# HyperDegrade: From GHz to MHz Effective CPU Frequencies

Alejandro Cabrera Aldaya Billy Bob Brumley
*Tampere University, Tampere, Finland*
*{alejandro.cabreraaldaya,billy.brumley}@tuni.fi*

## Abstract

Performance degradation techniques are an important complement to side-channel attacks. In this work, we propose HYPERDEGRADE—a combination of previous approaches and the use of simultaneous multithreading (SMT) architectures. In addition to the new technique, we investigate the root causes of performance degradation using cache eviction, discovering a previously unknown slowdown origin. The slowdown produced is significantly higher than previous approaches, which translates into an increased time granularity for FLUSH+RELOAD attacks. We evaluate HYPERDEGRADE on different Intel microarchitectures, yielding significant slowdowns that achieve, in select microbenchmark cases, three orders of magnitude improvement over state-of-the-art. To evaluate the efficacy of performance degradation in side-channel amplification, we propose and evaluate leakage assessment metrics. The results evidence that HYPERDEGRADE increases time granularity without a meaningful impact on trace quality. Additionally, we designed a fair experiment that compares three performance degradation strategies when coupled with FLUSH+RELOAD from an attacker perspective. We developed an attack on an unexploited vulnerability in OpenSSL in which HYPERDEGRADE excels—reducing by three times the number of required FLUSH+RELOAD traces to succeed. Regarding cryptography contributions, we revisit the recently proposed Raccoon attack on TLS-DH key exchanges, demonstrating its application to other protocols. Using HYPERDEGRADE, we developed an end-to-end attack that shows how a Raccoon-like attack can succeed with real data, filling a missing gap from previous research.

## 1 Introduction

Side Channel Analysis (SCA), is a cryptanalytic technique that targets the implementation of a cryptographic primitive rather than the formal mathematical description. Microarchitecture attacks are an SCA subclass that focus on vulnerabilities within the hardware implementation of an Instruction Set Architecture (ISA). While more recent trends exploit speculation [36, 39], classical trends exploit contention within different components and at various levels. Specifically for our work, the most relevant is cache contention.

Percival [49] and Osvik et al. [44] pioneered access-driven L1 data cache attacks in the mid 2000s, then Acıiçmez et al. [3] extended to the L1 instruction cache setting in 2010. Most of the threat models considered only SMT architectures such as Intel's HyperThreading (HT), where victim and spy processes naturally execute in parallel. Yarom and Falkner [61] removed this requirement with their groundbreaking FLUSH+RELOAD technique utilizing cache line flushing [31], encompassing cross-core attacks in the threat model by exploiting (inclusive) Last Level Cache (LLC) contention.

In this work, we examine the following Research Questions (RQ).

**RQ 1: Research Question 1**

With respect to SMT architectures, are CPU topology and affinity factors in performance degradation attacks?

Allan et al. [7] proposed DEGRADE as a general performance degradation technique, but mainly as a companion to FLUSH+RELOAD attacks. They identify hot spots in code and repeatedly flush to slow down victims—in the FLUSH+RELOAD case, with the main goal of amplifying trace granularity. Pereida García and Brumley [50] proposed an alternate framework for hot spot identification. We explore RQ 1 in Section 3 to understand what role physical and logical cores in SMT architectures play in performance degradation. Along the way, we discover the root cause of DEGRADE which we subsequently amplify. This leads to our novel HYPERDEGRADE technique, and Section 4 shows its efficacy, with slowdown factors in select microbenchmark cases remarkably exceeding three orders of magnitude.

**RQ 2: Research Question 2**

Does performance degradation lead to FLUSH+RELOAD traces with statistically more info. leakage?

Nowadays, FLUSH+RELOAD coupled with DEGRADE is a standard offensive technique for academic research. While both Allan et al. [7] and Pereida García and Brumley [50] give convincing use-case specific motivation for why DEGRADE is useful, neither actually show the information-theoretic advantage of DEGRADE. Section 5 closes this gap and partially answers RQ 2 by utilizing an existing SCA metric to demonstrate the efficacy of DEGRADE as an SCA trace amplification technique. We then extend our analysis to our HYPERDEGRADE technique to resolve RQ 2. At a high level, it shows HYPERDEGRADE leads to slightly noisier individual measurements yet positively disproportionate trace granularity.

**RQ 3: Research Question 3**

Can HYPERDEGRADE reduce adversary effort when attacking crypto implementations?

RQ 2 compared HYPERDEGRADE with previous approaches from a theoretical point of view. In Section 6 we compare the three approaches from an *applied* perspective, showing a clear advantage for HYPERDEGRADE over the others.

**RQ 4: Research Question 4**

Can a Raccoon attack (variant) succeed with real data?

Merget et al. [40] recently proposed the Raccoon attack (e.g. CVE-2020-1968), a timing attack targeting recovery of TLS 1.2 session keys by exploiting DH key-dependent padding logic. Yet the authors only model the SCA data and abstract away the protocol messages. Section 6 answers RQ 4 by developing a microarchitecture timing attack variant of Raccoon, built upon FLUSH+RELOAD and our new HYPERDEGRADE technique. Our end-to-end attack uses real SCA traces and real protocol (CMS) messages to recover session keys, leading to loss of confidentiality. We conclude in Section 7.

# 2 Background

## 2.1 Memory Hierarchy

Fast memory is expensive, therefore computer system designers use faster yet smaller caches of slower yet larger main memory to benefit from locality without a huge price increase. A modern microprocessor has several caches (L1, L2, LLC) forming a cache hierarchy [45, Sect. 8.1.2], the L1 being the fastest one but smaller and tightly coupled to the processor. Caches are organized in cache lines of fixed size (e.g., 64 bytes). Two L1 caches typically exist, one for storing instructions and the other for data. Regarding this work, we are mainly interested in the L1 instruction cache and remaining levels.

When the processor needs to fetch some data (or instructions) from memory, it first checks if they are already cached in the L1. If the desired cache line is in the L1, a *cache hit* occurs and the processor gets the required data quickly. On the contrary if it is not in the L1, a *cache miss* occurs and the processor tries to fetch it from the next, slower, cache levels or in the worst case, from main memory. When gathering data, the processor caches it to reduce latency in future loads of the same data, backed by the principle of locality [45, Sect. 8.1.5].

## 2.2 Performance Degradation

In contrast to generic CPU monopolization methods like the "cheat" attack by Tsafrir et al. [56] that exploit the OS process scheduler, several works have addressed the problem of degrading the performance of a victim using microarchitecture components [29, 30, 33, 42]. However, in most cases it is not clear whether SCA-based attackers gain benefits from the proposed techniques.

On the other hand, Allan et al. [7] proposed a cache-eviction based performance degradation technique that enhances FLUSH+RELOAD attack SCA signals (traces). This method has been widely employed in previous works to mount SCA attacks on cryptography implementations. For instance RSA [10], ECDSA [8], DSA [51], SM2 [58], AES [18], and ECDH [24].

The performance degradation strategy proposed by Allan et al. [7], DEGRADE from now on, consists of an attacker process that causes cache contention by continuously issuing `clflush` instructions. It is an unprivileged instruction that receives a virtual memory address as an operand and evicts the corresponding cache line from the entire memory hierarchy [1].

This attack applies to shared library scenarios, which are common in many OSs. This allows an attacker to load the same library used by the victim and receive a virtual address that will point to the same physical address, thus, same cache line. Therefore, if the attacker evicts said cache line from the cache, when the victim accesses it (e.g., executes the code contained within it), a cache miss will result, thus the microprocessor must fetch the content from slower main memory.

## 2.3 Leakage Assessment

Pearson's correlation coefficient, Welch's T-test, Test Vector Leakage Assessment (TVLA), and Normalized Inter-Class Variance (NICV) are established statistical tools in the SCA field. Leakage assessment leverages these statistical tools to identify leakage in procured traces for SCA. A short summary follows.

Pearson's correlation coefficient measures the linear similarity between two random variables. It is generally useful for leakage assessment [19, Sect. 3.5] and Point of Interest (POI)

identification within traces, for example in template attacks [17] or used directly in Correlation Power Analysis (CPA) [16]. POIs are the subset of points in an SCA trace that leak sensitive information.

Welch's T-test is a statistical measure to determine if two sample sets were drawn from populations with similar means. Goodwill et al. [26] proposed TVLA that utilizes the T-test for leakage assessment by comparing sets of traces with fixed vs. random cryptographic keys and data.

Lastly, Bhasin et al. [11] propose NICV for leakage assessment. It is an ANalysis Of VAriance (ANOVA) F-test, a statistical measure to determine if a number of sample sets were drawn from populations with similar variances.

## 2.4 Key Agreement and SCA

Merget et al. [40] recently proposed the Raccoon attack that exploits a specification-level weakness in protocols that utilize Diffie-Hellman key exchange. The key insight is that some standards, including TLS 1.2 and below, dictate stripping leading zero bytes from the shared DH key (session key, or pre-master secret in TLS nomenclature). This introduces an SCA attack vector since, at a low level, this behavior trickles down to several measurable time differences in components like compression functions for hash functions. In fixed DH public key scenarios, an attacker observes one TLS handshake (the target) then repeatedly queries the victim using a large number of TLS handshakes with chosen inputs. Detecting shorter session keys through timing differences, the authors use these inputs to construct a lattice problem to recover the target session key, hence compromising confidentiality for the target TLS session.

# 3 HyperDegrade: Concept

The objective of HYPERDEGRADE is to improve performance degradation offered by DEGRADE when targeting a victim process, resulting in enhanced SCA traces when coupled with a FLUSH+RELOAD attack. Under a classical DEGRADE attack, the degrading process continuously evicts a cache line from the cache hierarchy, forcing the microprocessor to fetch the cache line from main memory when the victim needs it.

It would be interesting to evaluate the efficacy of the DEGRADE strategy, seeking avenues for improvement. The root cause of DEGRADE as presented in [7] is the cache will produce more misses during victim execution—we present novel results on this later. Therefore, the cache miss to executed instructions ratio is a reasonable metric to evaluate its performance.

For this task, we developed a proof-of-concept victim that executes custom code located in a shared library. This harness receives as input a cache line index, then executes a tight loop in said cache line several times. Figure 1 shows the code

```
                                5000:   add    $0x1,%rsi
                                5004:   sub    $0x1,%rsi
.p2align 12                     5008:   add    $0x1,%rsi
L0:                             500c:   sub    $0x1,%rsi
.rept 64                        5010:   add    $0x1,%rsi
    .rept 6                     5014:   sub    $0x1,%rsi
        add $1, %rsi            5018:   add    $0x1,%rsi
        sub $1, %rsi            501c:   sub    $0x1,%rsi
    .endr                       5020:   add    $0x1,%rsi
    add $1, %rsi                5024:   sub    $0x1,%rsi
    sub $2, %rsi                5028:   add    $0x1,%rsi
    jz END                      502c:   sub    $0x1,%rsi
    jmp *%rdi ; L0              5030:   add    $0x1,%rsi
    .p2align 6                  5034:   sub    $0x2,%rsi
.endr                           5038:   je     6000 <END>
                                503e:   jmpq   *%rdi      ; L0
```

Figure 1: Victim single cache line loop (code and disasm.).

snippet of this loop at the left, and one cache line disassembled code at the right.

For our experiments the number of iterations executed is $2^{16}$ (defined by `rsi`). Therefore, we expect the number of instructions executed in the selected cache line is about 1M. Under normal circumstances, every time the processor needs to fetch this code from memory, the L1 cache should serve it very quickly.

## 3.1 Degrade Revisited

On the DEGRADE attacker side, we developed a degrading process that loads the same shared library and continuously evicts the victim executed cache line using `clflush`. We use the Linux `perf` tool to gather statistics about victim execution under a DEGRADE attack. For this task, we used the `perf` (commit `13311e74`) FIFO-based performance counters control to sync their sampling with the victim and degrade processes. `perf` uses two FIFOs for this task, one for enabling/disabling the performance counters and another for giving ACKs. The sync procedure in our measurement tooling is the following:

1. The degrade process executes and it blocks until receiving an ACK packet from `perf` using FIFO `A`.

2. `perf` executes with counters disabled ("-D -1" option), using FIFO `C` for control and `A` for ACKs. Then it runs `taskset` that executes the victim pinned to a specific core.

