Two methods for exploiting speculative control flow hijacks

Andrea Mambretti[1,2], Alexandra Sandulescu[2], Matthias Neugschwandtner[2],
Alessandro Sorniotti[2] and Anil Kurmus[2]

[1]Northeastern University, Boston, USA
[2]IBM Research, Zurich, Switzerland
mbr@ccs.neu.edu
{asa, aso, kur}@zurich.ibm.com
mneug@iseclab.org

Abstract

Touted as the buffer overflows of the age, Spectre and Meltdown have created significant interest around microarchitectural vulnerabilities and have been instrumental for the discovery of new classes of attacks. Yet, to-date, real-world exploits are rare since they often either require gadgets that are difficult to locate, or they require the ability of the attacker to inject code. In this work, we uncover two new classes of gadgets with very few restrictions on their structure, making them suitable for real-world exploitation. We demonstrate – through PoCs – their suitability to leak one bit and one byte respectively per successful attack, achieving high success rates and low noise on the constructed side-channel. We test our attack PoC on various kernels with default mitigations enabled, showing how they are insufficient to protect against them. We also show that hardening the configuration of mitigations successfully prevents exploitation, making a case for their wider adoption.

1 Introduction

Spectre [8] and Meltdown [11] have demonstrated that design-level CPU vulnerabilities exist, and have opened the floodgates to microarchitectural attack research. Yet, the space of possible attacks and their variants has not yet been thoroughly explored and understood.

A suitable historical parallel can be drawn with memory corruption attacks: it took decades of research after the seminal work around buffer overflows to thoroughly understand the prevalence of control flow hijacking, and design mitigations around that pivotal component of this class of attacks. The same holds for speculative executions attacks: their full scope is still largely unknown, and so are appropriate mitigations.

A subset of speculative executions attacks are of particular interest: speculative control flow hijacks (SpCFH), which allow an attacker to redirect execution to an attacker-chosen address that will be speculatively executed within the context of the victim thread. At that address resides a speculative gadget ending in a *side-channel-send* code, or *spadget* [6], which leaks information through a microarchitectural side channel. The attacker can then provide *side-channel-receive* code to read out the leaked information. The resulting attack may allow the attacker to read out arbitrary (and possibly secret) data out of the victim process.

The three known attacks that fall in the category of SpCFH attacks are Spectre v2 (branch target injection) [8], Spectre returns [9] and Speculative buffer overflows [7]. All three either target the branch target buffer (used to predict indirect calls and jumps), or the return stack buffer (used to predict returns).

The exploitability of SpCFH attacks is mainly dependent on the availability of suitable spadgets. To our knowledge, all SpCFH PoCs known to date require the ability to inject code or return into attacker-provided code (as in the Google Project Zero eBPF-based Spectre v2 exploit), showing that suitable spadgets have been hard to find. This motivates the research for new classes of spadgets.

In this paper, we show two new classes of spadgets that can be used in SpCFH attacks, such as Spectre v2. The first uses the instruction cache as a send an receive channel to leak a bit, dependent on a forced control flow in a spadget. The second uses BTI itself as a send and receive channel. While both side channels are known [1, 2, 5, 10], we propose novel variants for them, and analyse their use as part of a transient execution attack. Our results show that both can be used to successfully leak data from a proof of concept, SMT-colocated, victim program with the default spectre mitigation configuration options on the tested Linux distributions. We also verify that hardening the configuration of applicable mitigations (STIBP in this case) is an effective mitigation, making a case for their wider adoption.

This paper makes the following contributions:

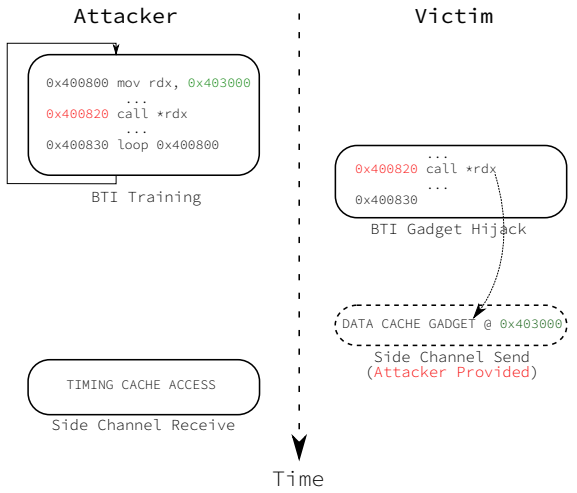

Figure 1: Overview of Spectre v2, a SpCFH attack: the attacker performs BTI at first; the victim speculatively executes the injected gadget whose cache side effects are later measured by the attacker.

