplayModel signatures provided by the CPUID instruction[4]:

Table 5: Processors with different RSB behavior

Family	Model
06H	4EH
06H	5EH
06H	55H
06H	66H
06H	67H
06H	8EH
06H	9EH

[4] Additional processors may exhibit vulnerable RSB behavior that are not listed in this table.

When the RSB "stack" is empty on these processors, a RET instruction may speculate based on the contents of the indirect branch predictor, the structure that retpoline is designed to avoid. The RSB may become empty under the following conditions:

1. Call stacks deeper than the minimum RSB depth (16) may empty the RSB when executing RET instructions. This includes CALL instructions and RET instructions within aborting TSX transactions.
2. IBPB command may empty the RSB.
3. Certain instructions may empty the RSB[5]:
    a. WRMSR to *0x79* (microcode update) , *0x7A* (SGX activation).
    b. WRMSR/RDMSR to/from *0x8C-0x8F* (SGX Launch Enclave Public Key Hash).
    c. SGX instructions (ENCLS, ENCLU) and SGX CPUID leaf.
    d. Imbalance between CALL instructions and RET instructions that leads to more RET instructions than CALL instructions. For example:
        i. OS context switch
        ii. C++ exception
        iii. `longjmp`
4. Entering sleep state of C6 or deeper (for example, MWAIT) may empty the RSB.

There are also a number of events that happen asynchronously from normal program execution that can result in an empty RSB. Software may use "RSB stuffing" sequences whenever these asynchronous events occur:

1. Interrupts/NMIs/traps/aborts/exceptions which increase call depth.
2. System Management Interrupts (SMI) (see BIOS/Firmware Interactions).
3. Host VMEXIT/VMRESUME/VMENTER.
4. Microcode update load (`WRMSR 0x79`) on another logical processor of the same core.

Software may avoid RSB underflow by inserting an "RSB stuffing" sequence following all of the above conditions.

These sequences, with an example of one instance shown below, can be removed using runtime patching techniques in the same way as the retpoline sequences on processors that do not require this mitigation.

Table 6: RSB stuffing

```
void rsb_stuff(void)
{
        asm(".rept 16\n"
            "call 1f\n"
            "pause ; lfence\n"
            "1: \n"
```

[5] RSB stuffing techniques help avoid use of the indirect branch predictor. Even though the IBPB command empties the RSB, it also mitigates exploit content in the indirect branch predictor. RSB manipulation following an IBPB command may not provide additional meaningful mitigation.

```
            ".endr\n"
            "addq $(8 * 16), %rsp\n");
}
```

5.3 Virtual Machine CPU Identification

A valuable tool in modern data centers is live migration of virtual machines (VMs) among a cluster of bare-metal hosts. However, those bare-metal hosts often differ in hardware capabilities. These differences could prevent a virtual machine that started on one host from being migrated to another host that has different capabilities. For instance, a virtual machine using Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instructions could not be live-migrated to an older system without Intel® AVX-512.

A common approach to solving this issue is exposing the oldest processor model with the smallest subset of hardware features to the VM. This addresses the live-migration issue, but results in a new issue: Software using model/family numbers from CPUID can no longer detect when it is running on a newer processor that is vulnerable to exploits of Empty RSB conditions.

To remedy this situation, an operating system running as a VM can query bit 2 of the IA32_ARCH_CAPABILITIES MSR, known as “RSB Alternate” (RSBA). When RSBA is set, it indicates that the VM may run on a processor vulnerable to exploits of Empty RSB conditions regardless of the processor’s DisplayFamily/DisplayModel signature, and that the operating system should deploy appropriate mitigations. Virtual machine managers (VMM) may set RSBA via MSR interception to indicate that a virtual machine might run at some time in the future on a vulnerable processor.

5.4 Recompilation

Mitigation with retpoline requires that all code in a program (or OS kernel) is compiled with a retpoline-enabled compiler in order to make sure vulnerable indirect branches are replaced with the retpoline sequence. In practice, this means that retpoline can only be applied in environments where recompilation and redeployment of updated binaries is possible. This includes instances where full source code is available, or where instructions are generated by a JIT compiler.

However, retpoline is not a practical mitigation for environments where full recompilation itself is not practical. Other mitigations may be appropriate in those environments.

6.0 BIOS/Firmware Interactions

System Management Interrupt (SMI) handlers can leave the RSB in a state that OS code does not expect. In order to avoid RSB underflow on return from SMI, an SMI handler may implement RSB stuffing (for parts identified in Table 5) before returning from System Management Mode (SMM).

7.0 Summary

There are a number of possible mitigation techniques for the branch target injection (Spectre variant 2) exploit. The retpoline mitigation technique presented in this document is resistant to exploitation and has attractive performance properties compared to other mitigations.

8.0 References

- Google* article on retpoline: Retpoline: a software construct for preventing branch-target-injection
- Side Channel Security Issue: Software Support

Appendix A Linux Implementation Details*

A.1 Enabling and Enumerating Retpoline Support

The Linux kernel implements retpoline to protect the kernel from exploits. The `CONFIG_RETPOLINE` build option is used to enable support. You can check for support on many distributions by running the following command:

```
grep CONFIG_RETPOLINE /boot/config-`uname -r`
```

This build option indicates whether retpoline support was requested in the build. However, even with this option set, you can successfully build the kernel even if the compiler does not support retpoline. In this case, the kernel will only contain minimal mitigations with retpoline in assembly code. These kernels will indicate that they are still "Vulnerable" to branch target injection (Spectre variant 2), as shown below:

```
# cat /sys/devices/system/cpu/vulnerabilities/spectre_v2
Vulnerable: Minimal generic ASM retpoline
```

Kernels which were built with a compiler that does support retpoline will indicate that they are mitigated and are no longer vulnerable:

```
# cat /sys/devices/system/cpu/vulnerabilities/spectre_v2
Mitigation: Full generic retpoline
```