Amplifying Side Channels Through Performance Degradation

Thomas Allan*, Billy Bob Brumley†, Katrina Falkner*, Joop van de Pol‡, Yuval Yarom§
* The University of Adelaide
Email: tom.allan@student.adelaide.edu.au, katrina.falkner@adelaide.edu.au
† Tampere University of Technology
Email: billy.brumley@tut.fi
‡ University of Bristol
Email: joop.vandepol@bristol.ac.uk
§ The University of Adelaide and NICTA
Email: yval@cs.adelaide.edu.au

***Abstract*—Interference between processes executing on shared hardware can be used to mount performance-degradation attacks. However, in most cases, such attacks offer little benefit for the adversary. In this paper, we show that performance-degradation attacks can be used to amplify side-channel leaks, enabling the adversary to increase both the amount and the quality of information captured.**

We describe a new microarchitectural performance-degradation attack that can slow victims down by a factor of over 150. We identify a new information leak in the OpenSSL implementation of the ECDSA digital signature algorithm. We show how to use the performance-degradation attack to amplify a side-channel enough to enable exploiting the new information leak. Using the combined attack, an adversary can break a private key of the `secp256k1` curve, used in the Bitcoin protocol, after observing only 6 signatures. This result is over four times better than any previously described attack.

I. Introduction

Executing multiple clients' workloads on a single hardware platform can help achieve high resource utilisation. A consequence of this resource sharing is that workloads of different clients can interfere with each other due to shared-resource contention [51, 52].

Malicious clients can exploit this interference to mount performance-degradation attacks against co-resident clients [13, 17, 34, 41]. Fortunately, such attacks tend to suffer from two main limitations on their usability. First, the attacks target resources that are broadly used by a large variety of workloads. Thus, the attacks do not target a specific client, affecting instead all of the co-resident workloads. The broad-spectrum nature of the attacks exposes them and may facilitate detection and identification of the adversary.

The second limitation of performance-degradation attacks is that, in most cases, the attacker does not gain any direct benefit from the attack. The main benefit an attacker gets from mounting a performance-degradation attack is harming the victim. An exception is the attack of Varadarajan et al. [47], in which the attack frees resources for the attacker's use by forcing the victim to wait on other resources.

In this paper, we present a new microarchitectural performance-degradation attack that overcomes these limitations. Rather than generating contention on a shared microarchitectural component, our attack targets specific code paths within the victim. Consequently, workloads that do not take the attacked code path are not affected by the attack.

The attack exploits a property of the Intel x86 and x86-64 architectures, which allows processes to manipulate the caching status of read-only memory. When the victim and the attacker have shared access to the victim's code — e.g. in shared library scenarios — the attacker can repeatedly evict the shared code from the cache, forcing the victim to wait while the processor retrieves the shared code from memory.

We focus on two different choices of code paths to attack. The first is attacking the main loop of a program. We demonstrate that using this technique an attacker can slow programs down by a factor of over 150, with a mean slow-down factor of 18 over the integer SPEC 2006 [18] benchmarks and 15 over the floating-point benchmarks.

A second choice of code path is inner loops in cryptographic primitives. We show that this kind of attack can be used to amplify the side-channel signal of the affected primitive, increasing the vulnerability of the affected primitive to side-channel attacks.

As an example, we analyse the FLUSH+RELOAD attack [15, 54]. We demonstrate that the attack has a maximum resolution which depends on the number of memory locations it attempts to probe. We further show a relationship between the resolution of events in the victim and the likelihood of the attacker missing an event.

To demonstrate the channel amplification, we improve the attack of van de Pol et al. [46] on the OpenSSL implementation of ECDSA with the `secp256k1` curve. In a nutshell, van de Pol et al. [46] trace the use of point addition and point doubling used throughout the scalar multiplication of ECDSA signature generation. From this sequence, it infers information on the ephemeral key used for the signature. The long term private key is then recovered from the information collected from multiple signatures by using a lattice attack.

We observe that tracing point inversions can increase the amount of information collected on each nonce, potentially reducing the number of signatures required for breaking the

key. However, tracing point inversion introduces two problems: (1) due to the high resolution required, there is a high probability of missing point inversions; (2) adding the trace for point inversion increases the number of memory locations we trace, limiting the applicability of the attack.

To overcome these limitations, we apply our performance-degradation attack against the scalar multiplication code. Our attack slows elliptic group operations by a factor of over 40, and the scalar multiplication by a factor of 32, allowing a virtually error-free trace of the operations. By using this technique, we can break the private key of the `secp256k1` curve used in Bitcoin after observing as few as 6 signatures.

The contributions of this paper are:

- We develop a new microarchitectural performance-degradation attack and demonstrate that it is about 8 times more potent than previously disclosed attacks. (Section IV)
- We analyse the probability of a FLUSH+RELOAD attack missing a monitored event based on the rate of the events. (Section V)
- We identify point inversions as a new source of leaked information in the implementation of ECDSA over prime fields in OpenSSL and show how to exploit this information. (Section VI)
- We use our performance-degradation attack to amplify the side channel and capture error-free traces of the scalar multiplication. (Section VII)

II. BACKGROUND

A. The Memory Hierarchy

The cache is part of the memory hierarchy that exploits the spatial and temporal locality of memory access to bridge the performance gap between the fast processor and the slower memory. Modern processors feature a hierarchy of caches, with higher-level caches, which are closer to the processor core, being smaller but faster than lower-level caches, which are closer to the main memory. In recent Intel architecture, there are, typically, three levels of cache. Each core has two levels of caches, called the L1 and L2 caches. The cores share access to a larger Last-Level Cache (LLC).

To exploit spatial locality, caches are organised in fixed-size *lines*, which are the units of allocation and transfer of data in the memory hierarchy. When the processor needs to access a memory address, it first checks if the line containing the address is cached in the top-level L1 cache. In a *cache hit*, the data is served from a copy of the data in the cache. Otherwise, in a *cache miss*, the processor repeats the search for the line in the next lower level in the memory hierarchy. When the line is found, the processor stores its contents in the cache, reducing the time required for accessing it in the near future. See [39, Ch. 8] for a good overview of caching in computer architecture.

Modern caches are typically *set associative*. A set associative cache is divided into multiple *sets*, each consisting of multiple *ways*. Each memory line is mapped to a single cache set. The memory line can only be cached in the set it is mapped to, but can be cached in any of the ways of the set. Typically, the set a memory line maps to is determined by a sequence of bits in the physical address of the memory line. However, the LLC in modern Intel processor uses a more complex hash function to determine the mapping [20, 30, 55].

Several cache optimisations result in memory lines being brought to the cache without the code accessing data in these lines. In the Intel architecture, the *spatial prefetcher* pairs consecutive memory lines and attempts to fetch the pair of a missed line [21]. Another optimisation is to detect sequences of accesses to consecutive memory addresses and prefetch memory lines that the processor anticipates may be required [21]. A third optimisation is *speculative execution*, where the processor attempts to follow both paths of a conditional branch before the branch condition is evaluated [45], bringing the code of both paths into the cache.

When multiple programs share the same cache, one program's use of the cache may evict another program's data from the cache, which due to the timing difference between cache hits and cache misses may create noticeable timing variations in the sharing programs. These timing variations have been used to mount side-channel attacks [1, 5, 28, 38, 40, 42, 56].

B. The FLUSH+RELOAD Attack

FLUSH+RELOAD [54] is a cache-based side-channel attack technique. Unlike other techniques, which infer the memory lines the victim accesses based on activity in cache sets, FLUSH+RELOAD positively identifies access to memory lines, giving it high accuracy, high signal to noise ratio and high resolution. The attack has been used in various settings, including between non-trusting processes, between isolated containers and across virtual machines and has been shown to be effective against multiple algorithms [3, 14, 22, 23, 46, 53, 57].

FLUSH+RELOAD relies on memory sharing between the victim and the adversary. Such sharing could be achieved via the use of shared libraries or using page de-duplication [2, 48]. To identify victim access to a shared memory line, the adversary flushes or evicts the memory line from the cache, waits a bit and then measures the time it takes to reload the memory line. If the victim accesses the line during the wait, the line will be cached and the reload will retrieve it from the cache. Otherwise, the line will not be cached and reloading will have to retrieve it from the main memory. As retrieving the line from the memory takes longer than accessing a cached copy, the adversary can distinguish between the two options and identify whether the victim has accessed the line during the wait.