- An Icache attack proof-of-concept: uses the instruction cache as a side channel, as part of a BTI attack to leak one bit of information at a time from a victim program.
- A Double BTI attack proof-of-concept: uses the branch target buffer (BTB) as a side channel, as part of a BTI attack to leak one byte of information at a time.
- An Analysis of current branch target injection mitigations on Linux, showing that both attacks work on user space programs with default settings.

Threat model: For both attacks, we assume the same threat model as Spectre v2, that is a local attacker, who knows the code of the target program, is able to bypass ASLR (possibly by using the BTB itself as side channel [5]), and is able to invoke (or predict accurately) code leading to a target indirect branch in the victim program.

2 Background and Related Work

Because there already is litterature available on summarizing existing speculative execution attacks [4, 12, 13] – we here describe background and related work specifically relevant to the two methods presented in this paper.

2.1 Speculative Control Flow Hijacking Attacks

SpCFH attacks, at a high level, rely on a central component of modern CPUs: branch prediction. The CPU needs to predict control flow transfers for filling its deep pipeline of instructions in flight, and the state used for this prediction is crucially shared, in time or in space, between attacker and victim execution threads.

These attacks can be decomposed in four essential steps: 1. SpCFH train, 2. SpCFH trigger, 3. Side channel send, 4. Side channel receive. . We describe those steps by using the initial Spectre v2 attack [8] as an example, as shown in Figure 1. The first step injects an entry into the BTB by training an indirect call. The BTB functions essentially as a hashtable, indexed by a function of the current program counter as well as the history of taken/not taken decisions on past branches. This means that an attacker replicates a similar history, and then invokes an indirect call from a virtual address equal to (or aliasing with, in general) the address of the target indirect branch to hijack, to create an entry in the BTB. On the second step, the attacker typically invokes the victim to trigger a code path leading to the targeted indirect branch. When branch prediction on the victim's indirect branch looks up the BTB, it uses the attacker-injected target. This leads to a speculative control flow hijack. In the eBPF Spectre v2 exploit for (hypervisor) kernels, the code targeted by this hijack is loaded by a host-resident, unprivileged userspace attacker into the kernel, by using an optional eBPF feature of the Linux kernel. This code, *side-channel-send* code, uses a Spectre v1-like, data cache-based gadget to leak kernel data into a shared array (Step 3). In Step 4, after speculative execution completes (and discards the wrongly executed architectural state), the attacker probes this shared array, with *side-channel-receive* code, to read out the data from the cache side channel.

2.2 Related Side channels

Existing transient execution attacks mostly use data cache side channels with a few exceptions (NetSpectre-AVX [14], SMoTherSpectre [3]). In this paper, we consider two other side channels that have not been demonstrated yet in transient execution attacks: instruction cache and BTB.

For the instruction cache, Acıiçmez [1] first demonstrates that secret-dependent control flow attacks against vulnerable cryptographic libraries can also be mounted by using the icache. In this attack, the attacking process forces eviction of the cache lines corresponding to the target victim's code in a loop, and times each iteration of this loop. Loops which run slower correspond to times where the victim process is executing targeted code. In contrast, the icache side channel demonstrated in this paper is akin to a Prime+Probe attack, but on the instruction cache. Indeed, in a scenario where (read-only) code is shared between two threads on the same core, the second

execution of the code will run faster due to caching. On Intel CPUs, their are multiple levels of caching for code: the well-known cache hierarchy (LLC, L2, L1) with L1 being split between code and data. Closer to the execution units, because decoding can be a bottleneck on `x86`, CPUs have microcode caches that cache previously decoded instructions. Although all these caches could be leveraged in an instruction-cache-based side channel, our timing measurements in our proof-of-concepts are mainly in the range of an L1i vs. DRAM difference.

