Spectre Returns!
Speculation Attacks Using the Return Stack Buffer

Esmaeil Mohammadian Koruyeh
B2B Department
Samsung Research America
Mountain View, CA 94043 USA

Khaled N. Khasawneh
Department of Electrical and Computer Engineering (ECE)
George Mason University
Fairfax, VA 22030 USA

Chengyu Song and Nael Abu-Ghazaleh
Department of Computer Science and Engineering
University of California
Riverside, CA 92521 USA

Editor’s notes:
This article describes a new Spectre-class attack that exploits the return stack buffer and does not rely on the branch predictor unit.
—Gang Qu, University of Maryland, USA

The recent transient execution attacks such as Spectre and Meltdown demonstrated how speculative execution can be exploited to enable the disclosure of secret data across both software and hardware isolation boundaries. Specifically, attackers can misguide the processor to execute a read speculatively instruction with an address under their control. Although the speculatively read values are not visible to programs through the architectural state, since the misspeculation effects are eventually undone, they can be communicated out using a covert channel. In particular, Spectre attacks rely on the attacker manipulating the branch predictors within the processor. The first exploration and disclosure of the attack identified and attacked two predictors: the direction predictor (Spectre v1) and the branch target predictor (Spectre v2). In this article, we identified a new Spectre class that targets the return stack buffer (RSB), a structure within modern processors used to predict the target of return instructions. Specifically, we show how an attacker can manipulate the state of the RSB to cause speculative execution of a payload attack gadget that reads and exposes sensitive information from kernel space or otherwise protected data.

Background
This section overviews some background: branch predictor structures in modern processors and Spectre attacks.
Branch prediction

Branch prediction is a critical component of modern processors that support speculative out-of-order execution. When a control flow instruction (branch, call, or return) is encountered, the result of the instruction (e.g., whether or not a conditional branch will be taken or what the target value is of an indirect branch or a return) is generally not known at the front end of the pipeline. As a result, to continue to fill the pipeline and utilize the available resources of the processor, branch prediction is used.

Modern processors employ sophisticated predictors which typically consist of three components.

- **Direction predictor** is responsible for predicting the direction of a conditional branch. Although a number of implementations have been studied, modern predictors typically implement a two-level context-sensitive predictor. The first level is a simple predictor that hashes each branch address to a direction predictor (typically a 2-bit saturating counter). This predictor is used either when a branch is not being successfully predicted or when the predictor has not been trained yet. When the predictor is trained, it typically uses a second prediction algorithm, often a variant of a gshare predictor, which uses the global history of a branch in addition to its address to hash to a direction predictor as before. The advantage is that the same branch can have different predictions based on the control flow path used to reach it.

- **Target predictor** is used by indirect jump and indirect call instructions which jump to an address held in a register or a memory location, which is unknown at the front end of the pipeline. This predictor typically uses the hash of the branch address to index a cache holding the branch targets called the branch target buffer (BTB). BTBs are shared across threads on a virtual core: one value used by a process could be used by another process whose branch has a matching address in the BTB.

- **Return address stack**: Since returns are not well predicted using the BTB, and often follow strict call-return semantics, their target is predicted using a return address stack of fixed size. When a call instruction executes, the return address is pushed on this hardware stack; if overflow happens, previous entries are overwritten [10]. When a return is encountered, the top of the stack is popped and used as the return target.

Speculation attacks

Spectre attacks have recently shown that the speculation behavior of modern processors can be exploited. In general, these attacks exploit four properties.

- **P1**: branch prediction validation happens deep in the central processing unit (CPU) pipeline. As a result, speculative instructions near the branch can access unprivileged memory locations.

- **P2**: speculative instructions leave side effects in microarchitectural structures such as caches, which can be inferred using well-known timing side-channel attacks like Flush + Reload and Prime + Probe.

- **P3**: the branch predictor can be mistrained (Spectre 1) or directly polluted (Spectre 2). It is shared across all programs running on the same physical core [9], [5], allowing code running in one privilege domain to manipulate branch prediction in another domain (e.g., kernel, VM, hypervisor, another process, or software guard extensions (SGXs) enclave). Our attacks replace this step with speculation control through the RSB.

