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 hundreds of gadgets that can be used to leak information. Finally, we demonstrate proof-of-concept attacks against the OpenSSH server, creating oracles for determining four host key bits, and against an application performing encryption using the OpenSSL library, creating an oracle which can differentiate a bit of the plaintext through gadgets in `libcrypto` and `glibc`.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

KEYWORDS

side-channel; simultaneous multithreading; speculative execution; attack; microarchitecture

[*]firstname.surname@epfl.ch
[†]{asa, eug, aso, kur}@zurich.ibm.com

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363194

ACM Reference Format:
Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus . 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *2019 ACM SIGSAC Conference on Computer & Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3363194

1 INTRODUCTION

Spectre [23, 24, 29] and Meltdown [26] 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 [15], retpolines [33], 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 [21].

One mitigating factor is that so far, with the exception of Netspectre-AVX [29], 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 [23]: 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, 11, 35]. 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 an up-to-date OpenSSL version. Further, we attack the OpenSSH server, leaking bits from the host's RSA key.

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
- Real world attacks where we target BTI gadgets in the OpenSSH server and in the latest version of OpenSSL, along with 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 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.

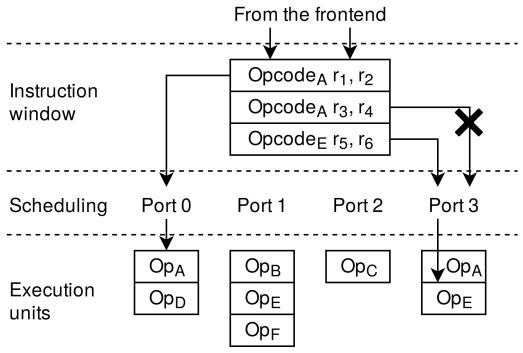


Figure 1: Instructions from the window are scheduled to ports shared by sets of execution units. A single instruction may be scheduled per port per cycle.

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 *re-order 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 can be shared between hyperthreads [6, 18] 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-timing 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 micro-architectural 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.

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 μop 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 μop 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. Instruction-to-port mappings are available through reverse engineering [10] or the Intel Architecture Code Analyzer (IACA) tool.

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 experiments in later sections of this paper, we shall establish various pairs of sequences to be SMoTher-differentiable. After collecting attacker timings distributions for each victim sequence in our controlled environment, we shall use the Student's t-test to establish statistical difference between them with at-least 95%-confidence. We argue that an attacker, in an adversarial scenario, can correlate its own timing distribution with either of the a-priori distributions to identify the victim sequence.

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` (port 1) or `ror` (port 06) 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` (port 06) instructions or `popcnt`. In this case, the attacker times a sequence of `bts` (port 06) instructions with both operands as registers.

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.

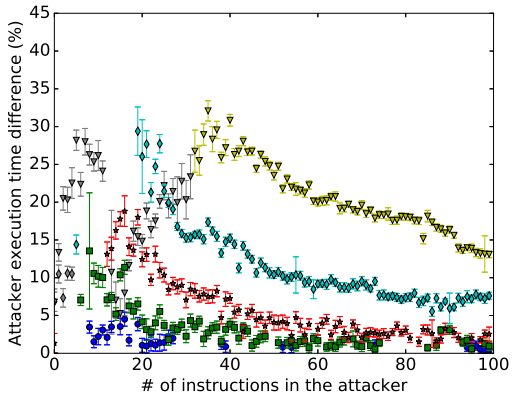
Results. Figure 2 plots the average difference in attacker execution time between the two sequences of victim instructions for each experiment. The length of the sequence for the victim was taken from the set $\{1, 4, 8, 16, 32\}$ while the attacker sequence varied in length between one and 100 instructions.

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`.

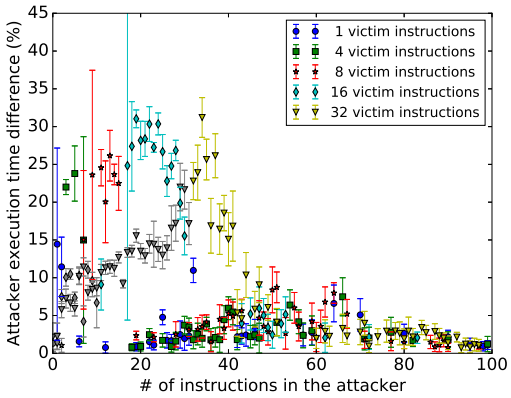
The measured differences vary from close to 0% to 40%. Longer sequences of instructions in the victim lead to higher differences and less variability in measurements. Only 48% of `popcnt` measurements with sequence of 1 victim instruction are SMoTher-differentiable, as opposed to 83% for a sequence of 4, and 100% for a sequence of 32 victim instructions. 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



(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.

