Port Contention for Fun and Profit

Alejandro Cabrera Aldaya*, Billy Bob Brumley†, Sohaib ul Hassan†, Cesar Pereida García†, Nicola Tuveri†
*Universidad Tecnológica de la Habana (CUJAE), Habana, Cuba
†Tampere University of Technology, Tampere, Finland

***Abstract*—Simultaneous Multithreading (SMT) architectures are attractive targets for side-channel enabled attackers, with their inherently broader attack surface that exposes more per physical core microarchitecture components than cross-core attacks. In this work, we explore SMT execution engine sharing as a side-channel leakage source. We target ports to stacks of execution units to create a high-resolution timing side-channel due to port contention, inherently stealthy since it does not depend on the memory subsystem like other cache or TLB based attacks. Implementing said channel on Intel Skylake and Kaby Lake architectures featuring Hyper-Threading, we mount and end-to-end attack that recovers a P-384 private key from an OpenSSL-powered TLS server using a small number of repeated TLS handshake attempts. Furthermore, we show that traces targeting shared libraries, static builds, and SGX enclaves are essentially identical, hence our channel has wide target application.**

1. Introduction

Microarchitecture side-channel attacks increasingly gain traction due to the real threat they pose to general-purpose computer infrastructure. New techniques emerge every year [1, 2], and they tend to involve lower level hardware, they get more complex but simpler to implement, and more difficult to mitigate, thus making microarchitecture attacks a more viable attack option. Many of the current microarchitecture side-channel techniques rely on the persistent state property of shared hardware resources, e.g., caches, TLBs, and BTBs, but non-persistent shared resources can also lead to side-channels [3], allowing leakage of confidential information from a trusted to a malicious process.

The microprocessor architecture is complex and the effect of a component in the rest of the system can be difficult (if not impossible) to track accurately: especially when components are shared by multiple processes during execution. Previous research [4, 5] confirms that as long as (persistent and non-persistent) shared hardware resources exist, attackers will be able to leak confidential information from a system.

In this work, we present a side-channel attack vector exploiting an inherent component of modern processors using Intel Hyper-Threading technology. Our new side-channel technique PORTSMASH is capable of exploiting timing information derived from port contention to the execution units, thus targeting a non-persistent shared hardware resource. Our technique can choose among several configurations to target different ports in order to adapt to different scenarios, thus offering a very fine spatial granularity. Additionally, PORTSMASH is highly portable and its prerequisites for execution are minimal, i.e., does not require knowledge of memory cache-lines, eviction sets, machine learning techniques, nor reverse engineering techniques.

To demonstrate PORTSMASH in action, we present a complete end-to-end attack in a real-world setting attacking the NIST P-384 curve during signature generation in a TLS server compiled against OpenSSL 1.1.0h for crypto functionality. Our *Spy* program measures the port contention delay while executing in parallel to ECDSA P-384 signature generation, creating a timing signal trace containing a noisy sequence of *add* and *double* operations during scalar multiplication. We then process the signal using various techniques to clean the signal and reduce errors in the information extracted from each trace. We then pass this partial key information to a recovery phase, creating lattice problem instances which ultimately yield the TLS server's ECDSA private key.

We extend our analysis to SGX, showing it is possible to retrieve secret keys from SGX enclaves by an unprivileged attacker. We compare our PORTSMASH technique to other side-channel techniques in terms of spatial resolution and detectability. Finally, we comment on the impact of current mitigations proposed for other side-channels on PORT SMASH, and our recommendations to protect against it.

In summary, we offer a full treatment of our new technique: from microarchitecture and side-channel background (Section 2); to the nature of port contention leakage when placed in an existing covert channel framework (Section 3); to its construction as a versatile timing side-channel (Section 4); to its application in real-world settings, recovering a private key (Section 5); to discussing (lack of) detectability and mitigations (Section 6). We conclude in Section 7.

2. Background

2.1. Microarchitecture

This section describes some of Intel's microarchitectural components and how they behave with Intel SMT implementation (i.e., Hyper-Threading technology). Intel launched its SMT implementation with the Pentium 4 MMX

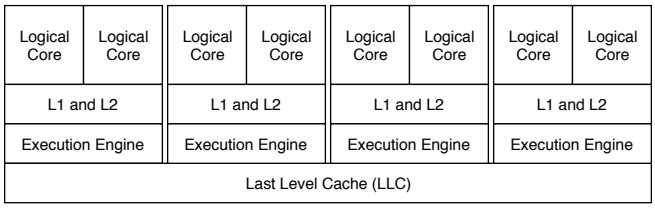

Figure 1. Intel i7 Core processor.

processor [6]. Hyper-Threading technology (HT) aims at providing parallelism without duplicating all microarchitectural components in a physical processor. Instead, a processor supporting Hyper-Threading has at least two logical cores per physical core where some components are shared between the logical ones.

Figure 1 shows a high-level description of the layout of an Intel i7 processor [7]. This figure shows four physical cores, each with two logical cores. In this setting the operating system sees a processor with eight cores.

Figure 1 sketches some microarchitectural components with a sharing perspective. L1 and L2 caches are shared between a pair of logical cores in the same physical core. The next level depicts how an Execution Engine is also shared between two logical cores. This component is very important for this manuscript as the presented microarchitectural side-channel relies on this logical-core-shared feature. On the other hand, the last level cache (LLC) is shared between all cores.

Generally speaking the Execution Engine is responsible for executing instructions therefore it is closely related to the concept of pipeline [7, 8]. A simplified pipeline model consists of three phases: (1) fetch, (2) decode, and (3) execute. While these phases have complex internal working details, Figure 2 provides a high-level abstraction focusing mainly on the Execution Engine part, and its description below also follows the same approach. For more information about its inner working details we suggest the reader to consult [6–8].

Each logical core has its own registers file, and the pipeline fetches instructions from memory according to the program counter on each of them. For the sake of processing performance fairness, this fetching is interleaved between the logical cores. After the fetch stage, a decoding phase decomposes each instruction into simpler micro-operations (*uops*). Each micro-operation does a single task, therefore this splitting helps out-of-order execution by interleaving their executions for the sake of performance. After this point, all processing is done on *uops* instead of instructions. The decoding phase then issues these *uops* to the execution scheduler.

At the scheduler there is a queue of *uops* that belongs to both logical cores. One task of the scheduler is issuing these *uops* to the Execution Ports while maximizing performance. An Execution Port is a *channel* to the execution units, the latter being where *uops* are actually executed. Figure 2 shows execution units as grey-colored boxes with labels indicating their functionality. For example, ports 0, 1, 5,

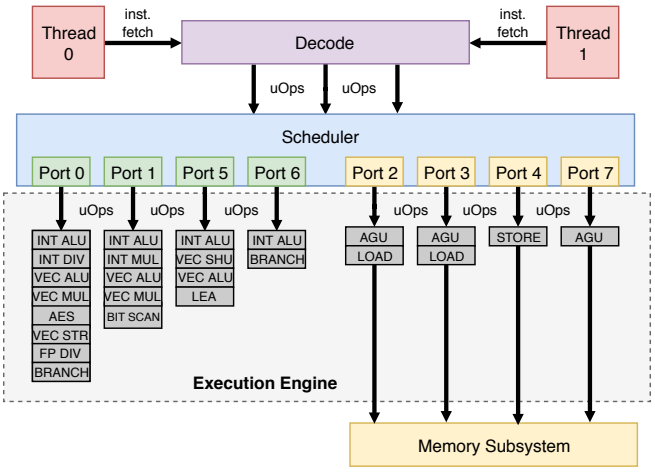

Figure 2. Skylake/Kaby Lake microarchitecture.

and 6 can be used to execute simple arithmetic instructions, because each of them is a channel to an ALU execution unit. While ports 2, 3, 4, and 7 are dedicated to memory-based *uops* (e.g. loads and stores).

As an illustrative example of how the whole process happens in this simplified model, let us consider the `adc mem, reg` instruction (AT&T syntax), which adds (with carry) the value at memory location `mem` into the content in register `reg`. According to Fog's instruction table for Skylake microarchitecture [9], this instruction splits into two *uops*: one arithmetic *uop* (that actually performs the addition) and another for loading a value from memory. The former can be issued to ports 0 or 6, while the latter to port 2 and 3 [9]. However, if we change the operand order in the original instruction (i.e., now the addition result is stored back in the memory location `mem`), the new instruction splits into three *uops*: two are essentially the same as before and another one is issued for storing the result back to memory (i.e., an operation that is handled by port 4).

This execution sequence behaves exactly the same in the presence of Hyper-Threading. At the scheduler, there are *uops* waiting for dispatch to some port for execution. These *uops* could actually belong to instructions fetched from any logical core, therefore, these cores share the Execution Engine in a very granular approach (at *uops* level).

2.2. SMT: Timing Attacks

Timing attacks on microprocessors featuring SMT technology have a long and old history with respect to side-channel analysis. Since the revival of SMT in 1995 [10], it was noted that contention was imminent, particularly in the memory subsystem. Arguably, timing attacks became a more serious security threat once Intel introduced its Hyper-Threading technology on the Pentium 4 microarchitecture. Researchers knew that resource sharing leads to resource contention, and it took a remarkably short time to notice that contention introduces timing variations during execution, which can be used as a covert channel, and as a side-channel.

In his pioneering work, Percival [11] described a novel cache-timing attack against RSA's Sliding Window Exponentiation (SWE) implemented in OpenSSL 0.9.7c. The attack exploits the microprocessor's Hyper-Threading feature and after observing that threads "share more than merely the execution units", the author creates a spy process that exfiltrates information from the L1 data cache. The L1 data cache attack correctly identifies accesses to the precomputed multipliers used during the SWE algorithm, leading to RSA private key recovery. As a countermeasure, to ensure uniform access to the cache lines, irrespective of the multiplier used, the OpenSSL team included a "constant-time" Fixed Window Exponentiation (FWE) algorithm paired with a scatter-gather method to mask table access [12].

Cache-based channels are not the only shared resource to receive security attention. Wang and Lee [3] and Acıiçmez and Seifert [13] analyzed integer multiplication unit contention in old Intel Pentium 4 processors with SMT support [6]. In said microarchitecture, the integer multiplication unit is shared between the two logical cores. Therefore contention could exist between two concurrent processes running in the same physical core if they issue integer multiplication instructions. Wang and Lee [3] explore its application as a covert channel, while Acıiçmez and Seifert [13] expand the side-channel attack approach.

