Google

Retpoline: a software construct for preventing branch-target-injection

Author: Paul Turner, Senior Staff Engineer, Technical Infrastructure

At Google, we have been researching mitigation strategies for the new class of exploits discovered by the Project Zero team affecting speculative execution. We wanted to share a binary modification technique that we have developed for protecting against “Branch target injection”, also referred to as “Spectre”. It is predicated on the fact that many CPUs implement a separate predictor for function returns. When available, this predictor is used with high priority, allowing for the construction of an indirect branch which is safe from speculation-based attacks.

Note: While some of the specific details and examples below are x86 specific, the ideas underlying the construction are commonly applicable.

Executive Summary

“Retpoline” sequences are a software construct which allow indirect branches to be isolated from speculative execution. This may be applied to protect sensitive binaries (such as operating system or hypervisor implementations) from branch target injection attacks against their indirect branches.

The name “retpoline” is a portmanteau of “return” and “trampoline.” It is a trampoline construct constructed using return operations which also figuratively ensures that any associated speculative execution will “bounce” endlessly.

(If it brings you any amusement: imagine speculative execution as an overly energetic 7-year old that we must now build a warehouse of trampolines around.)

Background

Example: A common C++ indirect branch

```
class Base {
 public:
  virtual void Foo() = 0;
};

class Derived : public Base {
 public:
  void Foo() override { … }
};

Base* obj = new Derived;
obj->Foo();
```

Recall that an indirect branch is one for which the target must be determined at run-time; common examples being polymorphic code or a jump-table. In the example above, when the virtual method `Foo()` is invoked, dynamic lookup must be performed to determine the location of the matching implementation. In this case, rather than stalling speculative execution until this target can be determined, the hardware may try to *guess*.

The strategies used to make this prediction vary between hardware implementations, they are commonly not isolated between security domains to reduce complexity and improve performance. While this has been previously exploited by probing the state of these predictors to infer the layout of another domain[1], there were not previously known observable **data side-effects**.

Now that a data side-effect has been found, it is possible to adversarially bias the speculative execution of `Foo()` so that a gadget such as Variant 1’s “Bounds check bypass” is instead temporarily speculatively executed.

Importantly, this steering may occur:

- Between user and kernel execution on the same CPU
- Between processes on the same CPU
- Between guests and their hypervisors
- Between execution on SMT or CPU siblings (prediction hardware may be shared)

While this does present a gloomy picture, there are two significant hurdles an attacker must overcome:

1. The gadget which victim indirect execution is steered towards, must be part of the victim's address space. This means that detailed knowledge of the victim binary and its current address space layout is required.
2. There must **also** also exist a channel on which side-effects of our gadget above can be observed. This is much more challenging in the absence of an attacker-accessible shared memory mapping with the victim. (See the appendix for some further details here.)

Despite the criticality of this attack, the baseline difficulty to construct an attack is reasonably high due to the degree of tailoring required. This complexity is greatly further increased if either of these conditions is not satisfied. While this is beneficial in that we do not need to (strongly) worry for most binaries, there are several classes of software for which these preconditions are more readily satisfied. The primary example being the host operating system: here, even if the exact operating system version is not known, there will always exist large commonalities which may be reasonably targeted (e.g. Between Linux or Windows versions). Further, in this example, application's own address space represents a shared channel with the hosting operating system that may be used to clear the second hurdle. (It should be noted that hardware protection techniques such as SMAP may be potentially bypassed in this case as the operating system's direct map can be used as an alias for triggering cache presence.)

Unfortunately, it is not practical or reasonable to avoid indirect branches in the construction of this software. This means that we need an efficient method of constructing an indirect branch that is not subject to external manipulation in its retirement.

(Un-)Directing Speculative Execution

The crux of our problem, is that if we want to transfer control to a runtime-target we must ultimately retire *some* form of indirect branch.

Example: A compiled x86 indirect branch

```
jmp *%rax; /* indirect branch to the target referenced by %rax */
```

While strategies such as serialization may allow us to potentially reduce the windows of execution for which %rax is not yet loaded (e.g. consider prior unretired load), the speculative execution here is a property of the hardware itself. There is no way to directly instruct the CPU that we want to transfer control to the address contained within %rax, but that its speculative execution may make no guesses about what that may be. Even if it is immediately resolvable.

This necessitates an alternate approach: While we can't prevent speculative execution in software, what if we could instead control it? Just as a potential attacker may be using branch-target-injection to manipulate the hardware prediction logic, what if we could perform our own injection that we could guarantee would be chosen over any potential external manipulation?