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 [29] 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

SMoTherSpectre is a speculative code-reuse attack technique which starts at an indirect jump on the victim's usual execution path. The attacker leverages Branch Target Injection (BTI) to "poison" the CPU's branch predictor such that when the victim's fetch unit asks for the target of the indirect jump, it is sent the address of a separate data-dependent conditional jump within the victim's binary with SMoTher-differentiable fall-through and target sequences. During the period of the speculative execution, the victim evaluates the condition and jumps to either the target or fall-through sequences. The attacker times a sequence of relevant instructions to identify which sequence is run on the victim (SMoTher), thereby inferring the outcome of the condition and learning some information about the victim's data.

SMoTherSpectre complements and extends existing attacks [5, 23, 24] 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). Almost every conditional jump can be part of a SMoTher-gadget, requiring only its fall-through and target to be

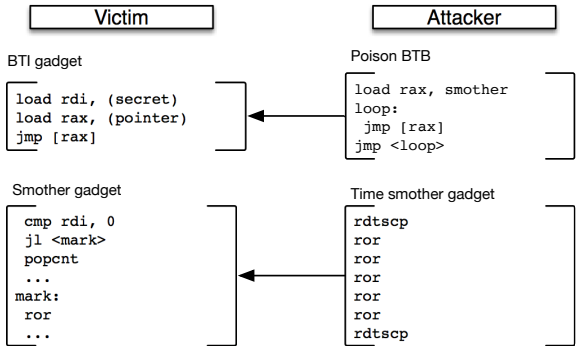


Figure 3: Overview of the SMoTherSpectre components.

SMoTher-differentiable. For example, `libcrypto` from the OpenSSL library contains more than 12,000 readily usable gadgets.

The second observation has lead to the proposal of defenses that ensure that *all* changes to microarchitectural state be undone [21]. However, the port-contention based side-channel 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. Finally, we discuss the characteristics of some SMoTher-gadgets we found in common system libraries, and what information they may be used to leak.

4.1 Attacker model

The objective of a SMoTherSpectre adversary is to extract secret information from a victim process and 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 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 the 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, 13].

Existing mitigations against BTI include software (retpolines) and a set of hardware interfaces for flushing the indirect branch predictors at the appropriate times and for not sharing them across SMT threads (IBRS, IBPB and STIBP in Intel). These mitigations come with a potentially severe performance impact [31]. 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 adversary also needs to know the victim's code base, which is possible through the use of common libraries and open-sourced applications, and where it is located in memory. It must be able to circumvent ASLR and similar controls: the literature contains several examples [9, 19, 30] 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 SMoTherSpectre target, with SMoTher-differentiable (see Section 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 fall-through contain SMoTher-differentiable instruction sequences (`popcnt` and `ror`).

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 BTI 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 `fall-through` 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 our experimental testbed to characterize the SMoTherSpectre side channel, an orchestrator process 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 (similar to the testbed in

Section 3.2). Attacker and victim processes 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; *iv*) evict the cache-line containing the indirect jump pointer. As other works have shown, the random ASLR offset can be leaked in a real-life attack [9, 30], and BTI can be performed by aliasing addresses (in the BTB) with very high success rates [18]. Therefore, we disregard these factors while creating our proof-of-concept (PoC). Evicting the jump pointer allows us to extend the duration of the victim's speculative execution, in order to establish an upper bound for accuracy and throughput for the channel. In alternate settings, we have noticed that usual victim computation can evict the pointer from the L1 cache. The resulting period of speculation is enough for our attack to work.

Further, we introduce 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 obtained 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 the victim code contains only one indirect jump.

The timed instruction sequence in the attacker consists of a series of 42 `crc32` instructions operating over randomly chosen, nonzero values. The victim process contains an equivalent sequence of `crc32` instructions at the fall-through 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 (0,1,5,6) and thus display less contention with the attacker.

We 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 (i7-6700K). 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 fall-through 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 on the victim. Based on the results of Figure 4 we choose a time-stamp counter difference of 89 as a threshold: if the mean of the samples is higher than the threshold we conclude that the secret is 1, and 0 otherwise. The experiment is run, 5 samples each