For the BTB as a side channel, Evtyushkin et al. [5] propose a powerful attack to partially derandomize ASLR. By placing indirect calls in the attacking process at different offsets and measuring their execution times, the attacker can infer if the chosen offset is aliased with the victim because location with BTB entries will result in faster execution, thereby leaking code location information by essentially bruteforcing possible offsets. In contrast, the BTB side channel in this paper is even more powerful, in that the attacker process speculatively executes the indirect branch, and places marker in multiple location at once (256 in our proof-of-concept) to infer which location has been trained and therefore leak multiple bits at once from the BTB.

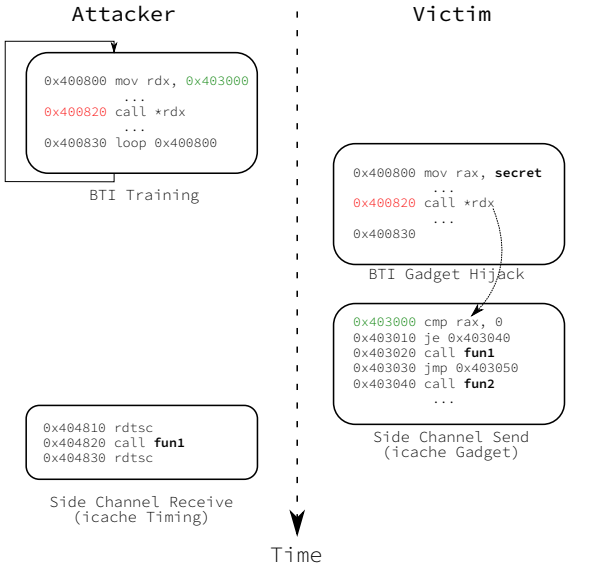

Figure 2: Description of the icache attack: the attacker performs BTI at first; the victim speculatively executes one of two functions depending on the content of a register; the attacker later times the execution of either function to learn one bit of the condition register.

3 Icache attack

The first contribution of the paper is the *icache attack*. Informally, this attack is based on the following observation: while the CPU strives to undo the effects of speculatively executed but not retired instructions, it does not hide effects on the instruction cache. As such, the instruction cache may be used to build a side channel between a gadget speculatively executed by a victim process and a gadget executed by an attacker process.

This attack makes use of speculative control flow hijack in order to redirect the victim to a gadget, henceforth referred to as the *icache gadget*. The icache gadget has the following characteristics: *i*) a compare-like instruction followed by a conditional jump; *ii*) target and fallthrough block of the jump leaving *measurable* and *distinct* side effects in the instruction cache; *iii*) the gadget is mapped by both the victim and the attacker. By measurable we mean that another process should be able to observe changes to the instruction cache left by the speculative execution of the gadget, for instance by attempting to execute either block (target or fallthrough) and measuring the speedup (or lack thereof) induced by the fact that the instructions of the block are present in the instruction cache. By distinct we mean that the effect left by speculative execution of one block should be different from those left by the other block. These two conditions constitute a *side-channel-send* operation over the information constituted by the condition of the jump. Clearly this information must be valuable from a security perspective: the condition may for instance depend on a compare instruction where the content of the register argument contains a secret for the victim. The last condition is required for the *side-channel-receive* operation, since cache line tagging in the instruction cache will not produce cache hits unless the cache lines have identical (physical) tags. Virtual indexing and ASLR also plays a role which will be discussed later in the section.

Figure 2 describes the attack. Attacker and victim are two co-located processes (either interleaved on the same hardware thread or running on different hardware threads in the same core). At first the attacker performs standard branch target injection by training an indirect jump to redirect the control flow to a specific address. The attacker chooses this address as that of the icache gadget. Whenever the attacker is successful, the control flow of the victim will be (speculatively) redirected to the icache gadget. In the figure, the gadget compares the content of `rax` to an immediate, and based on the result jumps to a block that performs a direct call – either to `fun1` or `fun2`. We assume that `rax` contains a secret, loaded before the indirect jump of the victim is executed. If BTI was successful, the attacker may later time the execution of either of the two functions to receive the leaked bit through the side channel. Note that the schedule of attacker and victim only needs to be loosely synchronised: the attacker's BTI training needs to be scheduled before the victim's targeted jump, and the attacker's icache tim-

ing must be scheduled after the speculative control flow hijack takes place. The attacker is thus able to leak one bit for each successful round. By varying the icache gadget to point to gadgets that leak different bits of the secret, the attacker may be able to partially or entirely reconstruct the secret.