3. The victim enables the counters by writing to `C`, then it blocks until it receives an ACK from the degrade process using another FIFO.

4. When `perf` receives the enable counters command, it sends an ACK using `A` to the degrade process. When the latter receives the ACK, it forwards it to the victim. When the victim receives this packet, it starts executing its main loop (Figure 1).

5. Once the victim finishes, it disables the counters in `perf`.

Table 1: NODEGRADE and DEGRADE statistics.

| Parameter | NODEGRADE | DEGRADE |
|---|---|---|
| `inst_retired.any` | 1.5M | 1.5M |
| `L1-icache-load-misses` | 4,115 | 33,785 |

This procedure considerably reduces measurement tooling overhead, but some remains. The NODEGRADE strategy does not use a degrade process, however we used a dummy process that follows the FIFO logic to unify the sync procedure among experiments. We repeated each experiment 100 times, gathering the average and relative standard deviation. In all reported cases the latter was less than 4%, therefore we used the average for our analysis. We recorded the number of L1 instruction cache misses and the number of instructions retired by the microprocessor. For these experiments, we used the environment setup Coffee Lake detailed in Table 3.

We collected data while the victim was running standalone (i.e., NODEGRADE strategy) and while it was under DEGRADE effect. Table 1 shows the results for each `perf` event. The number of retired instructions is roughly the same between both experiments, where the difference from expected (1M) is likely due to the measurement tooling overhead. Nevertheless, the number of L1 instruction cache misses was 4k for the NODEGRADE test and 33k for DEGRADE. However, 33k is still far below one cache-miss per executed instruction (1M).

## 3.2 The HyperDegrade Technique

In order to increase the performance impact of DEGRADE, we attempt to maximize the number of cache misses. For this task we made the hypothesis that in an SMT architecture, if the degrade process is pinned to the victim's sibling core, then the number of cache misses will increase.

According to an expired patent from Intel concerning `clflush` [46], the microarchitectural implementation of this instruction in the ISA distinguishes if the flushed cache line is already present in the L1 or not. While it is not explicitly stated in that document as there is no latency analysis, it is our belief that the flushed cache line would be evicted from the L1 before others caches, e.g., due to the proximity wrt., for instance, the LLC controller. Figure 2 illustrates this idea, where the arrows represent `clflush` actions and the dashed ones are slower than the others.

Following this hypothesis, we present HYPERDEGRADE as a cache-evicting degrade strategy that runs in the victim sibling core in a microarchitecture with SMT support. From an architecture perspective it does the same task as DEGRADE, but in the same physical core as the victim. However, the behavior at the microarchitecture level is quite different because, if our hypothesis is correct, it should produce more cache misses due to the local proximity of the L1. To support this claim, we repeated the previous experiment while pinning the

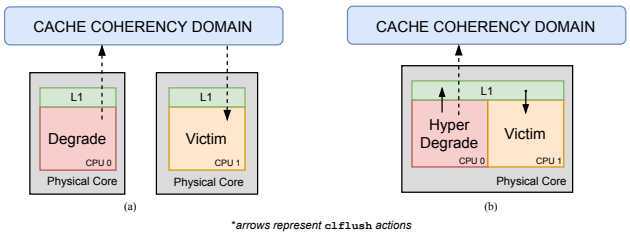


Figure 2: DEGRADE vs HYPERDEGRADE from `clflush` perspective.

Table 2: HYPERDEGRADE improvement.

| Parameter | NODEGRADE | DEGRADE | HYPERDEGRADE |
|---|---|---|---|
| `inst_retired.any` | 1.5M | 1.5M | 1.5M |
| `L1-icache-load-misses` | 4,115 | 33,785 | 992,074 |
| `cycles` | 1,252,211 | 12,935,389 | 504,395,314 |
| `machine_clears.smc` | $< 1$ | 28,375 | 983,348 |

degrade process to the victim sibling core.

Table 2 shows the results of HYPERDEGRADE in comparison with the previous experiment. Note that with HYPERDEGRADE there are about 33x cache misses[1] than with DEGRADE, translating to a considerable increase in the number of CPU cycles the processor spends executing the victim. At the same time, the number of observed cache misses increased considerably, approaching the desired rate. This result, while not infallible proof, supports our hypothesis that sharing the L1 with the victim process should produce higher performance degradation.

On the other hand, note the number of CPU cycles increases by a higher factor (43x), which leads us to suspect there could be another player that is influencing the performance degradation; further research is needed. After repeating the experiment for several `perf` parameters, we found an interesting performance counter that helps explain this behavior.

It is the number of *machine clears* produced by *self-modifying code* or SMC (`machine_clears.smc`). According to Intel, a *machine clear or nuke* causes the entire pipeline to be cleared, thus producing a *severe performance penalty* [1, 19-112].

Regarding the SMC classification of the machine clear, when the attacker evicts a cache line, it invalidates a cache line from the victim L1 instruction cache. This might be detected by the microprocessor as an SMC event.

The machine clears flush the pipeline, forcing the victim to re-fetch some instructions from memory, thus increasing the number of L1 cache misses due to the degrade process action. Therefore, it amplifies the effect produced by a cache miss, because sometimes the same instructions are fetched more than once.

Moreover, this analysis reveals an unknown performance degradation root cause of both DEGRADE and HYPER-

[1] after subtracting NODEGRADE cache misses to remove non-targeted code activity

DEGRADE, thus complementing the original research on DEGRADE in [7]. The performance degradation occurs due to an increased number of cache misses and due to increased machine clears, where the latter is evidenced by the significant increase from zero (NODEGRADE) to 28k (DEGRADE). Likewise, HYPERDEGRADE increases the number of cache misses and machine clears, thus, further amplifying the performance degradation produced by DEGRADE. This demonstrates that the topology of the microprocessor and the affinity of the degrade process have significant influence in the performance degradation impact, answering RQ 1.

We identified SMC machine clears as an additional root cause for both DEGRADE and HYPERDEGRADE, however, there could be others. In this regard, we highlight that our root cause analysis, albeit sound, is not complete. Moreover, achieving such completeness is challenging due to the undocumented nature of the microarchitecture, providing an interesting research direction for continued research. Indeed, in concurrent work, Ragab et al. [54] analyze machine clears in the context of transient execution.

**Contention test and pure SMC scenario.** For the sake of completeness, we compared the CPU cycles employed by different experiments using the previous setup. However, in this case, we vary the number of iterations in the tight loop over a single cache line. We ranged this value in the set $\{2^{16}, 2^{17}, .., 2^{25}\}$. Therefore, the number of executed instructions by the victim will be `victim_num_inst` $=$ $16 \times$ `num_iter`.

This comparison involves five experiments: one for each degrade strategy, plus a contention test and a pure SMC scenario (presented later). The contention test is equivalent to HYPERDEGRADE; however, this time the `clflush` instruction will flush a cache line *not used* by the victim. This test allows to evaluate the performance impact of co-locating a degrade process while it does not modify the victim's cache state.

Figure 3 visualizes the results of this comparison, where both axes are $\log_2$-scaled. The *y*-axis represents the ratio cycles per `victim_num_inst`. It can be appreciated that the contention test and NODEGRADE have very similar performance behavior (i.e., their curves overlap at the bottom). Hence, the performance degradation of a HYPERDEGRADE process will only be effective if it flushes a victim cache line, whereas additional resource contention related to executing the `clflush` instruction in a sibling core can be neglected.

We included a pure SMC scenario as an additional degrade strategy. Figure 4 illustrates the degrade process core. This code continuously triggers machine clears due to its self-modifying code behavior. Its position in Figure 3 shows it has about the same performance degrading power as DEGRADE. However, this pure SMC alternative does not depend on shared memory between the victim and degrade processes, thus the presence of—and finding—hot cachelines [7] is not a requirement.

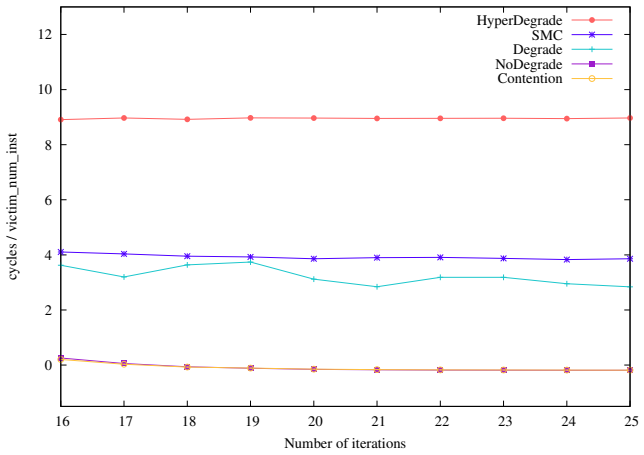


Figure 3: Experiments performance comparison ($\log_2$-scale).

```
    mov  $0x40, %dil     ; 0x40 is part of the opcode at L0
    lea  L0(%rip), %rcx  ; rcx points to L0
L0: mov  %dil, (%rcx)
    jmp  L0
```

Figure 4: Main SMC degrade process instructions.

**Limitations.** HYPERDEGRADE offers a significant slowdown wrt. previous performance degradation strategies. On the other hand, it is tightly coupled to SMT architectures because it requires physical core co-location with the victim process. Therefore, it is only applicable to microprocessors with this feature. In this regard, HYPERDEGRADE has the same limitation as previous works that exploit SMT [3, 6, 12, 27, 49, 62]. SCA attacks enabled by SMT often have no target shared library requirement, which is a hard requirement for FLUSH+RELOAD to move to cross-core application scenarios. For example, neither the L1 dcache spy [49] nor the L1 icache spy [3] require victims utilizing any form of shared memory on SMT architectures. Yet HYPERDEGRADE retains this shared library requirement, since our applet is based on `clflush` to induce the relevant microarchitecture events. However, since SMT is a common feature in modern microarchitectures and shared libraries are even more common, HYPERDEGRADE is another tool on the attacker's belt for performing FLUSH+RELOAD attacks.

## 4 HyperDegrade: Performance

With our HYPERDEGRADE applet from Section 3, the goal of this section is to evaluate the efficacy of HYPERDEGRADE as a technique to degrade the performance of victim applications that link against shared libraries. Section 5 will later explore the use of HYPERDEGRADE in SCA, but here we focus purely on the slowdown effect. Applied as such in isolation, HYPERDEGRADE is useful to effectively monopolize the CPU comparative to the victim, and also increase the CPU time billed to the victim for the same computations performed

Table 3: Various SMT architectures used in our experiments.

| Family | Model | Base Freq. | Cores / Threads | Details |
|---|---|---|---|---|
| Skylake | i7-6700 | 3.4 GHz | 4 / 8 | Ubuntu 18, 32 GB RAM |
| Kaby Lake | i7-7700HQ | 2.8 GHz | 4 / 8 | Ubuntu 20, 32 GB RAM |
| Coffee Lake | i7-9850H | 2.6 GHz | 6 / 12 | Ubuntu 18, 32 GB RAM |
| Whiskey Lake | i7-8665UE | 1.7 GHz | 4 / 8 | Ubuntu 20, 16 GB RAM |

by the victim.

Allan et al. [7, Sect. 4] use the SPEC 2006CPU benchmark suite, specifically 29 individual benchmark applications, to establish the efficacy of their DEGRADE technique as a performance degradation mechanism. In our work, we choose a different suite motivated from several directions.

First, unfortunately SPEC benchmarks are not free and open-source software (FOSS). In the interest of Open Science, we instead utilize the BEEBS benchmark suite by Pallister et al. [47, 48] which is freely available[2]. The original intention of BEEBS is microbenchmarking of typical embedded applications (sometimes representative) to facilitate device power consumption measurements. Nevertheless, it suits our purposes remarkably.

These 77 benchmark applications also differ in the fact that they are not built with debug symbols, which is required to apply the Allan et al. [7] methodology. While debug symbols themselves should not affect application performance, they often require less aggressive compiler optimizations that, in the end, result in less efficient binaries which might paint an unrealistic picture for performance degradation techniques outside of research environments.

We used the BEEBS benchmark suite off-the-shelf, with one minor modification. By default, BEEBS statically links the individual benchmark libraries whereas HYPERDEGRADE (and originally DEGRADE) target shared libraries. Hence, we added a new option to additionally compile each benchmark as a shared library and dynamically link the benchmark application against it.

