SMoTherSpectre: exploiting speculative execution through port contention

Atri Bhattacharyya
EPFL

Alexandra Sandulescu
IBM Research – Zurich

Matthias Neugschwandtner
IBM Research – Zurich

Alessandro Sorniotti
IBM Research – Zurich

Babak Falsafi
EPFL

Mathias Payer
EPFL

Anil Kurmus
IBM Research – Zurich

Abstract

Spectre, Meltdown, and related attacks have demonstrated that kernels, hypervisors, trusted execution environments, and browsers are prone to information disclosure through micro-architectural weaknesses. However, it remains unclear as to what extent other applications, in particular those that do not load attacker-provided code, may be impacted. It also remains unclear as to what extent these attacks are reliant on cache-based side channels.

We introduce SMoTHERSPECTRE, a *speculative code-reuse attack* that leverages port-contention in simultaneously multi-threaded processors (SMoTHER) as a side channel to leak information from a victim process. SMoTHER is a fine-grained side channel that detects contention based on a single victim instruction. To discover real-world gadgets, we describe a methodology and build a tool that locates SMoTHER-gadgets in popular libraries. In an evaluation on `glibc`, we found more than hundred gadgets that can be used to leak some information. Finally, we demonstrate a proof-of-concept attack against encryption using the OpenSSL library, leaking information about the plaintext through gadgets in `libcrypto` and `glibc`.

1 Introduction

Spectre [20, 21, 26] and Meltdown [23] form a new class of micro-architectural attacks. These attacks leverage weaknesses in speculative execution (Spectre) or separation between privileged and unprivileged code (Meltdown) to leave micro-architectural traces [5]. Both Spectre and Meltdown leverage a side channel based on the memory architecture to leak data from the address space of a target (e.g. from another process or from the kernel).

While micro-architectural side channels were known before the discovery of Meltdown and Spectre, their applicability was mostly limited to targets applying data-dependent control flow patterns or memory accesses. In this older class of vulnerabilities, an attacker would observe the micro-architectural changes to shared resources caused by the execution of a victim. For example, in a cache-based attack, the adversary would prime the cache, let the victim execute, and then detect which locations have been evicted from the cache. Such a side channel leaks addresses and allows the adversary to learn information from data-dependent execution. An effective mitigation strategy is to eliminate data-dependent control flow over sensitive data, such as cryptographic material.

In contrast, Spectre and Meltdown render this class of attacks generic and significantly harder to mitigate through software changes only. The side channel is now used *indirectly*, in a way that – crucially – does not rely on poor choices in the development of the target application. In Spectre, for instance, the attacker first primes the speculation engine (e.g., by preparing the branch target buffers) as well as the cache-based side channel; the victim then misspeculates at an attacker-controlled location and thereby leaks information [5]. The attacker can then read out the cache-based side channel. In light of these new attack vectors, architectural, system-wide defenses such as Kernel Page-Table Isolation [13], retpolines [30], or microcode updates must be rolled out to protect the system against attacks. One proposed microarchitectural defense is to revert all side effects of speculative execution [18].

One mitigating factor is that so far, with the exception of Netspectre-AVX [26], all existing attacks rely on side channels that are invariably cache-based to read out information. This in turn requires the presence of specific gadgets in the victim, which are often hard to find. Consider the example of Branch Target Injection (BTI), the technique used in Spectre v2 [20]: in the initial exploit, no suitable gadget was identified in the kernel. The attack was successful because it redirected speculative control flow to externally provided code, in the form of eBPF kernel code. This observation justifies why mitigations such as retpoline are not employed at large by user-space programs.

In this paper, we show that speculation attacks (e.g., through branch target injection) can leak arbitrary secrets from generic user-space programs through a side channel that

is not based on the memory architecture. In particular, we show that branch target injection can be used on existing program code, without requiring the injection of attacker code. To this end, we first show that port contention can be used as a powerful side channel when executing with simultaneous multi-threading (SMoTHer). We then exploit port contention as a side channel to transmit information during speculative execution (SMoTHerSpectre). This shows that, because the transmission occurs before speculative execution ends, reverting side effects of speculative execution would not be sufficient as a defense. Finally, we show how suitable portions of code can be found in target binaries automatically.

Other related work has looked at execution-unit-sharing as a side channel [1, 2, 10, 32]. Portsmash [2] concurrently developed to our work, demonstrates that port sharing leaks code access patterns and successfully extracts secrets from a known vulnerable version of OpenSSL. We are however the first to characterize this side channel and leverage it for a speculative execution attack, providing a full working proof of concept that leaks data from a recent OpenSSL.

This paper makes the following contributions:

- A precise characterization of the port-contention side channel (SMoTHer);

- A speculative execution attack (SMoTHerSpectre) that demonstrates the suitability of non-cache-based side channels to leak information. We show an end-to-end attack using speculation based on BTI by combining it with the port contention side channel;

- An automated technique to find target speculative gadgets in programs; and

- A real world attack where we target a BTI gadget in the latest version of OpenSSL and a SMoTHer gadget from the libc.

2 Background

The work in this paper relies on the complex interplay between software and hardware. In the following, we provide the background information necessary to understand SMoTHer and SMoTHerSpectre.

CPU Microarchitecture.
A modern CPU is typically split into two main components: the *frontend* and the *backend* (or execution engine). The frontend predicts where to fetch instructions from and creates a program-order stream of instructions to be executed by the backend. The instructions are either decoded and executed "as-is" in RISC ISAs (e.g., IBM POWER or ARM) or broken down into RISC-like instructions called μops in CISC ISAs (e.g., x86 or IBM Z). For brevity we refer to all instructions executed by the backend as μops. Once fetched and decoded, the μops are placed in an *instruction window* (also referred
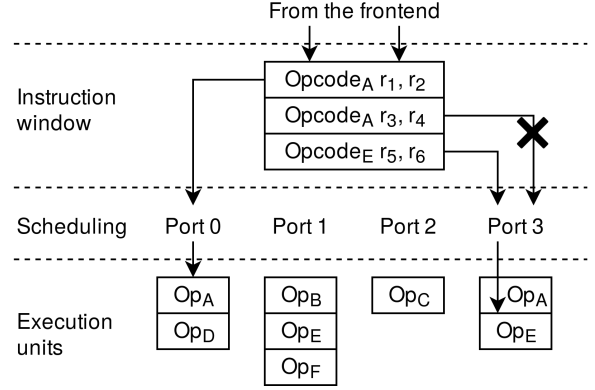

Figure 1: Instructions from the window are scheduled to ports shared by sets of execution units. A single instruction may be scheduled per port per cycle.

to as issue queue or reservation stations) to be scheduled and dispatched to execution units when their operands are ready. Every cycle, the scheduler searches the instruction window to identify which μops are ready for execution and which execution unit is available to dispatch them to. μops can execute out of program order (e.g., a later μop in program order can execute earlier) if their operands are ready and a relevant execution unit is available. Ideally, all execution units would be designed to handle every type of operation to maximize throughput. In practice, execution units are specialized and only the more commonly used ones are replicated. A group of execution units share a port, indicating their availability in a given cycle. Contention for a port leads to delays in execution. Figure 1 demonstrates scheduling instructions from an execution window containing three μops, where contention for port 3 prevents the second μop from being scheduled in the same cycle as the other two.

Speculative Execution. Because the stream of μops is *predicted* but is not guaranteed to *execute, complete and make its state visible to software*, the backend also contains a *reorder buffer* that commits the state of each completed μop in program order to the software visible structures (i.e., register file and memory). This execution of μops is *speculative* because the frontend may have mispredicted the direction and/or the target address of a branch operation. Upon misprediction, the pipeline flushes all μops in the re-order buffer and restarts fetching and decoding μops. While executing on the mispredicted path, the processor accesses the cache hierarchy leaving side-effects which lead to cache-based side channels even though the values accessed are discarded and do not impact the executing software.