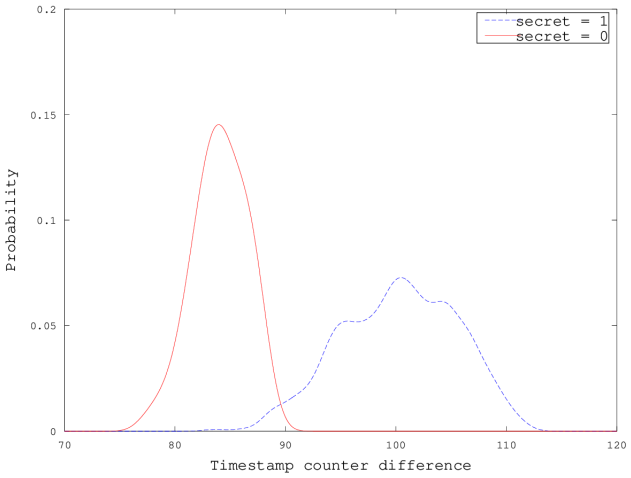


Figure 4: Probability density function (estimated using kernel density estimation) for the timing information of an attacker measuring `crc32` operations when running concurrently with a victim speculatively executing a SMoTher-gadget.

	glibc	ssl	pthread	ld	crypto	z	stdc++	Together
rax	14	12	9	7	11	8	8	21
rbx	6	2	0	1	6	1	1	9
rcx	8	1	2	1	5	1	2	8
rdx	10	2	5	6	7	2	3	14
rsi	8	4	1	2	3	1	1	10
rdi	8	2	0	2	3	1	0	11
rsp	2	0	0	0	0	0	2	3
rbp	5	3	0	0	9	0	0	13

Table 2: Number of different leakable register bits (out of 64) using SMoTher gadgets from common system libraries, specifically `test-jxx` one at a time, on multiple iterations of the victim with the same register state.

are collected for 20,000 secret bits. The attacker is able to correctly guess the victim's secret with a success rate of over 98%. The entire experiment takes 0.83*s* as reported by `time` yeilding a sample rate of 120,000 samples/second, and a leakage rate of 24,000bit/s. A similar experiment with guesses based on 1, 2, 3 and 4 samples result in accuracies of 72%, 78%, 83%, and 90% respectively. As expected, there is a trade-off between accuracy and leakage rate.

We repeated this experiment on a Haswell processor (i7-4790) using the same attacker-timed sequence and victim's SMoTher-gadget. With a threshold of 85 cycles, the attacker was able to guess the victim's secret bit with an accuracy of 53%, 62%, 69%, 70% and 76% based on 1, 2, 3, 4 and 5 samples respectively. We also validated the attack on a Broadwell processor (i3-5005u).

4.4 Discussion about SMoTher-gadgets

A SMoTher gadget is defined by two attributes: the condition on which the jump depends and the sequences on both paths following the jump. The latter determines whether the gadget can be used

in a SMoTherSpectre attack: the sequences must be SMoTher-differentiable. The former determines the information leaked by the gadget. In this section, we shall discuss some of the SMoTher-gadgets found in real libraries, and what they can leak.

Common instructions which set the condition flags in SMoTher-gadgets we found are `cmp`, `test`, `add`, `sub`. `cmp-jxx` sequences compare a value in a register (or loaded from memory) against other registers or against a constant. Each gadget reveals a constraint on the value. `test-jxx` and `and-jxx` sequences perform a bitwise-and of two values, setting flags based on the result. When one of the values is a constant with a single bit set, the gadget can be used to test whether specific bits are set in the first operand. Such gadgets reveal the corresponding bits to the attacker. When the second operand is not a constant, but a register whose value may be predicted or controlled, the attacker gains the power to check bits other than those specified by the constant gadgets.

Of the over 12,000 gadgets found in `libcrypto`, approximately:

- 2,800 are `cmp` operations
- 3,900 are `test` operations
- 1,500 are `add` operations
- 970 are `or` operations

There are around 350 `cmp-jump` gadgets which compare a value in a register or in memory against zero. Around a 100 gadgets, which check for greater-than/lesser-than conditions against zero, can be used to leak whether the value is positive or negative. Another 294 gadgets compare against (the constant) one, and 807 gadgets compare against other constants. Around 370 `cmp-jump` gadgets have a memory operand of which 118 compare with non-zero constants. 300+ gadgets compare with values on the stack, of which 33 are against non-zero constants. Of all `cmp-jump` gadgets, more than 400 check for signed or unsigned greater-than. The number of signed or unsigned lesser-than is about the same.

For victims running in a loop, there are cases where the register/memory state of interest will be the same across iterations. For example, a register/memory location storing a secret, cryptographic key can be expected to hold the same value across multiple calls to the encryption function. The attacker can leverage the BTI gadget to redirect the victim to different SMoTher-gadgets on different iterations, each time leaking different information about the secret. Over multiple iterations, the attacker can effectively leak multiple bits of the same secret, chaining the leaks from different SMoTher-gadgets. In Table 2, we show how many bits we can leak from the registers by chaining SMoTher-gadgets found in commonly used system libraries. To illustrate this for one specific register, Appendix A lists the gadgets which can be chained to leak 21 bits from `rax`.