## 4.1 Experiment

Before presenting and discussing the empirical results, we first describe our experiment environment. Since HYPERDEGRADE targets HT architectures specifically, we chose four consecutive chip generations, all featuring HT. Table 3 gives an overview, from older to younger models.

Our experiment consists of the following steps. We used the `perf` utility to definitively measure performance, including clock cycle count. In an initial profiling step, we exhaustively search (guided by `perf` metrics) for the most efficient cache line to target during eviction. We then run three different tests: a baseline NODEGRADE, classical DEGRADE, and our HYPERDEGRADE from Section 3. Each test that involves degradation profiles for the target cache line independently:

[2] https://github.com/mageec/beebs

Table 4: Statistics (aggregated from Table 8 and Table 9) for different performance degradation strategies targeting BEEBS shared library benchmarks, across architectures.

| Family | Method | Median | Min | Max | Mean | Stdev |
|---|---|---|---|---|---|---|
| Skylake | DEGRADE | 11.1 | 1.4 | 33.1 | 13.1 | 8.0 |
| Skylake | HYPERDEGRADE | 254.0 | 10.4 | 1101.9 | 306.3 | 226.7 |
| Kaby Lake | DEGRADE | 10.6 | 1.4 | 36.5 | 12.0 | 7.5 |
| Kaby Lake | HYPERDEGRADE | 266.4 | 10.2 | 1060.1 | 330.6 | 229.0 |
| Coffee Lake | DEGRADE | 12.2 | 1.5 | 39.0 | 14.0 | 7.9 |
| Coffee Lake | HYPERDEGRADE | 317.5 | 13.0 | 1143.7 | 382.5 | 246.9 |
| Whiskey Lake | DEGRADE | 12.5 | 1.5 | 43.9 | 14.4 | 9.2 |
| Whiskey Lake | HYPERDEGRADE | 364.3 | 13.5 | 1349.3 | 435.8 | 280.9 |

i.e. the target cache line for DEGRADE is perhaps not the same as HYPERDEGRADE. We then iterate each test to gather statistics, then repeat for all 77 BEEBS benchmarks, and furthermore across the four target architectures. We used the `taskset` utility to pin to separate physical cores in the DEGRADE case, and same physical core in the HYPERDEGRADE case.

## 4.2 Results

While Table 8 and Table 9 contain the full statistics per architecture, strategy, and BEEBS microbenchmark, Table 4 and Figure 5 provide high level overviews of the aggregate data. Table 4 shows the efficacy of HYPERDEGRADE over classical DEGRADE is striking, with median slowdown factors ranging from 254 to 364, and maximum slowdown factors ranging from 1060 to 1349. These maximum slowdowns are what our title alludes to—for example, in the Skylake i7-6700 case (maximum), reducing the 3.4 GHz base frequency to a 3.1 MHz effective frequency when observed from the victim application perspective.

Figure 5 visualizes the aggregate statistics from Table 8 and Table 9. Due to the magnitude of the slowdowns, the $x$-axis is logarithmic. Please note these data points are for identifying general trends; the location of individual points within separate distributions (i.e. different benchmarks) may vary.

Finally, Table 10 and Table 11 for the PARSEC [13] macrobenchmark suite are analogous to Table 8 and Table 9 for the BEEBS microbenchmarks. In this case, the slowdowns have a noticeably smaller magnitude. We attribute the difference to benchmarking goals. While BEEBS microbenchmarks are typically CPU-bound and capable of running on bare metal, that is not the case for PARSEC macrobenchmarks where the focus is parallelism. The combined results from the two benchmark suites demonstrate that while a typical binary will not experience a slowdown of three orders of magnitude, microbenchmarks with small, tight loops usually exhibit more significant slowdowns. This is convenient since the typical application of performance degradation mechanisms is in conjunction with side-channel attacks that target such hot spots.

In summary, the empirical data in this section validates the

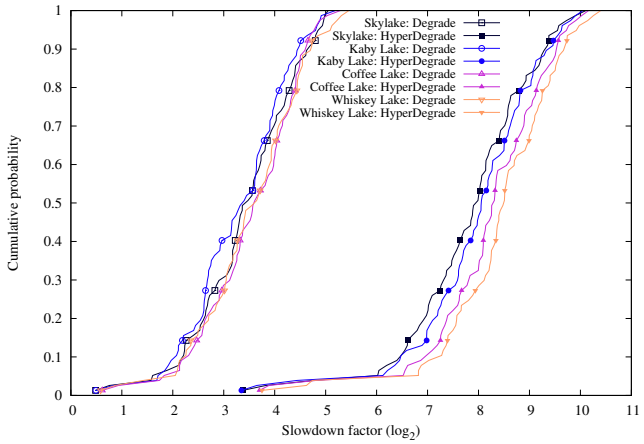


Figure 5: Distributions (computed from Table 8 and Table 9) for different performance degradation strategies targeting BEEBS shared library benchmarks, across architectures. Note the $x$-axis is logarithmic ($\log_2$).

HYPERDEGRADE concept and answers RQ 1 authoritatively. The data shows a clear advantage—even reaching three orders of magnitude in select microbenchmark cases—of HYPERDEGRADE over classical DEGRADE. Therefore, as a pure performance degradation mechanism, HYPERDEGRADE outperforms DEGRADE.

## 5 HyperDegrade: Assessment

Applying the HYPERDEGRADE concept from Section 3, Section 4 subsequently showed the efficacy of HYPERDEGRADE as a performance degradation technique. Similar to the classical DEGRADE technique, we see the main application of HYPERDEGRADE in the SCA area to improve the granularity of microarchitecture timing traces. That is the focus of this section.

We first enumerate some of the shortcomings in previous work on performance degradation. Allan et al. [7, Sect. 5] show that decreasing the FLUSH+RELOAD wait time—while indeed increasing granularity—generally leads to a higher number of missed accesses concerning the targeted line. This was in fact the main motivation for their DEGRADE technique. Applying DEGRADE [7, Sect. 7], the authors argue why missed accesses are detrimental to their end-to-end cryptanalytic attack. While the intuition for their argument is logical, the authors provide no evidence, empirical or otherwise, that DEGRADE actually leads to traces containing statistically more information, which is in fact the main purpose of performance degradation techniques. The motivation and intuition by Pereida García and Brumley [50] is similar—albeit with a different framework for target cache line identification—and equally lacks evidence.

The goal of this section is to answer RQ 2, rectifying these shortcomings inspired by information-theoretic methods. We do so by utilizing an established SCA metric to demonstrate that classical DEGRADE leads to statistically more leakage than FLUSH+RELOAD in isolation. Additionally, our HYPERDEGRADE technique further amplifies this leakage.

```
1200 <x64_victim_0>:                 1400 <x64_victim_1>:
1200: mov $CNT,%r10                  1400: mov $CNT,%r10
1207: add $0x1,%r10                  1407: add $0x1,%r10
120b: sub $0x1,%r10                  140b: sub $0x1,%r10
...                                  ...
12e7: add $0x1,%r10                  14e7: add $0x1,%r10
12eb: sub $0x1,%r10                  14eb: sub $0x1,%r10
12ef: sub $0x1,%r10                  14ef: sub $0x1,%r10
12f3: jnz 1207 <x64_victim_0+0x7>    14f3: jnz 1407 <x64_victim_1+0x7>
12f9: retq                           14f9: retq
```

Figure 6: Functions of a shared library (`objdump` view) used to construct an ideal victim for our SCA leakage assessment experiments.

### 5.1 Experiment

Figure 6 depicts the shared library we constructed to use throughout the experiments in this section. The code has two functions `x64_victim_0` and `x64_victim_1` that are essentially the same, but separated by 512 bytes. The functions set a counter (`r10`) from a constant (`CNT`, in this case 2k), then proceed through several effective nops (`add` and `sub` instructions that cancel), then finally decrement the counter and iterate.

We designed and implemented an ideal victim application linking against this shared library. The victim either makes two sequential `x64_victim_0` calls ("0-0") or `x64_victim_0` followed by `x64_victim_1` ("0-1"). We then used the stock FLUSH+RELOAD technique, probing the start of `x64_victim_0` (i.e. at hex offset `1200`).

Pinning the victim and spy to separate physical cores, we then procured 20k traces, in two sets of 10k for each of 0-0 and 0-1, and took the mean of the sets to arrive at the average trace. Figure 7 (Top) represents these two baseline FLUSH+RELOAD cases with NODEGRADE strategy as the two plots on the far left.

The next experiment was analogous, yet with the classical DEGRADE strategy. We degraded two cache lines—one in `x64_victim_0` and the `x64_victim_1`, both in the middle of their respective functions. These are the two middle plots in Figure 7 (Top). Here the victim, spy, and degrade processes are all pinned to different physical cores.

Our final experiment was analogous, yet with our novel HYPERDEGRADE strategy and pinning the victim and degrade processes to two logical cores of the same physical core—degrading the same two cache lines—and the spy to a different physical core. These are the two plots on the far right in Figure 7 (Top).

What can be appreciated in Figure 7 (Top), is that both performance degradation strategies are working as intended—they are stretching the traces. The remainder of this section focuses on quantifying this effect. In fact HYPERDEGRADE

Table 5: POI counts and ratios at various NICV thresholds across degrade strategies (see Figure 7). The ratios (x) are between the different strategies.

| Threshold | NoDegrade | Degrade | HyperDegrade |
|---|---|---|---|
| 0.1 | 233 | 1212 (5.2x) | 13151 (56.4x, 10.9x) |
| 0.2 | 188 | 1149 (6.1x) | 11664 (62.0x, 10.2x) |
| 0.3 | 167 | 1097 (6.6x) | 11159 (66.8x, 10.2x) |
| 0.4 | 147 | 1049 (7.1x) | 10194 (69.3x, 9.7x) |
| 0.5 | 117 | 969 (8.3x) | 6003 (51.3x, 6.2x) |

stretches the traces to such an extreme that the NoDegrade data on the far left is scantily discernible in this visualization.

## 5.2 Results

Recalling from Section 2.3, NICV suits particularly well for our purposes, since it is designed to work with only public data and is agnostic to leakage models [11]. The latter fact makes NICV pertinent as a metric to compare the quality of traces [11, Sect. 3]. The metric—in the interval $[0,1]$—is defined by

$$\mathrm{NICV}(X,Y) = \frac{\mathrm{Var}[\mathrm{E}[Y|X]]}{\mathrm{Var}[Y]} \tag{1}$$

with traces $Y$, classes $X$, and E the expectation (mean). The square root of the NICV metric, or the correlation ratio, is an upper bound for Pearson's correlation coefficient [53, Corollary 8]. Two classes (0-0 and 0-1) suffice for our purposes, simplifying Equation 1 as follows.

$$\mathrm{NICV}(X,Y) = \frac{(\mathrm{E}[Y|X=0] - \mathrm{E}[Y|X=1])^2}{4 \cdot \mathrm{Var}[Y]}$$

Figure 7 (Bottom) illustrates applying this metric to the two sets of measurements for each degrade strategy—baseline NoDegrade, Degrade, and HyperDegrade—and visualizing the square root, or maximum correlation. With simple thresholding to identify POIs (i.e., those points that exceed a fixed value), this leads to the POI statistics in Table 5. To give one extremely narrow interpretation, with `CNT` set to 2k in Figure 6, less than 2k POIs indicates information is being lost, i.e. the victim is running faster than the spy is capable of measuring. With this particular victim in this particular environment, it implies neither NoDegrade nor Degrade achieve sufficient trace granularity to avoid information loss, while HyperDegrade does so with ease.

In conclusion, this definitively answers RQ 2; the Degrade strategy leads to statistically more information leakage over stock Flush+Reload due to the significant POI increase. Similarly, it shows HyperDegrade leads to significantly more POIs compared to Degrade, but at the same time (on average) slightly lower maximum correlation for each POI.