Simultaneous Multithreading. Out-of-order processors provision a large fraction of silicon area to mechanisms that exploit speculation and parallelism in execution. While these mechanisms are designed for peak parallelism, most structures (e.g., execution units, branch tables, physical registers,

instruction window, re-order buffer) remain underutilized on average. Simultaneous MultiThreading (SMT) is a technique to improve utilization of these structures by allowing μops from multiple threads (e.g., two in x86 and eight in IBM POWER) to execute simultaneously on a single core. Individual SMT threads maintain their own architectural state, but share many microarchitectural structures in the processor pipeline simultaneously. SMT (or HyperThreading as Intel brands its implementation) is entirely transparent to software to which a single core appears as multiple logical cores. Besides the execution units, physical registers and instruction window, it is an implementation's choice as to which other structures SMT threads share. Experiments have proven that the branch predictor is shared between hyperthreads [6, 15] on Intel CPUs.

Speculative Execution Attacks. Speculative execution can be exploited by priming the branch predictor with sufficient history such that it is tricked into predicting the wrong target for a branch. Because branch direction history (i.e., taken or not taken) is a shared resource, an attacking process can prime the branch predictor of its victim. Similarly, a branch target buffer predicting the target address for a branch can be primed by an attacking process. This works for both conditional branches as well as indirect branches. In a conditional branch, such as an array-size check in Spectre V1, the CPU can be tricked into speculatively executing an out-of-bounds array access in spite of the failing length check. If the target address of the length check is not in the cache then the memory fetch will take longer than the following speculatively executed instructions. In an indirect branch, the CPU can be tricked into speculatively executing arbitrary code in a victim process by providing a malicious branch history through a temporally or spatially (in the case of SMT) co-located attacker process. We discuss related work in section 7.

Cache Side Channel. Speculative execution attacks, such as Spectre, exploit the fact that a speculatively executed and then discarded operation does have side effects on the microarchitectural state, even if it has none on the architectural state. For example, an instruction that operates on a value stored in memory will need to fetch that value and cause the corresponding memory region to be pulled into the cache. The side-effect that the memory region is now cached is not undone when the instruction is discarded instead of retired, and can be measured using cache side channels. For example, in Spectre V1 the victim code uses two dependent array lookups, where the result of the lookup of the first array is used as an index into the second array. This index can be leaked by measuring access times to the second array through a flush and reload attack. By ensuring that the second array has been flushed from the cache before the victim code executes, and measuring the access times afterwards, only the lookup of the index that has been used by the victim code will be significantly faster. We discuss related work in section 7.

3   Smother

In this section, we describe and evaluate SMOTHER, a side channel based on port-contention, present in SMT architectures. SMOTHER is based on the following observation: two co-located (i.e., running on the same physical core) hardware threads of execution share execution units. Instructions that are scheduled to execute on the same execution port will contend for the available resources. We show how this contention can be measured, at first in a coarse-grained way, i.e., with large sequences of instructions scheduled on the same port on both threads, and then in a fine-grained way, i.e., with minimal sequences of instructions. The result is that an unprivileged *attacker process* can detect whether a co-located *victim process* is running an instruction on a given port.

3.1   Ideal covert channel

In this experiment, we demonstrate port contention between two threads running simultaneously on the same physical core and describe how it can be measured in ideal conditions.

3.1.1   Experiment design

Executing instructions that occupy a specific port and measuring their timing enables inference about other instructions executing on the same port. We first choose two instructions, each scheduled on a single, distinct, execution port. One thread runs and times a long sequence of single micro-op instructions scheduled on port a, while simultaneously the other thread runs a long sequence of instructions scheduled on port b. We expect that, if $a = b$, contention occurs and the measured execution time is longer compared to the $a \neq b$ case.

3.1.2   Experimental setup

We run experiments on an Intel Core i7-6700K CPU running Ubuntu 16.04.4 stock kernel, version 4.15.0. Both attacker and victim are pinned to different hardware threads on the same physical core. The CPU governor is set to *Performance* for a constant clock frequency. The "performance" state is configured below the turbo frequency range to lower non-deterministic factors in the environment. Apart from these changes, all other settings are kept to their defaults. Most notably, speculative-execution-related mitigations are left enabled.

In the measuring thread, we execute and time a sequence of 1,200 `shl`, a single-micro-operation instruction that executes on port 0 or port 6, which we denote port 06, on this CPU. The colocated thread runs a sequence of either 1,200 `shl` or `popcnt` instructions: the `shl` instructions directly contend for port 06 while the `popcnt` instructions will introduce no contention as they execute only on port 1.

3

3.1.3 Results and discussion

We report averages over 10,000 runs, together with a 95%-confidence interval calculated using the Student's t-distribution. The experiment successfully demonstrates that port contention occurs and that the SMOTHER side channel can be used to extract information, as we can see in Table 1. Indeed, the run time of the contention experiment is about twice of the non-contended one. This indicates that port contention is likely the main bottleneck in this experiment.

This result shows how SMOTHER can be used as a reliable covert communication channel between two co-located threads. However, as this experiment requires precisely choosing the type and number of instructions running in one of the two threads, it is yet unclear if port contention may serve as a practical side channel. We explore this aspect in the next section.

Experiment	Execution Time (cycles)
Port contention	1214 ± 67
No port contention	674 ± 13

Table 1: Port contention covert channel: a thread running a long sequence of port 06 instructions is twice as slow when a co-located thread executes a long sequence of port 06 instructions, when compared to a co-located thread executing a long sequence of port-1-only instructions

3.2 Characterization of the side channel

We now analyse whether SMOTHER is effective as a side channel for distinguishing realistic sequences of instructions on a simultaneously executing, co-located victim process. Specifically, we want to explore whether an attacker can distinguish between the different sequences of instructions from a known set which the victim may run. To encapsulate this property of the set, we define the term SMOTHER-*differentiability*.

SMOTHER differentiability. Let us consider that the victim runs one sequence out of a set $V = \{V_0, V_1, ...\}$. The attacker is allowed to craft any sequence of instructions A and time multiple iterations of A running concurrently with the victim. If the attacker can infer which sequence $V_i \in V$ the victim was running based on its timing measurements, the sequences in V are said to be SMOTHER-differentiable. On its part, the attacker has a-priori knowledge of what timing to expect when A runs concurrently with each of $V_i \in V$. It can use experiments in a similar, but controlled, environment to generate this knowledge. Further, the attacker is allowed to use *any* statistical test or metric to make its decision. Examples of such metrics include the mean or the median of the timings, or their distribution. In the experiments we use statistical significance at 95%-confidence for the Student's t-test.

At its core, SMOTHER-differentiability implies that the

sequences in V have differing degrees of utilization on some specific port(s) and vice-versa. The attacker would ideally choose a sequence of instructions scheduled solely on these ports to maximize the chance of encountering different levels of contention across the different possible V_i. Through our experiments, we wish to explore how short SMOTHER-differentiable sequences can be and the ideal length of attacker sequences to differentiate them.

Experiment design and setup. In our first experiment, we consider a victim running sequences of either `popcnt` or `ror` and an attacker timing a sequence of `popcnt`. We vary the length of both attacker and victim sequences, and check for SMOTHER-differentiability by noting the percentage change in mean execution time for the attacker. In a second experiment, the victim runs either `cmovz` instructions or `popcnt`. In this case, the attacker times a sequence of `bts` instructions.