5 GADGET DISCOVERY

As described in Section 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 System V 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 can cause a speculation window of upto a few hundred (~200) 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. Similarly, calls to functions in dynamically-loaded ELF (Executable and Linkable Format) libraries also employ an indirect jump, using a pointer from the Global Offset Table (GOT) to facilitate dynamic symbol resolution. Arguments in such calls may contain sensitive information which can be compromised by an attacker using these jumps as BTI gadgets.

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 leading to SMoTher-differentiable sequences. 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. 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 port 0) 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 fall-through 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 $P = p_0 \dots p_7$. 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 [25] 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. On our targeted Skylake processors, the ports 2, 3, 4 and 7 are used for scheduling these. We include the inverse of the sum, r_m, of the cycles spent on these ports 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 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. Our search looks for valid instruction sequences starting at every offset in the binary. Therefore, it would detect any SMoTher-gadget resulting from an unintended sequences of bytes (starting from the middle of an intended instruction) which might decode to valid instructions. We show the results in Table 3, the libraries analyzed are taken from a regular Ubuntu 18.04 LTS installation. We focus on

	RDI	RSI	RDX	RCX	R8	R9
glibc 2.23	1155	1502	3864	4256	568	615
	1040	932	257	1029	135	29
stdc++ 6.0	189	400	869	1399	97	73
	209	65	98	276	58	14
ld 2.23	105	130	412	359	41	31
	46	47	29	110	6	0
pthread 2.23	23	56	70	82	25	8
	23	2	7	34	3	0
z 1.2.11	76	85	138	338	66	80
	24	29	8	96	16	5
crypto 1.1	1132	1048	1659	2566	45	29
	310	319	224	1036	239	167
ssl 1.1	243	239	376	500	39	21
	95	32	29	239	12	1

Table 3: SMoTher-gadgets we found in common system libraries, for the registers used to pass arguments in the System V x86_64 calling convention. First line: number of SMoTher-gadgets that use the value in the register, second line: number of gadgets that use its pointee.

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. However, none of the gadgets found were formed from instructions decoded from unintended byte sequences.

One under-approximating limitation of our gadget search algorithm is that it assumes that gadgets start from the latest flag-setting instruction before the jump. Suppose a sequence in the victim's code is `shl 8, rax; test 1, rax; jz 0xadd;`. Our tool will find `test 1, rax; jz 0xadd;` as a SMoTher gadget which leaks the least significant bit (LSB) of `rax`. However, the instructions preceding this might perform computations which cause the gadget to leak different information. The entire sequence is a different SMoTher gadget which leaks the 9-th least significant bit of `rax`. The space of usable SMoTher-gadgets exceeds the ones we have found, and depend on the particular victim's code.

6 REAL WORLD ATTACK

We demonstrate real-world attacks on OpenSSH and OpenSSL, two commonly used programs handling sensitive secrets that have been extensively hardened against regular and side-channel attacks.

6.1 OpenSSH attack

OpenSSH is widely used to securely and privately connect to servers over untrusted networks. The confidentiality of the server's private key is essential to the security of the overall system. Leaking the private server key allows an attacker to impersonate the server, acting as a man in the middle. In the OpenSSH attack, we find a BTI

```
.rept 8;
addl r8d, r9d;
addl r10d, r11d;
addl r8d, r9d;
addl r10d, r11d;
.endr;
```

(a) Attacker-timed code

```
0x6f8dc:  testl 0x100, (rdi)
0x6f8e2:  je    6f8ef
0x6f8e4:  mov   0x10(rbx),rax
0x6f8e8:  sub   0x8(rbx),rax
0x6f8ec:  sub   rax,rsi
0x6f8ef:  mov   rbx,rdi
...
```

(b) Victim SMoTher gadget (glibc)

```
static void (* volatile ssh_bzero)
                (void *, size_t) = bzero;
void explicit_bzero(void *p, size_t n) {
  ...
  ssh_bzero(p, n);
  ...
}
```

(c) Victim BTI gadget (OpenSSH)

Figure 5: Gadgets from real-world libraries used in our SMoTherSpectre exploit for leaking the $7th$ least significant bit of rdx's pointee

gadget in the default OpenSSH (version 7.2) SSH server binary available on Ubuntu 16.04 LTS, together with four SMoTher gadgets in glibc version 2.23, and leak bits of the host key. As shown by Heninger and Shacham [17], leaking a small fraction of bits enables recovery of the entire key.