3.1 Discussion

Anatomy of an icache gadget As discussed, the icache gadget presents relatively few restrictions and it is thus expected to be widely available to an attacker. In particular, the requirement of a shared memory mapping is satisfied in the (common) case of two processes (attacker and victim) using a common shared library, or the attacker mapping the executable of the victim. This ensures that instruction cache lines will have identical (physical) tags. Restrictions on virtual addressing will be discussed later in the section. The gadget shown in Figure 2 requires target and fallthrough of the conditional jump to contain a call to different functions. However, at its core, the gadget only requires that the icache-observable side effect be different depending on the outcome of the conditional jump. With this criterion we may eliminate gadgets whose size is a single cache line, or gadgets that will be prefetched in their entirety irrespective of the actual (speculated) control flow. No further restriction is imposed on the gadget. Finally, we stress that the icache gadget does *not* require the presence of the secret-dependent control flow antipattern in the victim code, e.g., as in previous icache-based attacks [1, 2]. While the icache gadget indeed performs a conditional jump based on the value of a secret, the secret is set by the victim in the completely unrelated BTI gadget.

ASLR The presence of ASLR on most modern systems introduces an obstacle for the attacker; indeed, while the requirement on a shared mapping of the icache gadget ensures that cache lines will have identical (physical) tags, they must also have identical (virtual) indices. The attacker may either target a shared icache gadget that is not built as position-independent code (e.g. (rare) a shared library built without the `fPIC` or equivalent compiler option; or (more common) an executable built without the `fPIE` or equivalent compiler option), or utilise well-known means of discovering the ASLR offset [5, 15].

Alternative *side-channel-receive* In the icache attack, the side channel is read by timing the execution of either the target or the fallthrough block of the jump in the icache gadget. An alternative to this approach is to perform a standard cache timing attack, by simply reading the code to probe it, instead of executing. Given that in our target platforms L1 data and instruction caches are separate, we did not try this experiment because the side channel would be noisier due to the smaller time difference between L2 cache and main memory.

4 Double BTI attack

In this section we describe the second attack, called *Double BTI attack*. The Double BTI attack also exploits speculative control flow hijack, as first shown by the Spectre v2 PoC. The original Spectre v2 POC, depicted in Figure 1, requires the ability of the attacker to inject a gadget into the victim address space, namely, the data cache gadget used to perform the *side-channel-send* operation.

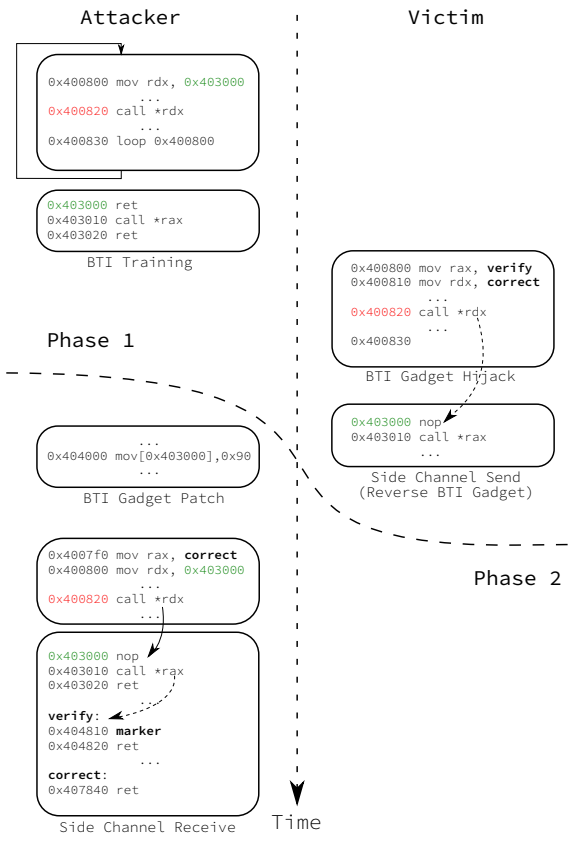

Figure 3: Description of the Double BTI attack: the attacker performs BTI at first; the victim speculatively executes the "reverse" BTI Gadget that further trains the branch predictor with the value of a register or a memory location; the attacker later execute the same "reverse" BTI Gadget and based on the side effects of wrong prediction (e.g. executing an instruction marker to a given location) can guess the value of the register or memory location