Spectre (Variant 1) is presented in Figure 1. In this code, a victim process reads values from array 1 using the offset provided by the attacker. Then, the resulting value is used to perform an access into array 2. As we discussed above, accesses into array 2 can be used by the attacker to deduce the value of the index. The index, in turn, is controlled by the attacker since the attacker controls the offset. Therefore, the attacker can use a carefully selected value of offset to read arbitrary memory address which then will result in cache access observable by the attacker. However, the if statement ensures there are no out-of-bounds memory accesses allowed. Unfortunately, the attacker can exploit speculative execution and behavior of the branch predictor to force the victim process to perform an out-of-bounds memory access in the following way.

```c
if (offset < array1_size) {
    y = array2[array1[offset] * 64];
}
```

**Figure 1. Spectre attack variant 1.**
a. The attacker mistrains the branch predictor by executing the code several times with the value of the offset such that the if statement is true (branch instruction not taken).

b. Next, to make the speculative window larger, the attacker evicts array1_size from the cache, so that the CPU has to load the value from memory. Since the speculation result will not be resolved until this value arrives, forcing it to come from memory expands the size of the speculation window to allow more elaborate speculative gadgets to be executed.

c. Finally, the attacker chooses the malicious offset such that it is larger than array1_size. The trained branch predictor unit predicts the branch not taken so that the CPU executes two memory accesses speculatively and discloses the secret value through the cache side channel.

Unlike Spectre variants, Meltdown does not rely on using misspeculation. Since an exception eventually will be raised, this attack requires the ability to tolerate and recover from the raised exception.

Attack principles: Reverse engineering the return stack buffer

In this section, we explain the operation of the RSB, which is the microprocessor structure our attacks exploit to implement speculation attacks that bypass all existing defenses. On modern processors, sophisticated branch predictors are used to predict the direction and target of conditional and indirect branches and calls. Return instructions challenge such predictors because the return address depends on the call location from which a function is invoked, which for many functions that are called from different locations of a program can lead to poor branch predictor performance. For example, consider a function such as printf() which may be called from many different locations of a program. Relying on the previous history of where it returned to can lead to very low prediction performance through the branch predictor. We verify each of these mechanisms on two Intel processors (a Haswell and a Skylake).

RSB overview

To overcome this problem, the return address is predicted using the RSB as follows. The RSB is a hardware stack buffer where the processor pushes the return addresses every time a call instruction is executed and uses that as a return target prediction when the matching return is encountered. Figure 2a shows an example of the state of the RSB after two function calls (F1 and F2) have been executed. The figure also shows the state of the software stack for the program where the stack frame information and the return address of the function are stored. Figure 2b shows how the values on these stacks are used when the return instruction from function F2 is executed. At this point, the return address from the fast shadow stack is used to speculate about the
return address location quickly. The instructions executed at this point are considered speculative. Meanwhile, the return address is fetched from the software stack as part of the teardown of the function frame. The return address is potentially in the main memory (not cached) and is received several hundred cycles later. Once the return address from the software stack is resolved, the result of the speculation is determined: if it matches the value from the RSB, the speculated instructions can be committed. If it does not, then a misspeculation has occurred and the speculatively executed instructions must be squashed. This behavior is similar to speculation through the branch predictor, except it is triggered by return instructions. Note that the misspeculation window could be substantially larger since the return could be issued out of order, and other dependencies have to be resolved before it is committed.

RSB sources of misspeculation

The RSB mis speculate s when the return address value in the RSB does not match the return address value in the software stack, leading the program to mis speculate to the address in the RSB. If this mis speculation can be triggered intentionally by an attacker, spectre-like attacks become possible through the RSB. Thus, in this subsection, we explain the sources of misspeculation through the RSB and discuss whether they provide a vector for attackers to trigger speculation attacks. We label these sources as S1–S4 to be able to refer to them in the attack descriptions.