Acıiçmez and Seifert [13] stated this side-channel attack is very specific to the targeted Intel Pentium 4 architecture due to the fact that said architecture only has one integer multiplier unit. They illustrated an attack against the SWE algorithm in OpenSSL 0.9.8e. For this purpose they developed a proof-of-concept, modifying OpenSSL source code to enhance the distinguishability between square and multiplication operations in the captured trace. In addition to integer multiplication unit sharing, their attack relies on the fact that square and multiplication operations have different latencies, an unnecessary assumption in our work.

In a 2016 blog post[1], Anders Fogh introduced Covert Shotgun, an automated framework to find SMT covert channels. The strategy is to enumerate all possible pairs of instructions in an ISA. For each pair, duplicate each instruction a small number of times, then run each block in parallel on the same physical core but separate logical cores, measuring the clock-cycle performance. Any pairwise timing discrepancies in the resulting table indicate the potential for a covert channel, where the source of the leakage originates from any number of shared SMT microarchitecture components. Fogh explicitly mentions caching of decoded *uops*, the reorder buffer, port congestion, and execution unit congestion as potential sources, even reproducing the `rdseed` covert channel [14] that remarkably works across physical cores.

Covert channels from Covert Shotgun can be viewed as a higher abstraction of the integer multiplication unit contention covert channel by Wang and Lee [3], and our side-channel a higher abstraction of the corresponding side-channel by Acıiçmez and Seifert [13]. Now limiting the discussion to port contention, our attack focuses on the port sharing feature. This allows a *darker-box* analysis of the targeted binary because there is no need to know the exact instructions executed by the victim process, only that the attacker must determine the distinguishable port through trial and error. This feature is very helpful, for example, in a scenario where the targeted code is encrypted and only decrypted/executed inside an SGX enclave [15].

Analogous to [11], Acıiçmez et al. [16] performed a cache-timing attack against OpenSSL DSA, but this time targeting the L1 instruction cache. The authors demonstrate an L1 instruction cache attack in a real-world setting and using analysis techniques such as vector quantization and hidden Markov models, combined with a lattice attack, they achieve DSA full key recovery on OpenSSL version 0.9.8l. They perform their attack on an Intel Atom processor featuring Hyper-Threading. Moreover, due to the relevance and threat of cache-timing attacks, the authors list and evaluate several possible countermeasures to close the cache side-channels.

More recently, Yarom et al. [5] presented CacheBleed, a new cache-timing attack affecting some older processors featuring Hyper-Threading such as Sandy Bridge. The authors exploit the fact that cache banks can only serve one request at a time, thus issuing several requests to the same cache bank, i.e. accessing the same offset within a cache line, results in bank contention, leading to timing variations and leaking information about low address bits. To demonstrate the attack, the authors target the RSA exponentiation in OpenSSL 1.0.2f. During exponentiation, RSA uses the scatter-gather method adopted due to Percival's work [11]. More precisely, to compute the exponentiation, the scatter-gather method accesses the cache bank or offset within a cache line according to the multiplier used, which depends on a digit of the private key. Thus, by detecting the used bank through cache bank contention timings, an attacker can determine the multiplier used and consequently digits of the private key. The attack requires very fine granularity, thus the victim and the spy execute in different threads in the same core, and after observing 16,000 decryptions, the authors fully recover 4096-bit RSA private keys.

In 2018, Gras et al. [4] presented TLBleed, a new class of side-channel attacks relying on the Translation Lookaside Buffers (TLB) and requiring Hyper-Threading to leak information. In their work, the authors reverse engineer the TLB architecture and demonstrate the TLB is a (partially) shared resource in SMT Intel architectures. More specifically, the L1 data TLB and L2 mixed TLB are shared between multiple logical cores and a malicious process can exploit this to leak information from another process running in the same physical core. As a proof-of-concept, the authors attack a non constant-time version of 256-bit EdDSA [17] and a 1024-bit RSA hardened against FLUSH+RELOAD as implemented in libgcrypt. The EdDSA attack combined with a machine-learning technique achieves a full key recovery success rate of 97%, while the RSA attack recovers 92% of the private key but the authors do not perform full key recovery. Both attacks are possible after capturing a single trace.

1. https://cyber.wtf/2016/09/27/covert-shotgun/

3. Instantiating Covert Shotgun

Being an automated framework, Covert Shotgun is a powerful tool to detect potential leakage in SMT architectures. But due to its black-box, brute force approach, it leaves identifying the root cause of leakage as an open problem: "Another interesting project would be identifying the subsystem which are being congested by specific instructions". In this section, we fill this research gap with respect to port contention. Our intention is not to utilize this particular covert channel in isolation, but rather understand how the channel can be better optimized for its later conversion to a side-channel in Section 4.

3.1. Concept

Assume cores C_0 and C_1 are two logical cores of the same physical core. To make efficient and fair use of the shared execution engine, a simple strategy for port allocation is as follows. Denote i the clock cycle number, and $j = i \bmod 2$, and $\mathcal{P}$ the set of ports.

1) C_j is allotted $\mathcal{P}_j \subseteq \mathcal{P}$ such that $|\mathcal{P} \setminus \mathcal{P}_j|$ is minimal.
2) C_{1-j} is allotted $\mathcal{P}_{1-j} = \mathcal{P} \setminus \mathcal{P}_j$.

There are two extremes in this strategy. For instance, if C_0 and C_1 are executing fully pipelined code with no hazards, yet make use of disjoint ports, then both C_0 and C_1 can issue in every clock cycle since there is no port contention. On the other hand, if C_0 and C_1 are utilizing the same ports, then C_0 and C_1 alternate, issuing every other clock cycle, realizing only half the throughput performance-wise.

Consider Alice and Bob, two user space programs, executing concurrently on C_0 and C_1, respectively. The above strategy implies the performance of Alice depends on port contention with Bob, and vice versa. This leads to a covert timing channel as follows. Take two latency-1 instructions: `NOP0` that can only execute on port 0, and `NOP1` similarly on port 1. Alice sends a single bit of information to Bob as follows.

1) If Alice wishes to send a zero, she starts executing `NOP0` continuously; otherwise, a one and `NOP1` instead.
2) Concurrently, Bob executes a fixed number of `NOP0` instructions, and measures the execution time t_0.
3) Bob then executes the same fixed number of `NOP1` instructions, and measures the execution time t_1.
4) If $t_1 > t_0$, Bob receives a one bit; otherwise, $t_0 > t_1$ and a zero bit.

The covert channel works because if both Alice and Bob are issuing `NOP0` instructions, they are competing for port 0 and the throughput will be cut in half (similarly for `NOP1` and port 1). On the other hand, with no port contention both `NOP0` and `NOP1` execute in the same clock cycle, achieving full throughput and lower latency.

TABLE 1. SELECTIVE INSTRUCTIONS. ALL OPERANDS ARE REGISTERS, WITH NO MEMORY OPS.

Instruction	Ports	Latency	Reciprocal Throughput
`add`	0 1 5 6	1	0.25
`crc32`	1	3	1
`popcnt`	1	3	1
`vpermd`	5	3	1
`vpbroadcastd`	5	3	1

3.2. Implementation

In this section, we give empirical evidence that Intel Hyper-Threading uses the previous hypothetical port allocation strategy for SMT architectures strategy (or one indistinguishable from it for our purposes). Along the way, we optimize the channel with respect to pipeline usage, taking into account instruction latencies and duplicated execution units.

In these experiments, we used an Intel Core i7-7700HQ Kaby Lake featuring Hyper-Threading with four cores and eight threads. Using the `perf` tool to monitor *uops* dispatched to each of the seven ports and the clock cycle count for a fixed number of instructions, we determined the port footprint and performance characteristics of several instructions, listed in Table 1. We chose this mix of instructions to demonstrate the extremes: from `add` that can be issued to any of the four integer ALUs behind ports 0, 1, 5, or 6, to `crc32` and `vpermd` that restrict to only ports 1 and 5, respectively. Furthermore, to minimize the effect of the memory subsystem on timings (e.g. cache hits and misses), in this work we do not consider any explicit store or load instructions, or any memory operands to instructions (i.e. all operands are registers).

Given the results in Table 1, we construct the covert channel as follows: `crc32` (port 1) will serve as the `NOP0` instruction, and `vpermd` (port 5) as `NOP1`. Being latency-3 instructions, we construct a block of three such instructions with disjoint operands to fill the pipeline, avoid hazards, and realize a throughput of one instruction per clock cycle. We repeated each block 64 times to obtain a low ratio of control flow logic to overall instructions retired. The Alice program sends a zero bit by executing the repeated `crc32` blocks in an infinite loop. Concurrently on the receiver side, using `perf`, we measured the number of clock cycles required for the Bob program to execute 2^{20} of the repeated `crc32` blocks, then again measured with the same number of repeated `vpermd` blocks. We then repeated the experiment with the Alice program sending a one bit analogously with the `vpermd` instruction. We carried out the experiments with both Alice and Bob pinned to separate logical cores of the same physical core, then also different physical cores. As a rough estimate, for full throughput we expect $3 \cdot 64 \cdot 2^{20} \approx 201$ million cycles (three instructions, with 64 repetitions, looping 2^{20} times); even with a latency of three, our construction ensures a throughput of one. Of course there is some overhead for control flow logic.

TABLE 2. RESULTS OVER A THOUSAND TRIALS. AVERAGE CYCLES AND STANDARD DEVIATION ARE IN THOUSANDS.

		Diff. Phys. Core		Same Phys. Core	
Alice	Bob	Cycles	Dev.	Cycles	Dev.
Port 1	Port 1	203331	645	**408322**	221
Port 1	Port 5	203322	514	203820	134
Port 5	Port 1	203334	639	203487	144
Port 5	Port 5	203328	524	**404941**	183

Table 2 contains the results, averaged over a thousand trials. First on separate physical cores, we see that the cycle count is essentially the same and approaches our full throughput estimate, regardless of which port Alice and/or Bob are targeting. This confirms the channel does not exist across physical cores. In contrast, the results on the same physical core validates the channel. When Alice and Bob target separate ports, i.e. the port 1/5 and 5/1 cases, the throughput is maximum and matches the results on different physical cores. However, when targeting the same port, i.e. the port 1/1 and 5/5 cases, the throughput halves and the cycle count doubles due to the port contention. This behavior precisely matches the hypothesis in Section 3.1.