To run this experiment, an orchestrator process is used to fork the victim and attacker processes, and to set their core affinities so that they share a physical core. We require the execution of the target sequence in the victim to temporally overlap with the (timed) execution of the attacker sequence to assure port contention. Therefore, the processes use a synchronization barrier which ensures that any following instructions will be run concurrently. Thereafter, each process runs their respective sequence, using `rdtscp` to take timestamps at the beginning and end of each run. The timestamps tell us the number of cycles taken to execute the sequence and were used to also check that the executions were properly synchronized. Atomic operations on variables in shared memory were used to implement the synchronization. We repeat this process to collect multiple timing samples.

In this set of experiments, we keep the same hardware and OS configuration as used in the covert channel experiment, while precisely controlling the synchronization of threads through the additional instrumentation described above.

Results. Figure 2 plots the average difference in attacker execution time between the two sequences of victim instructions for each experiment. The length of the sequence for the victim was taken from the set $\{1, 4, 8, 16, 32\}$ while the attacker sequence varied in length between one and 100 instructions.

Our measurements confirm that timing short sequences of instructions is feasible: for a vast majority of sequence-length combinations the victim sequences were SMOTHER-differentiable using the Student's t-test on the attacker's running time distributions. While timing `popcnt`, 83% of all combinations plotted in Figure 2a showed significant differences in means between the victim's sequences of `popcnt` and `ror`.

The measured differences vary from close to 0% to 40%. Longer sequences of instructions in the victim lead to higher differences and less variability in measurements. Only 48% of `popcnt` measurements with sequence of 1 victim instruction are SMOTHER-differentiable, as opposed to 83% for a sequence of 4, and 100% for a sequence of 32 victim in-
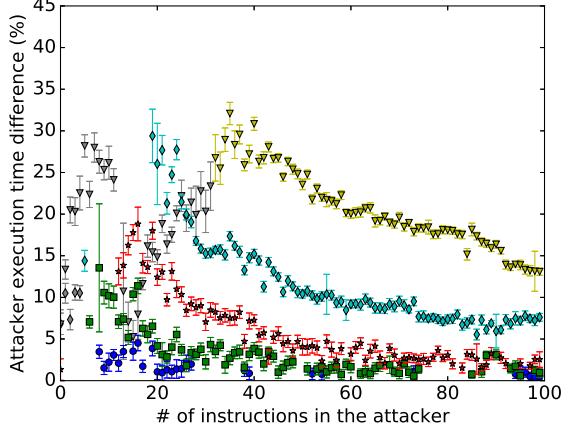
(a) SMOTHER attack using `popcnt` to detect if the co-located victim runs on port 1.
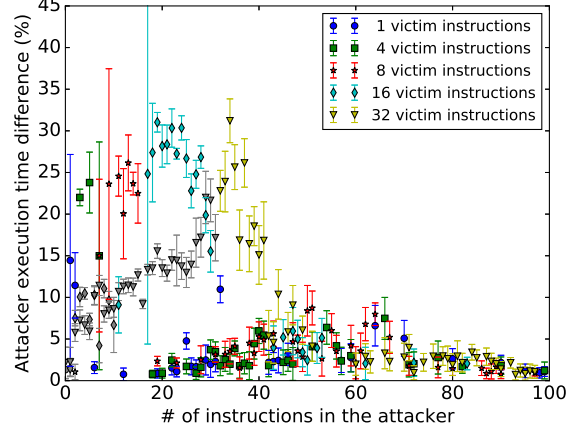
(b) SMOTHER attack using `bts` to detect if the co-located victim runs on port 06.

Figure 2: SMOTHER side channel characterization. Each data point represents the difference between the average execution time of the attacker thread, between the port contention scenario and the baseline. We do not plot the few data points where Student's t-test shows no statistically significant difference between both distributions at 95%-confidence. The data points for which the attacker runs fewer instructions than the victim are plotted in grey.

structions. This means that distinguishing a sequence of one victim instruction (max. 9% difference and more variability) is much harder than a sequence of 32 victim instructions (max. 38% difference and less variability).

We observe that there is an optimal number of attacker instructions to measure a victim instruction sequence of a given length, which increases with the number of victim instructions: from 10 attacker instructions for one victim instruction to 45 instructions for 32 victim instructions. This is explained by the following observations: contention for longer instruction sequences in the attacker is easier to time, since attacker and victim sequences are more likely to overlap. This effect fades when the attacker sequence becomes significantly longer than the victim's, at which point only a small portion of the executed instructions will contend, thereby leading to a smaller difference.

To show the breadth of possible SMOTHER-

differentiability results, we perform a second experiment, with a victim running instructions which may be scheduled to more than one port. Specifically, the victim runs either `cmovz` (port 06) or `popcnt` (port 1). The attacker times a sequence of `bts` instructions (port 06) to measure the contention on ports zero and six. Figure 2b shows that multiport instructions are still SMOTHER-differentiable. However, variance is higher, and we notice a steeper cut-off point beyond the optimal number of attacker instructions. Indeed, intuitively, with more execution ports available, the instructions are less likely to contend. In practice, this means the attacker may need more runs to extract information, and the choice of the number of attacker instructions is more important than in the previous experiment. As in the previous experiment, we observe that the optimal number of attacker instructions increases with the number of victim instructions. Beyond this number, most experiments show lower SMOTHER-differentiability, with

most between 0 and 5%.

While our results show that the SMOTHER side channel exists and can be measured even for a small sequence of instructions, we have noted a number of takeaways and pitfalls to avoid during measurements, namely:

- Synchronisation of the target code sequence in the victim and the timed code sequence in the attacker is extremely important, more so when the target code sequence in the victim is short;

- Pipeline bottlenecks other than port contention may occur and overshadow the side channel. One such example is read-after-write hazards;

- The CPU may eliminate the execution of some instructions based on their operands (one such case is *zero idioms*). This results in those operands not being executed, and removing contention;

- Some instructions (e.g., those from the SSE and AVX extensions) are subject to aggressive power-saving features on modern CPUs. This makes measuring port contention more difficult (and the power savings may in fact serve as its own side channel [26] separately from SMOTHER).

Finally, we note that practical instruction sequences are unlikely to be identical repeated instructions. However, this is not required for practical SMOTHER side channels: it is only required that, among a sequence of instructions, they exercise different degees of port pressure on the port that the attacker is measuring. We further expand on this idea in section 5 for practical SMOTHER-differentiable sequences.

4 SMOTHERSPECTRE

So far, we have seen how an attacker can identify SMOTHER-differentiable sequences executing on a co-located victim. For a data-dependent conditional jump in the victim's binary whose fall-through and target sequences are SMOTHER-differentiable, identifying which sequence is executed after the jump reveals the outcome of the condition. This, in turn, reveals some information about the data.

SMOTHERSPECTRE is a speculative code-reuse attack technique where the attacker influences the victim into speculatively executing such a conditional jump, leveraging a technique previously used by Spectre's variant 2 (BTI). A separate, indirect jump is found in the usual execution path of the victim. The CPU branch predictor is "poisoned" by the co-located attacker such that when the victim's fetch unit asks for the target of the indirect jump, it is sent the address of the conditional jump. During the subsequent period of speculative execution, the victim will evaluate the conditional jump and run one out of the SMOTHER-differentiable sequences. Concurrently, the attacker will time a sequence of relevant

instructions to identify which sequence the victim is running, thereby completing the information leakage.

SMOTHERSPECTRE complements and extends existing attacks [5, 20, 21] which use cache-based side channels to exfiltrate secrets. Using such channels implies that these exploits *i*) require the presence of *special* gadgets in the victim code, or the ability to inject them; and *ii*) depend on speculative execution leaving persistent, measurable microarchitectural side-effects.