The FLUSH+RELOAD attack needs processor support for evicting memory lines from the cache. So far, all published reports of the attack use the `clflush` instructions of the x86 and x86-64 instruction sets. In those instruction sets, `clflush` is an unpriviliged instruction, which every process can use.

Gruss et al. [14] suggest a variant of FLUSH+RELOAD, called EVICT+RELOAD, which does not require a specific instruction for evicting the memory line. Instead, they evict the victim memory line by accessing a number of memory lines that map to the same cache set as the victim line. Evicting the victim memory line using this technique takes significantly longer than using the `clflush` instruction. (325 cycles compared with 41 for `clflush`.) Furthermore, the eviction may fail, resulting in a false positive.

Both FLUSH+RELOAD and EVICT+RELOAD need to evict the victim cache line from all of the caches that the victim uses. When the victim and the adversary do not execute on the same core, they do not share the L1 and L2 caches. In this case, the attack relies on the *inclusion* property of the LLC. The contents of an inclusive cache is a superset of the contents of all higher level caches. To maintain the inclusion property, when a memory line is evicted from the LLC, the processor also evicts it from all of the L1 and L2 caches above it. All of the published attacks run on Intel processors, which use inclusive LLCs. Yarom and Falkner [54] report that the FLUSH+RELOAD attack does not work on AMD processors due to their non-inclusive LLCs.

C. Related Work

Several works have investigated performance-degradation attacks by co-located adversaries. Grunwald and Ghiasi [13] implement two attacks against Intel HyperThreading (HT), a Simultaneous Multithreading (SMT) technique. The first attack uses denormalised floating point numbers [12], which flush the instruction pipeline of the Pentium 4 processor used. The second attack uses self-modifying code, which results in flushing both the pipeline and the processor's trace cache. To test the attack, they use a compute-bound victim which repeatedly calculates the MD5 hash. The victim is slowed by about 120% with the first attack and by a factor of 20 with the second.

Heat stroke [17] is a performance-degradation attack that exploits the thermal management of the processor chip. Certain components of the chip tend to overheat when experiencing high utilisation, forcing the processor to reduce the utilisation of the hot components until they cool down. The authors use a simulated multi-threaded processor to test the attack. The adversary generates many register accesses causing overheating in the shared register file. The processor responds to overheating by slowing access to the register file. The attack achieves a mean slow down by a factor of 8 over the SPEC 2000 benchmark suite.

Matthews et al. [29] compare the performance isolation properties of virtualisation. They implement multiple adversaries, each attempting to monopolise a system resource. The main finding is that OS-level virtualisation (e.g. Solaris containers) provides less isolation than system-level hypervisors such as VMware or Xen. In particular, it performs poorly under memory or process number pressure. Other than that, all systems at most experience minor interference.

Moscibroda and Mutlu [34] note that the scheduling policy of memory banks favours requests for the currently open DRAM row. Consequently, an adversary that issues many requests to the same row can cause memory-access delays for programs that access the same DRAM bank. These delays can slow the victim down by a factor of 2.9 for one adversary and up to a factor of 4 for multiple adversaries. The suggested fix is to change the DRAM scheduling algorithm.

Woo and Lee [50] investigate attacks against a shared LLC. The attacks aim to evict entries from the LLC and rely on the LLC inclusiveness to also evict data from the victim L1. Two forms of attack are suggested and are tested using a simulator—no tests on an real processor are performed. Attacks using load instructions slow victims down by 50% on average, with a maximum slowdown of 100%. (The amount of degradation is estimated from the graphs provided due to the absence of exact figures.) The second form of attack uses atomic instructions which lock access to the bus. The mean slowdown with this attack is by a factor of 5, with a maximum slowdown factor of 10.

Another LLC monopolising attack is suggested by Weng et al. [49] which demonstrate a significant performance drop in co-resident VMs. The paper does not present exact figures, but judging from the supplied graphs, the performance seems to drop by about 30%. As a countermeasure, Weng et al. [49] suggest not scheduling non-trusting VMs concurrently on the same processor package.

Cardenas and Boppana [7] also use an adversary that tries to monopolise the LLC. The attack reduces the performance of the victim by 50% with a single attacking thread and up to 75% with multiple threads. Based on the observation that the adversary also suffers LLC misses, the paper suggests using the performance management unit (PMU) to identify the adversary and eventually mitigate the attack. We note that because our adversary does not suffer cache misses, this mitigation does not apply to our attack.

Richter et al. [41] investigate multiple techniques for degrading the performance of a shared PCI bus. They show that when I/O virtualisation is used, a malicious VM cause cause a drop of 27% in TCP throughput. With multiple attackers the drop reaches 35%.

Swiper [8] generates adversarial I/O workload to slow a target application down, achieving a reduction of up to 31% in the throughput of Web and media servers.

In all the systems described above, the only motivation for adversarial behaviour is the damage it causes to the victim. Performance degradation attacks are, therefore, a form of vandalism, whose only benefit is harming the victim. Varadarajan et al. [47] is the only prior work to offer direct benefits to the adversary. The resource freeing attack suggested uses a performance-degradation attack to slow a victim down. The adversary can then benefit from the victim slow down by using resources that the victim would otherwise use. The paper demonstrates how increasing the load on the victim gives the adversary a 60% performance boost.

III. THREAT MODEL

In the attack scenario, the adversary executes code concurrently with victim code on the same hardware. This scenario is common in multi-user operating systems and in virtualised environments. The operating system or the hypervisor prevent the adversary from accessing the victim's data.

We assume that the system supports a form of read-only sharing between the adversary and the victim. This sharing could be based on file mapping, e.g. shared libraries, or it can be based on coalescing identical contents through memory de-duplication. Memory de-duplication is known to be vulnerable to side-channel attacks [44], and is one of the requirements for the FLUSH+RELOAD attack [54]. We show that it also enables performance-degradation attacks. Like the FLUSH+RELOAD

attack, we also assume a shared inclusive LLC and require an efficient method of evicting memory lines from the cache.

IV. A Performance Degradation Attack

The performance-degradation attack we describe is based on the observation that programs tend to spend a significant part of their execution within a small "hot" section of the program code. Under normal execution, the frequently executed code is in the processor cache, hence access to it is fast.

If the memory that contains the hot code is shared between the adversary and the victim, the adversary can evict memory lines that contain that code from the last-level cache. This forces the victim to wait until the processor loads the code from the memory, introducing delays to the victim's process. Repeatedly evicting the hot code would negate the performance benefits of the cache, slowing the victim down.

The amount of slowdown depends primarily on the difference between the latencies of the cache and the memory. We measure the time it takes to load data from the L1 cache and from memory on an HP Elite 8300 running CentOS 6.5. (Intel i5-3470 processor, running at 3.2 GHz, with 8 GiB of DDR3-1600 CL-11 memory.) Figure 1 shows the distribution over 100,000 measurements.

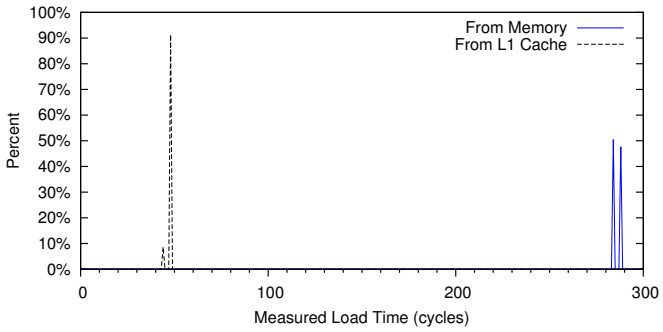

Fig. 1. Distribution of L1 cache and memory access times.

As we can see, virtually all loads from the L1 cache take 48 cycles. Over 98% of the loads from the memory take between 280 and 290 cycles, with the rest spread over the interval 250–1200 cycles.

In addition to data access latency, the measurements include the overhead of the measurement code. Due to optimisations, such as instruction pipelining and reordering, parallel use of multiple functional units and data prefetching, we cannot measure this overhead. Given that the L1 cache latency is 4 cycles [21], we can conclude that the memory latency is around 240 cycles.