4. From Covert to Side-Channel

One takeaway from the previous section is that, given two user space programs running on two separate logical cores of the same physical core, the clock cycle performance of each program depends on each other's port utilization. Covert Shotgun leaves extension to side-channels as an open problem: "it would be interesting to investigate to what extent these covert channels extend to spying". In this section, we fill this research gap by developing PORTSMASH, a new timing side-channel vector via port contention.

At a high level, in PORTSMASH the goal of the Spy is to saturate one or more ports with a combination of full instruction pipelining and/or generous instruction level parallelism. By measuring the time required to execute a series of said instructions, the Spy learns about the Victim's rate and utilization of these targeted ports. A higher latency observed by the Spy implies port contention with the Victim, i.e. the Victim issued instructions executed through said ports. A lower latency implies the Victim did not issue such instructions, and/or stalled due to a hazard or waiting due to e.g. a cache miss. If the Victim's ability to keep the pipeline full and utilize instruction level parallelism depends on a secret, the Spy's timing information potentially leaks that secret.

As a simple example conceptually related to our later application in Section 5, consider binary scalar multiplication for elliptic curves. Each projective elliptic curve point double and conditional addition is made up of a number of finite field additions, subtractions, shifts, multiplications, and squarings. These finite field operations utilize the pipeline and ports in very different ways and have asymptotically different running times. For example, shifts are extremely parallelizable, while additions via add-with-carry are strictly serial. Furthermore, the number and order of these finite field operations is not the same for point doubling and addition. The Spy can potentially learn this secret sequence of doubles and conditional additions by measuring its own performance through selective ports, leading to (secret) scalar disclosure.

```
mov $COUNT, %rcx

1:
lfence
rdtsc
lfence
mov %rax, %rsi

#ifdef P1
.rept 48
crc32 %r8, %r8
crc32 %r9, %r9
crc32 %r10, %r10
.endr
#elif defined(P5)
.rept 48
vpermd %ymm0, %ymm1, %ymm0
vpermd %ymm2, %ymm3, %ymm2
vpermd %ymm4, %ymm5, %ymm4
.endr
#elif defined(P0156)
.rept 64
add %r8, %r8
add %r9, %r9
add %r10, %r10
add %r11, %r11
.endr
#else
#error No ports defined
#endif

lfence
rdtsc
shl $32, %rax
or %rsi, %rax
mov %rax, (%rdi)
add $8, %rdi
dec %rcx
jnz 1b
```

Figure 3. The PORTSMASH technique with multiple build-time port configurations `P1`, `P5`, and `P0156`.

Figure 3 lists our proposed PORTSMASH Spy process. The first `rdtsc` wrapped by `lfence` establishes the start time. Then, depending on the architecture and target port(s), the Spy executes one of several strategies to saturate the port(s). Once those complete, the second `rdtsc` establishes the end time. These two counters are concatenated and stored out to a buffer at `rdi`. The Spy then repeats this entire process. Here we choose to store the counter values and not only the latency, as the former helps identify interrupts (e.g. context switches) and the latter can always be derived offline from the former, but the converse is not true. It is also worth mentioning the Spy must ensure some reasonable number of instructions retired between successive `rdtsc` calls to be able to reliably detect port contention; we expand later.

In general, strategies are architecture dependent and on each architecture there are several strategies, depending on what port(s) the Spy wishes to measure. We now provide and describe three such example strategies (amongst several others that naturally follow) for Intel Skylake and Kaby Lake: one that leverages instruction level parallelism and targets multiple ports with a latency-1 instruction, and two that leverage pipelining and target a single port with higher latency instructions.

Multiple ports. In Figure 3, the `P0156` block targets ports 0, 1, 5, and 6. These four `add` instructions do not create hazards, hence all four can execute in parallel to the four integer ALUs behind these ports, and as a latency-1 instruction in total they should consume a single clock cycle. To provide a window to detect port contention, the Spy replicates these instructions 64 times. With no port contention, this should execute in 64 clock cycles, and 128 clock cycles with full port contention.

Single port. In Figure 3, the `P1` and `P5` blocks target port 1 and 5, respectively, in a similar fashion. Since these are latency-3 instructions, we pipeline three sequential instructions with distinct arguments to avoid hazards and fill the pipeline, achieving full throughput of one instruction per cycle. Here the window size is 48, so the block executes with a minimum $3 \cdot 48 + 2 = 146$ clock cycles with no port contention, and with full port contention the maximum is roughly twice that.

4.1. Comparison

Our PORTSMASH technique relies on secret-dependent *execution port footprint*, a closely related concept to secret-dependent *instruction execution cache footprint*. Although similar in spirit to L1 icache attacks or LLC cache attacks, since both rely on a secret-dependent footprint in a microarchitecture component, we demonstrate that PORTSMASH offers finer granularity and is stealthier compared to other techniques. To differentiate PORTSMASH from previous techniques, we compare them with respect to spatial resolution, detectability, cross-core, and cross-VM applicability. We admit that detectability is extremely subjective, especially across different microarchitecture components; our rating is with respect to a malicious program while the target victim is idle, i.e. waiting to capture.

Initially, Osvik et al. [18] proposed the PRIME+PROBE technique against the L1 dcache, relying on SMT technology to provide asynchronous execution. Newer enhancements to this technique allowed cross-core (and cross-VM) successful attacks [22–24]. The spatial resolution of this attack is limited to cache-set granularity, that is usually at least of 512 bytes. Typically, the PRIME+PROBE technique occupies all cache sets, moderately detectable if cache activity monitoring takes place.

Later, Yarom and Falkner [19] proposed the FLUSH+RELOAD technique, a high resolution side-channel providing cache-line granularity with an improved eviction strategy. Closely related, Gruss et al. [20] proposed FLUSH+FLUSH, a stealthier version of FLUSH+RELOAD. Both techniques rely on shared memory between Victim and Spy processes, in addition to the `clflush` instruction to evict cache lines from the LLC. While this is a typical setting in cross-core scenarios due to the use of shared libraries, the impact in cross-VM environments is limited due to the common practice of disabling page de-duplication [25, Sec. 3.2]. FLUSH+RELOAD constantly reloads from the same address it flushes, hence is highly detectable if the number of cache-misses is monitored (Section 6 expands on this). In contrast, FLUSH+FLUSH does not perform loads at all; stealthiness is one of its strengths.

More recently, Gras et al. [4] proposed TLBLEED as another microarchitecture attack technique. Even if this is not a "pure" cache technique, it exploits TLBs, a form of cache for memory address translations [7]. Interestingly, this subtle distinction is sufficient for making it stealthier to cache countermeasures [4]. On the downside, the spatial resolution of this attack is limited to a memory page (4 KB). Since no cross-core improvements have been proposed for either TLBLEED or PORTSMASH, it could be seen as a drawback of these attacks. However, attackers can launch multiple Spy processes to occupy all cores and ensure co-location on the same physical core; see [26, Sec. 3.1] for a related discussion.

```
30f0  <x64_foo>:
30f0  test   %rdi,%rdi
30f3  je     4100 <x64_foo+0x1010>
30f9  jmpq   4120 <x64_foo+0x1030>
....
4100  popcnt %r8,%r8
4105  popcnt %r9,%r9
410a  popcnt %r10,%r10
410f  popcnt %r8,%r8
4114  popcnt %r9,%r9
4119  popcnt %r10,%r10
411e  jmp    4100 <x64_foo+0x1010>
4120  vpbroadcastd %xmm0,%ymm0
4125  vpbroadcastd %xmm1,%ymm1
412a  vpbroadcastd %xmm2,%ymm2
412f  vpbroadcastd %xmm0,%ymm0
4134  vpbroadcastd %xmm1,%ymm1
4139  vpbroadcastd %xmm2,%ymm2
413e  jmp    4120 <x64_foo+0x1030>
4140  retq
```

```
4150  <x64_bar>:
4150  test   %rdi,%rdi
4153  je     5100 <x64_bar+0xfb0>
4159  jmpq   5140 <x64_bar+0xff0>
....
5100  popcnt %r8,%r8
5105  popcnt %r9,%r9
510a  popcnt %r10,%r10
510f  popcnt %r8,%r8
5114  popcnt %r9,%r9
5119  popcnt %r10,%r10
511e  popcnt %r8,%r8
5123  popcnt %r9,%r9
5128  popcnt %r10,%r10
512d  popcnt %r8,%r8
5132  popcnt %r9,%r9
5137  popcnt %r10,%r10
513c  jmp    5100 <x64_bar+0xfb0>
513e  xchg   %ax,%ax
5140  vpbroadcastd %xmm0,%ymm0
5145  vpbroadcastd %xmm1,%ymm1
514a  vpbroadcastd %xmm2,%ymm2
514f  vpbroadcastd %xmm0,%ymm0
5154  vpbroadcastd %xmm1,%ymm1
5159  vpbroadcastd %xmm2,%ymm2
515e  vpbroadcastd %xmm0,%ymm0
5163  vpbroadcastd %xmm1,%ymm1
5168  vpbroadcastd %xmm2,%ymm2
516d  vpbroadcastd %xmm0,%ymm0
5172  vpbroadcastd %xmm1,%ymm1
5177  vpbroadcastd %xmm2,%ymm2
517c  jmp    5140 <x64_bar+0xff0>
517e  retq
```

Figure 4. Two Victims with similar port footprint i.e. port 1 and port 5, but different cache footprint. Left: Instructions span a single cache-line. Right: Instructions span multiple cache-lines.