**S1: Overfill or underfill of the RSB due to limited structure size:**

The RSB structure is typically sized to match common nesting depths of call stacks in programs. On low-end machines, the RSB can be as shallow as four entries in size. More typically, on desktops, it is in the range of 16 entries, and for server-class processors, it can be larger (e.g., 24 entries on the AMD Ryzen). As illustrated in Figure 3, when the RSB overfills, it typically overwrites the older entries in the stack. Eventually, when the stack is unrolled as the nested calls return, we reach the function whose value has been overwritten causing an underfill of the stack (in Figure 3, the entry for F1 got overwritten).

In an underfill, there is no value available on the RSB to guide speculation. Different CPUs handle this situation differently. For example, the Intel CPUs that we checked switch over to the branch predictor if the RSB is empty, which can be used to trigger attacks through the branch predictor. However, AMD appears not to follow this strategy.

**S2: Direct pollution of the RSB:**

This is the primary vector that we use in our proof of concept attacks. Call instructions implicitly push a return address to the RSB and the software stack. However, an attacker can then replace the address on the software stack (by writing directly to that location), or just remove it altogether (as shown in Figure 4a). In this case, the value in the RSB remains and does not match a value on the software stack causing mis speculation when a return is executed (as shown in Figure 4b). By controlling the call address, the attacker can control the misspeculation address. It is also possible to convert a call instruction into a push and jump, in which case a return value exists on the software stack that is not matched by a value in the RSB. A return could also be replaced by a pop and a jump, causing a value to remain in the RSB that has been removed from the software stack.

**S3: Speculative pollution of the RSB:**

Speculatively executed calls push a value on the stack, although the details are specific to the architecture. Once mis speculation is discovered, the call is squashed but the speculatively pushed return address remains on the RSB. This provides the opportunity for a malicious attacker to push a return address that is outside the address space accessible by the program (e.g., a kernel address) without raising an exception or having to handle the side effects of a call.
S4: RSB use across execution contexts: on a context switch, the RSB values left over from an executing thread are reused by the next thread. Once we switch to a new thread, if the thread executes a return, then it will missspeculate to an address provided by the original thread. The same is true with a switch over to the operating system (provided RSB refilling is not implemented), or to an SGX context.

SpectreRSB attack
We investigate different vectors of SpectreRSB showing first an attack that works within the same address space. We also demonstrate attacks across different threads and also the operating system (OS) kernel.

Basic attack example
In this section, we illustrate the attack principles by showing a basic speculation attack launched from a process to part of its address space that it cannot directly access (similar to Spectre variant 1). This attack represents the simplest instance of SpectreRSB and therefore we use it to explain the attack in detail. It is unlikely to be practical: it is difficult to implement the gadget to manipulate the stack using high-level sandboxing primitives to allow the attack to break sandbox boundaries. On an unpatched machine, this attack enables the attacker to read kernel memory via the Meltdown bug. However, kernel page-table isolation (KPTI) prevents using it to allow the user code to read kernel data. We note that this attack does not rely on any speculation through branches or the branch predictor. For this reason, the attack bypasses defenses that focus only on securing speculation through the branch predictor.

Figure 5 presents an overview of a basic SpectreRSB attack. The attack starts at line 20 with the call to speculative, with an argument which is the memory address of the sensitive data to be read. Speculative calls gadget, which serves two purposes: 1) the return address is pushed to the RSB (the return address is to line 15 where we have the payload gadget to be executed speculatively); and 2) we jump to the (inline assembly) function gadget which will manipulate the software stack to create the mismatch between the RSB and the software stack. In this case, the gadget cleans up the effects of the function call to itself, popping off the frame including the return address.

At this point, before the return, the stack state is consistent with a return from speculative back to main. However, the RSB holds a return value from gadget to speculative. Thus, in line 10 when the return executes, the CPU speculatively executes at line 15. The flush of the top of the stack (line 10) ensures that the true value of the return address will be fetched from memory rather than from the caches.
creating a large speculation window. Note that the speculation window is based on the return. Speculative execution at line 15 reads the secret which can be any mapped address even if inaccessible to the user process during normal execution and then communicates it out through the flush reload cache side channel by accessing a data-dependent index in the array (line 16). Finally, the real return value is obtained, and the misspeculation is squashed, returning us to line 21, where we probe the cache to identify which data-dependent cache set was accessed to expose the value of the secret.