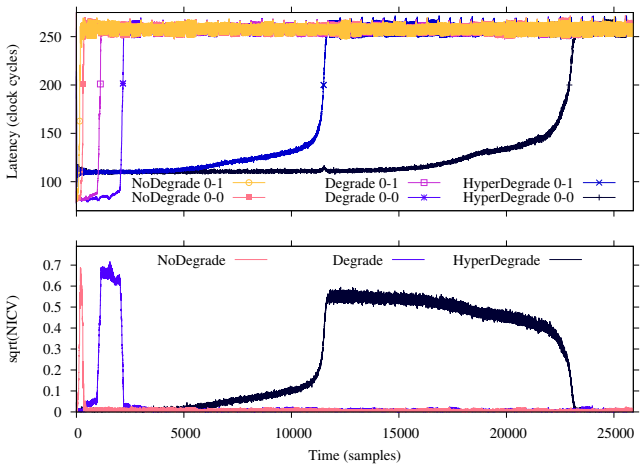


Figure 7: Top: averaged traces across different degrade strategies and different victim execution paths (i.e. classes, 0-0 and 0-1). The legend corresponds to the plots from left to right. Bottom: the NICV metric's square root, or maximum correlation. The legend again corresponds to the plots from left to right. The plots align and display the same time slice.

# 6 HyperDegrade: Exploitation

While Section 5 shows that HyperDegrade leads to more leakage due to the significant increase in POIs, the Figure 6 shared library and linking victim application are unquestionably purely synthetic. While this is ideal for leakage assessment, it does not represent the use of HyperDegrade in a real end-to-end SCA attack scenario. What remains is to demonstrate that HyperDegrade applies in end-to-end attack scenarios and that HyperDegrade has a quantifiable advantage over other degrade strategies wrt. attacker effort. That is the purpose of this section.

**The leak.** Recalling Section 2.4, the original Raccoon attack exploits the fact that Diffie-Hellman as used in TLS 1.2 and below dictates stripping leading zeros of the shared DH key during session key derivation. The authors note that *not* stripping is not foolproof can also lead to oracles [40, Sect. 3.5], pointing at an OpenSSL function that is potentially vulnerable to microarchitecture attacks [40, Appx. B]. They leave the investigation of said function—unrelated to TLS—as future work: a gap which this section fills.

Figure 8 shows that function, which is our target within the current (as of this writing) state-of-the-art OpenSSL 1.1.1h DH shared secret key derivation. The shared secret is computed at line 36; however, OpenSSL internals strip the leading zero bytes of this result. Therefore, at line 40 this function checks if the computed shared secret needs to be padded. Padding is needed if the number of bytes of the shared secret and the DH modulus differ.

**The leakage model.** Considering a theoretical leakage model, the binary result of the line 40 condition leaks whether the

```
33 int DH_compute_key_padded(unsigned char *key,
   ↪ const BIGNUM *pub_key, DH *dh)
34 {
35     int rv, pad;
36     rv = dh->meth->compute_key(key, pub_key, dh);
37     if (rv <= 0)
38         return rv;
39     pad = BN_num_bytes(dh->p) - rv;
40     if (pad > 0) {
41         memmove(key + pad, key, rv);
42         memset(key, 0, pad);
43     }
```

Figure 8: The target vulnerability in OpenSSL 1.1.1h Diffie-Hellman shared key derivation for our end-to-end attack.

shared secret has at least eight leading zero bits (branch taken) or not (branch not taken). This model—capable of extracting at most eight bits of information—affects the key sizes that are in scope. While 2048/256-bit DH parameters are more consistent with current key size recommendations, the original Raccoon attack [40, Table 3] is unable to target eight bits of leakage in this setting: the authors explicitly leave it as an open problem [40, Sect. 6.2]. They instead target legacy (1024/160-bit) or non-standard (1036/160-bit) DH parameters for an eight-bit leak. We follow suit, targeting legacy keys (see Section 6.1) for the exact same reasons.

**The victims.** Our next task was to identify callers to the Figure 8 code from the application and protocol levels, since it is unrelated to TLS. We successfully identified PKCS #7 (RFC 2315 [35]) and CMS (RFC 5652 [34]) as standards where Figure 8 might apply. We subsequently used the TriggerFlow tool [28] to verify that OpenSSL's `cms` and `smime` command line utilities have the Figure 8 function in their call stacks.

## 6.1 Attack Outline and Threat Model

In our end-to-end attack, all message encryptions and decryptions are with OpenSSL's command line `cms` utility. We furthermore assume Alice has a static DH public key in an X.509 certificate and, wlog., the DH parameters are the fixed 1024/160-bit variant from RFC 5114 [38]. OpenSSL supports these natively as named parameters, used implicitly. We carried out all experiments on the Coffee Lake machine from Table 3.

Our Raccoon attack variant consists of the following steps. (i) Obtain a target CMS-encrypted message from Bob to Alice. (ii) Based on the target, construct many chosen ciphertexts and submit them to Alice for decryption. (iii) Monitor Alice's decryptions of these ciphertexts with HyperDegrade and Flush+Reload to detect the key-dependent padding. (iv) Use the resulting information to construct a lattice problem and recover the original target session key between Bob and Alice, leading to loss of confidentiality for the target message. The original Raccoon attack [40] abstracts away most of these steps, using only simulated SCA data.

**Threat model.** Our attack makes several assumptions discussed below, which we borrow directly from the existing literature. (i) Our threat model assumes the attacker is able to co-locate on the same system with Alice (victim), and furthermore execute on the same logical and physical cores in parallel to Alice. See the end of Section 3 for a discussion of this standard assumption. (ii) We also assume that Alice decrypts messages non-interactively, due to the number of queries required. This is a fair assumption not only because DH is literally Non-Interactive Key Exchange (NIKE) [22] from the theory perspective, but also because CMS (the evolution of PKCS #7) has ubiquitous use cases, e.g. including S/MIME. Chosen ciphertext decryptions is a standard assumption from the applied SCA literature [23, Sect. 1.1] [24, Sect. 1.4] [55, Sect. 3]. (iii) We assume the attacker is able to observe one encrypted message from Bob to Alice. This is a passive variant of the standard Dolev-Yao adversary [21] that is Man-in-the-Middle (MitM) capable of eavesdropping, and the exact same assumption from the original Raccoon attack [40, Fig. 1]. Ronen et al. [55, Sect. 3] call this *privileged network position* since it is a weak assumption compared to full MitM capabilities. To summarize, the overall threat model used by Ronen et al. [55, Sect. 3] is extremely similar to ours and encompasses all of the above assumptions. The only slight difference is a stronger notion of co-location in our case—from same CPU to same physical core.

**Case study: triggering oracle decryptions.** We briefly explored the non-interactive requirement discussed above. Specifically, two arenas: automated email decryption, and automated decryption of certificate-related messages.

Recent changes in Thunderbird (v78+) migrate from the Enigmail plugin to native support for email encryption and/or authentication (PGP, S/MIME). Automated, non-interactive decryption for various purposes (e.g., filtering) appears to be a non-default (yet supported) option.[3] Quoting from that thread: "A lot of companies e.g. in the finance sector decrypt the messages at a central gateway and then forward them internally to the respective recipient."

We also found explicit code meeting our non-interactive requirement in the realm of automated certificate management. (i) The Simple Certificate Enrollment Protocol (SCEP, RFC 8894 [32]) supports exchanging (public key encrypted, CMS formatted) confidential messages over an insecure channel, such as HTTP or generally out-of-band. This is in contrast to the Automatic Certificate Management Environment (ACME) protocol (RFC 8555 [9], e.g., Let's Encrypt [2]), which relies on the confidentiality and authenticity guarantees of TLS. The open source[4] Apache module `mod_scep` dynamically links against OpenSSL to provide this functionality. (ii) The Certificate Management Protocol (CMP, RFC 4210 [41]) provides

[3] https://bugzilla.mozilla.org/show_bug.cgi?id=1644085
[4] https://redwax.eu/rs/docs/latest/mod/mod_scep.html

similar functionality (i.e., public key encrypted, CMS formatted messages) with similar motivations (automated certificate management over insecure channels). Yet the implementation integrated into upcoming OpenSSL 3.0 does not currently support encrypted protocol messages.[5]

## 6.2 Degrade Strategies Compared

This section aims at answering RQ 3 by means of comparing three performance degradation strategies (NoDegrade, Degrade, HyperDegrade) when paired with a Flush+Reload attack to exploit this vulnerability. We reuse the following setup and adversary plan later during the end-to-end attack (Section 6.3).

**Experiment.** We monitor the cache line corresponding to the `memmove` function call and its surrounding instructions, i.e. near line 41 of Figure 8. If `memmove` is executed, at least two cache hits should be observed: (i) when the function is called, (ii) then when the function finishes (`ret` instruction). Therefore, if two cache hits are observed in a trace *close* to each other, that would mean the shared secret was padded, and in contrast a single cache hit only detects flow surrounding line 41 of Figure 8.

We select the first cache line where the function `memmove` is located as the degrading cache line. It is the stock, unmodified, uninstrumented `memmove` available system-wide as part of the shared C standard library `libc`. Degrading during `memmove` execution should increase the time window the spy process has to detect the second cache hit (i.e., increase time granularity).

We strive for a *fair* comparison between the three degradation strategies during a Flush+Reload attack. It is challenging to develop an optimal attack for each degradation strategy and even harder to maintain fairness. Therefore, we developed a single attack plan and swept its parameters in order to provide a meaningful and objective comparison.

Table 6 summarizes the attack parameters and the explored search space. The first parameter, $r$, affects trace capturing—it specifies the number of iterations the Flush+Reload wait loop should iterate. The remaining parameters belong to the trace processing tooling. The second parameter, $t$, refers to the threshold in CPU clock cycles used to distinguish a cache hit from a miss. After some manual trace inspection, we observed this threshold varies between degradation strategy and Flush+Reload wait time; we decided to add it to the search space. The last parameter, $d$, specifies the distance (in number of Flush+Reload samples) between two cache hits to consider them as close.

We explore this parameter search space, and for each parameter set—i.e., triplet $(r,t,d)$—evaluate the attack performance, estimating the true positive (TP) and false positive rates (FP). For this task, we generated two pairs of DH keys (i.e., attacker and victim). We selected one of these pairs such that the shared secret needs padding after a DH key exchange, while for the other it does not.

Table 6: Attack parameters search space.

| Parameter | Range |
|---|---|
| Flush+Reload wait time ($r$) | $\{128, 256\}$ |
| Cache hit/miss threshold ($t$) | $\{50, 100, 150, 200\}$ |
| Cache hits closeness distance ($d$) | $\{1, 5, 10, \ldots, 95\}$ |

Table 7: Best results for degrade strategies.

| Strategy | Trace count | Param. set $(r,d,t)$ |
|---|---|---|
| NoDegrade | 651510 | $(128, 1, 100)$ |
| Degrade | 181189 | $(256, 1, 170)$ |
| HyperDegrade | 53721 | $(256, 1, 170)$ |

We then captured 1k traces for each key pair, parameter set, and degradation strategy under consideration and estimated the TP and FP rates. We are interested in finding which parameter sets lead to more efficient attacks in terms of number of traces to capture, i.e. number of attacker queries. Therefore, we focused on those results with zero false positives for the comparison, thus it is a best case analysis for all degradation strategies.

**Results.** For 1024-bit DH, the lattice-based cryptanalysis requires 173 samples where padding occurred (explained later). Therefore, the following equation defines the average number of traces that need to be captured, where $\Pr[\text{pad}] = 1/177 \approx 0.00565$ with the fixed RFC 5114 [38] parameters.

$$\text{num\_traces} = 173/(\Pr[\text{TP}] \cdot \Pr[\text{pad}])$$

Considering the very low probability the leakage occurs and the increased complexity of lattice-based cryptanalysis in the presence of errors (see [4, Sect. 6]), reducing the number of traces is important for attack effectiveness.

Table 7 shows the best parameter set results for each degrade strategy that could lead to a successful attack. Note that HyperDegrade clearly reduced the number of required traces to succeed by at least a factor of 3.3 when compared with Degrade (the second best performer). This translates into a considerable reduction in the number of traces: from 181k to 53k. Moreover, Figure 9 shows there is not just a single parameter set where HyperDegrade performs better than Degrade, but rather there are 88 of them. These results provide evidence that HyperDegrade can perform better than the other two degrade strategies for mounting Flush+Reload attacks on cryptography applications, answering RQ 3.