Recent microarchitecture attacks have been proposed achieving intra cache-line granularity. Yarom et al. [5] demonstrated that intra-cache granularity is possible—at least in older Intel microprocessors—with their CacheBleed attack. This attack proposes two techniques to achieve this granularity: cache bank conflicts and *write-after-read* false dependencies. Cache bank conflicts have a limited impact, considering the authors discovered that current Intel microprocessors no longer have cache banks; thus this technique does not apply to newer microprocessors. To that end, Moghimi et al. [21] improved the previous work and proposed a *read-after-write* false dependency side-channel. The authors highlight the potential 5 cycle penalty introduced when a memory write is closely followed by a read, a more critical condition compared to a read closely followed by a memory write. This technique gives a 4-byte granularity on the cache-lines, thus allowing them to exploit the 5 cycle delay to perform a key recovery attack against a constant-time AES implementation on Intel IPP library. Both attacks are less detectable than FLUSH+RELOAD since they do not utilize `clflush` at all.

Table 3 compares the previously mentioned techniques in their original version. As can be appreciated, our PORTSMASH technique enjoys the highest spatial resolution among them, since it goes beyond the cache-line and instead, it considers individual *uops* dispatched to the execution

2. Cache-set size depends on the microprocessor specifications and can be calculated as *(cache line size* $\times$ *cache associativity)*.

TABLE 3. COMPARISON OF MICROARCHITECTURE ATTACK TECHNIQUES (ORIGINAL VERSIONS)

Attack	Spatial Resolution	Size	Detectability	Cross-Core	Cross-VM
TLBLEED [4]	Memory Page (Very low)	4 KB	Low	No	Yes/SMT
PRIME+PROBE [18]	Cache-set (Low)	512 bytes[2]	Medium	Yes	Yes/SharedMem
FLUSH+RELOAD [19]	Cache-line (Med)	64 bytes	High	Yes	Yes/SharedMem
FLUSH+FLUSH [20]	Cache-line (Med)	64 bytes	Low	Yes	Yes/SharedMem
CacheBleed [5]	Intra cache-line (High)	8 bytes	Medium	No	Yes/SMT
MemJam [21]	Intra cache-line (High)	4 bytes	Medium	No	Yes/SMT
PORTSMASH	Execution port (Very High)	*uops*	Low	No	Yes/SMT

units. As an example, consider the two functions `x64_foo` and `x64_bar` in Figure 4. These two functions get passed an argument of either zero or one (e.g. a secret bit): in the former case, they start executing pipelined `popcnt` instructions in a loop, and `vpbroadcastd` instructions in the latter. The `x64_foo` function has all its functionality for both branches within a single cache line (64B), starting at address `0x4100`. In contrast, the `x64_bar` function has distinct cache lines for each branch: the zero case starts at address `0x5100` and the one case at `0x5140`, and the control flow for each corresponding loop restricts to its single cache line.

The `x64_bar` function is a potential target for L1 icache attacks, FLUSH+RELOAD attacks, FLUSH+FLUSH attacks, etc. since there are two different targets that span two different cache lines. In contrast, the `x64_foo` control flow resides in a single cache line: L1 icache attacks, FLUSH+RELOAD attacks, FLUSH+FLUSH attacks, etc. only have cache line granularity, and are not designed to distinguish varying code traversal within a single line. Remarkably, both `x64_foo` and `x64_bar` are potential targets for our new method. In this light, at a very high level what CacheBleed accomplished for dcache attacks—the ability to target at less than data cache line granularity—our method accomplishes for the code side, and furthermore potentially with a single trace instead of averaging traces.

To validate our findings, we ran the following set of PORTSMASH experiments. First, we configured the Victim process to execute the `x64_foo` function passing 0 as argument, causing the Victim to issue `popcnt` commands, using port 1. In parallel, we configured the Spy process with the `P1` strategy in the sibling logical core to issue and time `crc32` commands, thus creating contention and the Spy successfully tracks the Victim state by observing high latency. Then, we repeated the experiment but this time we passed 1 as argument to the Victim process, executing `vpbroadcastd` instructions, using port 5. Since the Spy process is still using the `P1` strategy, i.e. timing `crc32` instructions, port contention does not happen, hence the Spy successfully tracks the Victim state by observing low latency. Figure 5 (Top) shows the resulting trace for both cases i.e. contention vs no-contention from a Spy process perspective configured with the `P1` strategy. We then re-configured the Spy to use the `P5` strategy, and repeated the experiments, shown in Figure 5 (Bottom). This raw empirical data—that is clearly linearly separable—confirms not only the validity of our new side-channel in general,

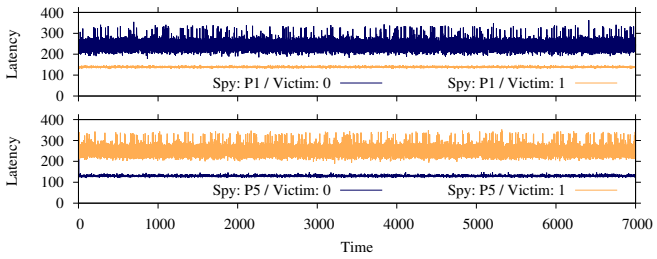

Figure 5. Top: Timings for the PORTSMASH Spy when configured with `P1`, in parallel to the Victim executing `x64_foo` with `rdi` as both zero and one in two consecutive runs. Bottom: Analogous but with the Spy configured with `P5`.

but furthermore the symmetry in the plots confirms that our technique even allows to leak code traversal information with a granularity finer than cache-line, since in this case it is dependent on port utilization by the executed instructions within the cache-line.

5. Applications

In the previous section, we developed a generic PORT SMASH Spy process to procure timing signals that detect port contention. In this section, we present the first attack using our technique in a real-world setting. We start with some background on ECC, and explain why P-384 is a highly relevant standardized elliptic curve, and examine its scalar multiplication code path within OpenSSL 1.1.0h and earlier. We then design and implement an end-to-end P-384 private key recovery attack that consists of three phases:

1) In the *procurement phase*, we target an stunnel TLS server authenticating with a P-384 certificate, using our tooling that queries the TLS server over multiple handshakes with the Spy measuring port contention in parallel as the server produces ECDSA signatures.
2) In the *signal processing phase*, we filter these traces and output partial ECDSA nonce information for each digital signature.
3) In the *key recovery phase*, we utilize this partial nonce information in a lattice attack to fully recover the server's P-384 private key.

We close this section with a discussion on applications to statically linked binaries and SGX enclaves.

5.1. ECC and P-384

Koblitz [27] and Miller [28] introduced elliptic curves to cryptography during the mid 1980's. By 1995, the National Security Agency (NSA) became a strong supporter of Elliptic Curve Cryptography (ECC) [29] and pushed for the adoption of ECDSA, the elliptic curve variant of the (then) recently approved Digital Signature Algorithm (DSA) [30].

In 2005, NSA's support of ECC was clear, mandating its use "for protecting both classified and unclassified National Security information[..], the NSA plans to use the elliptic curves over finite fields with large prime moduli (256, 384, and 521 bits) published by NIST" [31]. Shortly after, the NSA announced Suite B, a document recommending cryptography algorithms approved for protecting classified information up to Secret and Top Secret information, including P-256 at 128 bits of security, and P-384 at 192 bits of security, respectively.

During 2012, the Committee for National Security Systems (CNSS) issued CNSSP-15 [32], a document defining the set of public key cryptographic standards recommended to protect classified information until public standards for post-quantum cryptography (PQC) materialize, further pushing the support for both curves, P-256 and P-384. Suddenly in August 2015, and after a long history of ECC support, the NSA released a statement [33] urging the development of PQC and discouraging the late adoption of ECC, and instead focusing on the upcoming upgrade to quantum-resistant algorithms. Parallel to this statement, the Suite B recommendation was updated, mysteriously removing P-256 from the list of approved curves without giving any reason, and leaving P-384 as the only ECC option to protect information up to Top Secret level. In January 2016, the NSA issued a FAQ [34] derived from the statement released five months prior. They informed about the replacement of Suite B with an updated version of CNSS-15, and also finally commented on the removal of P-256 from the previous Suite B. We cherry-pick three statements from the document: (1) "equipment for NSS that is being built and deployed now using ECC should be held to a higher standard than is offered by P-256"; (2) "Elimination of the lower level of Suite B also resolves an interoperability problem raised by having two levels"; and (3) "CNSSP-15 does not permit use of P-521".

To summarize, P-384 is the only compliant ECC option for Secret and Top Secret levels. Unfortunately, its implementations have not received the same scrutiny as P-256 and P-521; we expand later in this section.

ECDSA. For the purpose of this paper we restrict to short Weierstrass curves over prime fields. With prime $p > 3$, all of the $x, y \in GF(p)$ solutions to the equation

$$E : y^2 = x^3 + ax + b$$

along with the point-at-infinity (identity) form a group. The domain parameters of interest are the NIST standard curve that set p Mersenne-like prime and $a = -3 \in GF(p)$.

The user's private-public keypair is (d_A, Q_A) where d_A is chosen uniformly from $[1\,..\,n)$ and $Q_A = [d_A]G$ holds. G is a generator $G \in E$ of prime order n. A digital signature on message m compute as follows.

1) Select a secret nonce k uniformly from $[1\,..\,n)$.
2) Compute $r = (k[G])_x \bmod n$.
3) Compute $s = k^{-1}(h(m) + d_A r) \bmod n$.
4) Return the digital signature tuple (m, r, s).

The hash function h can be any "approved" function, e.g. SHA-1, SHA-256, and SHA-512. Verification is not relevant to this work, hence we omit the description.

ECDSA and P-384 in OpenSSL. In OpenSSL, each elliptic curve has an associated method structure containing function pointers to common ECC operations. For ECDSA, scalar multiplication is the most performance and security-critical ECC operation defined in this method structure, and the actual algorithm to perform scalar multiplication depends on several factors, e.g. curve instantiated, scalar representation, OpenSSL version, and both library and application build-time options. Due to the long history of timing attacks against ECDSA and the possibility of improving the performance of some curves, over the years OpenSSL mainlined several implementations for scalar multiplication, especially for popular NIST curves over prime fields.