The threat model for this attack assumes a local attacker that is able to initiate TCP connections to the ssh daemon. As before, we assume that ASLR is disabled (or can be bypassed through other means). Since the target BTI gadget runs pre-authentication, the attacker only needs to connect and does not need to authenticate to the server. In our PoC, the local attacker is running on the same host. However, the same attack can be run from a colocated VM, assuming the VMM schedules both attacker and victim VMs on the same physical core. We also assume that the attacker is able to spawn processes on the same physical core as the victim SSH process: the assumption is realistic, as shown for example by Zhang *et al.* [37].

Our BTI gadget resides in the explicit_bzero function (Figure 5c) which clears regions of memory. The function is extensively used to zero out sensitive data before memory is released as a countermeasure against data leakage if that memory region is reused for another purpose. To eliminate the possibility of dead-store optimization by the compiler, explicit_bzero calls the standard bzero function using a *volatile* function pointer. We exploit the indirect jump generated for this function pointer call as the BTI gadget, knowing that the first argument to the function (stored in register rdi according to the System V calling convention).

In particular, we exploit an invocation of the BTI gadget where the pointer refers to the server's private host key (e.g., RSA key). This invocation is present in the code path handling new connections, when the server loop forks new processes for each incoming connection and loads the private host key from disk with the key_load_private function. The cryptographic values (e.g., the exponents and modulus of the RSA key) are kept in memory to later perform the ssh handshake but the buffer used to read out the file from disk is zeroed out and freed. This gadget is particularly convenient since the attacker gets an arbitrary number of attempts at discovering different bits of the same private key. Also, the attacker can control when the victim process is spawned by initiating connections to the ssh daemon.

An abridged version of the SMoTher gadget is shown in Figure 5b (see Appendix C for the full assembly listing). Our chosen SMoTher gadget differs slightly from that described in Section 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 fall-through path differ in utilization of execution ports 0156. This gadget is taken from glibc and demonstrates the availability of SMoTher-gadgets in commonly linked libraries. The attacker times a sequence of add instructions with register operands (port 0156) shown in Figure 5a to specifically target the same ports.

We ran our attack on a slightly-modified sshd server. The ssh server is modified to setup relevant performance counters to be used for statistical and monitoring purposes. These counter values are ignored by the actual attack. The other modification is to synchronize the attacker with the BTI gadget (as in Section 4.3). For other targets (i.e., OpenSSL), we have investigated alternate synchronization mechanisms that do not require victim modification and have good results. The server was compiled using the default options for Linux on x86_64.

In the PoC of the attack, an orchestrator process randomly sets the bit to be leaked before launching the server and attacker on colocated logical cores. The attacker process is responsible for "poisoning" the BTB to cause mis-speculation on the victim process handling the incoming ssh connection. Prior to BTI, the attacker also performs a series of cache accesses that result in the eviction of the server's cache line containing the function pointer ssh_bzero. This forces the victim's indirect call instruction to miss in the cache and speculate for a few cycles, increasing the BTI success rate. The attacker process is otherwise identical to the victim and follows the same code path, increasing the probability of the attacker having the same branching history as the victim at the call site, thereby increasing the success rate of BTI. The orchestrator launches an ssh client on a separate physical core to connect to the server and trigger the creation of the victim and attacker processes. Victim and attacker process execute and the attacker is able to collect a SMoTher-timing sample correlated to the value the LSB in byte 1 of the host private key.

The attack can be extended in two ways. First, we can pair our BTI gadget with other SMoTher gadgets in the victim, enabling us to leak other bits of the host private key. Second, we can find other occurrences of the explicit_bzero BTI gadget (or other BTI gadgets) where different secrets are held in registers or in memory.

In the explicit_bzero BTI gadget, we found that the value of the register r12 equals the value of rdi, both pointing to the host key in memory at the point of attack. Therefore, we are able to use three other SMoTher-gadgets which dereference r12. These gadgets allow us to leak extra bits from the host key, specifically the

4^{th} LSB in byte 13, the 4^{th} LSB in byte 14 and the 5^{th} LSB in byte 56. The corresponding assembly listings are shown in Appendix C.2.

Additionally, we can also find other BTI gadgets, or invocations of `explicit_bzero` with different secrets. Other secrets erased by this function include contents of the `/etc/shadow` file and client passwords in cleartext received during login attempts.

6.2 OpenSSL attack

For OpenSSL, we target a BTI gadget in the `libcrypto` library (version 1.1.1b, dated 26-Feb-2019) which is widely used for performing cryptographic functions and a SMoTher gadget from `glibc` version 2.27.

Over the years, considerable effort was devoted to thwarting 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 (such as ECB, CBC, GCM) with the same block cipher. OpenSSL uses a context variable that stores function pointers for encryption/decryption. These pointers are set during the initialization phase depending on the user-specified cipher mode.