[5] https://www.openssl.org/docs/manmaster/man1/openssl-cmp.html

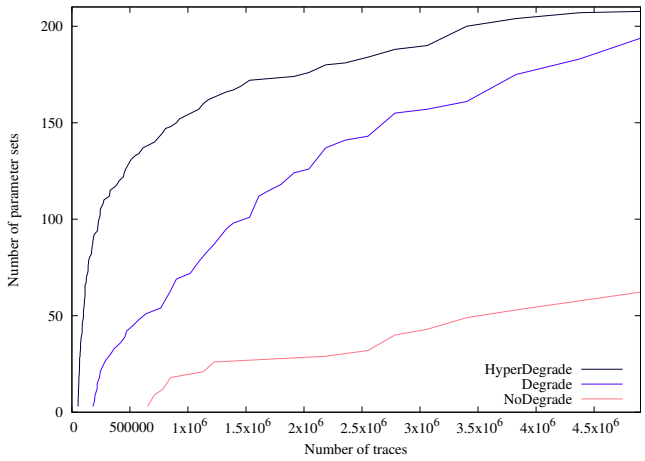


Figure 9: Degrade strategies comparison: how many parameter sets can be used to mount an attack using $x$-number of traces.

## 6.3 End-to-End Attack Instance

The remainder of this section answers RQ 4. We begin with lattice details, then finish with the results of our end-to-end attack.

**Lattice construction.** Alice's public key $g^a$ is readily available and the attacker observes $g^b$ from the original target query, along with ciphertext encrypted under the shared session key $g^{ab}$ (private). Then the attacker proceeds with chosen queries, crafting ciphertext $g^b g^{r_i}$ and random $r_i$ for submitting to Alice for decryption. Alice then computes $(g^b g^{r_i})^a = g^{ab} \cdot (g^a)^{r_i}$ with the attacker measuring if padding occurs. This is an instance of the hidden number problem (HNP) by Boneh and Venkatesan [14]—to recover $\alpha = g^{ab}$ given many $t_i = (g^a)^{r_i}$.

We use the lattice construction by Nguyen and Shparlinski [43] verbatim, stated here for completeness. Restricting to the $t_i$ where padding occurred, our SCA data tells us $0 < \alpha t_i < p/2^\ell$, where we set $\ell = 8$ due to the nature of this particular side channel; recall "branch taken" in Figure 8 says at least the top eight bits are clear. Denoting $u_i = p/2^{\ell+1}$ yields $v_i = |\alpha t_i - u_i|_p \leq p/2^{\ell+1}$ where $|x|_p$ is signed modulo $p$ reduction centered around zero. Then there are integers $\lambda_i$ where $\mathrm{abs}(\alpha t_i - u_i - \lambda_i) \leq p/2^{\ell+1}$ holds, and this is the key observation for lattice attacks; the $u_i$ approximate $\alpha t_i$ since they are closer than a random integer modulo $p$. Consider the rational $d+1$-dimension lattice generated by the rows of the following matrix.

$$B = \begin{bmatrix} 2Wp & 0 & \dots & \dots & 0 \\ 0 & 2Wp & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & 2Wp & 0 \\ 2Wt_1 & \dots & \dots & 2Wt_d & 1 \end{bmatrix}$$

When we set $W = 2^\ell$, $\vec{x} = (\lambda_1, \dots, \lambda_d, \alpha)$, $\vec{y} = (2Wv_1, \dots, 2Wv_d, \alpha)$, and $\vec{u} = (2Wu_1, \dots, 2Wu_d, 0)$ we get the linear relationship $\vec{x}B - \vec{u} = \vec{y}$. Solving the Closest Vector Problem (CVP) with inputs $B$ and $\vec{u}$ yields $\vec{x}$, and hence the target session key $\alpha$. We also use the traditional CVP-to-SVP (Shortest Vector Problem) embedding by Goldreich et al. [25, Sec. 3.4]. Pereida García et al. [52] suggest weighting on the average logarithm, hence we set $W = 2^{\ell+1}$ in our $\ell = 8$ scenario. Lastly, to set the lattice dimension we use the heuristic from [59, Sect. 9.1] verbatim, which is $d = 1 - e^{-c}$. With their suggested confidence factor $c = 1.35$ (to improve key bit independence), in our case it leads to $d = 173$; this explains the constant from Section 6.2. We use the same BKZ block size parameter as [40, Table 3], $\beta = 60$.

**Results.** Following the results of Section 6.2, we proceeded to capture 60k traces using HYPERDEGRADE and the parameter set shown in Table 7. Our capture tooling implements precisely step (i) to (iii) in Section 6.1, obtaining a target ciphertext, constructing chosen ciphertexts, and querying the oracle. In the end, at each capture iteration the attacker only needs to modify a public key field in an ASN.1 structure to produce a new chosen ciphertext, then take the measurements while Alice performs the decryption.

Considering the padding probability $\Pr[\text{pad}] = 1/177$, the expected number of padded traces in the 60k set is 339. After processing each trace, our tooling detected 3611 paddings. It indicates that, with high probability, these also contain false positives. Albrecht and Heninger [4] focus on lattice-based cryptanalysis of ECDSA and suggest adjusting lattice parameters at the cost of increased computation to compensate for errors. We instead use a different approach in the DH setting—not applicable in the ECDSA setting due to its usage of nonces—to counteract those false positives.

To reduce the FP rate, for each trace where we detected padding, we retry the query seven times and majority vote the result. From the 3611 traces detected as padded, only 239 passed the majority voting. Therefore, the total number of traces captured was $60\text{k} + 7 \cdot 3611 = 85277$.

Even with the majority voting, some false positives could remain; we sorted the 239 samples by the vote count. Then we selected the highest ranked 173 samples to build the HNP instances. For the sake of completeness, we verified there were 47 false positives in the 239 set; however, all had the lowest vote count of four.

We implemented our lattice using BKZ reduction from `fpylll`[6], a Python wrapper for the `fplll` C++ library [20]. We constructed 24 lattice instances from our SCA data, and executed these in parallel on a 2.1 GHz dual CPU Intel Xeon Silver 4116 (24 cores, 48 threads across 2 CPUs) running Ubuntu 20 with 256 GB memory. The first instance to recover the session key did so in one hour and five minutes with a

[6] https://github.com/fplll/fpylll

single BKZ reduction. With no abstractions, utilizing real trace data at the application level, and real protocol messages, our end-to-end attack resolves RQ 4.

**OpenSSL disclosure.** We contacted the OpenSSL security team to disclose our results regarding the exploitability of this leak. We also designed, implemented and tested a fix[7] that avoids executing the leaky branch. We achieved this by changing the default behavior of the `dh->meth->compute_key` function pointer to always return a fixed-length array (i.e., the public byte length of $p$) in constant time, ensuring variable `pad` is zero in Figure 8 (i.e., `BN_num_bytes(dh->p)` and `rv` are equal). Retaining the branch and not simply removing it is due to backwards compatibility issues with OpenSSL engines [57]. OpenSSL merged our fix on 10 January 2021, included as of version 1.1.1j.

## 7 Conclusion

HYPERDEGRADE increases performance degradation with respect to state-of-the-art. The difference depends on the targeted process, but we achieved slowdown factors up to three orders of magnitude in select microbenchmark cases. In addition to increased cache misses, we discovered the cache-based performance degradation root cause is due to the increased number of machine clears produced by the processor detecting a cache line flush from L1 as self-modifying code (RQ 1). We analyzed the impact of DEGRADE and HYPERDEGRADE on FLUSH+RELOAD traces from a theoretical point of view using leakage assessment tools, demonstrating that HYPERDEGRADE tremendously increases the number of POIs, which reflects in an increased time granularity (RQ 2). From an applied perspective, we designed a fair experiment that compares the three degrade strategies NODEGRADE, DEGRADE, and HYPERDEGRADE when coupled with a FLUSH+RELOAD attack wrt. the number of traces needed to recover a secret from a cryptography implementation (RQ 3). Our resulting data demonstrates the benefits of HYPERDEGRADE, requiring three times less traces and attacker queries to succeed, the latter being the standard metric in applied SCA literature. Regarding cryptography, we answered an open problem from the recently published Raccoon attack, providing experimental evidence that such an attack applies with real data (RQ 4).

**Availability.** In support of Open Science and to ensure reproducibility, we released our tooling for the Section 4 BEEBS experiment as a freely available research artifact [5] for the benefit of the community. The artifact can be used to reproduce Table 8 and Table 9.

**Future work.** Our work either reinforces or illuminates several new avenues for continued related research.

In Section 6.2, we noted how the cache hit threshold varies depending on various spy parameters. We have also noted this behavior in other FLUSH+RELOAD scenarios, outside this work. It would definitely be an interesting future research line to investigate its root cause.

In general, our off-the-shelf applied lattice techniques in Section 6.3, while serving their purpose for proof-of-concept, are likely not optimal. Fundamental lattice-based cryptanalysis improvements (e.g., the recent [4]) are beyond the scope of our work, but could reduce dimension and subsequently attacker queries. Similar to our Raccoon variant, the original Raccoon attack [40] is unable to target 2048/256-bit DH with eight bits or less of leakage. The authors leave this as an open problem, and we concur; indeed, improved lattice methods to compensate for these significantly larger finite field elements is an interesting research direction.

Several previous studies gather widespread certificate and key usage statistics. For example, the original Raccoon attack authors gather statistics for static DH keys in X.509 certificates for TLS 1.2 (and lower) authentication, and/or ephemeral-static DH keys in TLS 1.2 (and lower) cipher suites [40, Sect. 7] from public services. Bos et al. [15] gather publicly-available elliptic curve keys from protocols and services such as TLS, SSH, BitCoin, and the Austrian e-identity card; Valenta et al. [60] consider IPSec, as well. Not specific to any particular public key cryptosystem, Lenstra et al. [37] gather publicly-available PGP keys and X.509 certificates for TLS 1.2 (and lower) authentication. Along these lines, although it is beyond the scope of our work, we call for future studies that gather and share S/MIME key usage statistics, paying particular attention to legal issues and privacy since, different from PGP, we are not aware of any general public (distributed) repositories for S/MIME keys.

**Acknowledgments.** 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). Supported in part by CSIC's i-LINK+ 2019 "Advancing in cybersecurity technologies" (Ref. LINKA20216).

## References

[1] Intel 64 and IA-32 architectures software developers manual. Volume 3B 325462-073US, Intel, November 2020. URL https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf.

[2] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J. Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, Seth D. Schoen, and Brad Warren. Let's Encrypt: An automated certificate authority to encrypt the entire web. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK,*

[7] https://github.com/openssl/openssl/pull/13772

Table 8: BEEBS performance degradation results (cycles, thousands) on Skylake and Kaby Lake.