Based on work by Käsper [35]—and as a response to the data cache-timing attack by Brumley and Hakala [36]—OpenSSL introduced `EC_GFp_nistp256_method`, a constant-time scalar multiplication method for the NIST P-256 curve (and analogous methods for P-224 and P-521). This method uses secure table lookups (through masking) and a fixed-window combing during scalar multiplication. Later, Gueron and Krasnov [37] introduced a faster constant-time method with their `EC_GFp_nistz256_method`. This method uses Intel AVX2 SIMD assembly to increase the performance of finite field operations, thus providing a considerable speedup when compared to `EC_GFp_nistp256_method` that is portable C. The NIST curve P-256 quickly became (arguably) the most widely used, fast, and timing-attack resistant of all NIST curves in OpenSSL. Unfortunately, P-384 was neglected, and it missed all of the previous curve-specific improvements that provided timing attack security for P-224, P-256, and P-521. Instead, P-384 follows a generic non constant-time scalar multiplication algorithm—i.e. interleaved scalar multiplication by Möller [38, Sec. 3.2]—which uses a modified windowed Non-Adjacent Form (wNAF) for scalar representation. Although this implementation is a known vulnerability [36, 39–41], it has never been exploited in the context of P-384 in OpenSSL.

In OpenSSL 1.1.0h and below, P-384 calls `ecdsa_sign_setup` @ `crypto/ec/ecdsa_ossl.c` when generating an ECDSA signature. There, the underlying `ec_wNAF_mul` function gets called to perform scalar multiplications, where $r = [k]G$ is the relevant computation for this work. That function first transforms the scalar k to its wNAF representation and then, based on this

representation, the actual scalar multiplication algorithm executes a series of *double* and *add* operations. To perform double and add operations, OpenSSL calls `ec_GFp_simple_dbl` and `ec_GFp_simple_add` respectively. There, these methods have several function calls to simpler and lower level Montgomery arithmetic, e.g. shift, add, subtract, multiply, and square operations. A single ECC double (or add) operation performs several calls to these arithmetic functions. Among the strategies mentioned in Section 4, we found that for our target the `P5` strategy results in the least noisy trace overall.

In summary, by using the PORTSMASH technique during OpenSSL P-384 ECDSA signature generation, we can measure the timing variations due to port contention. More specifically, we capture the port contention delay during double and add operations, resulting in an accurate raw signal trace containing the sequence of operations during scalar multiplication, and leaking enough LSDs of multiple nonces k to later succeed in our *key recovery phase*.

5.2. Procurement Phase: TLS

Stunnel[3] provides TLS/SSL tunneling services to servers (and clients) that do not speak TLS natively; during the *procurement phase* we used stunnel 5.49 as the TLS server. We compiled it from source and linked it against OpenSSL 1.1.0h for crypto functionality. Our setup consists of an Intel Core i7-6700 Skylake 3.40GHz featuring Hyper-Threading, with four cores and eight threads, running Ubuntu 18.04 LTS "Bionic Beaver", and TurboBoost disabled.

We configured the stunnel server with a P-384 ECDSA certificate and `ECDHE-ECDSA-AES128-SHA256` as the TLS 1.2 cipher suite. We wrote a custom TLS client to connect to our TLS server. Typically, during a TLS handshake, the client and the server exchange several protocol messages, including `ClientHello`, `ServerHello` and `ServerKeyExchange` parameters. These messages are concatenated, hashed and digitally signed by the server. Then, the client verifies the signature before finally establishing a session with the server.

Our custom TLS client serves two purposes: (1) it starts the TLS handshake with the stunnel server, collecting protocol messages and digital signatures; and (2) it launches the Spy process, which captures the scalar multiplication operations performed by OpenSSL on the server side during the handshake, both for Diffie-Hellman key agreement and ECDSA signature generation. Figure 6 (Top) shows a trace captured by the Spy process, containing the two scalar multiplication operations during TLS handshake, i.e. ECDH and ECDSA respectively.

The client drops the handshake as soon as the server presents the digital signature; since we are only interested in capturing up to the digital signature generation, this allows us to rapidly capture more traces. Additionally, our client concatenates the protocol messages, hashes the resulting concatenation, and stores the message digest. Similarly, it stores the respective DER-encoded P-384 ECDSA signatures for each TLS handshake. This process is repeated as needed to build a set of traces, digest messages and digital signatures that our lattice attack uses later in the *key recovery phase*.

3. https://www.stunnel.org

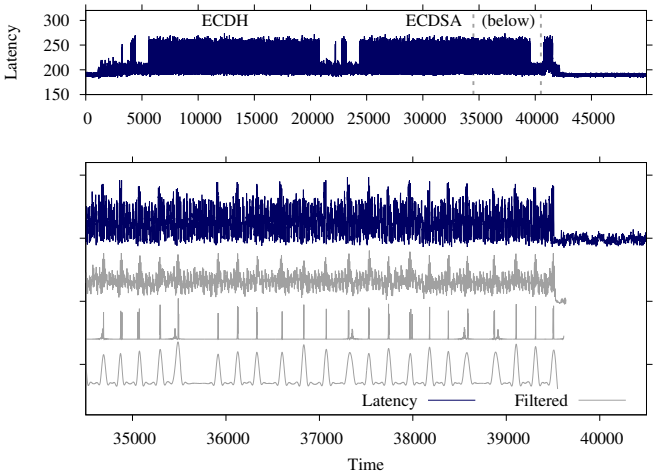

Figure 6. Multiple TLS trace stages. Top: Raw TLS handshake trace showing scalar multiplication during ECDH and ECDSA. Bottom: Zoom at the end of the previous ECDSA trace, peaks (filtered) represent Add operations. For example, this trace ends with an add operation, indicating the nonce is odd.

Once the data tuples are captured, we proceed to the *signal processing phase*, where the traces are trimmed and filtered to reduce the noise and output useful information. Figure 6 (Bottom) shows a zoom at the end of the (Top) trace, where the filters reveals peaks representing addition operations, separated by several double operations.

At a high level—returning to the discussion in Section 4—the reason our signal modulates is as follows. The wNAF algorithm executes a (secret) sequence of double and add operations. In turn, these operations are sequences of finite field additions, subtractions, multiplications, and squarings. Yet the number and order of these finite field operations are not identical for double and add. This is eventually reflected in their transient port utilization footprint.

5.3. Signal Processing Phase

After verifying the existence of SCA leakage in the captured TLS traces, we aim to extract the last Double and Add sequence to provide partial nonce information to the *key recovery phase*. Although visual inspection of the raw trace reveals the position of Double and Add operations, this is not enough to automatically and reliably extract the sequence due to noise and other signal artifacts.

Since our target is ECDSA point multiplication, we cropped it from the rest of the TLS handshake by applying a root mean square envelope over the entire trace. This resulted in a template used to extract the second point multiplication corresponding to the ECDSA signature generation. To further improve our results, we correlated the traces to

the patterns found at the beginning and end of the point multiplication. This was possible as the beginning shows a clear pattern (trigger) due to OpenSSL precomputation, and the end of the trace has a sudden decrease in amplitude.

We then used a low pass filter on the raw point multiplication trace to remove any high frequency noise. Having previously located the end of point multiplication, we focused on isolating the Add operations to get the last Add peak, while estimating the Doubles using their length. For doing this we applied source separation filtering method known as Singular Spectrum Analysis (SSA) [42]. SSA was first suggested in SCA literature for power analysis to increase signal to noise ratio in DPA attacks [43], and later used as a source separation tool for extracting Add operations in an EM SCA attack on ECDSA [44].

The SSA filter performs an eigen-spectra decomposition of the original signal using a trajectory matrix into different components which are then analyzed and selected accordingly for reconstructing a filtered signal. The first step *embedding* converts the single dimension signal $\{m_k\}_{k=1}^{N}$ of length N into a multidimensional trajectory matrix M which contains I column vectors each of size w where $I = N - w + 1$. The window size $1 < w \leq N/2$ dictates the quality and performance of the reconstruction phase. The second step *singular value decomposition* (SVD) decomposes the trajectory matrix M into non-zero eigenvalues λ_k of $MM^\top$ sorted in decreasing ranks of their magnitudes along with their corresponding eigenvectors u_k. With $v_k = M^\top u_k \sqrt{\lambda_k}$ and $Y_k = u_k v_k$ the projection matrices, SVD can be shown as:

$$M = \sum_{k=1}^{d} \sqrt{\lambda_k} Y_k^\top$$

To obtain the reconstructed components $\{y_i\}_{i=1}^{N}$, next perform a diagonal averaging also known as Hankelization by computing the average over the skewed diagonal of the projection matrices Y_k [45]. The original signal can thus be reproduced by summing all the reconstructed components:

$$\{m_i\}_{i=1}^{N} = \sum_{k=1}^{d} \{y_i^k\}_{i=1}^{N}$$

For source separation, only the useful components can be chosen, leaving out the noisy ones from all the d possible choices.

For our purpose, we decided to threshold the window size as suggested in [43]. Since the total length of the trace was around 15000 samples, this gave us a window size of 30. However, based on experimentation, a window of size 20 yielded optimal results using the second and the third component.

The traces occasionally encountered OS preemptions corrupting them due to the Spy or Victim being interrupted. We detect Spy interrupts as high amplitude peaks or low amplitude gaps, depending on whether they happened while during or between latency measurement windows. Similarly the Victim interrupts exhibit a low amplitude gap in our traces, since there was no Victim activity in parallel. In any case, we discarded all such traces (around 2.5%) when detecting any interrupt during the last Double and Add sequence.

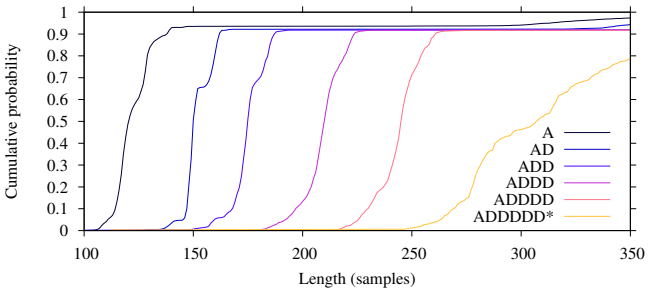

Figure 7. Length distributions for various patterns at the end of scalar multiplication.

Finally by applying continuous wavelet transform [46] in the time-frequency domain we were able to detect the high energy Add peaks, therefore isolating them. Moreover, a root-mean-square of the resulting peaks smoothed out any irregularities. Figure 6 illustrates the results of signal processing steps on a TLS trace from top to bottom. Even after applying these steps, some traces where the Adds were indistinguishable due to noise still occur, decreasing the accuracy of our results by about 2%.