Calls using function pointers in C/C++ are traditionally implemented by indirect calls in assembly. While exploitable indirect jumps are prevalent in most programs, the first observation limits the set of available gadgets for ultimately leaking secrets. This scarcity, along with the overheads of some software-only mitigations, justifies the use of user-space programs to not deploy countermeasures such as retpolines or STIBP by default. In contrast, SMOTHER-differentiable gadgets are easily found (as we demonstrate in section 5) making them prime targets for SMOTHERSPECTRE. Almost every conditional jump can be part of a SMOTHER-gadget, requiring only its fallthrough and target to be SMOTHER-differentiable. Common examples from x86 are `cmp-jx` and `test-jx` sequences. We later discuss (in subsection 4.2) why SMOTHER-gadgets persist in applications which have otherwise moved to eliminate data-dependent control flow in regions of code dealing with secrets (such as cryptographic keys). The reader will notice that we are *not* in presence of a data-dependent control flow anti-pattern. Indeed, the application developer has little control over the existence of SMOTHER gadgets.

The second observation has lead to the proposal of defenses that ensure that *all* changes to microarchitectural state be undone [18]. However, the port-contention based side-channel used by SMOTHERSPECTRE persists even if the CPU were able to perform a perfect roll-back of changes caused by non-retired instructions. The very fact that instructions are speculatively executed remains a measurable quantity. These characteristics allow SMOTHERSPECTRE to present a more powerful avenue of attack.

In this section, we first present the attacker model and objectives for SMOTHERSPECTRE. We then explain the basic premise of the attack, the conditions required and how we ensure these are met in our proof-of-concept. We then present a characterization of the SMOTHERSPECTRE side channel. Later, we discuss practical considerations for real-world attacks to ensure that these conditions are met.

4.1 Attacker model

The objective of a SMOTHERSPECTRE adversary is to extract secret information from a victim process. In the context of the SMOTHERSPECTRE attack, we make the following assumptions about the attacker: *i*) they control code in a process co-located with the victim process; *ii*) they can launch branch
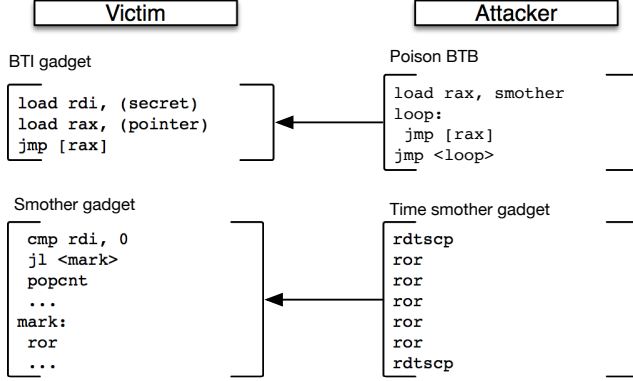
```
   ┌─────────────┐          ┌─────────────┐
   │    Victim   │          │   Attacker  │
   └─────────────┘          └─────────────┘

   BTI gadget                Poison BTB
  ┌───────────────────┐    ┌───────────────────┐
  │ load rdi, (secret)│◄───│ load rax, smother │
  │ load rax,(pointer)│    │ loop:             │
  │ jmp [rax]         │    │   jmp [rax]       │
  └───────────────────┘    │ jmp <loop>        │
                           └───────────────────┘

   Smother gadget            Time smother gadget
  ┌───────────────────┐    ┌───────────────────┐
  │ cmp rdi, 0        │    │ rdtscp            │
  │ jl <mark>         │    │ ror               │
  │ popcnt            │◄───│ ror               │
  │ ...               │    │ ror               │
  │ mark:             │    │ ror               │
  │   ror             │    │ ror               │
  │   ...             │    │ rdtscp            │
  └───────────────────┘    └───────────────────┘
```

Figure 3: Overview of the SMOTHERSPECTRE components.

target injection attacks. The first assumption is justified: if the attacker can execute code on the same machine of the victim, the scheduler may schedule attacker and victim on two different threads of the same physical core. An example of such colocation may exist in public cloud offerings where compute resources are shared at a fine granularity between tenants: for IaaS, virtual cores for different customers may map to the same physical core, for PaaS/SaaS processes for different tenants may be similarly scheduled [4, 11]. As for the second assumption, two main mitigations have been designed against Spectre v2, namely, a new set of interfaces to the CPU to prevent BTI by flushing the indirect branch predictors at the appropriate times, or not sharing them across co-located hyperthreads (IBRS, IBPB, and STIBP on Intel) and retpolines. These mitigations come with a potentially severe performance impact [28]. As such, these controls have been enabled only for selected system components such as the kernel, and none of the user-space programs we have analysed make use of them. The ability to launch BTI attacks also implies that the adversary knows the victim program and that it is able to create an own program with an address space that is adequately congruent to that of the victim. The attacker must thus be able to circumvent ASLR and similar controls: the literature contains several examples [9, 16, 27] of how this is achievable in practice, including an approach using the same BTB weaknesses that make BTI possible.

4.2 Attack principle

Figure 3 shows a side-by-side layout of the code of a victim and an attacker in the SMOTHERSPECTRE setting. As the figure shows, the attack requires two types of gadgets in the victim code:

- A BTI gadget: Stores secret data into memory or a register (called the SMOTHERSPECTRE *target*) followed by an indirect branch that can be poisoned by the attacker;

- A SMOTHER gadget: A data-dependent conditional jump whose control variable is the SMOTHERSPEC-

TRE target, with SMOTHER-differentiable (see subsection 3.2) target and fall-through code paths.

The example BTI gadget in Figure 3 stores the secret into the register `rdi`, a pointer into `rax` and finally jumps to the location pointed to by `rax`. The corresponding SMOTHER gadget contains an `rdi`-dependent conditional branch where the jump target and fallthrough contain SMOTHER-differentiable instruction sequences (`popcnt` and `ror`). In section 5, we investigate the likelihood of such gadgets existing and describe strategies for the attacker to find them.

Note an important difference between traditional data-dependent control flow sequences and SMOTHERSPECTRE. Data-dependent control flow sequences over confidential data are considered vulnerabilities, especially when found in cryptographic libraries. SMOTHERSPECTRE does not require such a vulnerability to be present in the victim. It connects the loading of a secret variable to a register or memory location (BTI gadget) with an altogether independent, speculatively executed sequence, which happens to perform a compare-and-jump over that same register or memory location (SMOTHER gadget). The two sets of instructions may well be entirely uncorrelated from a software development perspective, making the pattern harder if not entirely impossible to eliminate.

The attacker proceeds in two main steps, as shown in Figure 3: in the first phase the attacker performs traditional, Spectre v2 style branch target injection and then enters in a busy wait sequence, for instance a sequence of `nop` instructions. The purpose of the latter is to align the second phase of the attack with the speculative execution of the `mark` or `fallthrough` sequence in the victim. In the second phase the attacker performs a SMOTHER-style timing of a carefully selected sequence of instructions – `ror` in the example. The attacker then proceeds to a statistical analysis of the gathered timing information to learn one bit of information. This entire process can be repeated with different gadgets to leak different bits, and thereby reconstruct the secret. Note that while the example utilizes the indirect-branch prediction hardware to steer speculative execution to gadgets, any existing branch redirection method may be used for this purpose (for example the return stack buffer).