Recall that function return is itself an indirect branch. However, unlike other indirect branches, its target may be directly cached for exact future prediction at the point of function call. It's also useful that lookup may be be implemented as a simple stack around calls and returns. As this is low cost, high accuracy, and high frequency, this is a predictor that is extremely commonly implemented in some form. Specific examples of this include the *return stack buffer (RSB)* on Intel CPUs, the *return address stack (RAS)* on AMD CPUs, and the *return stack* on ARM.

We provide exact constructions for x86-type architectures below; however, there are no key dependencies beyond the core idea of using return prediction to control the path that speculative execution may take.

Construction (x86)

On x86 architectures, function call and return is implemented using the `call` and `ret` instructions. `call` accepts a target (which may be direct or indirect) and branches execution to that target, while `ret` returns (to the instruction following) the last `call`. (This is implemented by call pushing the return target to the stack, prior to branching.) The key property here is that the hardware's *cache* for the return target (the RSB) and the *actual destination* which is maintained on stack are distinct. The RSB entry is a hardware detail and invisible to the underlying application. However, we may manipulate its generation to **control** speculative execution while modifying the visible, on-stack value to direct how the branch is actually retired.

Now we reach the actual construction. Let us first consider an indirect branch to `*%r11`. We will subsequently use this as a building block in constructing an indirect call.

Indirect branch construction

`jmp *%r11`	`  call set_up_target;  (1)`
	`capture_spec:          (4)`
	`  pause;`
	`  jmp capture_spec;`
	`set_up_target:`
	`  mov %r11, (%rsp);    (2)`
	`  ret;                 (3)`

In execution order, we:

1. Make a direct `call` to `set_up_target`; this is known at compile time and will not trigger speculative target resolution. This generates distinct stack and RSB entries with a return target of `capture_spec`.
2. Modify the on-stack entry generated by (1), to instead direct to `%r11`. Note that this does not affect the RSB entry generated above, which will still target `capture_spec`.
3. Return against our original `call`
   a. Speculative execution consumes the RSB entry generated by (1), and is captured within the pause loop at (4). These instructions are only executed by the speculative path.
   b. Our return is ultimately retired, the on-stack value is used to locate the actual new instruction pointer and the benign results of any speculative execution in the loop at (4) are discarded

Importantly, in the execution above there was no point at which speculative execution could be controlled by an external attacker, while accomplishing our goal of branching to an indirect target.

Indirect call construction	
`call *%r11`	`  jmp set_up_return;`
	`inner_indirect_branch:`
	`  call set_up_target; }`
	`capture_spec:         }`
	`  pause;              }`
	`  jmp capture_spec;   } Indirect branch`
	`set_up_target:        } sequence.`
	`  mov %r11, (%rsp);   }`
	`  ret;                }`
	`set_up_return:`
	`  call inner_indirect_branch; (1)`

Here we use two `call`s. Unlike an indirect branch, for the `call` case, our target needs to be able to eventually return control. The outer `call` sets up the return frame that will be used for this, while the inner uses the indirect branch construction above to perform the actual control transfer. This has the particularly nice property that the RSB entry and on-stack target installed by (1) is both valid and used. This allows the return to be correctly predicted so that our simulated indirect jump is the only introduced overhead.

Out-of-line construction (x86)

The construction above can be used as an in-place replacement for any indirect branch. It is compatible with shared (or otherwise relocated code) as the new sequence is position independent. With the coordination of shared trampoline sequences, this may be improved so that indirect calls do not need to individually duplicate the construction above. The challenge here is that we need a vector to communicate our indirect target to the trampoline, as its load must occur after we have transferred control. This can be accomplished using a "per-target" trampoline, effectively encoding the destination into entry point.

Out of line construction	
`jmp *%r11`	`jmp retpoline_r11_trampoline;`
`call *%r11`	`call retpoline_r11_trampoline;`
Shared Trampoline	`retpoline_r11_trampoline:`
	`  call set_up_target;`
	`capture_spec:`
	`  pause;`
	`  jmp capture_spec;`
	`set_up_target:`
	`  mov %r11, (%rsp);`
	`  ret;`

The sequence above can be duplicated to also support offset encoding, e.g., `retpoline_60_rcx_trampoline`, which loads `0x60(%rcx)`. Alternatively, a compiler may to use only a single sequence, materializing the target within a consistent target (e.g. known register which is always used) before invoking the external trampoline.

Correctness Details

Return stack underflow

Function return is itself an indirect branch. Specifically here, we do not refer to the return gadgets constructed above, but rather the natural function returns which will still exist protected application. While the transformation above may be used to protect other indirect branches, we must also guarantee that returns do not lead to speculation.

For example, on x86 the RSB[2] or RAS[3] define fixed capacities. Behavior with exhausted return stack predictors is not well-specified, which means we must potentially avoid underflow. For example, in the case that the hardware chose to instead turn to another predictor. To prevent this, in some cases it may be necessary to "refill" the return stack to guarantee that underflow cannot occur. (See the appendix for an example.)