The output of this phase, for each trace, is the distance from the last Add operation to the end of the point multiplication: estimating the number of trailing Doubles by counting the number of samples. Figure 7 depicts the CDF of the resulting sequences using our distance metric, having clear separation for each trailing Double and Add sequence.

5.4. Key Recovery Phase: Lattices

For the lattice attack stage we build on the work presented in [40], using the BKZ reduction algorithm to efficiently look for solutions to the Shortest Vector Problem (SVP). The output of the *signal processing phase* eventually provides us with partial nonce information, as the trailing sequence tells us the bit position of the lowest set bit.

We first describe exploring the lattice parameter space using traces modeled without errors. We then combine this study with profiling of the experimental data and the constraints of the computational resources at our disposal to launch a real world end-to-end attack.

Exploration of the lattice parameter space. The main parameters to tune in implementing the lattice attack are the size (d) of the set of signatures used to build the lattice basis and the block size (β) for the BKZ reduction algorithm. The parameter d is directly proportional to the amount of bits of knowledge available in the lattice structure; the product $d \times \beta$ is inversely related to the average number of BKZ iterations required to retrieve the private key, but is directly related to the execution time of a single iteration.

Theoretically, given an infinite amount of time, if the selected subset of signatures does not contain any error and if the lattice embeds more bits of knowledge than the bit-length of the secret key, it should eventually succeed. In this scenario, optimizing for the smallest d that delivers enough bits of knowledge to recover the private key would be the preferred metric, as it requires less overall data from the *procurement phase* (lowering the risk of detection) and also improves success chances of the heuristic process (dealing with potential errors in the *signal processing phase*).

In a more realistic scenario we want to model the lattice parameters to succeed in a "reasonable" amount of time. This definition is not rigorous and largely depends on the capabilities of a dedicated attacker: in this academic contest, constrained by the grid computing resources available to us, we define a lattice instance as *successful* if it correctly retrieves the secret key in under 4 hours when running on a single 2.50 GHz Xeon E5-2680 core.

We modeled our preliminary experiments using random nonces, biased to have a trailing sequence of zero bits: i.e., this is equivalent to assuming error-free traces from the *signal processing phase*. We run two sets of experiments, one with a bias mask of `0x3`, i.e., with at least two trailing zero bits (using the notation from [40], $z \geq 2$ and $l \geq 3$), and the other with a bias mask of `0x1`, i.e., with at least one trailing zero bit ($z \geq 1$ and $l \geq 2$).

Regarding the block size β, our experiments (for which detailed results are provided in Appendix A) confirm the findings in [40]: "The lower block sizes perform better in the higher dimensions, as the high-dimensional lattices already contain much information and strong reduction is not required." Building on their results, we started with $\beta \in \{10, 20, 15\}$ and analyzed the trade-off between an increase in the number of iterations against a generally faster execution time for each iteration. In terms of overall execution time, we ultimately determined that $\beta = 20$ performed better (even for extremely high lattice dimensions).

To determine the optimal d for each bias case, we ran 10000 instances of the lattice algorithm against the two sets of modeled perfect traces and measured the corresponding amount of known nonce bits, the number of iterations for successful instances, the overall execution time for successful instances, and the *success* probability. The results indicate $d = 450$ is optimal for the `0x1` biased ideal traces, with success probability exceeding 90% coupled with a small number of iterations as well as overall execution time. Analogously, $d = 170$ is optimal for the `0x3` bias case.

Experimental parameters with real traces. Real traces come with errors, which lattices have no recourse to compensate for. The traditional solution is oversampling, using a larger set of t traces (with some amount e of traces with errors), and running in parallel a number (i) of lattice instances, each picking a different subset of size d from the larger set. Picking the subsets uniformly random, the probability for any subset to be error-free is:

$$\Pr(\textit{No error in a random subset of size } d) = \frac{\binom{t-e}{d}}{\binom{t}{d}}$$

For typical values of $\{t, e, d\}$, the above probability is small and not immediately practical. But given the current capabilities for parallelizing workloads on computing clusters, repeatedly picking different subsets compensates:

$$\Pr(\geq 1 \textit{ error-free subset over } i \textit{ inst.}) = 1 - \left(1 - \frac{\binom{t-e}{d}}{\binom{t}{d}}\right)^{i} \quad (1)$$

Profiling the *signal processing phase* results, we determined to utilize thresholding to select traces belonging to the "AD", "ADD", "ADDD" and "ADDDD" distributions of Figure 7. In our setup, other traces are either useless for our lattice model or have too low accuracy. To ensure accuracy, we determined very narrow boundaries around the distributions to limit overlaps at the cost of very strict filtering. Out of the original 10000 captures, the filtering process selects a set of 1959 traces with a `0x1` bias mask including $e = 34$ (1.74%) errors. Combining this with $d = 450$ from our empirical lattice data, (1) leads us to $i \geq 36000$ instances required to achieve a probability $\geq 99\%$ of picking at least one subset without errors. This number of instances is beyond the parallel computational resources available to us, hence we move to the remaining case.

Filtering out also the 1060 traces categorized as `'AD'` delivers a total of 899 traces with a `0x3` bias mask, including $e = 14$ (1.56%) errors. Combining this with $d = 170$ for the higher nonce bias and substituting in (1) leads us to $i \geq 200$ instances to achieve a probability $\geq 99.99\%$ of picking at least one subset without errors.

When using the actual attack data we noticed that while our filtering process increasing accuracy, it has the side-effect of straying from the statistics determined in ideal conditions. We speculate this is due to filtering out longer trailing zero bits (low accuracy) decreasing the average amount of known nonce bits per signature, resulting in wider lattice dimensions with lower than expected useful information. This negatively affects the overall success rate and the amount of required iterations for a successful instance. We experimentally determined that when selecting only nonces with a bias mask between `0x3` and `0xF`, $d = 290$ compensates with a success rate (for an error-free subset) of 90.72%. Using these values in (1) leads us to $i = 2000$ instances to achieve a probability 99.97% of picking at least one subset without errors—well within the computing resources available to us.

Finally, running the entire process on the real data obtained from the *signal processing phase* on the original 10000 captures, using parameters $t = 899$, $e = 14$, and $d = 290$ over $i = 2000$ instances running in parallel on the described cluster resulted in 11 instances that successfully retrieved the secret key, the fastest of which terminated in 259 seconds after only two BKZ reduction iterations.

5.5. SGX

Intel Software Guard Extensions (SGX)[4] is a micro architecture extension present in modern Intel processors. SGX aims at protecting software modules by providing integrity and confidentiality to their code and memory contents. In SGX terminology, an SGX *enclave* is a software module that enjoys said protections. SGX was designed to defend processes against tampering and inspection from OS-privileged adversaries, therefore, it provides strong isolation between enclave memory regions and the outer world. Despite these strong protections, side-channel attacks are still considered a major threat for enclaves, as SGX by itself does not protect against them [47]. In this regard, as the SGX threat model considers an OS-level adversary, it is even possible to mount more powerful side-channel attacks against enclaves where the measurement noise can be reduced considerably [48–50].

From a practical perspective, it is interesting to know which unprivileged side-channel techniques are a threat for SGX enclaves. Regarding cache attacks, FLUSH+RELOAD and FLUSH+FLUSH do not apply for the unprivileged scenario since they require shared memory with the SGX enclave, but the latter does not share its memory [15, 47]. However, researchers use other attack techniques against SGX, such as L1-based PRIME+PROBE attacks [50], and false dependency attacks Moghimi et al. [21]. It is worth mentioning that these methods assume an attacker with privileged access, however, we strongly believe these attacks would succeed without this assumption at the cost of capturing traces with a higher signal-to-noise ratio. Finally, TLBLEED [4] could be a potential successful attack technique against SGX, yet the authors leave it for future work.

The rest of this section demonstrates our PORTSMASH technique against SGX enclaves. By definition, it should apply without issues since SGX technology does not protect the execution units. This fills a research gap left by Covert Shotgun as an open problem: "That would be especially interesting say in SGX scenarios." For our experiments we developed two Victim processes. One Victim is a standard processes linked against OpenSSL 1.1.0h, and the other is an Intel SGX SSL enclave. Both Victim processes follow the same scalar multiplication code path analyzed in Section 5, therefore we have two processes executing exactly the same code path with and without SGX protections.

Following the rationale that a PORTSMASH attack is oblivious to SGX enclaves, we applied the `P5` strategy employed in Section 5. We captured two traces on an Intel Core i7-7700HQ (i.e. Kaby Lake), one for each setting: SGX and non-SGX. Figure 8 shows both the raw and filtered traces for each of them. Note the similarities between both raw traces, and after applying a noise reduction filter, the similarities become more evident since the position of additions are clearly revealed in both traces as amplitude peaks. This experiment validates our hypothesis that a PORT SMASH attack can be applied to SGX enclaves as well as to non-SGX processes. Moreover, it also proves that the amount of noise does not significantly vary between both scenarios.

4. https://software.intel.com/en-us/sgx

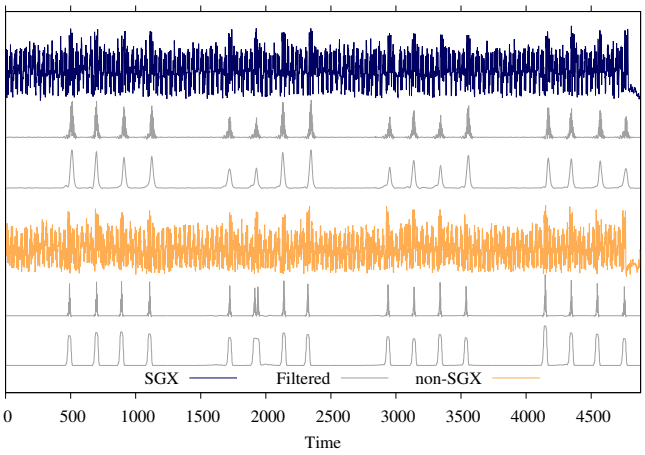

Figure 8. From top to bottom: raw trace of our SGX Victim; said trace after filtering; raw trace of our user space Victim; said trace after filtering. Both victims received the same input, i.e. a scalar that induces 16 point additions at the end of the trace, clearly identifiable by the peaks in the filtered traces. This demonstrates the leakage from SGX is essentially identically to the leakage outside SGX.