To measure the effects of the attack, we test it with the SPEC CPU 2006 [18] benchmark suite. To generate a baseline performance measurement, we pin the SPEC benchmarks to one core and run them on an otherwise idle machine. The measurements follow the SPEC reporting guidelines. In particular, we use the SPEC `ref` workload, and for each benchmark we use the median time of three runs.

We then measure the performance of the benchmark under the attack. We measure under two scenarios—with a single attacking thread and with three attacking threads running in parallel. To avoid affecting the SPEC benchmark through time sharing, we pin the SPEC benchmarks and the attacking threads, each to a separate core. As in the baseline case, the machine is otherwise idle.

To apply the attack, we need to identify the hot sections of each of the SPEC benchmarks. One possible way of doing that is to read and understand the code of each benchmark and use that understanding to identify frequently used code sections. However, due to the size of the code base, such an approach would require significant effort and is prone to errors due to limited understanding of the code [43].

Instead, we use automatic tools for analysing the SPEC benchmarks. We build the SPEC benchmarks with instrumentation for collecting code-coverage information. We then use the program `gcov` to find out which source lines are the most frequently executed. Our attack targets this code.

Because the instrumentation skews the performance of the program, we do not use the instrumented binary for the performance testing. Instead, we build optimised SPEC benchmarks with debugging symbols and use the these debugging symbols to find the memory addresses corresponding to the lines identified through code coverage. The result of this process is a list of candidate memory lines for the attack. We note that debugging symbols are not loaded into memory when the program executes and do not affect its performance.

Usually, to achieve an efficient attack, we cannot use all the candidate memory lines. The reason being that evicting a line from the cache takes time. If we try to evict too many memory lines, we reduce the frequency of evicting each of the lines. Hence lines stay longer in the cache, allowing the victim to benefit from faster access to them. With cache eviction taking around 70 cycles and memory access around 240, we should be able to evict 3 lines from memory before the first is reloaded. Hence, in our settings, evicting more than three cache lines in a single attacking thread reduces the efficiency of the attack.

To implement an efficient attack we, therefore, need to select a small number of the candidate memory lines identified above. A naïve approach is to pick memory lines corresponding to the most frequently accessed source lines. Such an approach, however, does not guarantee the most efficient attack. There are several scenarios in which attacking less frequently used memory lines may result in a more efficient attack.

One such scenario occurs when the most commonly used source line is replicated in the binary. Replication can occur through compiler optimisations, such as loop unrolling or inlining, or when the line is in a C++ template that the code instantiates multiple times. When the line is replicated, `gcov` reports the cumulative number of uses of the line, across all of the replicas. However, none of the replicas is used as often as reported and attacking any replica would not achieve the most effective attack.

Another possible scenario occurs when a memory line contains function calls. Each time the source line is executed, the memory line is accessed twice—once before the call and once on return. Hence, targeting the memory line would be twice as effective as the `gcov` output indicates.

TABLE I. SPEC CPU 2006 RUNNING TIMES (SECONDS)

	Baseline	One attacker	Three attackers
perlbench	396	3,052	20,922
bzip2	443	1,651	9,538
gcc	312	660	1,369
mcf	286	1,145	2,928
gobmk	446	970	2,180
hmmer	432	514	62,507
sjeng	513	2,048	4,288
libquantum	587	5,492	25,395
h264ref	523	7,381	15,482
omnetpp	290	723	2,935
astar	375	3,364	9,792
xalancbmk	219	602	1,990
SpecINT Mean	387	1,574	6,841
bwaves	756	8,004	46,993
gamess	689	12,493	16,367
milc	405	1,846	10,737
zeusmp	387	426	823
gromacs	375	1,050	5,390
cactusADM	660	817	6,408
leslie3d	628	1,426	13,695
namd	397	414	405
dealII	317	2,761	6,723
soplex	320	2,403	3,829
povray	163	1,439	6,759
calculix	780	18,558	121,759
GemsFDTD	674	1,740	4,859
tonto	465	739	2,956
lbm	370	1,506	8,868
wrf	586	752	2,473
sphinx3	591	11,640	20,225
SpecFP Mean	469	1,989	6,885

As we can see, in both cases, the reason for the behaviour is the gap between the high-level source code and the low-level machine code that implements it. Tools that can trace the sequence of memory accesses by a program, such as Cage [27], can possibly provide a better indication of which memory lines to attack. However, due to processor optimisations such as out-of-order execution, prediction based on accurate memory access traces may still be inaccurate.

Instead of attempting to accurately predict the best memory lines to use for the attack, we test the efficiency of the attack with several different selections of candidate memory lines. The results we report are for the selection that produced the most effective attack. We acknowledge that other selections may produce more effective attacks. Hence the results below may understate the strength of the attack.

Table I summarises the results of the tests. It shows the running times of each of the SPEC benchmarks, as well as the geometric mean for the integer (SPEC INT) and the floating point (SPEC FP) benchmarks. These results are visualised in Figure 2 and Figure 3.

As the results show, a single attacking thread reduces the mean execution speed to about a quarter of the normal speed, whereas 3 threads have a mean slowdown by a factor of 15–18. However, there is a large variance in the effectiveness of the attack. The effective slowdown with one attacker ranges from 4% (`namd`) to 2,279% (`calculix`). For three attackers the range is even bigger. `namd` is hardly affected whereas `calculix` is over a 150 times slower under the attack.

The attack is less effective on `namd` and `zeusmp` because both benchmarks do not have a tight internal loop. Instead, the internal loops in these benchmarks span a relatively large amount of code. For example, the main loop in `namd` contains

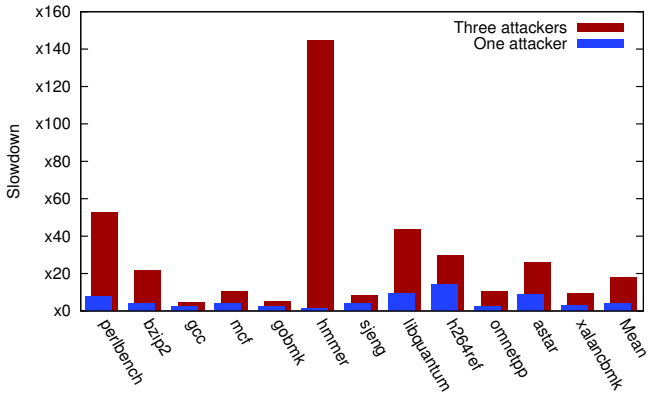

Fig. 2. SPEC CPU2006 Integer Results.

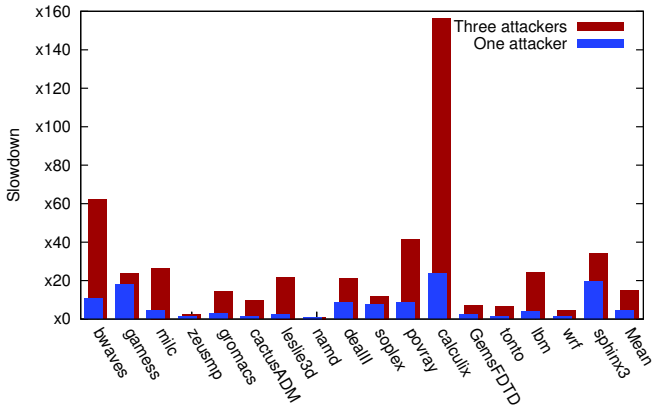

Fig. 3. SPEC CPU2006 Floating Point Results.

256 lines of C++ code, which span over 93 memory lines. The attack only evicts a small fraction of this code, so the overall performance hit is very small.

Considering that memory accesses are 60 times slower than cache accesses, the results for `hmmer` and `calculix` are surprising. The observed slowdowns by factors of 145 and 156, respectively, are much larger than would be expected from the cache vs. memory speed difference. We speculate that the reason for this slowdown is the interaction of instruction fetching with the cache. Under normal circumstances the processor fetches instructions in batches of up to five instructions. While each of these fetches takes four cycles, they execute in parallel, achieving a rate of one batch per cycle [10]. If the attack is very efficient, the targeted cache line could be evicted after fetching only one batch, potentially reducing the performance by a factor of 240.

Unlike previous microarchitectural performance-degradation attacks, which affect all of the programs that use a microarchitectural component, our attack is very specific. It only targets programs that use specific code segments. The rest of this paper describes how we exploit this property of the attack.