Cases which this applies to include:

- When we transfer control into protected execution (so that we do not perturb the steps that it may have taken to preserve the integrity of their hardware return prediction).
  - Guest to hypervisor transitions, context-switches into a protected process, interrupt delivery (and return).
- When we resume from from a hardware sleep state which may not have preserved this cache (e.g., `mwait`).
- When natural execution potentially exhausts the return stack in a protected application. (Note that this is a particular corner case, with more limited exploitability -- we expect that most binaries deploying retpoline protections will not require this specific mitigation.)

Binaries with shared linkage

While our initial focus has been the protection of operating system and hypervisor-type targets, there are classes of user application for which this coverage is valuable. In these cases, it should be called out that shared linkage and runtimes will lead to frequent additional interactions with indirect branches. Ubiquitous examples include the Program Link Table (PLT) and dynamically loaded standard libraries. We will be publishing additional optimization notes and techniques for these cases.

Performance Details

Overhead

Naturally, protecting an indirect branch means that no prediction can occur. This is intentional, as we are "isolating" the prediction above to prevent its abuse. Microbenchmarking on Intel x86 architectures shows that our converted sequences are within cycles of a native indirect branch (with branch prediction hardware explicitly disabled).

For optimizing the performance of high-performance binaries, a common existing technique is providing manual direct branch hints. I.e., Comparing an indirect target with its known likely target and instead using a direct branch when a match is found.

Example of an indirect jump with manual prediction

```
cmp %r11, known_indirect_target
jne retpoline_r11_trampoline
jmp known_indirect_target
```

One example of an existing implementation for this type of transform is profile guided optimization, which uses run-time information to emit equivalent direct branch hints.

Pause usage in support loop

The pause instructions in our speculative loops above are not required for correctness. But it does mean that non-productive speculative execution occupies less functional units on the processor.

Alignment

All of the sequences above benefit from having their internal targets aligned to the architecture's preference. (On x86 this is 16-bytes). See the appendix for example sequences, annotated with alignment improvements.

Available Implementations

An implementation for LLVM is is under review for official merge here.

An implementation for GCC is available here.

Terms

Google makes the constructions and sequences described in this article ("retpoline code") available to you under a perpetual, worldwide, nonexclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare

Appendix

Return stack refill (x86)

When refilling the return stack, there are two core requirements we must respect.

1. To guarantee an RSB entry, the calls we use should be of non-zero displacement. This is because of constructs such as `call &next_instruction`, commonly used to determine the current instruction pointer by relocatable code. These are commonly optimized out and ignored by return stack prediction.
2. We must guarantee that if our entry is consumed (due to underflow), it will itself be safe from speculation. These use an equivalent trap to our construction above to ensure safety.

Our refill construction is then to emit return-stack generating `call`s, then reset the stack, without actually unwinding our refilling calls. This ensures that new RSB entries exist, without perturbing the program's control flow.

Example x86 refill sequence

```
   mov $8, %rax;
    .align 16;
 3: call 4f;
3p: pause; call 3p;
    .align 16;
 4: call 5f;
4p: pause; call 4p;
    .align 16;
 5: dec %rax;
    jnz 3b;
    add $(16*8), %rsp;
```

This implementation uses 8 loops, with 2 `call`s per iteration. This is marginally faster than a single call per iteration. We did not observe useful benefit (particularly relative to text size) from further unrolling. This may also be usefully split into smaller (e.g. 4 or 8 `call`) segments where we can usefully pipeline with other operations.

Example construction with alignment prefixes

Example constructions including alignment prefixes

`call *%r11`	
	```
	  jmp set_up_return;
	.align 16;
	inner_indirect_branch:
	  call set_up_target;
	capture_spec:
	  pause;
	  jmp capture_spec;
	.align 16;
	set_up_target:
	  mov %r11, (%rsp);
	  Ret
	.align 16;
	set_up_return:
	  call inner_indirect_branch;
	```

Shared Trampoline	
	```
	.align 16;
	retpoline_r11_trampoline:
	  call set_up_target;
	capture_spec:
	  pause;
	  jmp capture_spec;
	.align 16;
	set_up_target:
	  mov %r11, (%rsp);
	  ret;
	```

(In these examples we do not benefit from aligning the `capture_spec` branch target, as it is only potentially speculatively executed.)

Footnotes

[1] http://www.cs.ucr.edu/~nael/pubs/micro16.pdf

[2] Intel® 64 and IA-32 Architectures Optimization Reference Manual -- 2.4.2.1.

[3] AMD® Software Optimization Guide -- 2.7.1.6

Was this article helpful? Yes No