Attacks across different threads

In this section, we investigate different vectors of SpectreRSB which exploit S4 (RSB use across execution context) to pollute the RSB. These attacks potentially allow an attacker to attack another process (similar to Spectre V2), perhaps even across virtual machines (VMs), making the attack dangerous on the cloud. In general, these attacks require a machine not implementing RSB refilling (pre-Skylake, or Xeon, for example), to make sure that a context switch does not overwrite the polluted addresses from the RSB (Figure 6).

The attacker establishes co-location with the victim on the same core similar to Spectre 2. The attack pattern proceeds as follows: 1) after a context switch to the attacker, s/he flushes shared address entries (for flush reload). The attacker also pollutes the RSB with the target address of a payload gadget in the victim’s address space; 2) the attacker yields the CPU to the victim; and 3) the victim eventually executes a return, causing speculative execution at the address on the RSB that was injected by the attacker. Steps 4 and 5 switch back to the attacker to measure the leakage.

1. Attack 2a: Attack across two colluding threads: In this attack, the attacker and the victim are two colluding threads following the steps in Figure 6. In the first attack, we let the two threads synchronize using futex operations to control their interleaving. The RSB pollution happens in the first thread which also flushes the top of the stack of the second thread, while the return happens in the second. The attack succeeded, proving that SpectreRSB works from one thread to another. However, since the return is in user mode, we cannot read kernel data. For the attack to be useful, we should launch an attack such that the victim colluding thread returns while in the kernel (enabling us to read kernel data while its memory is mapped).

2. Attack 2b: Attack with two colluding threads with return from inside kernel: Next, we wanted to see if we could use this attack to cause a return while the victim thread is in the kernel mode in step 3. To ensure this, we have the colluding victim execute a blocking system call, which typically has them deep inside a call stack in the kernel before blocking. The attacker after polluting the RSB, waits for the victim to unblock, perhaps even triggering the event that unblocks it. At this point, the victim continues execution inside the kernel and recurses back out of its call stack, with one or more returns, triggering the vulnerability. This attack requires a machine without supervisor mode execution prevention (SMEP) enabled. We demonstrated the attack with SMEP disabled.

Triggering an unmatched return: the RSB assumes that strictly paired call-return behavior. In attacks that cross execution boundaries, the attacker pollutes the RSB, but would then like to trigger a return in the victim process code (or OS/SCX code) to which they have no access. However, if the attacker manages to catch the victim inside of a function call, then when the victim executes again, it will encounter an unmatched return. This could rely on timing or a blocking call inside of a function that will cause the scheduler to unschedule the victim. In this proof-of-concept attack, we placed an unmatched return
directly in the enclave, but we expect to be able to do that using the strategies above for other enclaves.

**Attack 3: From user to kernel**

In this section, we briefly discuss the possibility of another attack where user code pollutes the RSB and then triggers an unmatched return in the kernel (we call this attack 4). This attack is likely to be difficult, if not impossible, so we describe it only for completeness. The main insight is that a return from the kernel to a polluted address in the RSB will cause speculation while in kernel mode. This means that the kernel address space is still mapped, allowing us to read from the kernel. This attack assumes the following ingredients: 1) that the RSB is shared between the user and the kernel: we find that this is the case on two Intel processors; 2) we need to be able to trigger an unmatched return in the kernel. Although some programming constructs such as tail recursion, continuations, setjmp/longjmp, and others can break call-return semantics, we have not attempted to find such unmatched returns in the kernel; and 3) we need to figure out the stack address of the kernel, and evict it from the cache. This last step is necessary to make sure that the speculation window is sufficiently large to execute a useful gadget speculatively (without this, we can only execute a gadget a few instructions long speculatively). Luckily, the mapping between the stack kernel address and the physical address is deterministic in Linux on x86-64 (it uses the Physmap address directly instead of double mapping it). This makes deriving the conflict set straightforward once we identify the kernel stack address.