V. Limitations of the Flush+Reload Attack

The main claim of this paper is that slowing down victims can allow the adversary to improve side-channel attacks. To better understand why this is true, we first study the Flush+Reload attack to see what limits its accuracy and resolution. Our focus is on asynchronous attacks, i.e. on attacks in which the adversary executes concurrently with the victim.

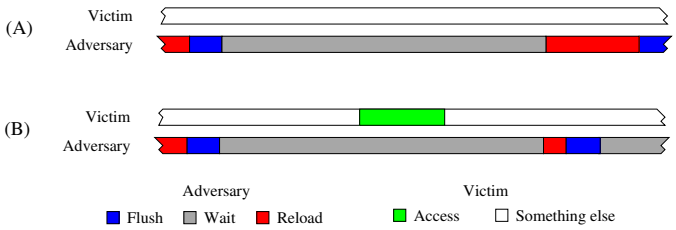

Fig. 4. Timing of Flush+Reload. (A) No Victim Access (B) With Victim Access

Typically, the adversary divides time into fixed length *slots*. At the start of a time slot, the monitored memory line is flushed from the cache hierarchy. The adversary, then, waits to allows the victim time to access the memory line. At the end of the slot, the adversary reloads the memory line, measuring the time to load it. If the victim accesses the memory line during the wait, the line will be available in the cache and the reload operation will take a short time. If, on the other hand, the victim has not accessed the memory line, the line will need to be brought from memory and the reload will take significantly longer. Figure 4 (A) and (B) show the timing of the attack phases without and with victim access.

The length of the time slot determines the granularity of the attack. The adversary cannot distinguish between multiple victim accesses to the probed memory line if they all occur within the same time slot. Consequently, a shorter time slot allows for a higher attack resolution. However, because the flush and reload operations are not instantaneous, they pose a lower bound on the length of the slot. This lower bound may be more significant when the adversary needs to monitor multiple lines, in which case the slot cannot be shorter than the time required for flushing and reloading all of the probed memory lines.

Another factor that limits the slot size is the probability of missing a victim access due to overlap. In an asynchronous attack, the victim operates independently of the adversary. As such, victim access to a memory location can occur at the same time the adversary reloads the location to test if it is cached, depicted in Figure 5 (A). In such a case, the victim access will not trigger a cache fill. Instead, the victim will use the cached data from the reload phase. Consequently, the adversary will miss the access.

A similar scenario occurs when the reload operation partially overlaps the victim access. In this case, depicted in Figure 5 (B), the reload phase starts while the victim is waiting for the data. The reload benefits from the victim access and terminates faster than if the data has to be loaded from memory. However, the timing may still be longer than a load from the cache. Whether the adversary recognises a partial overlap as a read from the cache or from memory depends on

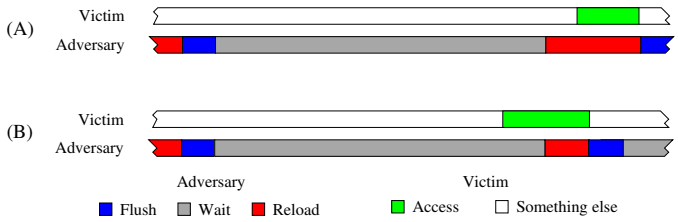

Fig. 5. Overlap in Flush+Reload. (A) Total overlap (B) Partial overlap

the time difference between the start of the victim access and the start of the adversary reload.

As we can see, there is a short *overlap period* that starts a bit before the adversary probe and ends when the adversary evicts the monitored line from the cache. Victim accesses to the monitored cache line during the overlap period are missed by the adversary. Because the victim access time is independent of the adversary probe, we can expect that the probability of a miss would be the ratio between the length of the overlap period and the interval between adversary probes.

To validate this expectation we measure the miss rate with different slot sizes. We run an adversary program that monitors a memory line at a fixed rate. In parallel, we run a victim program that accesses the monitored memory line 10,000 times, and count how many of these 10,000 accesses our adversary misses. Table II summarises the results.

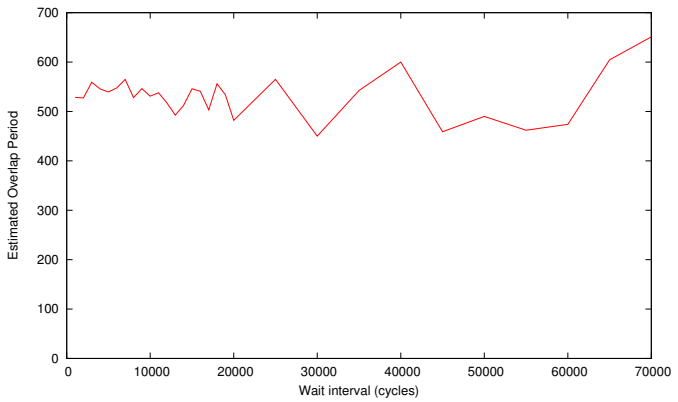

Fig. 6. Estimated length of the overlap period

We can now multiply the miss rate by the length of the slot to estimate the length of the overlap period. Figure 6 shows the estimated overlap period for each slot length. We can see that with a few outliers, due to noise, the estimated period is fairly stable. The average estimated period is 530 cycles. Figure 7 shows the overlap probability for each slot length along with the calculated value $(530/\text{slot})$.

A further aspect that affects the attack accuracy is operating system activity. The operating system may suspend the adversary execution to handle some system activity, such as a network or a timer interrupt. If the interruption is short enough to be wholly contained within a time slot, it will not affect the attack. If, however, the adversary is interrupted for a longer period, the adversary loses the ability to distinguish between and to order multiple events occuring during the interruption.

TABLE II. NUMBER OF MISSED ACCESSES FOR SLOT LENGTH (CYCLES)

Slot	Missed	Slot	Missed	Slot	Missed	Slot	Missed	Slot	Missed
1,000	5,286	7,000	807	13,000	379	19,000	281	45,000	102
2,000	2,637	8,000	660	14,000	376	20,000	241	50,000	98
3,000	1,864	9,000	607	15,000	364	25,000	226	55,000	84
4,000	1,364	10,000	531	16,000	338	30,000	150	60,000	79
5,000	1,079	11,000	589	17,000	296	35,000	155	65,000	93
6,000	913	12,000	431	18,000	209	40,000	150	70,000	93

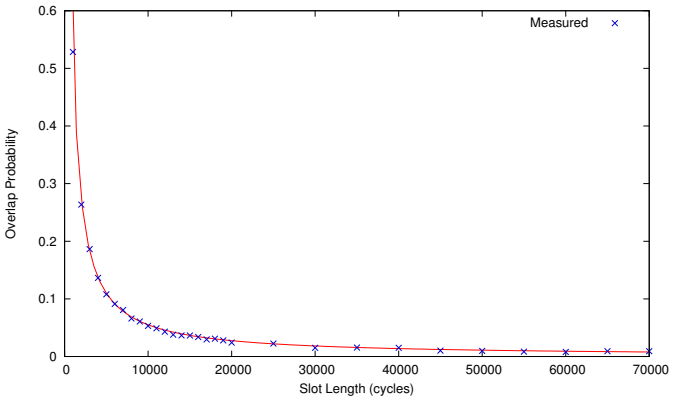

Fig. 7. Slot length and overlap probability

In our experience, shorter interruptions of about 5,000 cycles are quite common, occuring, on average, about 1,000 times per second. Longer interruptions of about 30,000 cycles or 9 μs occur at a rate of 50 per second. Significantly longer interruptions are possible when the operating system suspends the adversary in order to time-share the processor.

In summary, to achieve a high attack resolution, the adversary needs to use a short time slot. However, the length of the probe and the number of required probes present a lower limit on the slot length and, consequently, an upper limit on the attack resolution. Furthermore, the higher the attack resolution is the higher the probability of an error due to missing a victim access or being interrupted by the operating system is.

Several methods to overcome the large miss probability with short time slots have been suggested. Yarom and Falkner [54] monitor lines that are accessed in a loop. Their attack cannot distinguish between multiple consecutive accesses to the same line, but it can distinguish between periods of access and periods of no access to the line. Benger et al. [3] and van de Pol et al. [46] monitor memory lines that contain a call instruction. Such lines are accessed twice, once before the call and once upon return. Depending on the execution time of the called function, this approach can ensure that at most one of the two accesses is missed.