| | Skylake | | | | | Kaby Lake | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | NoDegrade | Degrade | | HyperDegrade | | NoDegrade | Degrade | | HyperDegrade | |
| aha-compress | 70583 | 848723 | (12.0x) | 16222580 | (229.8x) | 69754 | 697353 | (10.0x) | 18585074 | (266.4x) |
| aha-mont64 | 21954 | 463471 | (21.1x) | 12066723 | (549.6x) | 22148 | 356097 | (16.1x) | 11607976 | (524.1x) |
| bs | 2032 | 8991 | (4.4x) | 171277 | (84.3x) | 2180 | 7137 | (3.3x) | 152106 | (69.8x) |
| bubblesort | 332386 | 9244976 | (27.8x) | 249121317 | (749.5x) | 305248 | 9197521 | (30.1x) | 229279506 | (751.1x) |
| cnt | 13237 | 191420 | (14.5x) | 4482733 | (338.6x) | 13488 | 187061 | (13.9x) | 5202446 | (385.7x) |
| compress | 8878 | 104299 | (11.7x) | 3248479 | (365.9x) | 9001 | 109617 | (12.2x) | 3767946 | (418.6x) |
| cover | 6982 | 172070 | (24.6x) | 2777996 | (397.9x) | 7165 | 95582 | (13.3x) | 2592937 | (361.9x) |
| crc | 7671 | 139690 | (18.2x) | 3745933 | (488.3x) | 7839 | 152098 | (19.4x) | 3516855 | (448.6x) |
| crc32 | 46875 | 1458728 | (31.1x) | 31199559 | (665.6x) | 47004 | 1716899 | (36.5x) | 33339067 | (709.3x) |
| ctl-stack | 37481 | 528762 | (14.1x) | 11150759 | (297.5x) | 38447 | 577541 | (15.0x) | 15785236 | (410.6x) |
| ctl-string | 31088 | 981144 | (31.6x) | 14581397 | (469.0x) | 32189 | 799759 | (24.8x) | 17316754 | (538.0x) |
| ctl-vector | 30742 | 367393 | (12.0x) | 7275579 | (236.7x) | 31266 | 372204 | (11.9x) | 8280240 | (264.8x) |
| cubic | 33333 | 201498 | (6.0x) | 3240751 | (97.2x) | 30686 | 184586 | (6.0x) | 4497482 | (146.6x) |
| dijkstra | 1965916 | 39394667 | (20.0x) | 962126944 | (489.4x) | 1979452 | 36038128 | (18.2x) | 1032841427 | (521.8x) |
| dtoa | 13236 | 76977 | (5.8x) | 1374173 | (103.8x) | 13422 | 77959 | (5.8x) | 1838910 | (137.0x) |
| duff | 6332 | 60486 | (9.6x) | 2488543 | (393.0x) | 6448 | 56663 | (8.8x) | 1876609 | (291.0x) |
| edn | 192602 | 4037466 | (21.0x) | 65758984 | (341.4x) | 191705 | 2978126 | (15.5x) | 86795945 | (452.8x) |
| expint | 29724 | 395738 | (13.3x) | 4513790 | (151.9x) | 30193 | 377277 | (12.5x) | 5138636 | (170.2x) |
| fac | 4595 | 64388 | (14.0x) | 1719178 | (374.1x) | 4761 | 60222 | (12.6x) | 1815137 | (381.2x) |
| fasta | 2218271 | 28380647 | (12.8x) | 628593367 | (283.4x) | 2216456 | 33215414 | (15.0x) | 644852892 | (290.9x) |
| fdct | 7759 | 23118 | (3.0x) | 935889 | (120.6x) | 7863 | 33953 | (4.3x) | 993673 | (126.4x) |
| fibcall | 2889 | 19083 | (6.6x) | 751337 | (260.0x) | 3054 | 19906 | (6.5x) | 824186 | (269.8x) |
| fir | 764841 | 18665980 | (24.4x) | 731941919 | (957.0x) | 764893 | 17441064 | (22.8x) | 685505153 | (896.2x) |
| frac | 12426 | 138238 | (11.1x) | 3326194 | (267.7x) | 12614 | 153789 | (12.2x) | 3927731 | (311.4x) |
| huffbench | 1400373 | 12636520 | (9.0x) | 185693684 | (132.6x) | 1424397 | 10053959 | (7.1x) | 187339402 | (131.5x) |
| insertsort | 4381 | 76538 | (17.5x) | 1937214 | (442.1x) | 4518 | 82371 | (18.2x) | 2423159 | (536.3x) |
| janne_complex | 2443 | 17672 | (7.2x) | 387808 | (158.7x) | 2589 | 15907 | (6.1x) | 403711 | (155.9x) |
| jfdctint | 11742 | 117469 | (10.0x) | 2469100 | (210.3x) | 11917 | 106261 | (8.9x) | 3063148 | (257.0x) |
| lcdnum | 2247 | 20278 | (9.0x) | 283506 | (126.1x) | 2419 | 14790 | (6.1x) | 357861 | (147.9x) |
| levenshtein | 151336 | 3413532 | (22.6x) | 94467885 | (624.2x) | 148115 | 3324556 | (22.4x) | 92479156 | (624.4x) |
| ludcmp | 8941 | 86599 | (9.7x) | 2064081 | (230.8x) | 9190 | 71764 | (7.8x) | 2182355 | (237.5x) |
| matmult-float | 67852 | 1760965 | (26.0x) | 45235894 | (666.7x) | 68377 | 1481444 | (21.7x) | 53988799 | (789.6x) |
| matmult-int | 438567 | 13438448 | (30.6x) | 371621846 | (847.4x) | 444120 | 9045445 | (20.4x) | 424979930 | (956.9x) |
| mergesort | 519862 | 9598330 | (18.5x) | 179589127 | (345.5x) | 517294 | 8497198 | (16.4x) | 207490953 | (401.1x) |
| miniz | 3405 | 17987 | (5.3x) | 224863 | (66.0x) | 3547 | 12738 | (3.6x) | 290558 | (81.9x) |
| minver | 6391 | 53104 | (8.3x) | 1275079 | (199.5x) | 6824 | 44293 | (6.5x) | 1331331 | (195.1x) |
| nbody | 250992 | 5342357 | (21.3x) | 164185274 | (654.1x) | 253584 | 5439927 | (21.5x) | 162955284 | (642.6x) |
| ndes | 113555 | 1740050 | (15.3x) | 34799379 | (306.5x) | 119918 | 1563197 | (13.0x) | 53176247 | (443.4x) |
| nettle-aes | 113306 | 489386 | (4.3x) | 14676852 | (129.5x) | 113123 | 488887 | (4.3x) | 21739421 | (192.2x) |
| nettle-arcfour | 87327 | 1784265 | (20.4x) | 22378097 | (256.3x) | 87136 | 1515589 | (17.4x) | 25846101 | (296.6x) |
| nettle-cast128 | 13957 | 23609 | (1.7x) | 198077 | (14.2x) | 13263 | 23692 | (1.8x) | 166519 | (12.6x) |
| nettle-des | 9179 | 27907 | (3.0x) | 250080 | (27.2x) | 9336 | 30056 | (3.2x) | 202710 | (21.7x) |
| nettle-md5 | 7482 | 35234 | (4.7x) | 549028 | (73.4x) | 7604 | 29737 | (3.9x) | 732806 | (96.4x) |
| nettle-sha256 | 14957 | 55160 | (3.7x) | 1459782 | (97.6x) | 15014 | 56483 | (3.8x) | 1314044 | (87.5x) |
| newlib-exp | 3667 | 23027 | (6.3x) | 344297 | (93.9x) | 3815 | 23708 | (6.2x) | 484350 | (126.9x) |
| newlib-log | 3176 | 14916 | (4.7x) | 352798 | (111.1x) | 3304 | 15313 | (4.6x) | 417836 | (126.4x) |
| newlib-mod | 2280 | 10949 | (4.8x) | 203870 | (89.4x) | 2486 | 11338 | (4.6x) | 221979 | (89.3x) |
| newlib-sqrt | 9448 | 140972 | (14.9x) | 5497784 | (581.8x) | 9608 | 143752 | (15.0x) | 3787532 | (394.2x) |
| ns | 27810 | 920498 | (33.1x) | 30642549 | (1101.9x) | 24270 | 718504 | (29.6x) | 25730490 | (1060.1x) |
| nsichneu | 12465 | 17453 | (1.4x) | 129175 | (10.4x) | 12680 | 17799 | (1.4x) | 128992 | (10.2x) |
| prime | 58915 | 934324 | (15.9x) | 11517642 | (195.5x) | 57920 | 922405 | (15.9x) | 14618565 | (252.4x) |
| qsort | 3868 | 34437 | (8.9x) | 636532 | (164.5x) | 4014 | 26922 | (6.7x) | 831894 | (207.2x) |
| qurt | 5456 | 56121 | (10.3x) | 972214 | (178.2x) | 5588 | 59091 | (10.6x) | 1089667 | (195.0x) |
| recursion | 7983 | 149886 | (18.8x) | 4806422 | (602.0x) | 7365 | 140916 | (19.1x) | 5193849 | (705.1x) |
| rijndael | 1831062 | 11743208 | (6.4x) | 235853943 | (128.8x) | 1830953 | 9728816 | (5.3x) | 280555490 | (153.2x) |
| select | 2499 | 18750 | (7.5x) | 286758 | (114.7x) | 2615 | 15980 | (6.1x) | 310185 | (118.6x) |
| sglib-arraybinsearch | 32695 | 941448 | (28.8x) | 24948746 | (763.1x) | 32073 | 913077 | (28.5x) | 25650916 | (799.7x) |
| sglib-arrayheapsort | 73813 | 964227 | (13.1x) | 28155013 | (381.4x) | 74316 | 1056981 | (14.2x) | 27101576 | (364.7x) |
| sglib-arrayquicksort | 35820 | 584648 | (16.3x) | 21360149 | (596.3x) | 35771 | 445614 | (12.5x) | 19761832 | (552.4x) |
| sglib-dllist | 103228 | 931802 | (9.0x) | 19577829 | (189.7x) | 103490 | 784713 | (7.6x) | 21655708 | (209.3x) |
| sglib-hashtable | 73515 | 720144 | (9.8x) | 13448462 | (182.9x) | 75750 | 672251 | (8.9x) | 16372567 | (216.1x) |
| sglib-listinsertsort | 148962 | 4155691 | (27.9x) | 57359384 | (385.1x) | 147850 | 4116964 | (27.8x) | 82759796 | (559.8x) |
| sglib-listsort | 82649 | 851066 | (10.3x) | 21974191 | (265.9x) | 83363 | 786974 | (9.4x) | 22087737 | (265.0x) |
| sglib-queue | 79274 | 976908 | (12.3x) | 20131755 | (254.0x) | 79976 | 999905 | (12.5x) | 28994719 | (362.5x) |
| sglib-rbtree | 194861 | 1795057 | (9.2x) | 39956917 | (205.1x) | 198968 | 1518989 | (7.6x) | 47673755 | (239.6x) |
| slre | 81356 | 961313 | (11.8x) | 22990277 | (282.6x) | 81068 | 885753 | (10.9x) | 27643530 | (341.0x) |
| sqrt | 278604 | 5427855 | (19.5x) | 66636453 | (239.2x) | 275896 | 4302655 | (15.6x) | 78476306 | (284.4x) |
| st | 46255 | 758154 | (16.4x) | 13044156 | (282.0x) | 45667 | 563498 | (12.3x) | 14099803 | (308.7x) |
| statemate | 5587 | 36137 | (6.5x) | 989900 | (177.2x) | 5768 | 36597 | (6.3x) | 1074219 | (186.2x) |
| stb_perlin | 170355 | 2261082 | (13.3x) | 64860091 | (380.7x) | 141765 | 2404186 | (17.0x) | 63274557 | (446.3x) |
| stringsearch1 | 15819 | 91324 | (5.8x) | 4618112 | (291.9x) | 16061 | 102834 | (6.4x) | 4935053 | (307.3x) |
| strstr | 5200 | 48645 | (9.4x) | 1224891 | (235.5x) | 5354 | 51939 | (9.7x) | 1345225 | (251.2x) |
| tarai | 2884 | 28585 | (9.9x) | 754962 | (261.7x) | 2965 | 18523 | (6.2x) | 682388 | (230.1x) |
| trio-snprintf | 16952 | 78512 | (4.6x) | 1100793 | (64.9x) | 17144 | 72171 | (4.2x) | 1273947 | (74.3x) |
| trio-sscanf | 21499 | 152353 | (7.1x) | 2933134 | (136.4x) | 22001 | 120026 | (5.5x) | 3577032 | (162.6x) |
| ud | 11215 | 69845 | (6.2x) | 1724508 | (153.8x) | 11106 | 75100 | (6.8x) | 2196806 | (197.8x) |
| whetstone | 335501 | 2891946 | (8.6x) | 55686230 | (166.0x) | 301591 | 2700492 | (9.0x) | 61531596 | (204.0x) |

Table 9: BEEBS performance degradation results (cycles, thousands) on Coffee Lake and Whiskey Lake.