We explore a proof-of-concept attack with an unmatched return in a kernel module that we build. Later, we discuss concrete possibilities for how to make this happen with multiple threads. The attack is shown in Figure 7 and works only on an unpatched machine or a machine not implementing RSB refilling. After polluting the RSB in steps 1–3, and flushing the top of the kernel stack in step 4, before issuing a system call to our kernel module with the unmatched return. The mismatched return triggers a misspeculation to step 7 to execute in supervisor mode. This attack does not work on patched Skylake+ processors due to RSB refilling but works on the Xeon machine. We also discover that SMEP checks are not speculatively bypassed since the speculative program counter is known at the time of speculation. Thus, the attack as shown requires SMEP to be disabled to enable the kernel to return to user code. An alternative strategy to bypass this limitation is to try to use the return address of the gadget in PhysMap (as discussed under S3 in Section 3), but most Linux distributions disable the execution of PhysMap addresses. We only demonstrated the attack with SMEP disabled. We also assumed that we knew the address of the kernel stack pointer to flush it in step 4.

### Research impact

**Impact of the paper on the industry**

SpectreRSB attack along with other transient execution attacks show that secure design principles have been overlooked in performance optimization techniques in modern processors and need more attention to make the processor secure.

In the article, we showed that state-of-the-art defenses against other variants of specter class attacks including Google’s Retpoline and Intel’s microcode patches do not stop SpectreRSB. So that it received a common vulnerabilities and exposures (CUEs) string (CVE-2018-15572) from the National Institute of Standards and Technology (NIST) vulnerability database. The article resulted in a patch to fix this vulnerability in the Linux mainline kernel distribution to always unconditionally fill RSB on context switches based on the recommendation of our article; the patch prevents some of the variants of SpectreRSB.
Our results were responsibly reported to the Intel security team since we successfully tested our attacks on Intel’s CPUs. Although we did not demonstrate attacks on AMD or ARM processors, they also use RSBs to predict return addresses. Therefore, we also reported our results to AMD and ARM. A few months after publishing our article, Canella et al. [4] showed that SpectreRSB attack is also possible on AMD and ARM processors.

Impact on the research community

SpectreRSB helps the research community have a better understanding of transient attacks in general. Also, SpectreRSB has been used as a base for some other attacks, as an example [1] has demonstrated that the SpectreRSB and SpectreBTB can be used to bypass memory corruption defenses like stack protectors and control flow integrity (CFI). In another study [2], SpectreRSB was used to enable the speculative port-contention side-channel attack (SMoTher-Spectre). In [8], SpectreRSB has been used to exploit Meltdown efficiently. In [3], researchers implemented the SpectreRSB attack on BOOM processors.

Also, there are many proposals for detecting and mitigating the SpectreRSB. Inspectre [6] proposed a formal analysis to check the existence of the spectre attack and specifically the SpectreRSB attack. Ghostbusting [7] proposed a mitigation technique for spectreRSB based on intraprocess memory isolation. Specfuzz [12] proposed dynamic testing for SpectreRSB as well as other speculative execution vulnerabilities. SPECCFI [11] proposed a complete hardware design to mitigate this attack.

In this article, we introduced a new type of speculation attack (SpectreRSB) that is triggered by the RSB, rather than the branch predictor unit. The RSB is used to predict the address of return instructions. We demonstrated a number of vectors that allow an attacker to cause RSB mispeculation. Using these techniques, we construct a number of attack vectors including attacks within the same process, attacks on SGX enclaves, attacks on the kernel, and attacks across different threads and processes. SpectreRSB bypasses all published defenses against Spectre, making it a highly dangerous vulnerability.