While these techniques can reduce or eliminate the probability of missing a victim access, they are not always applicable. In such scenarios, slowing the victim down can increase the interval between victim accesses and allow reducing the miss probability by using longer time slots. We describe such a scenario in the following sections.

VI. ATTACKING OPENSSL

A. ECDSA

The ElGamal Signature Scheme [9] is the basis of the US 1994 NIST standard, Digital Signature Algorithm (DSA). The ECDSA is the adaptation of one step of the algorithm from the multiplicative group of a finite field to the group of points on an elliptic curve. The main benefit of using this group as opposed to the multiplicative group of a finite field is that smaller parameters can be used to achieve the same security level [25, 31] due to the fact that the current best known algorithms to solve the discrete logarithm problem in the finite field are sub-exponential and those used to solve the ECDLP are exponential — see Galbraith and Gaudry [11] and Koblitz and Menezes [26, Sec. 2-3] for an overview of recent ECDLP developments.

Parameters: An elliptic curve E defined over a finite field $\mathbb{F}_q$; a point $G \in E$ of a large prime order n (generator of the group of points of order n). Parameters chosen as such are generally believed to offer a security level of $\sqrt{n}$ given current knowledge and technologies. Parameters are recommended to be generated following the Digital Signature Standard [37]. The field size q is usually taken to be a large odd prime or a power of 2. The implementation of OpenSSL uses both prime fields and $q = 2^m$; the results in this paper relate to the former case.

Public-Private Key pairs: The private key is an integer d, $1 < d < n - 1$ and the public key is the point $Q = dG$. Calculating the private key from the public key requires solving the ECDLP, which is known to be hard in practice for correctly chosen parameters.

Signing: Suppose Bob, with private-public key pair $\{d_B, Q_B\}$, wishes to send a signed message m to Alice. He follows the following steps:

1) Using an approved hash algorithm, compute $e = Hash(m)$, take $\bar{e}$ to be the leftmost ℓ bits of e (where $\ell = \min(\log_2(q)$, bitlength of the hash)).
2) Randomly select $k \leftarrow_R \mathbb{Z}_n$.
3) Compute the point $(x, y) = kG \in E$.
4) Take $r = x \bmod n$; if $r = 0$ then return to Step 2.
5) Compute $s = k^{-1}(\bar{e} + rd_B) \bmod n$; if $s = 0$ then return to Step 2.
6) Bob sends (m, r, s) to Alice.

Verifying: The message m is not necessarily encrypted, the contents may not be secret, but a valid signature gives Alice strong evidence that the message was indeed sent by Bob. She verifies that the message came from Bob by:

1) Checking that all received parameters are correct, that $r, s \in \mathbb{Z}_n$ and that Bob's public key is valid, that is $Q_B \neq$

$\mathcal{O}$ and $Q_B \in E$ is of order n.
2) Using the same hash function and method as above, compute $\bar{e}$.
3) Compute $\bar{s} = s^{-1} \bmod n$.
4) Find the point $(x, y) = \bar{e}\bar{s}G + r\bar{s}Q_B$.
5) Verify that $r = x \bmod n$ otherwise reject the signature.

Step 2 of the signing algorithm is of vital importance — inappropriate reuse of the random integer led to the highly publicised breaking of Sony PS3 implementation of ECDSA[1]. Knowledge of the random value k, a.k.a. the *ephemeral key* or the *nonce*, leads to knowledge of the secret key. All values (m, r, s) can be observed by an eavesdropper, $\bar{e}$ can be found from m, $r^{-1} \bmod n$ can be easily computed from n and r, and if k is discovered then an adversary can find Bob's secret key through the simple calculation

$$d_B = (sk - \bar{e})r^{-1}.$$

Our attack targets Step 3 of the OpenSSL implementation of ECDSA.

B. ECC in OpenSSL

For ECDSA signing, the performance-critical component is scalar multiplication (Step 3) that, for an ℓ-bit integer k computes

$$kP = \sum_{i=0}^{\ell-1} k_i 2^i P$$

where k_i denotes bit i of k. Two key avenues for improving the performance of this operation are using a low-weight representation for the scalar, coupled with a scalar multiplication algorithm that interleaves elliptic curve additions and doublings, both with a goal of reducing the number of group operations. What follows is a description of how OpenSSL carries out this computation.

Scalar representation: The fact that group element inversion is cheap for elliptic curves makes signed representations for scalars a viable option: "subtraction of points on an elliptic curve is just as efficient as addition" [16, p. 98]. Generally, signed representations reduce the amount of needed precomputation by a factor of 2. A popular choice for ECC is Non-Adjacent Form (NAF) that, with a window width w represents k using digit set $\{0, \pm 1, \pm 3, \ldots, \pm(2^{w-1} - 1)\}$ with the property that all non-zero digits are separated by at least $w - 1$ zeros, leading to lower average weight than other representations (e.g. binary). The modified version mNAF_w is otherwise the same but allows the most significant digit to violate the non-adjacent property if doing so decreases the length but keeps the same weight [33, Sec. 4.1]. It does so by applying the map $10^{w-1}\delta \mapsto 010^{w-2}\hat{\delta}$ if $\delta < 0$ where $\hat{\delta} = \delta + 2^{w-1}$. Figure 8 illustrates the mNAF_w algorithm. See function `bn_compute_wNAF` in `crypto/bn/bn_intern.c` for OpenSSL's implementation of this procedure. Lastly, it is worth noting that the most significant digit in NAF and mNAF_w for $k \geq 1$ is guaranteed to be positive.

[1]http://arstechnica.com/gaming/2010/12/ps3-hacked-through-poor-implementation-of-cryptography/

Input: Integer $k \geq 1$, width w
Output: $\text{mNAF}_w(k)$
$i \leftarrow 0$
while $k \geq 1$ **do**
  if k *is odd* **then** $k_i \leftarrow k \bmod 2^w$, $k \leftarrow k - k_i$ **else**
  $k_i \leftarrow 0$ $k \leftarrow k/2$, $i \leftarrow i + 1$
end
if $k_{i-1} = 1$ **and** $k_{i-1-w} < 0$ **then**
  $k_{i-1-w} \leftarrow k_{i-1-w} + 2^{w-1}$
  $k_{i-1} \leftarrow 0$, $k_{i-2} \leftarrow 1$, $i \leftarrow i - 1$
end
return $(k_{i-1}, \ldots, k_0)$

Fig. 8. Generating modified Non-Adjacent Form for scalars. Here mods takes residues from $-(2^{w-1} - 1)$ to $2^{w-1} - 1$.

Input: Integer $k \geq 1$, $P \in E(\mathbb{F}_q)$, width w
Output: kP
$(k_{\ell-1} \ldots k_0) \leftarrow \text{mNAF}_w(k)$
Precompute jP for all odd $0 < j < 2^{w-1}$
$Q \leftarrow k_{\ell-1}P$
for $i \leftarrow \ell - 2$ **to** 0 **do**
  $Q \leftarrow 2Q$
  if $k_i \neq 0$ **then** $Q \leftarrow Q + k_i P$
end
return Q

Fig. 9. Left-to-right double-and-add scalar multiplication with mNAF_w signed representation

Scalar multiplication: In the absence of any curve-specific routines, for curves over $\mathbb{F}_p$ OpenSSL implements interleaved scalar multiplication by Möller [32, Sec. 3.2] — see scalar multiplication function `ec_wNAF_mul` in `crypto/ec/ec_mult.c` for OpenSSL's implementation. While there are many paths through the code depending on inputs [4, Sec. 2.2], this work assumes the case of a single scalar input where no a priori precomputation structure is available. For this case, the function execution simplifies to a textbook left-to-right, double-and-add scalar multiplication routine — see e.g. Hankerson et al. [16, p. 100]. Figure 9 illustrates the algorithm that will perform ℓ point doublings and a number of point additions equaling the number of non-zero digits (minus the first point addition and plus the $2^{w-2} - 1$ point additions for ad hoc precomputation). Since point Q accumulates the partial scalar multiple, Q is termed the *accumulator*.