With the Double BTI attack we are able to lift this restriction, making speculative control flow hijack attacks far more pernicious. The intuition behind the attack is

that the gadget implementing the *side-channel-send* operation may be instantiated as simply as by a second indirect call. Crucially, this indirect call will cause a second, "reverse" BTI, where this time the attacker is subjected to branch target injection. If the attacker is able to measure the effects of this second BTI and learn one or more bits of information about the injected target, the side channel is successfully read.

At a high level, the attack has 2 main phases: in the first phase, the attacker performs standard BTI and, whenever successful, causes the victim's control flow to be (speculatively) hijacked to execute the reverse BTI gadget. This represents the *side-channel-send* operation. In the second phase, the attacker attempts to perform the *side-channel-receive* operation by observing the effects of the victim's speculative execution. We can see the two phases in detail in Figure 3.

4.1 Phase 1

Phase 1 starts with the attacker training the BTB by repeatedly executing an indirect call whose target address is identical to the one of the reverse BTI gadget in the victim. The attacker can execute this either on the same thread or on a twin thread on the same physical core of the victim process. The gadget (identified in the figure as the BTI training gadget) the attacker calls into initially consists of a return instruction followed by a register-indirect call instruction that is never executed in this phase.

When the training is over and BTI is successful, we assume that the victim speculatively executes the reverse BTI gadget. The reverse BTI gadget is identical to the BTI training gadget in the attacker, save for the fact that it starts with a nop. The nop may be replaced in practice with any instruction that doesn't disrupt the control flow and whose size still ensures that the indirect call in the victim's reverse BTI gadget has the same address as the (so-far unexecuted) indirect call in the attacker's BTI training gadget.

The reverse BTI gadget contains an indirect call which is speculatively executed. Crucially, our findings prove that the side effects caused on the BTB by its execution are not rolled back by the CPU. Further, we show that a single execution of the victim is sufficient to make this side effect persistent and observable. For these reasons, we can see the reverse BTI gadget as an implementation of the *side-channel-send* operation: if the information being sent depends on a secret of the victim, the attacker is later able to read it with a suitable *side-channel-receive* gadget in the next phase.

4.2 Phase 2

At this point phase 2 begins. In phase 2 the attacker "patches" its BTI training gadget by replacing the leading `ret` with a `nop`. This enables the attacker to perform the second indirect call without losing alignment with the victim and without requiring more complex gadgets to distinguish between training and measurement mode.

Subsequently, the attacker calls into the (now patched) BTI training gadget once more, finally executing the register-indirect call whose target was trained by the victim. If the victim's training was successful, the attacker will not execute the code at the `correct` label but rather at the victim-trained `verify` label. This is because the CPU tries to predict the target of the call, and uses the history left from the victim execution. The attacker structures its address space to contain suitable speculative execution markers. Observing the side effects left by the marker corresponds to the *side-channel-receive* operation.

4.3 Practical considerations

In our first proof-of-concept implementation, we instantiate the marker with a set of instructions that is measured by a specific Intel Performance Monitor Counter (PMC). The chosen event must be one that is triggered even if the responsible instructions do not retire. In practice we have chosen the failed store-to-load forward counter, which requires a sequence of 3 `mov` instructions. The performance counter related to the marker is incremented whenever the attack succeeds and the indirect call in phase 2 is speculatively redirected to the location trained by the victim. Clearly this technique is not applicable to a real-world setting since programming PMC counters requires root privileges.