As a result, cryptographic applications using `libcrypto` execute an indirect call (the BTI gadget) during every block encryption or decryption. Such gadgets are the result of commonly used coding practices, and do not directly perform any data-dependent actions based on the secret value. As in the OpenSSH attack, the use of function pointers leads to indirect calls in the compiled binary. While security was the motivating factor for OpenSSH, OpenSSL uses function pointers to support polymorphic-like behavior, enabling our transient execution attack.

Our BTI gadget is contained in `EVP_EncryptUpdate`, and is shown in Figure 6c. The third argument (`in`) contains a pointer to the plaintext to be encrypted (and is therefore a secret). In accordance with 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 3^{rd} LSB in byte 1 of the plaintext, referenced through `rdx`. An abridged version of the SMoTher gadget is shown in Figure 6b (see Appendix B for the full assembly listing).

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 encryptions using calls to `EVP_EncryptUpdate`. We have also instrumented the victim to setup relevant performance counters which are only used for statistical and monitoring purposes and are not used in the attack. The victim library does not contain any code to help the attacker synchronize with the execution of the BTI gadget.

The attacker triggers the encryptions on the victim. It also runs code that is almost identical to the victim apart from the following differences. First, it loads the call pointer with the location of the SMoTher gadget on the victim to trigger BTI on the victim process. Second, it replaces the code at the target location by a delay sequence and the SMoTher timing. The delay sequence consists of a series of dependent instructions that allows the attacker to delay for a controlled number of cycles, synchronizing with the victim's SMoTher gadget, before measuring the timing sample. 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, thus maximising the success of BTI. In each iteration, the attacker gets one timing measurement. Between iterations, the attacker performs a series of memory accesses designed to evict the victim's cache line holding the pointer to the encryption function from the L1 cache to increase the BTI success rate. We observed that other usual work being performed on the core can have the same effect.

Pointer register	Byte Offset	Bit mask	Δ SMoTher timing
`rdi`	0x01	0x01	0.32% ± 0.21%
`r12`	0x38	0x10	0.64% ± 0.62%
`r12`	0x0d	0x08	0.66% ± 0.47%
`r12`	0x0c	0x08	0.42% ± 0.33%

Table 4: SMoTherSpectre results leaking the `sshd` private key. Four gadgets, each targeting a different key bit identified by its byte offset and bit mask, were used. We also show the mean timing difference percentage for the attacker's SMoTher timing, separated according to the value of the randomly-set target bit: all show a difference at 95% confidence.

6.3 Experimental results

We run the OpenSSH attack on a quad-core, hyper-threaded Skylake CPU (i7-6700K) with the server and attacker pinned on logical cores 0 and 4 respectively (running on physical core 0). For each connection attempt to the server, the orchestrator randomly sets or resets the target bit, logs its value and the attacker measures a SMoTherSpectre timing sample. We run the attack 10,000 times and separate the collected samples based on the value of the target bit on that particular run, yielding two sets of attacker timings corresponding to the target bit being zero or one. Finally, we run the Student's t-test to check whether the sets are statistically distinguishable. We used this methodology on four SMoTher-gadgets described in Section 6.1. Table 4 shows the results: the distributions are differentiable with at-least 95% confidence for each of the four gadgets. The attack does not require extremely high BTI success rate: in our samples, we observe BTI success rates ranging between 16% and 25%. The whole experiment takes about 75 seconds of real-time, of which a total of 20 seconds are spent by the orchestrator waiting for the server to be fully setup before launching the client.

We run the OpenSSL attack on an i5-6200u CPU. A run of 100,000 encryptions is performed by the victim for each value of the secret bit. The large number of encryptions is necessary to estimate the probability density function for this SMoTher-gadget. A practical attack can confidently leak a bit with fewer encryptions. The attack takes about 950 ms of userspace time, leading to a measurement rate of more than 200,000 samples/second. The attacker succeeded in BTI with a success rate up to around 80%. We have found the time taken by the victim to reach the indirect call from the call triggering encryption entry to be highly predictable. The attacker is thus able to run the timing sequence concurrently with the victim's SMoTher-gadget without additional synchronization. Figure 7 shows the distribution of timestamp counter difference

```
.rept 8;
btrl r8d, r9d;
btrl r10d, r11d;
btsl r8d, r9d;
btsl r10d, r11d;
.endr;
```

(a) Attacker-timed code

```
0xf5393:  testq  0x400,(rdx)
0xf539a:  je     f5382
0xf539c:  mov    -0xb0(rbp),rdi
...
0xf5382:  add    0x1,rax
...
```