C. Attacking ECDSA

As mentioned, an attacker who knows the ephemeral key k used for a *single* signature (m, r, s) can obtain the secret key d_B from a simple calculation. It turns out that knowing a few bits of the nonces for *sufficiently many* signatures allows an attacker to obtain the secret key as well. One option is to embed the information for various signatures into a *lattice* such that the solution to a geometric lattice problem corresponds to the secret key [19, 35, 36].

But how does the attacker obtain any information on the ephemeral keys? As these keys are only used during the

computation, a natural approach is to obtain this information through a side-channel attack. Unfortunately, using the side-channel described above to attack the wNAF implementation of the scalar multiplication does not directly reveal a fixed number of bits of every ephemeral key. This is due to the fact that the side-channel only reveals when the relevant operations take place, but in the case of an addition it does not show which value is being added. Previous works obtain information on the ephemeral key k from the double and add chains in different ways.

L1 dcache targeting fixed lower bits: Brumley and Hakala [5] use the fact that the number of doubles after the last addition in the trace reveals an equal number of least significant bits of k: "From the side channel perspective, consecutive doublings allow inference of zero coefficients, and more than w point doublings reveals non-trivial zero coefficients" [5, Sec. 3.2]. They target signatures and traces that indicate a minimum of six zeros in the LSBs, in total requiring 2600 signatures and corresponding traces to recover the private key for curve `secp160r1` with a lattice attack [5, Sec. 6].

LLC targeting variable lower bits: The numerous drawbacks of the previous attack include (1) discarding on average $1-2^{-6}$ percent of the traces; (2) limiting to SMT architectures like Intel's HT; (3) rather noisy traces from the L1 data cache. Benger et al. [3] tackle all of these issues, while at the same time targeting the substantially larger and relevant curve `secp256k1`: "Prior work fixes a minimum value of [LSBs] and utilizes this single value in all equations ... If we do this we would need to throw away [the majority] of the executions obtained. By maintaining full generality ... we are able to utilize all information at our disposal" [3, Sec. 4]. As each trace reveals a different number of LSBs of the ephemeral key, they adjust the lattice problem accordingly and recover the private key with as little as 200 signatures and corresponding traces. However, to recover the private key with probability greater than 0.5, they require approximately 300 signatures.

LLC targeting full traces: Subsequently, van de Pol et al. [46] show how to use roughly half of the double and add chain for group orders of a special form, i.e., $q = 2^n + \varepsilon$ where $|\varepsilon| < 2^p$ for $p \ll n$. It relies on the fact that the positions of adds in the chain reveal the positions of non-zero wNAF digits in the representation of k. Two adds are separated by at least w doubles, and every additional double reveals that the corresponding bits of k are repeating. However, a single bit of information is lost for every pair of consecutive non-zero wNAF digits, because these repeating bits of k are either zero or one depending on whether the second wNAF digit was positive or negative. Note that this method requires perfect traces, because each double is required to determine the bit position of the various additions. Therefore, whenever a double is missed, the whole trace preceding the missed double will produce inaccurate information.

D. Point Inversion: A New Leak

An implementation of scalar multiplication in Figure 9 requires accompanying control logic — in particular, to handle negative k_i digits. We observe the following trends in open source elliptic curve libraries for inverting points.

Invert on-the-fly: While the cost of elliptic curve point inversion can vary depending on the coordinate system choice, for many systems $\mathbb{F}_p$ curves require only a finite field negation, i.e. flipping the sign of the y-coordinate. In these cases, since point inversion is so light many implementations opt for on-the-fly inversion. That is, when $k_i < 0$ compute $Q := Q + -(k_i P)$ inverting $k_i P$ to a temporary variable immediately preceding the point addition function call. For example, this is the approach taken by Bitcoin's libsecp256k1[2]. Scalar multiplication function `secp256k1_ecmult` in `src/ecmult_impl.h` calls macro `ECMULT_TABLE_GET_GE` which, in the case of a negative digit, calls `secp256k1_ge_neg` in `src/group_impl.h` to negate the point operand. The advantage to this approach is that it requires marginal additional storage overhead, and the disadvantage is that the algorithm will eventually end up inverting the same point more than once — duplicating a previously computed value.

Precompute inversions: As written, the precomputation table in Figure 9 requires storing 2^{w-2} points. Another strategy is to double the size of the table and additionally store the inverses of the required points. Then for negative digits, compute $Q := Q + (\hat{k}_i)P$ where $\hat{k}_i$ is the table index for $-k_i$. Normally this will be handled in the NAF coding itself by yielding e.g. indices $(0, 1, 2, 3)$ corresponding to digits $(1, 3, -3, -1)$. For example, this is the approach taken by NSS[3] — see scalar multiplication function `ec_GFp_pt_mul_jm_wNAF` in `lib/freebl/ecl/ecp_jm.c`. The advantage of this approach is that each point is only inverted a single time, and the disadvantage is that the required storage for the precomputation table doubles.

Invert the accumulator: Similar to the invert on-the-fly approach but without requiring a temporary point, another strategy is to track the sign of the accumulator in a variable and invert the accumulator as needed preceding point additions. That is, if the sign of the accumulator matches the sign of the digit $k_i \neq 0$, compute $Q := Q + |k_i|P$; otherwise $Q := -Q + |k_i|P$, inverting the accumulator before the point addition function call. Finally, after all digits are processed set $Q := -Q$ if the accumulator is in the inverted state. For example, this is the approach taken by OpenSSL — see scalar multiplication function `ec_wNAF_mul` in `crypto/ec/ec_mult.c` that calls `EC_POINT_invert` if *precisely* one of the following statements is true:

- the current (non-zero) digit is negative (variable `is_neg`);
- the accumulator is inverted (variable `r_is_inverted`).

While all of the above approaches have potential side-channel issues, we focus on the last approach since OpenSSL implements it. From the side-channel perspective, if we capture the sequence of elliptic curve point doublings, additions, and inversions for a particular k we can recover the signs of all non-zero k_i digits as follows. Denote the n inversions by $I_1 \dots I_n$ for $n > 1$. Note that n is always even since the accumulator (Q) always starts and ends in the non-inverted state, and (for completeness) that if $n = 0$ then all digits are positive. The accumulator toggles to the inverted state at I_1, then back to the non-inverted state at I_2, and so on — i.e. the accumulator

[2] https://github.com/bitcoin/secp256k1

[3] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS

enters the inverted state at I_j for odd j and non-inverted state for even j. Hence:

1) All the additions before I_1 correspond to positive digits.
2) For odd j, all the additions between I_j and I_{j+1} correspond to negative digits. This is due to the fact that the sign of the accumulator agrees with the sign of the current digit for such additions.
3) Similarly for even j, all the additions between I_j and I_{j+1} correspond to positive digits.

E. Exploiting the new leak

The improved side-channel described in this work allows us to determine whether the wNAF digits are positive or negative. This immediately gives one extra bit of information for each pair of consecutive adds in the top half of the double and add chain. Using the notation of van de Pol et al. [46], if there are two consecutive adds at positions m and $m+l$ for $p < m < n-l$ we can write

$$k = a \cdot 2^{m+l+1} + b \cdot 2^{m+w} + c,$$

where $0 \leq a < 2^{n-m-l}$, $2^{m-1} < c < 2^{m+w} - 2^{m-1}$, and $b = 2^{l-w}$ if the second wNAF digit is positive or $b = 2^{l-w} - 1$ if it is negative.

Now, if we define $t = (r/s) \cdot 2^{n-m-l-1} \bmod q$ and $u = (b + 1/2) \cdot 2^{n+w-l-1} - (h/s) \cdot 2^{n-m-l-1} \bmod q$, it follows that $|d_B \cdot t - u|_q < q/2^{l-w+2}$, where $|.|_q$ is reduction $\bmod q$ into the range $[-q/2, q/2)$. Writing $z = l - w + 1$, each such triple (u, t, z) provides z bits of information about the secret key d_B, but conversely increases the dimension of the closest vector problem in the lattice by one. To balance the hardness of the lattice problem with the information provided by the triples, we only used the 75 triples with the highest z-values which results in a lattice dimension of 76. Table I shows the result of this attack for a varying number of signatures σ. It was implemented using the `fplll` library[4] and executed on a single core of an Intel E5620 processor. Thus, given six error-free traces on different signatures allows an attacker to obtain the secret key in more than half the cases.