**Interestingly, there is a** patch that was proposed to protect against the behavior of Intel Core i7 Skylake generation and newer processors called RSB refilling. RSB refilling interferes with SpectreRSB attacks that experience at least one mode switch from user to kernel. We recommend that this patch should be deployed immediately across all processor generations (and not just Skylake+). In the long run, we believe that these patches are ad hoc and that new attack vectors will continue to emerge. Current systems are fundamentally insecure unless speculation is disabled. However, we believe that it is possible to design future generations of CPUs that retain speculation, but also close speculative leakage channels, for example, by keeping speculative data in separate CPU structures than committed data.

**References**

[1] A. Mambretti et al., “Bypassing memory safety mechanisms through speculative control flow hijacks,” 2020, arXiv:2003.05503.

[2] A. Bhattacharyya et al., “SMoTherSpectre: Exploiting speculative execution through port contention,” 2019, arXiv:1903.01843.

[3] C. Canella, K. N. Khasawneh, and D. Gruss, “The evolution of transient-execution attacks,” in Proc. Great Lakes Symp. VLSI, Sep. 2020, pp. 163–168.

[4] C. Canella et al., “A systematic evaluation of transient execution attacks and defenses,” in Proc. USENIX Secur. Symp., 2019, pp. 249–266.

[5] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “Jump over ASLR: Attacking branch predictors to bypass ASLR,” in Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), Oct. 2016, pp. 1–3.

[6] R. Guanciale, M. Balliu, and M. Dam, “InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., Oct. 2020, pp. 1853–1869.

[7] I. R. Jenkins et al., “Ghostbusting: Mitigating spectre with intraprocess memory isolation,” in Proc. 7th Symp. Hot Topics Sci. Secur., Sep. 2020, pp. 1–11.

[8] T. Kim and Y. Shin, “Reinforcing meltdown attack by using a return stack buffer,” IEEE Access, vol. 7, pp. 186065–186077, 2019.

[9] P. Kocher et al., “Spectre attacks: Exploiting speculative execution,” 2018, arXiv:1801.01203.

[10] E. Koruyeh et al., “Spectre returns! Speculation attacks using the return stack buffer,” in Proc. USENIX Workshop Offensive Technol. (WOOT), 2018, pp. 1–12.

[11] E. M. Koruyeh et al., “SpecCFI: Mitigating spectre attacks using CFI informed speculation,” in Proc. IEEE Symp. Secur. Privacy (SP), May 2020, pp. 39–53.
Direct questions and comments about this article to Esmaeil Mohammadian Koruyeh, B2B Department, Samsung Research America, Mountain View, CA 94043 USA; Emoha004@ucr.edu.

[12] O. Oleksenko et al., “SpecFuzz: Bringing spectre-type vulnerabilities to the surface,” in Proc. 29th USENIX Secur. Symp. (USENIX Security), 2020, pp. 1481–1498.

Esmaeil Mohammadian Koruyeh is a security researcher at Samsung Research America (SRA), Mountain View, CA 94043 USA. His research interests include computer architecture support for security, microarchitecture security, and system security. Mohammadian Koruyeh has a PhD from the University of California, Riverside, CA, USA. He is a Member of IEEE.

Khaled N. Khasawneh is an assistant professor in the Department of Electrical and Computer Engineering, George Mason University, Fairfax, VA 22030 USA. His research spans computer architecture, security, and machine learning. His current research focuses on hardware support for security, microarchitecture security, machine-learning security and privacy, and malware detection. He is a Member of IEEE.

Chengyu Song is an associate professor in the Department of Computer Science and Engineering, University of California, Riverside, CA 92521 USA. His research interests include system security, program analysis and verification, and operating systems. Song has a PhD in computer science from Georgia Institute of Technology, Atlanta, GA, USA. He is a Member of IEEE.

Nael Abu-Ghazaleh is a professor in the Department of Computer Science and Engineering and the Department of Electrical and Computer Engineering, University of California, Riverside, CA 92521 USA. His research interests include computer architecture support for security, parallel discrete event simulation, and networking and mobile computing. Abu-Ghazaleh has a PhD from the University of Cincinnati, Cincinnati, OH, USA. He is a Senior Member of IEEE.