(b) Victim SMoTher gadget (glibc)

```
if(ctx->cipher->do_cipher(ctx, out, in, inl))
{
    *outl = inl;
    return 1;
}
...
```

(c) Victim BTI gadget (OpenSSL)

Figure 6: Gadgets from real-world libraries used in our SMoTherSpectre exploit for leaking the 3^{rd} LSB of byte 1 of rdx's pointee

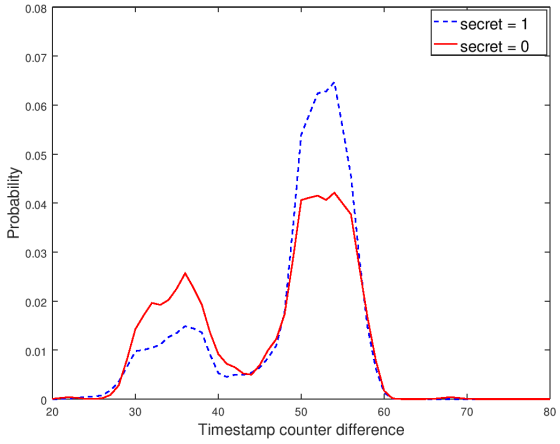


Figure 7: Probability density function (estimated using kernel density estimation) for the attacker's timing running our SMoTherSpectre attack on OpenSSL, for when the victim's secret bit is one versus zero.

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-test is able to successfully distinguish between them with 95% confidence. The test reports a timing difference of 10.69% ± 6.31%.

6.4 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 [12].

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 [33], 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 [23]. 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., Return Stack Buffer (RSB) overflow [27] or speculative return address overwrite [22]. 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" [18, 23]. 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 [24, 27]. 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 [22].

Meltdown [26] ("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 [34] 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 an attack. 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, 20]: 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 [29] 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 different tiers of the memory hierarchy. 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 data and 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 [28, 32].

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 [36]. Related to flush-and-reload, *flush-and-flush* [16] 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 [14] shows how the TLB can be used as a side channel to leak a cryptographic key. Aforementioned Netspectre-AVX [29] 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 units 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 [35] 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 [11] 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, SMoTherSpectre 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 the OpenSSH server and encryption using 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. 2007. Cheap hardware parallelism implies cheap security. In *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*. IEEE, 80–91.

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

[3] AMD. 2018. Speculative Store Bypass Disable. https://developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf.

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

[5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2018. A Systematic Evaluation of Transient Execution Attacks and Defenses. https://arxiv.org/abs/1811.05441.

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

[7] Jonathan Corbet. [n.d.]. Taming STIBP. https://lwn.net/Articles/773118/.

[8] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security Symposium*.

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

[10] Agner Fog. [n.d.]. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf.

[11] Anders Fogh. [n.d.]. Covert Shotgun. https://cyber.wtf/2016/09/27/covert-shotgun/.

[12] Anders Fogh and Christopher Ertl. [n.d.]. Wrangling with the Ghost: An inside story of mitigating speculative execution side channel vulnerabilities. https://i.blackhat.com/us-18/Thu-August-9/us-18-Fogh-Ertl-Wrangling-with-the-Ghost-An-Inside-Story-of-Mitigating-Speculative-Execution-Side-Channel-Vulnerabilities.pdf.

[13] Google [n.d.]. Google Compute Engine FAQ. https://cloud.google.com/compute/docs/faq. Accessed: 2019-02-13.

[14] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*.

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

[16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[17] Nadia Heninger and Hovav Shacham. 2009. Reconstructing RSA Private Keys from Random Key Bits. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Shai Halevi (Ed.), Vol. 5677. Springer, 1–17. https://doi.org/10.1007/978-3-642-03356-8_1

[18] Jann Horn. 2018. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html. *Project Zero* 3 (2018).

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

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

[21] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2018. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. *arXiv preprint arXiv:1806.05179* (2018).

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

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

[24] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Workshop On Offensive Technologies*.

[25] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. 2018. Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures. https://arxiv.org/abs/1809.00912.

[26] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.

[27] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Conference on Computer and Communications Security*.

[28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology*.

[29] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2018. NetSpectre: Read Arbitrary Memory over Network. https://arxiv.org/abs/1807.10535.

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

[31] Linus Torvalds. 2018. Linus on Spectre/Meltdown mitigations. https://lkml.org/lkml/2018/1/21/192.

[32] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology* (2010).

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

[34] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.