TABLE III. ATTACK RESULTS FOR A GIVEN NUMBER OF SIGNATURES σ

σ	Time (s)	Prob
4	15.08	0.005
5	13.94	0.165
6	12.51	0.545
7	11.50	0.735
8	9.69	0.840

VII. AMPLIFICATION ATTACK

In the previous section, we identified a new leak in the OpenSSL implementation of ECDSA and analysed the leak under the assumption that the adversary can obtain a perfect trace of the victim's operations. In this section, we investigate the practical issues of obtaining perfect traces. We first look at why error-free traces are required for the attack. We proceed with describing how past research used the FLUSH+RELOAD attack to achieve a high probability of obtaining perfect traces. We then explain why these techniques are not sufficient when we want to capture inversions. We show that amplification allows us to overcome the limitations and demonstrate how to use it to obtain perfect traces.

[4]http://perso.ens-lyon.fr/damien.stehle

A. The need for perfect traces

Suppose that the adversary manages to obtain an almost-perfect trace. That is, she knows the sequence of operations taken by the victim with the exception of a single error that causes it to either miss a double operation or add a spurious one. When inferring the positions of the non-zero wNAF digits, the error will propagate through the representation of the scalar, changing the position of all digits to the left of the error, which are the positions we use for the lattice attack. Consequently, the lattice attack will receive an erroneous input and will fail to find the key. Similarly, if the trace misses an inversion or contains a spurious one, the signs of any digit above the error locations are incorrect.

Even if the adversary knows or suspects that an error has occurred, correcting the error poses problems. If, for example, the adversary notes that the time between two operations in the trace is longer than expected, the adversary can suspect an error. However, because the victim may have been suspended while the processor executed some system function, the adversary cannot be certain that an error occurred.

The adversary could try to use known properties of perfect traces to identify and possibly correct errors in captured traces. However, there is very little information that the adversary can use. In particular, the adversary does not know for certain the number and position of point addition operations. She can detect, but not correct, errors like: (1) the number of point inversions must be even; (2) at least w point doublings must separate point additions. Finally, even though all the scalars used in the multiplication have the same bit length (due to a timing attack [6] resulting in `CVE-2011-1945`), the length of the wNAF representation may vary. For example, we look at the numbers 228 and 229. The binary representations of these is 11100100 and 11100101, i.e. both are 8 bit numbers. The 4-NAF representation of 228 is $1, 0, 0, 0, 0, 0, -7, 0, 0$. That is, $228 = 1 \cdot 2^8 - 7 \cdot 2^2$. The representation of 229 is $7, 0, 0, 0, 0, 5$ — $229 = 7 \cdot 2^5 + 5$. Hence, while the bit length of both numbers is 8, the length of their 4-NAF representations are 9 and 6. Consequently, the number of double operations in the trace is not fixed.

As we can see, the effects of errors in the trace are not localised, errors are hard to detect, and are almost impossible to correct. Combined with the sensitivity of the lattice attack to errors, every small error in the captured trace significantly reduces the probability of attack success. In particular, unless the adversary can get enough error free traces, she will not be able to apply the attack.

B. Obtaining perfect traces

van de Pol et al. [46] attack the same implementation that we target. Unlike us, they do not try to trace the accumulator inversions, focusing instead on add and double operations. van de Pol et al. [46] divide time into slots of 1,200 cycles and probe memory lines within the functions that implement the group add and group double operations. As Section V

demonstrates, with slots of 1,200 cycles the expected miss rate is around 44%.

To reduce the miss probability, van de Pol et al. [46] choose memory lines that contain a call to a field multiplication operation. As discussed above, the victim accesses memory lines that contain a call twice; once when executing the call and once when the call returns. Because these two accesses are related, their times are not independent and the probability of missing each is not independent of each other. Consequently, van de Pol et al. [46] manages to reduce the number of capture errors to 1 in 1,000 group operations. With around 300 operations in trace, the probability of capturing an error-free trace is 58%.

For our attack, we need to further trace accumulator inversions along with group addition and double. While the group inversion code contains a call instruction, we cannot probe the memory line that contains it. The reason is that due to speculative execution, all of the code up to the call instruction is prefetched into the cache, even if the execution does not take the path. As a result, monitoring these memory lines would result in a large number of false positives. Therefore, we have to monitor memory lines that follow the call to the field negation, which do not contain additional call instructions.

In our environment (OpenSSL 1.0.2a running on an HP Elite 8300 running CentOS 6.5 64 bit), add operations take on average 3,223 cycles and double operations take 3,427 cycles. As Benger et al. [3] discuss, the maximum slot length we can use is about half the length of the operations, or 1,600 cycles. With these time slots, the probability of missing the victim access to the memory line in the inversion code is about 33%. With such an error probability, and an expected number of 25 inversions in each scalar multiplication, the probability of capturing a perfect trace is less than 1/25,000, which is way too low for a practical attack.

One possibility of reducing the miss probability when tracing accumulator inversions is to monitor two memory lines within the code. The scalar multiplication code in OpenSSL invokes the generic elliptic curve point inversion function `EC_POINT_invert`. The function invokes the curve-specific point inversion function, which in the case of the `secp256k1` curve is `ec_GFp_simple_invert`. Said function invokes field subtraction (`BN_usub`) to negate the y component of the point. By probing the memory lines following the return of `BN_usub` and the return of `ec_GFp_simple_invert` we get the same effect as probing a memory line that contains a `call` instruction, with the adversary missing at most one of these accesses.

While this approach guarantees that the adversary does not miss accumulator inversions, it requires the adversary to monitor four memory lines: one in each of the double and add functions and two in the inversion functions. Each probe takes about 450 cycles, so probing four memory lines takes 1,800 cycles. When we set the slot size to 1,800 cycles, the traces loses accuracy because we can no longer determine the order of some of the operations in the sequence.

Increasing the slot length would allow us to consistently trace all accumulator inversions, however the speed of calculating the group addition and doubling limits the maximum slot length. To increase the limit, we can try slowing the group operations down.

C. A performance-degradation attack against OpenSSL

We use our performance-degradation attack to slow the group operations down. We target the `bn_mul_mont` function, which implements the field multiplication and square. We use one attacking thread and check the effect of repeatedly evicting one or two memory lines in the main loop of the function. Table IV summarises the run time of the add and the double operations under the attack. As we can see, repeatedly evicting one memory line in the field multiplication reduces the speed of the add group operation by a factor of 9. The double operation is slowed down by a factor of over 7. When we evict two memory lines, the group operations are slowed down by a factor of 47 and 36.

TABLE IV. SECP256K1 GROUP OPERATION TIMES (CYCLES)

	Add	**Double**
No attack	3,223	3,427
Evicting one line	29,605	25,264
Evicting two lines	152,409	125,660

With group operations taking over 100,000 cycles, we can safely increase the slot size and monitor the four memory lines required for obtaining the trace. We set the slot size to 17,000 cycles and captured 1,000 traces. Comparing the traces to the ground truth we find that only five of them show errors. Hence, our attack captures error-free traces almost every time. We can now use these traces with the lattice attack of Section VI, to break the long-term ECDSA key of the victim after observing as few as six signatures.

Table V compares the results of this work with previous cache-based attacks on OpenSSL ECDSA. As we can see, the attack requires less than a quarter of the previous best attack. About half of this improvement is due to exploiting the leak of point inversion and the other half comes from the increased accuracy of observing the side-channel. Employing the performance-degradation attack to amplify the side-channel underpins both these improvements.

TABLE V. OPENSSL ECC CACHE-TIMING ATTACK RESULTS COMPARED

Curve	**Source**	**Perfect traces**	**Signatures**
`secp160r1`	Brumley and Hakala [5]	-	2600
`secp256k1`	Benger et al. [3]	-	300
`secp256k1`	van de Pol et al. [46]	13	25
`secp256k1`	this work	6	6

VIII. CONCLUSION

Typical performance-degradation attacks usually do not provide any direct benefit to the attacker. Their main benefit is derived indirectly, through the damage they cause to the victim. In this paper we demonstrate that these attacks can offer tangible benefits to the attacker—it can be used to amplify a side-channel, allowing the attacker to receive more information through the channel than was otherwise possible.