4.3 Characterization of the Side Channel

In order to characterize the SMOTHERSPECTRE side channel we build an experimental test bed which is similar to the one described in subsection 3.2. In particular, an orchestrator process once more forks a victim and an attacker process, pins them to two threads on the same physical core and executes an attacker and a victim process. Attacker and victim process execute the body of a loop after synchronization using atomic operations on shared memory. The body of the loop is constructed as described in Figure 3.

In our proof-of-concept, we leverage the branch target buffer to redirect an indirect branch in the BTI gadget of

the victim to the SMOTHER gadget. In order to maximize the success rate, we *i)* insert a series of N always-taken branches just prior to the indirect branch; *ii)* ensure that the addresses of the branches (including the target of BTI) are located at congruent addresses between attacker and victim; *iii)* disable ASLR. While it is unrealistic to have such perfect conditions in a real-life attack, other works have shown that the random ASLR offset can be leaked [9, 27], and that BTI can be performed by aliasing addresses (in the BTB) with very high success rates [15]. Therefore, we disregard these factors while creating our PoC. In an effort to establish upper bounds for accuracy and throughput for the channel, we further ensure that the secret is in a register, while the branch target address must be retrieved from DRAM, in order to extend the length of speculative execution and amplify the degree of resource contention. These conditions are realistic since it is not unlikely for a program to store secret information in registers (for instance to perform cryptographic operations or compare secret strings), while function pointers are often initialized early, after which they are likely to be evicted from the cache before being used.

We introduce further instrumentation to obtain information about the success of the BTI attack. This information is supplied by the Performance Counter Monitor (PMC) infrastructure and can be obtain by using the `msr` kernel module. We use it to program the PMC counters to retrieve samples for the `BR_MISP_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET` event, which is triggered every time the target of a taken indirect jump is mispredicted. PMC counters are sampled at the start of every loop and once more at their end. BTI is successful whenever the difference in the value of the counter is 1, given that only one indirect jump (the target) is present.

The timed instruction sequence in the attacker consists of a series of 36 `crc32` instructions operating over randomly chosen, nonzero values. Every 6 instructions we interpose a set of shifts and `ors` over the same register operands to avoid zero idiom related pipeline optimizations and RAW hazard. The victim process contains an equivalent sequence of `crc32` instructions at the fallthrough of the branch: given that `crc32` instructions execute exclusively on port 1, if BTI is successful and the speculated conditional branch is not taken, the victim will be competing for execution on port 1 with the attacker. The target of the branch instead contains a sequence of instruction designed to be executable on more ports and thus display less contention with the attacker.

We thus collect two sets of samples: one when the victim's secret is set to zero, and one where it is set to a nonzero value. Figure 4 shows the results of the experiment on a Skylake platform (i7-6700). We obtain similar results on an i5-6200u. As we can see, the distributions obtained when the victim has a nonzero secret generates more contention on port 1 and thus causes the attacker to measure a higher time-stamp counter difference. This is justified by the fact that a nonzero secret causes speculative execution to be directed to the fallthrough
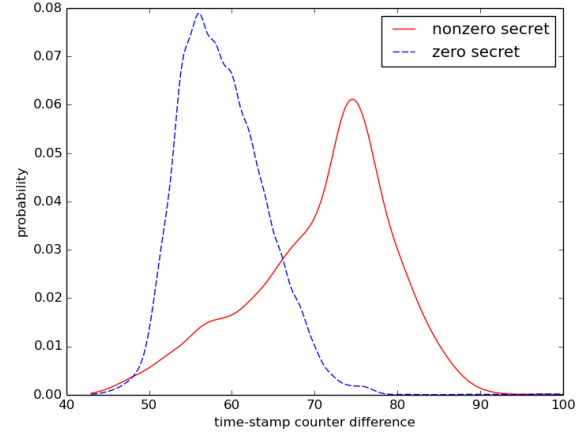

Figure 4: Probability density function for the timing information of an attacker measuring `crc32` operations when running concurrently with a victim process that speculatively executes a branch which is conditional to the (secret) value of a register being zero. The distribution with the blue dashed line shows the case of a zero secret, whereas the red solid line shows the case of a nonzero secret. The probability density function is estimated using Kernel density estimation.

of the branch, which we have designed with a competing sequence `crc32` instructions.

In the next phase of the attack, we use the results of this experiment as profiling information to read the side channel. To this end, a bit sequence is generated and set - bit by bit - as the secret value that is leaked in the experiment. The experiment is run, 100 samples are collected and distribution parameters are generated. Based on the results of Figure 4 we choose a time-stamp counter difference of 66 as a threshold: if the median of the distribution is higher than the threshold we conclude that the secret is 1, and 0 otherwise. We repeat the experiment 64 times: the attack displays a success rate of over 98% even with BTI rates as low as 10%.

As for the channel bandwidth, collecting samples for a single bit, involving fork, pinning, execution, synchronization and 100 repetitions of the attacker/victim loop takes 2*ms* as reported by `time`, yielding a lower bound for the bandwidth of 62*B/s*.

5 Gadget discovery

As described in subsection 4.2, we require two gadgets to be present in the victim code for SMOTHERSPECTRE. We investigate the characteristics of ideal gadgets and how to find them in a given piece of code. We introduce *port fingerprinting* to summarize the port utilization of an instruction sequence and assess the potential to be detected using SMOTHER. Port fingerprinting enables a comparison of the port utilization of two instruction sequences and rank combinations of instruction

sequences based on their difference in port utilization.

BTI Gadget. The purpose of the BTI gadget is to pass the secret through a register to an arbitrary code target in the same process. Depending on the attack scenario, the BTI gadget is the only piece of code that is strictly required to be present in the victim. Ideally, it just consists of two instructions: one that moves the secret into a register and an indirect control-flow transfer. In order to maximize the speculative execution window, the target of the indirect control-flow transfer should be retrieved from uncached memory. An archetype of an ideal BTI gadget is a virtual function call in C++, with the secret value being an argument to such a function call. In the x86_64 calling convention, the first six parameters of a function are passed in registers. Further, the typical implementation of a virtual function call uses indirection through a vtable to resolve the binding at runtime. Since the vtable is stored in memory, the target of the call needs to be loaded, which causes a speculation window of a few hundred cycles if the vtable has been evicted from the cache prior to the call. We can reasonably assume that this will happen in practice if objects are created by an early initialisation phase and used (potentially much) later in response to external events.

SMOTHER Gadget. A SMOTHER gadget is the receiving end of a BTI gadget. Depending on the attack scenario, it is either already part of the victim, or can be supplied via an additional attack vector. It starts with an instruction that compares the register to a known value. The known value can either be a known immediate in the code, or, more powerfully, an attacker-controlled value specified via an additional attack vector. The next instruction needs to be a conditional control flow transfer based on this comparison. It leads to two branches, which need to be composed of an instruction sequence that allows to be leaked via the SMOTHER port contention side channel. To maximize the chances of SMOTHER-differentiability, the instruction sequences should each have a distinct port fingerprint such that they can be clearly distinguished from one another. Of course this depends on the layout of the execution engine: on Intel Skylake, a prime example would be one branch with a sequence of AES instructions (only available on port zero) and another branch with a sequence of MMX instructions, predominantly limited to port five. Besides, the instructions should ideally not load from or store to memory, as potential cache misses introduce noise. Further, the more generic the instructions in the sequence are, the more likely it is that their execution unit does not require a warm-up phase during which execution is slow, again introducing noise.