We identify 2 realistic marker instances that implement a *side-channel-receive* operation. The first candidate uses instruction cache side effects. Assuming that the attacker knows the first 6 most significant bytes of `rax` and wants to discover the 7^{th}, it would layout its address space by placing at each of the 256 possible addressed an icache-differentiable gadget. This gadget would in practice contain a suitable amount of nop padding to account for the content of the least significant byte and a call instruction to one of 256 different functions, followed by an `lfence` instruction to stop speculative execution. The attacker would speculatively execute one such gadget as the first part of the *side-channel-receive* operation, and then time the execution of all functions as second part of the *side-channel-receive* operation. If only one of the functions executes in less time than a pre-computed threshold, its ordinal number corresponds to the leaked byte. This approach suffers from a rapidly deteriorating signal quality, due to the noise in-

duced in the instruction cache by the measuring process.

```
value0:
     mov rax, QWORD[array + 0 * 1024]
     ret
value1:
     mov rax, QWORD[array + 1 * 1024]
     ret
                ...
value255:
     mov rax, QWORD[array + 255 * 1024]
     ret
```

Figure 4: *side-channel-receive* approach using data cache access pattern

The second candidate uses data cache access as a measurable side effect. The setup is identical to the previous approach save for the fact that the 256 target functions each contain a different memory access (`load` operations on an array). When speculatively executed, this induces an effect in on the data cache, which can then be measured. This approach is described in Figure 4. With this approach, the side channel signal maintains its quality throughout the measuring process and allows the attacker to extract a full byte from the *side-channel-receive* operation.

5 Evaluation

5.1 icache attack

Experimental setup We test the icache attack on an Intel Core i7-6700K CPU running Ubuntu 16.04.6 LTS, kernel version 4.15.0. The attacker and victim processes are co-located. The following system setup is in place: ASLR is off to ensure consistent virtual addresses for BTI training, scaling governor is set to `performance` for constant clock frequency. Clock frequency is set below turbo. On the speculative execution mitigation side, the default setup is in place – `spectre_v2` set to `auto` and `spectre_v2_user` set to `auto`.

The attacker process is timing the execution of target code that is shared between victim and attacker. The same icache (physical) tags allow the attacker to determine the exact path taken in the victim icache gadget. To enforce this behaviour, we test our attack on two different setups: in the first, the shared code resides in a POSIX shared memory region; in the second, the shared code is part of a shared library. For this second part, we test with both `libhttp-parser`, part of nodejs and `libcrypto`, part of OpenSSL.

Attacker and victim use lightweight synchronisation for higher BTI success rate. In practice, this synchronisation is not required as long as we can assume that the attacker is able to trigger the victim and can thus time its execution accordingly. To maximise the signal of the icache side channel we flush the cache lines that correspond to the target code area before each loop. Given that the shared gadget is dynamically mapped, the icache timing gadget in the attacker does not time a direct call but a register-indirect one.

Secret	Success Rate
0	98.89%
1	97.78%

Table 1: icache attack experiment with a gadget from `libhttp-parser.so`: each row displays the success rate in guessing the value of the victim's secret. The success rate is computed as the rate between samples displaying an icache hit (resp. miss) when the value of the victim's secret was 0 (resp. 1). An icache hit is defined as an execution of the icache gadget timed below a pre-determined threshold.

Results and Discussion Our PoC program runs 100 repetitions of the attacker and victim. We notice that the attacker timing results are influenced by its prior knowledge of the target address of the `call` instruction causing what we assume to be prefetching. To eliminate this source of noise from the timing we change the `call` target address once in 11 loops, thus obtaining one valid sample out of 11 runs of attacker and victim loop. We run the PoC one thousand times and we therefore collect a total number of 9k samples out of 100k runs of the attack.

Table 1 shows results with a gadget chosen from `libhttp-parser.so`: in particular the chosen functions for `fun1` and `fun2` are 7 pages apart and are 29 and 870 bytes each. We obtain similar results for the other combinations (POSIX shared memory or different shared objects). The BTI success rate varies from 10% to 90% over all our experiments. The table shows results from an experiment where the BTI success rate lies between 32% and 34%. Each run collects one timing sample for the execution of the function `fun1` (with reference to Figure 2) corresponding to the function that the victim should execute in case of successful BTI and when the (secret) value of the condition register is 0. Intuitively, if the average of a distribution of timing samples is smaller than some threshold, the attacker should be able to conclude that the value of the secret is 0, and 1 otherwise. We determine the value of the threshold by timing the execution of `fun1` during a learning phase, building a distribution of timing samples for icache hits and setting a *hit threshold* as $ht = avg + 3 * \sigma$, where avg and σ are average and standard deviation of the distribution.