| | Coffee Lake | | | | | Whiskey Lake | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | NoDegrade | Degrade | | HyperDegrade | | NoDegrade | Degrade | | HyperDegrade | |
| aha-compress | 71096 | 813585 | (11.4x) | 21102464 | (296.8x) | 69668 | 816475 | (11.7x) | 23975673 | (344.1x) |
| aha-mont64 | 21931 | 468459 | (21.4x) | 12821078 | (584.6x) | 21687 | 492008 | (22.7x) | 15890278 | (732.7x) |
| bs | 1874 | 9676 | (5.2x) | 183221 | (97.8x) | 1763 | 8435 | (4.8x) | 207217 | (117.5x) |
| bubblesort | 335556 | 9468033 | (28.2x) | 322437859 | (960.9x) | 302489 | 11286756 | (37.3x) | 369824760 | (1222.6x) |
| cnt | 13046 | 231693 | (17.8x) | 5177506 | (396.8x) | 13030 | 240288 | (18.4x) | 6270126 | (481.2x) |
| compress | 8728 | 115043 | (13.2x) | 4971456 | (569.6x) | 8552 | 115566 | (13.5x) | 5197671 | (607.7x) |
| cover | 6779 | 115233 | (17.0x) | 3332393 | (491.6x) | 7151 | 118538 | (16.6x) | 2716975 | (379.9x) |
| crc | 7526 | 160400 | (21.3x) | 4230637 | (562.1x) | 7601 | 177394 | (23.3x) | 3880102 | (510.4x) |
| crc32 | 47198 | 1842254 | (39.0x) | 36548242 | (774.4x) | 46386 | 2035556 | (43.9x) | 37694482 | (812.6x) |
| ctl-stack | 37440 | 616068 | (16.5x) | 14702668 | (392.7x) | 37847 | 725898 | (19.2x) | 21964505 | (580.3x) |
| ctl-string | 31313 | 853770 | (27.3x) | 18846973 | (601.9x) | 31845 | 704006 | (22.1x) | 19457277 | (611.0x) |
| ctl-vector | 30486 | 436212 | (14.3x) | 9325473 | (305.9x) | 30914 | 442518 | (14.3x) | 10432897 | (337.5x) |
| cubic | 33872 | 209368 | (6.2x) | 9250622 | (273.1x) | 30295 | 215268 | (7.1x) | 9827263 | (324.4x) |
| dijkstra | 1970036 | 44216908 | (22.4x) | 1253757348 | (636.4x) | 1961091 | 42750800 | (21.8x) | 1470626853 | (749.9x) |
| dtoa | 13163 | 88265 | (6.7x) | 2015472 | (153.1x) | 13020 | 78937 | (6.1x) | 2348685 | (180.4x) |
| duff | 6198 | 73171 | (11.8x) | 2860949 | (461.6x) | 6043 | 63287 | (10.5x) | 2248195 | (372.0x) |
| edn | 194838 | 3877224 | (19.9x) | 86990331 | (446.5x) | 196602 | 4120570 | (21.0x) | 104584239 | (532.0x) |
| expint | 29623 | 468879 | (15.8x) | 4908536 | (165.7x) | 29713 | 453217 | (15.3x) | 5642536 | (189.9x) |
| fac | 4334 | 77443 | (17.9x) | 1858161 | (428.7x) | 4226 | 62654 | (14.8x) | 2150388 | (508.8x) |
| fasta | 2241455 | 36760694 | (16.4x) | 718824003 | (320.7x) | 2285382 | 38691950 | (16.9x) | 770811211 | (337.3x) |
| fdct | 7519 | 25085 | (3.3x) | 1528669 | (203.3x) | 7465 | 38172 | (5.1x) | 2103383 | (281.7x) |
| fibcall | 2713 | 20411 | (7.5x) | 827897 | (305.2x) | 2731 | 22243 | (8.1x) | 893172 | (327.0x) |
| fir | 772946 | 17543810 | (22.7x) | 804645036 | (1041.0x) | 855949 | 22698227 | (26.5x) | 827569902 | (966.8x) |
| frac | 12313 | 172861 | (14.0x) | 5878181 | (477.4x) | 12136 | 185299 | (15.3x) | 5985378 | (493.2x) |
| huffbench | 1417998 | 14160500 | (10.0x) | 216244934 | (152.5x) | 1449947 | 13955966 | (9.6x) | 229499477 | (158.3x) |
| insertsort | 4212 | 81626 | (19.4x) | 2250385 | (534.2x) | 4065 | 95022 | (23.4x) | 2758031 | (678.3x) |
| janne_complex | 2272 | 17739 | (7.8x) | 449771 | (197.9x) | 2171 | 18005 | (8.3x) | 484653 | (223.2x) |
| jfdctint | 11636 | 142194 | (12.2x) | 2969790 | (255.2x) | 11497 | 123960 | (10.8x) | 3773542 | (328.2x) |
| lcdnum | 2065 | 22535 | (10.9x) | 323111 | (156.4x) | 1950 | 16860 | (8.6x) | 395945 | (203.0x) |
| levenshtein | 153352 | 3282550 | (21.4x) | 108037727 | (704.5x) | 149114 | 4310603 | (28.9x) | 114304863 | (766.6x) |
| ludcmp | 8823 | 82479 | (9.3x) | 2434265 | (275.9x) | 8765 | 83908 | (9.6x) | 3208591 | (366.0x) |
| matmult-float | 68462 | 1689146 | (24.7x) | 50348408 | (735.4x) | 67895 | 1762057 | (26.0x) | 57860913 | (852.2x) |
| matmult-int | 443996 | 9399015 | (21.2x) | 375279612 | (845.2x) | 443126 | 11765284 | (26.6x) | 501250720 | (1131.2x) |
| mergesort | 525439 | 10632506 | (20.2x) | 222978650 | (424.4x) | 514642 | 10430846 | (20.3x) | 281787323 | (547.5x) |
| miniz | 3270 | 14691 | (4.5x) | 412024 | (126.0x) | 3143 | 13570 | (4.3x) | 487135 | (155.0x) |
| minver | 6194 | 53678 | (8.7x) | 1413440 | (228.2x) | 6373 | 52712 | (8.3x) | 1506640 | (236.4x) |
| nbody | 256466 | 6453615 | (25.2x) | 191264270 | (745.8x) | 252409 | 5888723 | (23.3x) | 216254299 | (856.8x) |
| ndes | 114508 | 1772845 | (15.5x) | 54788991 | (478.5x) | 119261 | 1683039 | (14.1x) | 65749412 | (551.3x) |
| nettle-aes | 114333 | 567288 | (5.0x) | 31052424 | (271.6x) | 112622 | 576548 | (5.1x) | 34106112 | (302.8x) |
| nettle-arcfour | 88005 | 2120577 | (24.1x) | 26954748 | (306.3x) | 86661 | 1436640 | (16.6x) | 35980953 | (415.2x) |
| nettle-cast128 | 13974 | 26723 | (1.9x) | 206736 | (14.8x) | 12886 | 23546 | (1.8x) | 317716 | (24.7x) |
| nettle-des | 9078 | 32316 | (3.6x) | 255601 | (28.2x) | 8921 | 23980 | (2.7x) | 238207 | (26.7x) |
| nettle-md5 | 7350 | 32253 | (4.4x) | 675813 | (91.9x) | 7217 | 29839 | (4.1x) | 937431 | (129.9x) |
| nettle-sha256 | 14889 | 65280 | (4.4x) | 1735502 | (116.6x) | 14623 | 62231 | (4.3x) | 1652764 | (113.0x) |
| newlib-exp | 3658 | 25479 | (7.0x) | 661410 | (180.8x) | 3352 | 18812 | (5.6x) | 639414 | (190.7x) |
| newlib-log | 3038 | 16828 | (5.5x) | 511174 | (168.2x) | 2908 | 12749 | (4.4x) | 546217 | (187.8x) |
| newlib-mod | 2098 | 12596 | (6.0x) | 286197 | (136.4x) | 2010 | 13121 | (6.5x) | 337327 | (167.8x) |
| newlib-sqrt | 9430 | 186240 | (19.7x) | 6294054 | (667.4x) | 9174 | 183378 | (20.0x) | 5230626 | (570.1x) |
| ns | 28058 | 823601 | (29.4x) | 32090464 | (1143.7x) | 23796 | 943435 | (39.6x) | 32109538 | (1349.3x) |
| nsichneu | 12323 | 18872 | (1.5x) | 160230 | (13.0x) | 12236 | 18332 | (1.5x) | 164601 | (13.5x) |
| prime | 58925 | 944634 | (16.0x) | 16115583 | (273.5x) | 57233 | 1005099 | (17.6x) | 18090661 | (316.1x) |
| qsort | 3625 | 34736 | (9.6x) | 810665 | (223.6x) | 3543 | 28630 | (8.1x) | 1046006 | (295.2x) |
| qurt | 5333 | 67598 | (12.7x) | 1442175 | (270.4x) | 5140 | 64387 | (12.5x) | 1418908 | (276.0x) |
| recursion | 7838 | 177321 | (22.6x) | 5943982 | (758.3x) | 6813 | 147406 | (21.6x) | 5670809 | (832.3x) |
| rijndael | 1846763 | 10954085 | (5.9x) | 298854160 | (161.8x) | 1829801 | 11877918 | (6.5x) | 301644496 | (164.9x) |
| select | 2304 | 17312 | (7.5x) | 335812 | (145.7x) | 2182 | 18485 | (8.5x) | 369037 | (169.1x) |
| sglib-arraybinsearch | 32531 | 1061435 | (32.6x) | 28361404 | (871.8x) | 31769 | 1107230 | (34.9x) | 31999324 | (1007.2x) |
| sglib-arrayheapsort | 74568 | 1174730 | (15.8x) | 41340255 | (554.4x) | 73905 | 1196702 | (16.2x) | 50085952 | (677.7x) |
| sglib-arrayquicksort | 36200 | 583738 | (16.1x) | 26300739 | (726.5x) | 35305 | 545617 | (15.5x) | 25654013 | (726.6x) |
| sglib-dllist | 104108 | 947183 | (9.1x) | 26737508 | (256.8x) | 102530 | 930507 | (9.1x) | 31315926 | (305.4x) |
| sglib-hashtable | 72956 | 752795 | (10.3x) | 23759580 | (325.7x) | 74857 | 798844 | (10.7x) | 27269711 | (364.3x) |
| sglib-listinsertsort | 150571 | 3901838 | (25.9x) | 79469309 | (527.8x) | 151002 | 5126991 | (34.0x) | 99854495 | (661.3x) |
| sglib-listsort | 83492 | 865237 | (10.4x) | 26506487 | (317.5x) | 83432 | 879144 | (10.5x) | 31338183 | (375.6x) |
| sglib-queue | 79923 | 1171442 | (14.7x) | 33365763 | (417.5x) | 79308 | 1213322 | (15.3x) | 41832831 | (527.5x) |
| sglib-rbtree | 197550 | 1975689 | (10.0x) | 55392435 | (280.4x) | 198785 | 1797773 | (9.0x) | 59307610 | (298.3x) |
| slre | 81343 | 966902 | (11.9x) | 30416350 | (373.9x) | 79819 | 1022986 | (12.8x) | 34449484 | (431.6x) |
| sqrt | 281429 | 6231013 | (22.1x) | 91157327 | (323.9x) | 286888 | 4148972 | (14.5x) | 87278192 | (304.2x) |
| st | 46787 | 879737 | (18.8x) | 13216784 | (282.5x) | 45113 | 591158 | (13.1x) | 16071550 | (356.2x) |
| statemate | 5426 | 48227 | (8.9x) | 1187334 | (218.8x) | 5319 | 41665 | (7.8x) | 1302996 | (244.9x) |
| stb_perlin | 170913 | 2951012 | (17.3x) | 81001658 | (473.9x) | 141537 | 3078432 | (21.7x) | 85481578 | (604.0x) |
| stringsearch1 | 15688 | 95590 | (6.1x) | 5010590 | (319.4x) | 15624 | 116866 | (7.5x) | 6019351 | (385.3x) |
| strstr | 5072 | 59257 | (11.7x) | 1662703 | (327.8x) | 4943 | 51456 | (10.4x) | 2064634 | (417.6x) |
| tarai | 2698 | 25798 | (9.6x) | 898884 | (333.1x) | 2566 | 24645 | (9.6x) | 937342 | (365.3x) |
| trio-snprintf | 16752 | 89957 | (5.4x) | 1578752 | (94.2x) | 16670 | 80103 | (4.8x) | 1886009 | (113.1x) |
| trio-sscanf | 21441 | 170602 | (8.0x) | 4218744 | (196.8x) | 21561 | 121913 | (5.7x) | 4357713 | (202.1x) |
| ud | 11003 | 70482 | (6.4x) | 2121308 | (192.8x) | 10692 | 77272 | (7.2x) | 2755305 | (257.7x) |
| whetstone | 337043 | 3331038 | (9.9x) | 87417253 | (259.4x) | 301729 | 3001540 | (9.9x) | 99761886 | (330.6x) |

Table 10: PARSEC performance degradation results (cycles, thousands) on Skylake and Kaby Lake.