To demonstrate side-channel amplification, we first identify a new microarchitectural attack vector, which is over

8 times more potent than previously published attacks. We further identify a new information leak in the OpenSSL implementation of the ECDSA signature scheme. Lastly, we show how using the new performance-degradation attack to amplify a cache side channel allows the attacker to exploit the information leak. Our combined attack allows the adversary to completely cryptanalyse the `secp256k1` elliptic curve used in Bitcoin after observing the side channel over only 6 signatures, less than a quarter of any prior result.

Disabling cross-domain memory sharing and disabling the `clflush` instructions are two suggested countermeasures for the FLUSH+RELOAD attack and for the Rowhammer attack [24]. These countermeasures also affect our performance-degradation attack. Instead of using the `clflush` instruction, the victim line can be evicted by creating a contention on the cache set it is stored in. Using this technique will also obviate the need for memory sharing. Further work is required to determine the effectiveness of this technique for performance degradation and whether it can be used for side channel amplification.

ACKNOWLEDGEMENTS

We would like to thank Dr Naomi Benger for the useful discussions, advice and support. We would also like to thank Camilla Beck and Diclehan Erdal for performing some of the experiments for this work.

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

Parts of this research was performed under contract to the Defence Science and Technology Group, Maritime Division, Australia.

This research was supported in part by COST Action IC1306.

The second author was supported in part by TEKES grant 4681/31/2014 INKA EAKR Hardware Rooted Security.

The fourth author was supported in part by EPSRC via grant EP/I03126X.

REFERENCES

[1] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES*, Santa Barbara, CA, US, 2010.

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *2009 Ottawa Linux Symp.*, pages 19–28, Montreal, Quebec, Canada, Jul 2009.

[3] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah, just a little bit": A small amount of side channel can go a long way. In *CHES*, volume 8731 of *LNCS*, pages 75–92, Busan, KR, September 2014.

[4] Billy Bob Brumley. Faster software for fast endomorphisms. In *6th COSADE*, pages 127–140, Berlin, DE, Apr 2015.

[5] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *15th ASIACRYPT*, pages 667–684, Tokyo, JP, Dec 2009.

[6] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *16th ESORICS*, Leuven, BE, 2011.

[7] Carlos Cardenas and Rajendra V Boppana. Detection and mitigation of performance attacks in multi-tenant cloud computing. In *1st International IBM Cloud Academy Conference*, Research Triangle Park, NC, US, 2012.

[8] Ron C. Chiang, Sundaresan Rajasekaran, Nan Zhang, and H. Howie Huang. Swiper: Exploiting virtual machine vulnerability in third-party clouds with competition for I/O resources. *Trans. Parall. & Distr. Syst.*, 26(6):1732–1742, Jun 2015.

[9] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, Santa Barbara, CA, US, 1985.

[10] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. http://www.agner.org/optimize/microarchitecture.pdf, Aug 2014.

[11] Steven D. Galbraith and Pierrick Gaudry. Recent progress on the elliptic curve discrete logarithm problem. IACR Cryptology ePrint Archive, Report 2015/1022, Oct 2015.

[12] David Goldberg. What every computer scientist should know about floating-point arithmetic. *Comput. Surveys*, 23(1):6–48, Mar 1991.

[13] Dirk Grunwald and Soraya Ghiasi. Microarchitectural denial of service: Insuring microarchitectural fairness. In *35th MICRO*, pages 409–418, Istanbul, TR, Nov 2002.

[14] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security*, Washington, DC, US, 2015.

[15] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *S&P*, pages 490–505, Oakland, CA, US, 2011.

[16] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Professional Computing. 2004.

[17] Jahangir Hasan, Ankit Jalote, T. N. Vijaykumar, and Carla E. Brodley. Heat stroke: Power-density-based denial of service in SMT. In *11th HPCA*, pages 166–177, San Francisco, CA, US, Feb 2005.

[18] J.L. Henning. SPEC CPU2006 benchmark descriptions. *Comp. Arch. News*, 34(4), Sep 2006.

[19] Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *DCC*, 23(3):283–290, Aug 2001.

[20] Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. Sep 2015.

[21] Intel 64 & IA-32 AORM. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, Apr 2012.

[22] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-VM attack on AES. In *RAID*, pages 299–319, Gothenburg, Sweden, Sep 2014.

[23] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *ASIACCS*, pages 85–96, Singapore, Apr 2015.

[24] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *41st ISCA*, pages 361–372, Jun 2014.

[25] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics Comput.*, 48(177):203–209, Jan 1987.

[26] Neal Koblitz and Alfred Menezes. A riddle wrapped in an enigma. IACR Cryptology ePrint Archive, Report 2015/1018, Nov 2015.

[27] Sarah Laing, Michael E. Locasto, and John Aycock. An experience report on extracting and viewing memory events via Wireshark. In *8th WOOT*, San Diego, CA, US, Aug 2014.

[28] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *S&P*, pages 605–622, San Jose, CA, US, May 2015.

[29] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrius Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation of virtualization systems. In *WS Experimental Comp. Sci.*, San Diego, CA, US, Jun 2007.

[30] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *RAID*, Kyoto, Japan, Nov 2015.

[31] Victor S. Miller. Use of elliptic curves in cryptography. In *CRYPTO'85*, pages 417–426, Santa Barbara, CA, US, Aug 1985.

[32] Bodo Möller. Algorithms for multi-exponentiation. In *SAC*, pages 165–180, Toronto, ON, CA, Aug 2001.

[33] Bodo Möller. Improved techniques for fast exponentiation. In *Inform. Security & Cryptology*, pages 298–302, Seoul, KR, Nov 2002.

[34] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *16th USENIX Security*, Boston, MA, US, 2007.

[35] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *J. Cryptology*, 15(2):151–176, Jun 2002.

[36] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *DCC*, 30(2):201–217, Sep 2003.

[37] NIST FIPS PUB 186-4. *Digital Signature Standard (DSS)*. NIST, 2013.

[38] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. http://www.cs.tau.ac.il/~tromer/papers/cache.pdf, Nov 2005.

[39] Daniel Page. *Practical Introduction to Computer Architecture*. Texts in Computer Science. 2009. URL http://dx.doi.org/10.1007/978-1-84882-256-6.

[40] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, CA, 2005.

[41] Andre Richter, Christian Herber, Holm Rauchfuss, Thomas Wild, and Andreas Herkersdorf. Performance isolation exposure in virtualized platforms with PCI passthrough I/O sharing. In *Architecture of Computing Systems*, pages 171–182. 2014.

[42] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *16th CCS*, pages 199–212, Chicago, IL, US, 2009.

[43] Vineet Sinha, David Karger, and Rob Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, pages 21–25, San Diego, CA, US, Oct 2005.

[44] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest OS. In *4th European Workshop on System Security*, Salzburg, AT, 2011.

[45] Augustus K. Uht and Vijay Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *28th MICRO*, pages 313–325, Nov 1995.

[46] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *2015 CT-RSA*, pages 3–21, San Francisco, CA, USA, Apr 2015.

[47] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M Swift. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *19th CCS*, Raleigh, NC, US, 2012.

[48] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *5th OSDI*, Boston, MA, US, 2002.

[49] Chuliang Weng, Jianfeng Zhan, and Yuan Luo. TSAC: Enforcing isolation of virtual machines in clouds. *Trans. Computers*, 64(5):1470–1482, May 2015.

[50] Dong Hyuk Woo and Hsien-Hsin S. Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *WS Chip Multiprocessor Memory Syst. & Interconnects*, Phoenix, AZ, US, 2007.

[51] Carole-Jean Wu and Margaret Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *Int. Symp. Performance Analysis Syst. & Softw.*, ISPASS '11, Austin, TX, US, 2011.

[52] Tianni Xu, Xiufeng Sui, Zhicheng Yao, Jiuyue Ma, Bao Yungang, and Lixin Zhang. Rethinking virtual machine interference in the era of cloud applications. In *15th HPCC*, pages 190–197, Zhangjiajie, Hunan, China, Nov 2013.

[53] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. IACR Cryptology ePrint Archive, Report 2014/140, Feb 2014.

[54] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security*, pages 719–732, San Diego, CA, US, 2014.

[55] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. IACR Cryptology ePrint Archive, Report 2015/905, Sep 2015.

[56] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *19th CCS*, pages 305–316, Raleigh, NC, US, 2012.

[57] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant side-channel attacks in PaaS clouds. In *21st CCS*, Scottsdale, AZ, US, 2014.