The overall success rate of the experiment shown

Family Name	Code	Success Rate
Coffee Lake	i7-8559U	66.8%
Skylake	i7-6700k	83.9%
Kaby Lake R	i7-8550U	68.6%
Kaby Lake R	i7-8650U	69.0%
Broadwell	i5-5250U	25.5%

Table 2: Double BTI attack success rate on leaking a one byte of secret

in Table 1 is computed by building a distribution of samples for each run of the attacker (100 repetitions as described above) and considering a hit the cases in which the average is below the hit threshold (resp. above) and the value of the secret is 0 (resp. 1). The table shows that the success of the overall experiment is always above 97% with either value of the secret.

5.2 Double BTI Attack

Experimental setup We tested our Double BTI attack on multiple Intel CPUs. On each machine, the attacker and the victim are co-located. In the PoC, the register (`rax`) that is the target of the indirect jump in the reverse BTI gadget is set as follows: the 3rd least significant byte is a secret value that the attacker wants to discover, prefixed by a (known) offset and suffixed by all zeroes. The prefix just ensures that the attacker can map its set of 256 markers at a non otherwise mapped location. In the PoC, the attacker uses Double BTI attack to learn the value of the secret byte. We use data cache timing markers as discussed in Section 4.3. During this experiment, the mitigations enabled against BTI are the default ones (see Section 6) enabled on a stock Ubuntu. In this attack, we do not employ any specific synchronisation between victim and attacker: the correct sequencing of the two processes is achieved simply by delaying the start of the victim by a suitable amount of time. We `clflush` the memory locations containing the indirect call targets to maximize the speculation window. With this setup, we measure the attack success rate over 1000 attempts to leak the unknown byte of `rax` by timing accesses to each of the 256 locations in the array that is filled by the corresponding markers (as described in Figure 4). The timing of the array is performed in non-linear order to avoid prefetching effects. The timing always reveals two different cases: either exactly one array location is below a pre-defined threshold (fixed at 80 clock cycles) or none is. The first case corresponds to a successful *side-channel-receive* operation.

Table 2 shows the results of our experiments on different platforms. We can see that we have non-negligible successes on all platforms, with success rates peaking above 80% and never below 20%. The quality of the side channel signal is excellent owing to the fact that the attacker performs both the initial (speculative) access followed in close succession by the timing of the array location accesses, yielding an extremely clean measurement environment.

6 Mitigations

Both the icache and double BTI method presented here use BTI for speculative control flow hijacking. Therefore, BTI mitigations from Spectre v2 are applicable.

Mitigations are available at the hardware and software level to prevent BTI attacks. At the software level, compiling with retpoline [16] mitigates BTI by rewriting all indirect calls to avoid CPU prediction, through the use of a carefully crafted return sequence. At the hardware-level, Intel added Indirect Branch Restricted Speculation (*IBRS*), Indirect Branch Predictor Barrier (*IBPB*) and Single Thread Indirect Branch Predictors (*STIBP*. *IBRS* essentially flushes all branch predictor state when switching between user and kernel mode. IBPB essentially flushes all branch predictor state upon execution, even within a process. Finally, STIBP stops sibling SMT threads branch predictor from influencing the branch predictor decisions on other siblings threads on the same core.

We tested our attacks against the current implementation of BTI mitigations on the stock kernel 4.15.0 of our Coffee Lake machine. The kernel offers two switches to enable Spectre v2 protections. The first, `spectre_v2`, controls mitigations for protecting the kernel from userspace attacks, as well as functions as a master switch for enabling userspace protections. It can be set to `on`, `off` or `auto`. The option `on` and `off` forces respectively all the protection to be enabled or disabled. In our experiment, we left `spectre_v2` to `auto`, the default setting in recent Ubuntu distributions, to be able to enforce a finer grain control over the BTI mitigations and test functionality.