6. Mitigations

6.1. Existing Work

Due to the copious amount of microarchitecture side-channel attacks in recent years, several countermeasures and mitigations appear in the literature; see [51] for a complete survey on countermeasures. From all the microarchitecture side-channel attacks proposed, cache-timing attacks and their respective techniques have arguably the most impact of all. This translates to the development of specific memory-based mitigations such as cache partitioning [11, 52], cache flushing [53, 54], and (partially) disabling caching [16]. Nevertheless, generally these solutions do not provide protections against non memory-based side-channels. To that end, another mitigation technique angle follows malware analysis methods. One way to categorize these countermeasures is by *binary* and *runtime* analysis.

Binary analysis looks for *code signatures* that allows classifying a binary file as a malicious file or not. Irazoqui et al. [55] proposed MASCAT, a binary analysis framework for detecting microarchitecture malware. This framework analyzes a target binary by looking for the signature of a set of instructions often used during microarchitecture attacks, e.g. high-resolution timers, fence instructions and cache-flushing instructions. Nevertheless, [15] showed that is possible to hide malicious code from static analysis of binaries.

Runtime analysis inspects potentially malicious processes while they execute, looking for dubious activities. Several approaches propose microarchitecture attack mitigations [56–58]. Most of them focus mainly on monitoring

hardware performance counters (HPC) to detect irregular execution patterns that may suggest an ongoing side-channel attack. Kulah et al. [56] and Zhang et al. [57] focus on unusual cache-activity rates, while Raj and Dharanipragada [58] aims at detecting an attack by measuring memory bandwidth differences.

Wichelmann et al. [59] recently proposed a combination of these categories. Their framework MicroWalk applies Dynamic Binary Instrumentation and Mutual Information Analysis to not only detect leakage in binaries, but also to locate the source of the leakage in the binary. The framework combines the memory footprint and the program control-flow to determine the side-channel leakage. They apply their technique successfully to closed source cryptographic libraries such as Intel IPP and Microsoft CNG.

From this brief survey, most of the work to mitigate microarchitecture side-channels is in the area of cache-based channels. Hence, many of these frameworks and techniques are not directly applicable to detect and mitigate our PORT SMASH technique. Since our technique does not target the cache, but instead focuses on the execution units, we argue it is extremely challenging to detect it. For example, when using an HPC-based countermeasure, it must distinguish normal port utilization between highly optimized code and PORTSMASH. At the end of the day, microprocessor manufacturers and code developers expect full core resource utilization. We agree that it is conceptually possible to adapt some of the previous countermeasures to detect our technique, but it is an open question how difficult, effective, and practical these countermeasures would be.

6.2. Recommendations

Our PORTSMASH technique relies on SMT and exploits transient microarchitecture execution port usage differences, therefore two immediate countermeasures arise: (1) remove SMT from the attack surface; and (2) promote execution port-independent code.

So far, the best and most recommended strategy against attacks relying on SMT—e.g. CacheBleed, MemJam, and TLBleed—is to simply disable this feature. Even OpenBSD developers [5] recently followed this approach, since it is the simplest solution that exists but it comes at the cost of performance loss on thread-intensive applications. In order to minimize this loss, Wang and Lee [3] proposed a selective approach by modifying the OS to support logical core isolation requests from user space, such that security-critical code can trigger it on demand. This selective SMT-disabling reduces performance loss but is costly to implement since it requires changes in the OS and the underlying libraries, hindering portability and large-scale adoption.

The second option, port-independent code, can be achieved through secret-independent execution flow secure coding practices, similar to constant-time execution. Constant-time implementations that execute the same set of instructions independently from the secret—i.e. all code and data addresses are assumed public—fulfill the port-independent code requirement we propose to mitigate this technique.

5. https://marc.info/?l=openbsd-cvs&m=152943660103446

7. Conclusion

We presented a new SCA technique exploiting timing information against a non-persistent shared HW resource, derived from port contention in shared CPU execution units on SMT architectures. Our PORTSMASH technique features interesting properties including high adaptability though various configurations, very fine spatial granularity, high portability and minimal prerequisites. We demonstrated it is a practical attack vector with a real-world end-to-end attack against a TLS server, successfully recovering an ECDSA P-384 secret key; we further demonstrate it is a viable side-channel to endanger the security of SGX enclaves and discussed potential mitigations.

Following responsible disclosure procedures, we reported our findings to the manufacturer and OS vendors, which resulted in the assignment of CVE-2018-5407 to track the vulnerability.

We leave as future work exploring the capabilities of PORTSMASH on other architectures featuring SMT, especially on AMD Ryzen systems.

Finally, we conclude with a remark on how this work, together with the increasingly fast-paced publications of scientific results in the same field, confirms once again SCA as a practical and powerful tool to find, exploit—and eventually mitigate—significant and often underestimated threats to the security of our data and communications.

Acknowledgments

We thank Tampere Center for Scientific Computing (TCSC) for generously granting us access to computing cluster resources.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 804476).

This article is based in part upon work from COST Action IC1403 CRYPTACUS, supported by COST (European Cooperation in Science and Technology).

We thank Nokia Foundation for funding a research visit of Alejandro Cabrera Aldaya to Tampere University of Technology during the development of this work.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[2] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks:

Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018. [Online]. Available: http://arxiv.org/abs/1801.01203

[3] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Proceedings of the 22nd Annual Conference on Computer Security Applications, ACSAC 2006, Miami Beach, FL, USA, December 11-15, 2006*. IEEE Computer Society, 2006, pp. 473–482. [Online]. Available: https://doi.org/10.1109/ACSAC.2006.20

[4] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 955–972. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[5] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A timing attack on OpenSSL constant time RSA," in *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, ser. Lecture Notes in Computer Science, B. Gierlichs and A. Y. Poschmann, Eds., vol. 9813. Springer, 2016, pp. 346–367. [Online]. Available: https://doi.org/10.1007/978-3-662-53140-2_17

[6] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman, "The microarchitecture of the Intel Pentium 4 processor on 90nm technology." *Intel Technology Journal*, vol. 8, no. 1, pp. 7 – 23, 2004.

[7] Intel, "Intel 64 and IA-32 architectures software developers manual," *https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf*, vol. Volume 1.

[8] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology architecture and microarchitecture." *Intel Technology Journal*, vol. 6, no. 1, 2002.

[9] A. Fog, "Instruction tables (2018)," *http://www.agner.org/optimize/instruction_tables.pdf*, vol. Version: 15-Sept-2018.

[10] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Retrospective: Simultaneous multithreading: Maximizing on-chip parallelism," in *25 Years of the International Symposia on Computer Architecture (Selected Papers).*, G. S. Sohi, Ed. ACM, 1998, pp. 115–116. [Online]. Available: http://doi.acm.org/10.1145/285930.285971

[11] C. Percival, "Cache missing for fun and profit," in *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*, 2005. [Online]. Available: http://www.daemonology.net/papers/cachemissing.pdf

[12] E. Brickell, G. Graunke, M. Neve, and J. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities," *IACR Cryptology ePrint Archive*, vol. 2006, no. 52, 2006. [Online]. Available: http://eprint.iacr.org/2006/052

[13] O. Acıiçmez and J. Seifert, "Cheap hardware parallelism implies cheap security," in *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2007, FDTC 2007: Vienna, Austria, 10 September 2007*, L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J. Seifert, Eds. IEEE Computer Society, 2007, pp. 80–91. [Online]. Available: https://doi.org/10.1109/FDTC.2007.4318988

[14] D. Evtyushkin and D. V. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 843–857. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978374

[15] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, ser. Lecture Notes in Computer Science, M. Polychronakis and M. Meier, Eds., vol. 10327. Springer, 2017, pp. 3–24. [Online]. Available: https://doi.org/10.1007/978-3-319-60876-1_1

[16] O. Acıiçmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, ser. Lecture Notes in Computer Science, S. Mangard and F. Standaert, Eds., vol. 6225. Springer, 2010, pp. 110–124. [Online]. Available: https://doi.org/10.1007/978-3-642-15031-9_8

[17] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang, "High-speed high-security signatures," *J. Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012. [Online]. Available: https://doi.org/10.1007/s13389-012-0027-1

[18] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, ser. Lecture Notes in Computer Science, D. Pointcheval, Ed., vol. 3860. Springer, 2006, pp. 1–20. [Online]. Available: https://doi.org/10.1007/11605805_1

[19] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. USENIX Association, 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[20] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., vol. 9721. Springer, 2016, pp. 279–299. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_14

[21] A. Moghimi, T. Eisenbarth, and B. Sunar, "MemJam: A false dependency attack against constant-time crypto implementations in SGX," in *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*, ser. Lecture Notes in Computer Science, N. P. Smart, Ed., vol. 10808. Springer, 2018, pp. 21–44. [Online]. Available: https://doi.org/10.1007/978-3-319-76953-0_2

[22] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 605–622. [Online]. Available: https://doi.org/10.1109/SP.2015.43

[23] G. I. Apecechea, T. Eisenbarth, and B. Sunar, "S$a: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 591–604. [Online]. Available: https://doi.org/10.1109/SP.2015.42

[24] M. Kayaalp, N. B. Abu-Ghazaleh, D. V. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*. ACM, 2016, pp. 72:1–72:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2897962

[25] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: the missing OS abstraction," *CoRR*, vol. abs/1810.05345, 2018. [Online]. Available: http://arxiv.org/abs/1810.05345

[26] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 549–564. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp

[27] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[28] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, ser. Lecture Notes in Computer Science, H. C. Williams, Ed., vol. 218. Springer, 1985, pp. 417–426. [Online]. Available: https://doi.org/10.1007/3-540-39799-X_31

[29] A. H. Koblitz, N. Koblitz, and A. Menezes, "Elliptic curve cryptography: The serpentine course of a paradigm shift," *Journal of Number Theory*, vol. 131, no. 5, pp. 781–814, 2011. [Online]. Available: https://doi.org/10.1016/j.jnt.2009.01.006

[30] F. 186-2, "Digital signature standard (DSS)," *National Institute of Standards and Technology (NIST)*, vol. 20, p. 13, 2000.