5.1 Ranking SMOTHER Gadgets

The instruction sequences we consider consist of basic blocks that start at the respective branch targets. To identify instruction sequences that are ideal for SMOTHER and compare them against one another, we need to measure their suitability for

SMOTHER. The primary criterion is that the compare instruction operand has to match the register that is loaded with the secret in the BTI gadget. Further, we evaluate the instruction sequence at the branch target and fallthrough by quantifying three properties: *i)* the port utilization difference of the two branch targets (r_p), *ii)* the difference of the two branch targets in terms of the length of the branches (r_l), and *iii)* the amount of memory operations in both branches (r_m). To compare instruction sequences based on these properties, we combine them using the rank product $RP(g) = (\prod_{i=1}^{k} r_{gi})^{1/k}$ for our $k (= 3)$ properties.

To compare the port utilization, we first use Intel's Architecture Code Analyzer (IACA) to obtain a port fingerprint P for a given instruction sequence. The port fingerprint is a summary that lists the total number of cycles spent on every port for a given instruction sequence $P = p_0 \ldots p_7$. IACA internally uses a microarchitecture-specific model of the processor to compute the cycles, taking out-of-order execution into account. It also models the divider pipe on Skylake, allowing port zero, which handles the complex div instruction, to be ready for the next μop in the next cycle, while the div is still being executed. As it cannot know better, IACA assumes all CPU resources to be fully available prior to execution of the sequence. An open-source alternative to IACA, OSACA [22] also supports AMD processors.

To compare two port fingerprints P and Q, we subtract them and then calculate the utilization difference as the sum over the vector: $r_p = \sum_{0..7}^{i} (|p_i - q_i|)$. The larger r_p, the higher the difference in port utilization of the two instruction sequences. The utilization difference will be high for long instruction sequences that do not share a port. Such instruction sequences lend themselves well to SMOTHER.

While a ranking based on the port utilization difference already captures the most important aspect, it has one drawback: gadgets where the branch instruction sequences are of different length, such as 2 instructions vs. 20 will rank high, whereas we prefer sequences of equal length for the timing. Therefore, we also include the inverse of the length difference $r_l = abs(l_1 - l_2)$ between the sequences of a gadget in the ranking.

Finally, we also take the potential noise into account that can be caused by memory operations. We include, the inverse of the sum r_m of the cycles spent on ports 2, 3, 4 and 7 in both branches as an additional ranking for the gadget. The final rank of a gadget g_i is given by $RP(g_i) = (r_{p_i} \cdot (max(r_l) - r_{l_i}) \cdot (max(r_m) - r_{m_i}))^{1/3}$.

5.2 Finding Gadgets

We develop a tool to aid gadget discovery based on the popular distorm3 disassembler and Intel's Architecture Code Analyzer, and use it to analyze a number of common system libraries that are likely to be linked to a victim executable. For the analysis we only consider gadgets with a branch length

	RDI	RSI	RDX	RCX	R8	R9
glibc 2.27	14/28	25/34	115/21	44/16	28/8	24/2
libstdc++ 6.0.25	0/10	3/3	18/4	33/1	6/1	2/1
ld 2.27	3/5	7/13	11/7	4/4	1/0	0/0
pthread 2.27	0/2	0/1	1/0	0/0	2/4	2/0
libz 1.2.11	0/2	0/0	2/0	5/0	7/0	4/0
libcrypto 1.1	7/19	15/6	25/11	13/5	16/2	9/1
libssl 1.1	1/5	1/3	10/1	10/0	2/0	0/0

Table 2: SMoTHER gadgets we found in common system libraries for the registers used to pass arguments in the x86_64 calling convention. We list both the number of SMoTHER gadgets that use the value in the register directly, as well as the number of those that compare against its pointee.

between 3 and 70 instructions, with 3 instructions being a reasonably low bound for smothering and 70 instructions being an upper bound for speculative execution. We show the results in Table 2, the libraries analyzed are taken from a regular Ubuntu 18.04 LTS installation. We focus on SMoTHER gadgets that compare against the registers used in the x86_64 calling convention and either use the value in the register directly, or use it as a pointer and compare to a value pointed to in memory. The rationale behind this is that BTI gadgets are typically indirect calls that pass a secret, such as a cryptographic key, as a parameter. The results show that we can find enough SMoTHER gadgets even in a single common library such as `glibc` alone. Note that this method applies irrespective of whether the library is loaded at runtime or is statically linked into the victim's binary.

6 Real world attack

For our real-world attack, we target a BTI gadget from OpenSSL's `libcrypto` (commit `f1d49ed`, dated 27-Nov-2018) library which is widely used for performing cryptographic functions and a SMoTHER gadget from `glibc` version 2.23.

Over the years, considerable effort has been invested to thwart potential attackers and to protect OpenSSL from side-channel attacks, primarily by removing data-dependent memory-access or control flow. Our attack, however, targets BTI gadgets (indirect jumps or calls) that are found in code used to choose between encryption modes, allowing for multiple modes of operation with the same block cipher. For example, OpenSSL features more than eight modes of operation for AES, including electronic codebook (ECB) and cipher-block chaining (CBC) modes. Such gadgets are the result of commonly used coding practices, and do not directly perform any data-dependent actions based on the secret value. In our attack, the use of function pointers leads to indirect calls in the compiled binary. The arguments to these function calls often contain secrets or pointers to secrets. Arguments are loaded into registers prior to the call as per the relevant calling convention, allowing us to learn information about

their values in our SMoTHER gadget.

OpenSSL uses a context variable that stores pointers to functions for encryption/decryption. These pointers are set during the initialization phase by allowing the user to choose which cipher mode they wish to use. We target indirect calls that use these function pointers by poisoning the branch predictor to point to our intended SMoTHER gadget instead. Specifically, we use calls to the cipher from `EVP_EncryptUpdate`, shown in Figure 5c. The third argument (`in`) contains a pointer to the plaintext to be encrypted. As per the System V calling convention, this pointer is stored in register `rdx` prior to the call. The secret in our chosen SMoTHER gadget is the first byte of the plaintext, referenced through `rdx`.

An abridged version of the SMoTHER gadget is shown in Figure 5b (see Appendix A for the full assembly listing). Our chosen SMoTHER gadget differs slightly from that described in subsection 4.2 in that it compares the value of a memory location pointed to by a register, not the value of the register itself. The target and fallthrough path differ in utilization of execution ports 0 and 6. The attacker times a sequence of `btr,bts` (Figure 5a) to specifically target the same ports. This gadget is taken from `glibc` and demonstrates the availability of SMoTHER gadgets in commonly linked libraries. Further, the expected difference in utilization of ports 0 and 6 between the two paths is a couple of cycles. Successfully using SMoTHER on such a gadget demonstrates the power of our attack methodology.

In our attack, we model a victim that encrypts text using OpenSSL's EnVeloP (EVP) API. After performing the necessary initializations, it performs a series of calls to `EVP_EncryptUpdate`. We have statically linked the victim with a slightly modified version of `libcrypto`. The first modification serves to flush the address holding the call pointer from the caches. This causes the victim to use the prediction from the BTB, making branch target injection reliable. The other modification helps synchronize the attacker with the victim using atomic operations on shared memory, similar to the prototype from subsection 4.3. Finally, we have instrumented the victim to setup relevant performance counters. We use these counters for statistical and monitoring purposes only and not directly for the attack. We will release all attack proof of concept implementations as open-source.

In a more realistic scenario, an attacker can ensure that certain variables are not cached by selectively polluting cache sets or forcing the victim along computational paths that would eventually evict the pointer from the caches. Such approaches have been described by previous attacks [25]. We have also found the time taken by the victim to reach the indirect call from the library entry to be highly predictable. This could provide the attacker an alternate method for synchronizing the timed section with the execution of the SMoTHER gadget.