[35] Zhenghong Wang and Ruby B. Lee. 2006. Covert and Side Channels Due to Processor Architecture. In *Proceedings of the 22Nd Annual Computer Security Applications Conference (ACSAC '06)*. IEEE Computer Society, Washington, DC, USA, 473–482. https://doi.org/10.1109/ACSAC.2006.20

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

[37] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 305–316. https://doi.org/10.1145/2382196.2382230

A GADGETS LEAKING 21 BITS OF RAX

The following table lists parts of SMoTher-gadgets which can be used to leak 21 bits of information from rax. We also show which library the gadget was found in.

Address	Comparison instruction	Library
0xd3900	test 0x1, al	glibc
0x1101cb	test 0x2, al	glibc
0x12f779	and 0x4, al	glibc
0x29709	and 0x8, al	glibc
0x126500	and 0x10, al	glibc
0x7e83	and 0x20, al	ld
0xc378e	and 0x40, al	glibc
0xd7e50	and 0x80, eax	glibc
0x12cad9	test 0x2, ah	stdc++
0xf1794	test 0x307, ax	libcrypto
0x5f661	and 0x2100, eax	glibc
0x11c7f6	and 0x2abd, eax	glibc
0x10ca11	and 0x8000, eax	glibc
0x17bcd4	test 0x100000, eax	libcrypto
0x268de	test 0x200000, eax	ssl
0xbe656	and 0x3084a5, eax	glibc
0x26f20	test 0x800000, eax	ssl
0xb3ba0	test 0x1000000, eax	glibc
0xb7db	test 0x40000000, eax	pthread
0x50e7b	test 0x80000000, eax	ssl
0xa6133	test 0x83000002, eax	libcrypto

B OPENSSL ATTACK GADGETS

B.1 SMoTher gadget

The following gadget leaks the 3^{rd} LSB from the byte at offset 1 from the pointer in `rdx`.

```
f5393:  testq  0x400, (rdx)
f539a:  je     f5382
f539c:  mov    -0xb0(rbp), rdi
f53a3:  mov    -0xf0(rbp), edx
f53a9:  mov    (rdi, rax, 8), rax
f53ad:  test   edx, edx
f53af:  mov    rax, 0x50(rbx)
...
f5382:  add    0x1, rax
f5386:  add    0x20, rdx
f538a:  cmp    rax, -0x100(rbp)
...
```

C OPENSSH ATTACK GADGETS

C.1 SMoTher gadget with `rdi` pointer

The following gadget leaks the LSB from the byte at offset 1 from the pointer in `rdi`.

```
...
6f8dc:  testl  0x100, (rdi)
6f8e2:  je     6f8ef
6f8e4:  mov    0x10(rbx), rax
6f8e8:  sub    0x8(rbx), rax
6f8ec:  sub    rax, rsi
6f8ef:  mov    rbx, rdi
6f8f2:  mov    ecx, 0xc(rsp)
6f8f6:  mov    edx, 0x8(rsp)
6f8fa:  mov    rsi, (rsp)
...
```

C.2 SMoTher gadgets with `r12` pointer

The following gadget leaks the 5^{th} LSB from the byte at offset 56 from the pointer in `r12`.

```
e8577:  testb  0x10, 0x38(r12)
e857d:  je     e8608
e8583:  sub    0x8, rsp
e8587:  push   rbx
e8588:  pushq  0x0
e858a:  pushq  0x0
e858c:  mov    edx, r8d
e858f:  mov    edx, r9d
e8592:  mov    r10d, ecx
e8595:  sub    r10d, r8d
e8598:  mov    r13, rsi
e859b:  mov    r12, rdi
...
e8608:  sub    0x8,rsp
e860c:  push   rbx
e860d:  push   r14
e860f:  push   r15
...
```

The following gadget leaks the 4^{th} LSB from the byte at offset 12 from the pointer in `r12`.

```
5220e:  testb  0x8, 0xd(r12)
52214:  je     52221
52216:  xor    edx, edx
52218:  xor    esi, esi
5221a:  xor    edi, edi
...
52221:  mov    r13, rcx
52224:  add    0x3, r13
52228:  sar    0x2, rcx
5222c:  cmp    0x6, r13
...
```

The following gadget leaks the 4^{th} LSB from the byte at offset 13 from the pointer in `r12`.

```
529a2:  testb  0x8, 0xc(r12)
529a8:  je     523da
529ae:  mov    -0x100(rbp), rcx
529b5:  lea    0xc(rcx), rdx
529b9:  cmp    rax, rdx
...
523da:  mov    -0xf8(rbp), r13d
523e1:  mov    (rax), edx
523e3:  add    -0xe8(rbp), r13d
...
```

D RESPONSIBLE DISCLOSURE

The attacks presented in this paper were disclosed to Intel, OpenSSL and AMD in late 2018.

IBM Corporation
New Orchard Road
Armonk, NY 10504