[31] N. S. Agency, "The Case for Elliptic Curve Cryptography," 2005-10-13. [Online]. Available: tinyurl.com/NSAandECC

[32] C. on National Security Systems, "Cnssp-15 national information assurance policy on the use of public standards for the secure sharing of information among national security systems," 2012-10-01. [Online]. Available: https://www.cnss.gov/CNSS/issuances/Policies.cfm

[33] N. S. Agency, "Commercial national security algorithm suite," 2015-08-19. [Online]. Available: https://apps.nsa.gov/iaarchive/programs/iad-initiatives/cnsa-suite.cfm

[34] ——, "Commercial national security algorithm suite and quantum computing faq," 2016-01-05. [Online]. Available: https://apps.nsa.gov/iaarchive/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm

[35] E. Käsper, "Fast elliptic curve cryptography in OpenSSL," in *Financial Cryptography and Data Security - FC 2011 Workshops, RLCPS and WECSR 2011, Rodney Bay, St. Lucia, February 28 - March 4, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, G. Danezis, S. Dietrich, and K. Sako, Eds., vol. 7126. Springer, 2011, pp. 27–39. [Online]. Available: https://doi.org/10.1007/978-3-642-29889-9_4

[36] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, ser. Lecture Notes in Computer Science, M. Matsui, Ed., vol. 5912. Springer, 2009, pp. 667–684. [Online]. Available: https://doi.org/10.1007/978-3-642-10366-7_39

[37] S. Gueron and V. Krasnov, "Fast prime field elliptic-curve cryptography with 256-bit primes," *J. Cryptographic Engineering*, vol. 5, no. 2, pp. 141–151, 2015. [Online]. Available: https://doi.org/10.1007/s13389-014-0090-x

[38] B. Möller, "Algorithms for multi-exponentiation," in *Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Revised Papers*, ser. Lecture Notes in Computer Science, S. Vaudenay and A. M. Youssef, Eds., vol. 2259. Springer, 2001, pp. 165–180. [Online]. Available: https://doi.org/10.1007/3-540-45537-X_13

[39] T. Allan, B. B. Brumley, K. E. Falkner, J. van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, S. Schwab, W. K. Robertson, and D. Balzarotti, Eds. ACM, 2016, pp. 422–435. [Online]. Available: http://doi.acm.org/10.1145/2991079.2991084

[40] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, ""ooh aah... just a little bit" : A small amount of side channel can go a long way," in *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, ser. Lecture Notes in Computer Science, L. Batina and M. Robshaw, Eds., vol. 8731. Springer, 2014, pp. 75–92. [Online]. Available: https://doi.org/10.1007/978-3-662-44709-3_5

[41] N. Tuveri, S. ul Hassan, C. Pereida García, and B. B. Brumley, "Side-channel analysis of SM2: A late-stage featurization case study," in *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3-7, 2018, San Juan, PR, USA*. New York, NY, USA: ACM, 2018, p. 14 pages. [Online]. Available: https://doi.org/10.1145/3274694.3274725

[42] R. Vautard, P. Yiou, and M. Ghil, "Singular-spectrum analysis: A toolkit for short, noisy chaotic signals," *Physica D: Nonlinear Phenomena*, vol. 58, no. 1, pp. 95 – 126, 1992. [Online]. Available: https://doi.org/10.1016/0167-2789(92)90103-T

[43] S. M. D. Pozo and F. Standaert, "Blind source separation from single measurements using singular spectrum analysis," in *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, ser. Lecture Notes in Computer Science, T. Güneysu and H. Handschuh, Eds., vol. 9293. Springer, 2015, pp. 42–59. [Online]. Available: https://doi.org/10.1007/978-3-662-48324-4_3

[44] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "ECDSA key extraction from mobile devices via nonintrusive physical side channels," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 1626–1638. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978353

[45] I. Markovsky, "Structured low-rank approximation and its applications," *Automatica*, vol. 44, no. 4, pp. 891–909, 2008. [Online]. Available: https://doi.org/10.1016/j.automatica.2007.09.011

[46] I. Daubechies, "The wavelet transform, time-frequency localization and signal analysis," *IEEE Trans. Information Theory*, vol. 36, no. 5, pp. 961–1005, 1990. [Online]. Available: https://doi.org/10.1109/18.57199

[47] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, no. 86, 2016. [Online]. Available: http://eprint.iacr.org/2016/086

[48] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 640–656. [Online]. Available: https://doi.org/10.1109/SP.2015.45

[49] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, ser. Lecture Notes in Computer Science, W. Fischer and N. Homma, Eds., vol. 10529. Springer, 2017, pp. 69–90. [Online]. Available: https://doi.org/10.1007/978-3-319-66787-4_4

[50] F. Dall, G. D. Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "CacheQuote: Efficiently recovering long-term secrets of SGX EPID via cache attacks," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 2, pp. 171–191, 2018. [Online]. Available: https://doi.org/10.13154/tches.v2018.i2.171-191

[51] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018. [Online]. Available: https://doi.org/10.1007/s13389-016-0141-6

[52] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proceedings of the first ACM Cloud Computing Security Workshop, CCSW 2009, Chicago, IL, USA, November 13, 2009*, R. Sion and D. Song, Eds. ACM, 2009, pp. 77–84. [Online]. Available: http://doi.acm.org/10.1145/1655008.1655019

[53] Y. Zhang and M. K. Reiter, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 827–838. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516741

[54] M. M. Godfrey and M. Zulkernine, "Preventing cache-based side-channel attacks in a cloud environment," *IEEE Trans. Cloud Computing*, vol. 2, no. 4, pp. 395–408, 2014. [Online]. Available: https://doi.org/10.1109/TCC.2014.2358236

[55] G. Irazoqui, T. Eisenbarth, and B. Sunar, "MASCAT: Preventing microarchitectural attacks before distribution," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, Z. Zhao, G. Ahn, R. Krishnan, and G. Ghinita, Eds. ACM, 2018, pp. 377–388. [Online]. Available: http://doi.acm.org/10.1145/3176258.3176316

[56] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas, "SpyDetector: An approach for detecting side-channel attacks at runtime," *International Journal of Information Security*, Jun 2018. [Online]. Available: https://doi.org/10.1007/s10207-018-0411-7

[57] T. Zhang, Y. Zhang, and R. B. Lee, "CloudRadar: A real-time side-channel attack detection system in clouds," in *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*, ser. Lecture Notes in Computer Science, F. Monrose, M. Dacier, G. Blanc,

and J. García-Alfaro, Eds., vol. 9854. Springer, 2016, pp. 118–140. [Online]. Available: https://doi.org/10.1007/978-3-319-45719-2_6

[58] A. Raj and J. Dharanipragada, "Keep the PokerFace on! Thwarting cache side channel attacks by memory bus monitoring and cache obfuscation," *J. Cloud Computing*, vol. 6, p. 28, 2017. [Online]. Available: https://doi.org/10.1186/s13677-017-0101-4

[59] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," *CoRR*, vol. abs/1808.05575, 2018. [Online]. Available: http://arxiv.org/abs/1808.05575

Appendix A.
Lattice Statistics

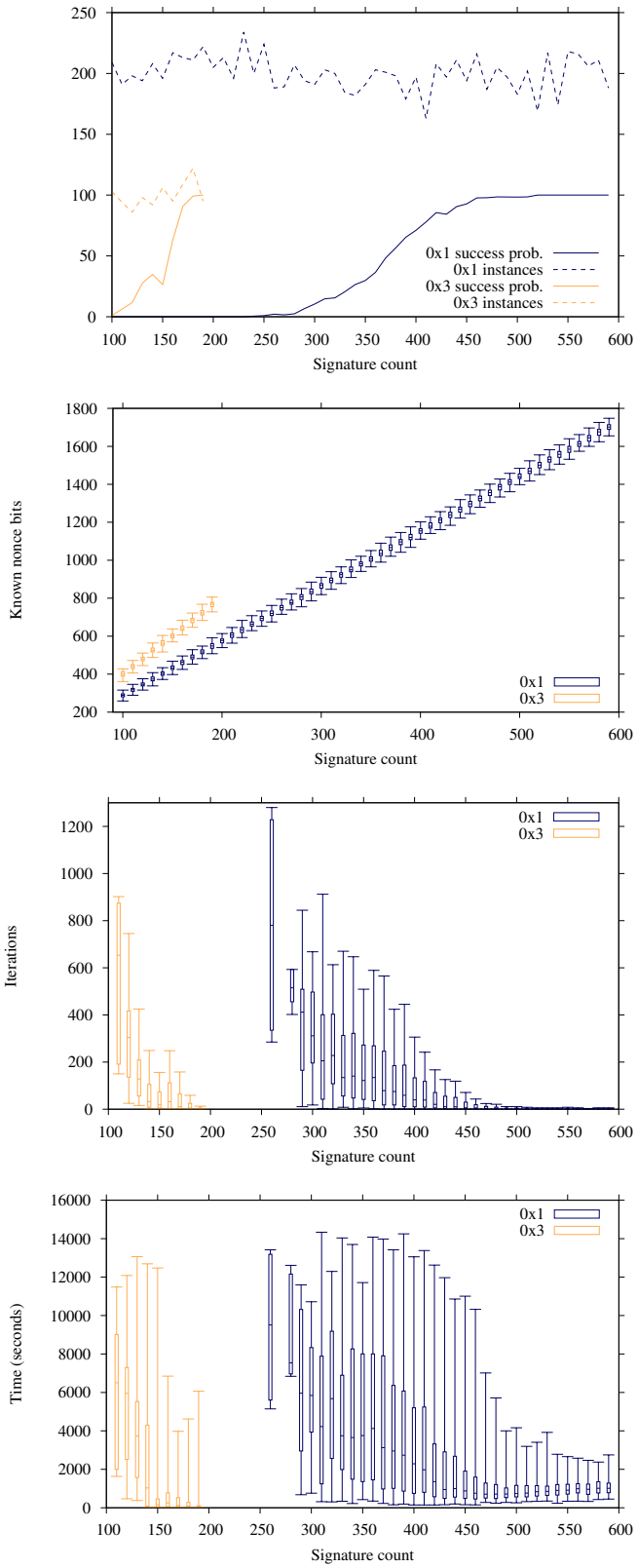

Figure 9. Empirical data for lattice experiments.