The attacker runs code that is practically identical to the

```
.rept 8;              0x89190: cmpb  0x0, (rdx)    if(ctx->cipher->do_cipher(ctx, out,
btrl r8d, r9d;        0x89193: je    0x8936e                               in, inl))
btrl r10d, r11d;      0x89199: movq  rcx, r13      {
btsl r8d, r9d;        ...                               *outl = inl;
btsl r10d, r11d;      0x8936e: add   0x88, rsp         return 1;
.endr;                ...                           }
                                                    ...
```

(a) Attacker-timed code   **(b)** Victim SMOTHER gadget from `glibc`   **(c)** Victim BTI gadget from OpenSSL

Figure 5: Gadgets from real-world libraries used in our SMOTHERSPECTRE exploit.
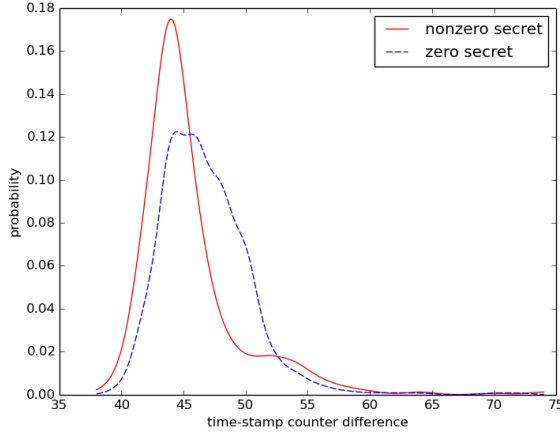
Figure 6: Probability density function for the timing information of an attacker when running concurrently with a victim process that speculatively executes a branch which is conditional to the (secret) value of a memory location being zero. The distribution with the blue dashed line shows the case of a zero secret, whereas the red solid line shows the case of a nonzero secret. The probability density function is estimated using Kernel density estimation.

victim apart from the following differences. First, it loads the call pointer with the location of the SMOTHER gadget on the victim. This is to trigger BTI on the victim process. Second, it replaces the code at the target location by the SMOTHER timing sequence. Otherwise, the attacker runs code that mimics the victim: it performs the same call to the encryption function where it follows the same sequence of checks and jumps. It also runs in a loop performing the same number of iterations. This increases the probability of the attacker having the same branching history as the victim at the call site, thereby increasing the success rate of BTI.

6.1   Results and Discussion

We ran the attack on an i5-6200u CPU with the attacker and victim pinned to the same physical core. A run of 1,000 encryptions was performed by the victim. The attacker succeeded in BTI with a success rate of around 55%. Figure 6

shows the distribution of timestamp counter difference measured by the attacker for the SMOTHER gadget. The distributions show a significant variation, with that corresponding to the zero-secret tending towards higher values. The Student's t-distribution test is able to successfully distinguish between them with 95% confidence.

6.2   Mitigating SMOTHERSPECTRE

Mitigations for SMOTHERSPECTRE can be subdivided in two categories: mitigations for SMOTHER and mitigations for BTI.

SMOTHER mitigations. The general idea of preventing SMOTHER attacks is to ensure that two threads with different privileges (in the general sense) do not compete for the same execution port.

Currently available software SMOTHER mitigations are limited. Apart from the straightforward but performance-costly possibility of disabling SMT in its entirety (up to 10-15% overhead on Intel), the OS scheduler can employ a side-channel aware strategy. For example, the OS scheduler can decide to only colocate (on threads on the same core) processes from the same user

Finally, CPU-level mitigations could be deployed in the future, possibly improving both security and performance over existing mitigations. For instance, alternatives to SMT can be considered to achieve thread-level parallelism within a core. These include coarse-grained and interleaved multithreading.

BTI mitigations. Mitigations against branch target injection are also known as Spectre v2 mitigations. These include retpolines, which rewrite code to remove indirect calls [30], as well as CPU-based controls. Intel has exposed to developers a set of security controls that limit an attacker's ability to perform BTI. While they have been applied in selected cases, they have not been widely adopted because of their overhead [7], and because in many cases, the required gadgets were simply not present [20]. Wide adoption of these mitigations may limit the SMOTHERSPECTRE attack.

Summary. Fully mitigating the attack in either of these two categories is sufficient to stop the attack presented in this paper. However, SMOTHERSPECTRE does not necessarily need to employ BTI: it can be generalized to use any other form

of speculative control flow hijack, e.g., RSB overflow [24] or speculative return address overwrite [19]. In those cases, corresponding mitigations would apply.

7 Related Work

Transient Execution Attacks. Transient execution attacks exploit instructions that are executed, yet not necessarily retired and thus cover both attacks based on speculative execution as well as out-of-order execution [5].

At the beginning of 2018, two security issues exploiting speculative execution were revealed under the name "Spectre" [15, 20]. Spectre V1 ("Bounds Check Bypass") exploits branch prediction on a conditional branch to achieve an out-of-bounds access during speculative execution: given a conditional branch that performs a bounds-check on an array, the branch predictor is trained to the in-bounds case by performing multiple executions of the corresponding code with a benign index. When the code is then executed with an out-of-bounds index, a misprediction occurs and the array access with the malicious index is performed. If the result is used in further computation such as another array access, it can be leaked through a side channel. Spectre V2 ("Branch Target Injection") exploits branch prediction on indirect control-flow transfers. To this end the attacker first trains the branch predictor for a given address to transfer control to an address of the attacker's choosing. The predictor will then use the branch history created by the attacker for a spatially or temporally co-located victim. Again, a cache side-channel can be used to leak data of the attackers choosing in the following. The return stack buffer, which is used for return statements in a similar fashion as the branch history is used for indirect jumps has also been leveraged as a speculative execution trigger [21, 24]. The return address on the stack has also been the target of other work, showing that through load-to-store forwarding it can be speculatively overwritten, leading to a speculative execution sibling of the classic stack buffer overflow [19].

Meltdown [23] ("Rogue Data Cache Load"), which was also revealed in early 2018 exploits out-of-order execution: a memory load instruction immediately after a high latency instruction might fetch data into the cache even if it is not permitted to access the actual memory location. The reason is that on certain CPUs, the corresponding permission check is not on the critical path for the data fetch and the exception is only triggered after the data fetch. On such CPUs this allows reading arbitrary kernel memory from userspace. Similarly, also privileged system register can be read ("Rogue System Register Read"). The more recent Foreshadow [31] attacks a similar phenomenon, "L1 Terminal Fault" in Intel nomenclature. If an instruction accesses a virtual address that is not in the translation lookaside buffer (TLB) and the corresponding page table entry's (PTE) present bit is not set, this is referred to as a "terminal fault". During out-of-order execution, the processor computes a physical address from the PTE, which

is used for a lookup in the L1 data cache. Until the instruction retires and a page fault is raised, cached data is forwarded to dependent instructions, which can be used in a side channel. This bypasses various access checks, including SGX protection, extended page table address translation and system management mode (SMM) checks, thus affecting virtualization and SGX enclaves (enclave data is not encrypted in L1D). Also related to out-of-order execution is the speculative store bypass [3, 17]: for a code sequence of a dependent store and a load instruction, the load instruction, if executed out-of-order before the store might retrieve stale data from memory that can be used in a side channel. This happens in cases where the CPU cannot detect the dependency in the code sequence.

Transient execution attacks are not only a local security issue that requires a victim device to execute attacker-controlled code. As Netspectre [26] demonstrates they also work remotely. While being less effective, they are still powerful enough to break for example address space layout randomization.

Cache Side Channels. Cache side channels leverage timing differences in accesses to memory locations based on whether the data in those memory locations is cached or not. Accesses to cached locations will be faster, whereas accesses to uncached locations will be slower, as the data needs to be fetched from main memory. This principle applies to both regular data as well as instructions: Execution of code whose instructions are not cached will take longer than execution of cached code.