The second `spectre_v2_user` controls mitigations for userspace programs, and is gated by the previous setting. It can be set to `on`, `off`, `auto`, `prctl/ibpb` and `seccomp/ibpb`. As for the previous switch, `on` and `off` enable and disable all the protections. Meanwhile, `auto` defers the decision to enable or disable each protection and their mode based on additional configuration. Instead, both `prctl/ibpb` and `seccomp/ibpb` set IBPB always-on but leave conditional STIBP that has to be enabled on request by the process. For seccomp processes the restriction is enabled automatically.

Among those settings, our attacks are prevented if and only if STIBP is enabled (forced globally or the victim

Distribution	Kernel	Generation Date	STIBP	Vulnerable?
Ubuntu 18.04.2 LTS	4.15.0-50-generic	May 6 18:46:08 UTC 2019	conditional	Yes
Ubuntu 18.04.2 LTS	4.18.0-18-generic	Apr 5 10:22:13 UTC 2019	conditional	Yes
Ubuntu 16.04.6 LTS	4.15.0-50-generic	May 8 15:55:19 UTC 2019	conditional	Yes
Ubuntu 18.04.2 LTS	4.19.0-041900-generic	Oct 22 22:11:45 UTC 2018	unsupported	Yes
Ubuntu 18.04.1 LTS	4.15.0-29-generic	Jul 17 15:39:52 UTC 2018	unsupported	Yes

Table 3: Default STIBP settings in the kernel used by the distributions tested in our evaluation

thread enables STIBP using `prctl`). Both attacks can also be prevented in software if the victim is compiled using retpoline. While non-SMT based BTI attacks can be mounted (i.e, attacker thread runs before and after victim threads, with two context switches), because of the enabled kernel mitigations flushing branch predictor state, these attacks do not apply.

Given the current performance penalties of enabling STIBP, this protection is set conditional by default or unsupported (as shown in Table 3) and therefore unless requested by the application, our attack is not mitigated. Furthermore, we verified that sensitive programs such as `passwd`, `sudo` and `nginx` do not make use of the `prctl` interface to enable currently such protection. Given these default settings and the risks posed by BTI-related attacks, and in particular those presented in this paper, we recommend sensitive applications to enable STIBP through `prctl` when assuming local attackers.

Finally, other types of speculative control flow hijacks, i.e., return prediction based [9, 7] remain unaffected by these mitigations, and the two methods presented in this paper could be applied for those attacks as well.

7 Conclusion

In this paper, we present two new attacks, icache, and Double BTI. With these two attacks, we are able to leak a bit and a byte respectively from a victim context for each run. Both attacks lower the requirements for Spectre v2 gadgets, since they do not require the injection of code inside the victim. We develop and test proofs of concept for both attacks on several CPUs showing their success rate and general viability. Also, we analyse the attacks against current available mitigations (e.g. STIBP) and confirm their success when mitigations are left with their default settings. We also verify that sensitive programs such as *sudo*, *passwd* do not make use of the *prctl* interface to enhance their protection against such attacks. We leave real world implementation of our attacks against such programs for future work. In the meantime, we recommend maintainers of sensitive userspace programs to consider enabling BTI mitigations.

References

[1] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007.

[2] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2010.

[3] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention, 2019.

[4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. `https://arxiv.org/abs/1811.05441`, 2018.

[5] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 40. IEEE Press, 2016.

[6] Richard Grisenthwaite. ARM Whitepaper: Cache Speculation Side-channels, 2018.

[7] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. `https://people.csail.mit.edu/vlk/spectre11.pdf`, 2018.

[8] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*, 2018.

[9] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *USENIX Workshop On Offensive Technologies*, 2018.

[10] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security 17)*, 2017.

[11] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[12] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Let's Not Speculate: Discovering and Analyzing Speculative Execution Attacks. https://domino.research.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c85256b360066f0d4/d66e56756964d8998525835200494b74, 2018.

[13] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution, 2019.

[14] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. https://arxiv.org/abs/1807.10535, 2018.

[15] Alexander Sotirov. Bypassing memory protections: The future of exploitation. In *USENIX Security*, 2009.

[16] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018.