| | Skylake | | | | | Kaby Lake | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | NoDegrade | Degrade | | HyperDegrade | | NoDegrade | Degrade | | HyperDegrade | |
| blackscholes | 1054281 | 1780761 | (1.7x) | 18930407 | (18.0x) | 1076981 | 1865230 | (1.7x) | 23938125 | (22.2x) |
| streamcluster | 1615085 | 5743487 | (3.6x) | 135072031 | (83.6x) | 1609993 | 5117245 | (3.2x) | 149581491 | (92.9x) |
| fluidanimate | 1811088 | 8432568 | (4.7x) | 152698071 | (84.3x) | 1819377 | 8398743 | (4.6x) | 176033235 | (96.8x) |
| swaptions | 1530971 | 5362703 | (3.5x) | 85922802 | (56.1x) | 1534618 | 5586172 | (3.6x) | 112188654 | (73.1x) |
| freqmine | 2287791 | 4632607 | (2.0x) | 59327806 | (25.9x) | 2312730 | 4426167 | (1.9x) | 70071636 | (30.3x) |
| canneal | 2682216 | 9233551 | (3.4x) | 104439434 | (38.9x) | 3017818 | 8427129 | (2.8x) | 123489532 | (40.9x) |

Table 11: PARSEC performance degradation results (cycles, thousands) on Coffee Lake and Whiskey Lake.

| | Coffee Lake | | | | | Whiskey Lake | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | NoDegrade | Degrade | | HyperDegrade | | NoDegrade | Degrade | | HyperDegrade | |
| blackscholes | 960317 | 1653042 | (1.7x) | 23307722 | (24.3x) | 897093 | 1728380 | (1.9x) | 35757406 | (39.9x) |
| streamcluster | 1612074 | 5753689 | (3.6x) | 150934680 | (93.6x) | 1438805 | 5510738 | (3.8x) | 263693538 | (183.3x) |
| fluidanimate | 1616088 | 7346424 | (4.5x) | 172996775 | (107.0x) | 1626180 | 9221194 | (5.7x) | 212095965 | (130.4x) |
| swaptions | 1614836 | 5397065 | (3.3x) | 92169463 | (57.1x) | 1356684 | 6204397 | (4.6x) | 149384195 | (110.1x) |
| freqmine | 2152394 | 4007422 | (1.9x) | 69069190 | (32.1x) | 2113832 | 4481302 | (2.1x) | 80329274 | (38.0x) |
| canneal | 2567086 | 7895577 | (3.1x) | 112026498 | (43.6x) | 2566824 | 9135742 | (3.6x) | 127757591 | (49.8x) |


*November 11-15, 2019*, pages 2473–2487. ACM, 2019. URL https://doi.org/10.1145/3319535.3363192.

[3] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *LNCS*, pages 110–124. Springer, 2010. URL https://doi.org/10.1007/978-3-642-15031-9_8.

[4] Martin R. Albrecht and Nadia Heninger. On bounded distance decoding with predicate: Breaking the "lattice barrier" for the hidden number problem. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *LNCS*, pages 528–558. Springer, 2021. URL https://doi.org/10.1007/978-3-030-77870-5_19.

[5] Alejandro Cabrera Aldaya and Billy Bob Brumley. *HyperDegrade Proof-of-Concept*. Zenodo, October 2021. URL https://doi.org/10.5281/zenodo.5549559.

[6] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 870–887. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00066.

[7] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 422–435. ACM, 2016. URL http://doi.acm.org/10.1145/2991079.2991084.

[8] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 225–242. ACM, 2020. URL https://doi.org/10.1145/3372297.3417268.

[9] Richard Barnes, Jacob Hoffman-Andrews, Daniel McCarney, and James Kasten. Automatic Certificate Management Environment (ACME). RFC 8555, RFC Editor, March 2019. URL https://datatracker.ietf.org/doc/rfc8555/.

[10] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *LNCS*, pages 555–576. Springer, 2017. URL https://doi.org/10.1007/978-3-319-66787-4_27.

[11] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. NICV: Normalized inter-class variance for detection of side-channel leakage. In *International Symposium on Electromagnetic Compatibility, EMC 2014, Tokyo, Japan, May 12-16, 2014, Proceedings*, pages 310–313, 2014. URL https://ieeexplore.ieee.org/document/6997167.

[12] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 785–800. ACM, 2019. URL https://doi.org/10.1145/3319535.3363194.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In Andreas Moshovos, David Tarditi, and Kunle Olukotun, editors, *17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, pages 72–81. ACM, 2008. URL https://doi.org/10.1145/1454115.1454128.

[14] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *LNCS*, pages 129–142. Springer, 1996. URL https://doi.org/10.1007/3-540-68697-5_11.

[15] Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, volume 8437 of *LNCS*, pages 157–175. Springer, 2014. URL https://doi.org/10.1007/978-3-662-45472-5_11.

[16] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004. URL https://doi.org/10.1007/978-3-540-28632-5_2.

[17] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *LNCS*, pages 13–28. Springer, 2002. URL https://doi.org/10.1007/3-540-36400-5_3.

[18] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR_DRBG. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1241–1258. IEEE, 2020. URL https://doi.org/10.1109/SP40000.2020.00046.

[19] Jean-Sébastien Coron, Paul C. Kocher, and David Naccache. Statistics and secret leakage. In Yair Frankel, editor, *Financial Cryptography, 4th International Conference, FC 2000 Anguilla, British West Indies, February 20-24, 2000, Proceedings*, volume 1962 of *LNCS*, pages 157–173. Springer, 2000. URL https://doi.org/10.1007/3-540-45472-1_12.

[20] The FPLLL development team. fplll, a lattice reduction library. 2016. URL https://github.com/fplll/fplll.

[21] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 1983. URL https://doi.org/10.1109/TIT.1983.1056650.

[22] Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-interactive key exchange. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings*, volume 7778 of *LNCS*, pages 254–271. Springer, 2013. URL https://doi.org/10.1007/978-3-642-36362-7_17.

[23] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *LNCS*, pages 444–461. Springer, 2014. URL https://doi.org/10.1007/978-3-662-44371-2_25.

[24] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 845–858. ACM, 2017. URL http://doi.acm.org/10.1145/3133956.3134029.

[25] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *LNCS*, pages 112–131. Springer, 1997. URL https://doi.org/10.1007/BFb0052231.

[26] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *Non-Invasive Attack Testing Workshop, NIAT 2011, Nara, Japan, September 26-27, 2011. Proceedings*. NIST, 2011. URL https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf.

[27] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 955–972. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/gras.

[28] Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley. Triggerflow: Regression testing by advanced execution path inspection. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren,

editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, volume 11543 of *LNCS*, pages 330–350. Springer, 2019. URL https://doi.org/10.1007/978-3-030-22038-9_16.

[29] Dirk Grunwald and Soraya Ghiasi. Microarchitectural denial of service: insuring microarchitectural fairness. In Erik R. Altman, Kemal Ebcioglu, Scott A. Mahlke, B. Ramakrishna Rau, and Sanjay J. Patel, editors, *Proceedings of the 35th Annual International Symposium on Microarchitecture, Istanbul, Turkey, November 18-22, 2002*, pages 409–418. ACM/IEEE Computer Society, 2002. URL https://doi.org/10.1109/MICRO.2002.1176268.

[30] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 897–912. USENIX Association, 2015. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[31] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 490–505. IEEE Computer Society, 2011. URL https://doi.org/10.1109/SP.2011.22.

[32] Peter Gutmann. Simple Certificate Enrolment Protocol. RFC 8894, RFC Editor, September 2020. URL https://datatracker.ietf.org/doc/rfc8894/.

[33] Jahangir Hasan, Ankit Jalote, T. N. Vijaykumar, and Carla E. Brodley. Heat stroke: Power-density-based denial of service in SMT. In *11th International Conference on High-Performance Computer Architecture (HPCA-11 2005), 12-16 February 2005, San Francisco, CA, USA*, pages 166–177. IEEE Computer Society, 2005. URL https://doi.org/10.1109/HPCA.2005.16.

[34] Russ Housley. Cryptographic message syntax (CMS). RFC 5652, RFC Editor, September 2009. URL https://datatracker.ietf.org/doc/rfc5652/.

[35] Burt Kaliski. PKCS #7: Cryptographic message syntax version 1.5. RFC 2315, RFC Editor, March 1998. URL https://datatracker.ietf.org/doc/rfc2315/.

[36] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00002.

[37] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Public keys. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *LNCS*, pages 626–642. Springer, 2012. URL https://doi.org/10.1007/978-3-642-32009-5_37.

[38] Matt Lepinski and Stephen Kent. Additional Diffie-Hellman groups for use with IETF standards. RFC 5114, RFC Editor, January 2008. URL https://datatracker.ietf.org/doc/rfc5114/.

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[40] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E). In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 213–230. USENIX Association, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/merget.

[41] Tero Mononen, Tomi Kause, Stephen Farrell, and Dr. Carlisle Adams. Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP). RFC 4210, RFC Editor, September 2005. URL https://datatracker.ietf.org/doc/rfc4210/.

[42] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In Niels Provos, editor, *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*. USENIX Association, 2007. URL https://www.usenix.org/conference/16th-usenix-security-symposium/memory-performance-attacks-denial-memory-service-multi.

[43] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Elliptic Curve Digital Signature Algorithm with partially known nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003. URL https://doi.org/10.1023/A:1025436905711.

[44] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006. URL https://doi.org/10.1007/11605805_1.

[45] Daniel Page. *Practical Introduction to Computer Architecture*. Texts in Computer Science. Springer, 2009. URL https://doi.org/10.1007/978-1-84882-256-6.

[46] Salvador Palanca, Stephen A. Fischer, and Subramaniam Maiyuran. CLFLUSH micro-architectural implementation method and system, 2003. US Patent 6,546,462.

[47] James Pallister, Simon J. Hollis, and Jeremy Bennett. BEEBS: Open benchmarks for energy measurements on embedded platforms. *CoRR*, abs/1308.5174, 2013. URL http://arxiv.org/abs/1308.5174.

[48] James Pallister, Simon J. Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *Comput. J.*, 58(1):95–109, 2015. URL https://doi.org/10.1093/comjnl/bxt129.

[49] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*, 2005. URL http://www.daemonology.net/papers/cachemissing.pdf.

[50] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 83–98. USENIX Association, 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia.

[51] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make sure DSA signing exponentiations really are constant-time". In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1639–1650. ACM, 2016. URL http://doi.acm.org/10.1145/2976749.2978420.

[52] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. Certified side channels. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2021–2038. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/garcia.

[53] Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. Statistical analysis of second order differential power analysis. *IEEE Trans. Computers*, 58(6):799–811, 2009. URL https://doi.org/10.1109/TC.2009.15.

[54] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1451–1468. USENIX Association, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/ragab.

[55] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of Bleichenbacher's CAT: new cache ATtacks on TLS implementations. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 435–452. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00062.

[56] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Secretly monopolizing the CPU without superuser privileges. In Niels Provos, editor, *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*. USENIX Association, 2007. URL https://www.usenix.org/conference/16th-usenix-security-symposium/secretly-monopolizing-cpu-without-superuser-privileges.

[57] Nicola Tuveri and Billy Bob Brumley. Start your ENGINEs: Dynamically loadable contemporary crypto. In *2019 IEEE Cybersecurity Development, SecDev 2019, Tysons Corner, VA, USA, September 23-25, 2019*, pages 4–19. IEEE, 2019. URL https://doi.org/10.1109/SecDev.2019.00014.

[58] Nicola Tuveri, Sohaib ul Hassan, Cesar Pereida García, and Billy Bob Brumley. Side-channel analysis of SM2: A late-stage featurization case study. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 147–160. ACM, 2018. URL https://doi.org/10.1145/3274694.3274725.

[59] Sohaib ul Hassan, Iaroslav Gridin, Ignacio M. Delgado-Lozano, Cesar Pereida García, Jesús-Javier Chi-Domínguez, Alejandro Cabrera Aldaya, and Billy Bob Brumley. Déjà vu: Side-channel analysis of Mozilla's NSS. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1887–1902. ACM, 2020. URL https://doi.org/10.1145/3372297.3421761.

[60] Luke Valenta, Nick Sullivan, Antonio Sanso, and Nadia Heninger. In search of CurveSwap: Measuring elliptic curve implementations in the wild. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 384–398. IEEE, 2018. URL https://doi.org/10.1109/EuroSP.2018.00034.

[61] Yuval Yarom and Katrina 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*, pages 719–732. USENIX Association, 2014. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[62] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *LNCS*, pages 346–367. Springer, 2016. URL https://doi.org/10.1007/978-3-662-53140-2_17.