To use an *evict-and-time* cache side channel one first primes the cache by executing a victim function and then measures how long the function takes to execute – this is the baseline run. One can now compare this baseline against further executions of the function, with different cache sets evicted. If the time the function takes to execute is slower than the baseline, the victim function depends on the evicted cache set.

To use a *prime-and-probe* cache side channel, one first primes the cache with known attacker-controlled addresses. One then waits for the victim code to run. Afterwards one measures the access time to addresses used for probing: it will be low for addresses touched by the victim code and high for others. The difference to evict-and-time is that the attacker measures her own operation in contrast to the execution of victim code. Both *evict-and-time* and *prime-and-probe* have been extensively used to attack AES implementations [25, 29].

Another technique that became popular with attacks leveraging a shared last-level cache (LLC) is *flush-and-reload*. It requires an instruction that allows an attacker to flush a certain cache line, such as `clflush` on x86_64. In a corresponding attack, the attacker first flushes a cache line and then waits for the victim code to execute. Afterwards the attacker times the access to the address, which will be fast if the victim accessed (reloaded) it and slow otherwise. Flush-and-reload is similar to prime-and-probe, but much more fine-grained as individual cache lines can be targeted. It has been used

to leak information from the LLC, which is typically shared among multiple CPU cores [33]. Related to flush-and-reload, *flush-and-flush* [14] is based on the observation, that `clflush` will take less time to execute when it is run on a location that is not cached. The advantage over flush-and-reload is that no actual access that would pull data into the cache is performed, making the attack stealthier.

Finally, *prime-and-abort* leverages Intel's transactional memory mechanism to detect when a cache set has been evicted without the need to probe the cache [8]. In contrast to all previous cache side channels, it does not need to time an operation. Transactional memory operations require transactional data to be buffered in the cache which has limited space. A transaction set up by the attacker will abort if the victim accesses a critical address.

Other Side Channels. Mitigations against cache-based side channels have led researchers to explore other shared resources as well. TLbleed [12] shows how the TLB can be used as a side channel to leak a cryptographic key. Aforementioned Netspectre-AVX [26] uses a side channel based on AVX instructions. This side channel exploits the fact that the execution unit processing those instructions employs aggressive power saving. When such instructions have not been used for a long time, they execute much slower.

In particular, execution-unit-sharing-based side channels in the SMT settings have been studied as early as in 2006: Wang and Lee [32] demonstrate a multiply-based covert channel making use of contention on execution units. Aciicmez and Seifert [1] extend this work by analyzing its applicability as a side channel. Anders Fogh [10] proposes a generalized result by analyzing contention results of the cross product of 12 curated instructions. Finally, Portsmash [2], concurrently and independently demonstrates how port contention can be used to leak sensitive cryptographic material from OpenSSL. Portsmash relies on a known vulnerable implementation of OpenSSL, and therefore does not require any mitigation beyond avoiding vulnerable code patterns. In contrast, SMOTH-ERSPECTRE does not require a secret-dependent control flow by combining port contention with BTI, and thereby showing broader applicability of the port contention side channel. Finally, in contrast with all previous works, this work provides a characterization of this side channel, including an analysis for low number of victim instructions.

8 Conclusion

We further our understanding of possible attacks in the space of speculative execution. This is crucial to design suitable defenses and to apply them to the right systems. In particular, we show that Branch Target Injection attacks against applications that do not load attacker-provided code are feasible, by crafting an exploit for OpenSSL. To this end, we present a precise characterisation of port contention, the non cache-based side channel we use for the attack, and develop a tool to help us find suitable gadgets in existing code. We will open-source our proof of concept implementation, gadget finder, as well as the data of our measurements to enable others to study this interesting side channel. As a consequence, it is now clear that in SMT environments defenses solely relying on mitigating cache side channels, or solely relying on reverting microarchitectural state after speculative execution, are insufficient.

In the immediate future, implementing existing BTI mitigations is sufficient to prevent SMOTHERSPECTRE. Future work may mitigate such attacks with lower performance overhead and better security guarantees, for instance through side-channel resistant ways of designing thread-level parallelism in upcoming CPUs.

References

[1] Onur Aciicmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*, pages 80–91. IEEE, 2007.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. Cryptology ePrint Archive, Report 2018/1060, 2018. `https://eprint.iacr.org/2018/1060`.

[3] AMD. Speculative store bypass disable. `https://developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf`, 2018.

[4] Zack Bloom. Cloud computing without containers. `https://blog.cloudflare.com/cloud-computing-without-containers/`, 2018.

[5] 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.

[6] Intel Coorporation. Intel 64 and ia-32 architectures optimization reference manual, 2016.

[7] Jonathan Corbet. Taming stibp. `https://lwn.net/Articles/773118/`.

[8] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel TSX. In *USENIX Security Symposium*, 2017.

[9] 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.

[10] Anders Fogh. Covert shotgun. `https://cyber.wtf/2016/09/27/covert-shotgun/`.

[11] Google compute engine faq. `https://cloud.google.com/compute/docs/faq`. Accessed: 2019-02-13.

[12] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security Symposium*, 2018.

[13] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[14] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[15] Jann Horn. Reading privileged memory with a side-channel. *Project Zero*, 3, 2018.

[16] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[17] Secure Windows Initiative. Speculative store bypass. `https://blogs.technet.microsoft.com/srd/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass-cve-2018-3639/`, 2018.

[18] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. *arXiv preprint arXiv:1806.05179*, 2018.

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

[20] 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.

[21] 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.

[22] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. Automated instruction stream throughput prediction for intel and amd microarchitectures. `https://arxiv.org/abs/1809.00912`, 2018.

[23] 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.

[24] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *Conference on Computer and Communications Security*, 2018.

[25] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Topics in Cryptology*, 2006.

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

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

[28] Linus Torvalds. Linus on spectre/meltdown mitigations. `https://lkml.org/lkml/2018/1/21/192`, 2018.

[29] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 2010.

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

[31] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[32] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22Nd Annual Computer Security Applications Conference*, ACSAC '06, pages 473–482, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*, 2014.

A OpenSSL gadgets

A.1 SMOTHER gadget

```
0x89190 <__argz_replace+48>:  cmpb   0x0, (rdx)
```

```
0x89193 <__argz_replace+51>:   je     0x8936e
                                      <__argz_replace+526>
0x89199 <__argz_replace+57>:   movq   rcx, r13
0x8919c <__argz_replace+60>:   movq   (rdi), rcx
0x8919f <__argz_replace+63>:   movq   rdx, rdi
0x891a2 <__argz_replace+66>:   movq   0x0, 0x60(rsp)
0x891ab <__argz_replace+75>:   movq   0x0, 0x68(rsp)
0x891b4 <__argz_replace+84>:   lea    0x70(rsp), r12
0x891b9 <__argz_replace+89>:   xorl   r15d,r15d
0x891bc <__argz_replace+92>:   movq   rcx, 0x18(rsp)
0x891c1 <__argz_replace+97>:   movq   (rsi), rcx
0x891c4 <__argz_replace+100>:  movq   rcx, 0x30(rsp)
...
0x8936e <__argz_replace+526>:  add    0x88, rsp
0x89375 <__argz_replace+533>:  mov    ebx, eax
0x89377 <__argz_replace+535>:  pop    rbx
0x89378 <__argz_replace+536>:  pop    rbp
0x89379 <__argz_replace+537>:  pop    r12
0x8937b <__argz_replace+539>:  pop    r13
0x8937d <__argz_replace+541>:  pop    r14
0x8937f <__argz_replace+543>:  pop    r15
...
```