DABANGG: Time for Fearless Flush based Cache Attacks

Anish Saxena
Indian Institute of Technology Kanpur
anishs@iitk.ac.in

Biswabandan Panda
Indian Institute of Technology Kanpur
biswap@cse.iitk.ac.in

Abstract

Flush based cache attacks like Flush+Reload and Flush+Flush are one of the highly effective cache attacks. In fact, the Flush+Flush attack is stealthy too. Most of the flush based attacks provide high accuracy in controlled environments where attacker and victim are the only two processes that are running on a system by sharing OS pages. However, we observe that these attacks lose their effectiveness (prone to low accuracy) on a noisy multi-core system where co-running applications run along with the attacker and the victim. Two root causes for the varying accuracy of flush based attacks are: (i) the dynamic nature of core frequencies that fluctuate depending on the system load, and (ii) the relative placement of victim and attacker threads in the processor (same logical core, same physical core, different physical cores). The variation in the processor frequencies and placement of threads affect one of the critical attack steps (the cache latency calibration step as the latency threshold set to distinguish a cache hit from a miss becomes inaccurate).

We propose a set of refinements (DABANGG refinements) to make existing flush attacks resilient to frequency changes and thread placement in the processor, and therefore system noise. We propose refinements to pre-attack and attack steps and make it conscious about the latency change. We evaluate DABANGG-enabled Flush+Reload and Flush+Flush attacks (DABANGG+Flush+Reload and DABANGG+Flush+Flush, respectively) against the standard Flush+Reload and Flush+Flush attacks across five scenarios for eight different combinations of system noise capturing different levels of compute, memory, and I/O noise intensities: (i) a side-channel attack based on user input (single-character and multi-character key-logging), (ii) a side-channel on AES, (iii) a side-channel on RSA, (iv) a covert-channel, and (v) a transient execution attack in the form the Spectre attack. We also discuss the cross-VM feasibility for each attack scenario. For all the scenarios, DABANGG+Flush+Reload and DABANGG+Flush+Flush outperform the standard Flush+Reload and Flush+Flush attacks in terms of F_1 score and accuracy.

1 Introduction

On-chip caches on the modern processors provide a perfect platform to mount side-channel and covert-channel attacks as attackers exploit the timing difference between a cache hit and a cache miss. A miss in the Last-level Cache (LLC) requires data to be fetched from the DRAM, providing a measurable difference in latency compared to a hit in the LLC. Some of the common cache attacks follow one of the following protocols: (i) flush based attacks in the form of Flush+Reload [29] and Flush+Flush [10] and (ii) eviction based attacks [17], [11], [22]. Compared to eviction based attacks, flush based attacks provide better accuracy as flush based attacks demand the notion of OS page sharing between the attacker and the victim, and the attacker can precisely flush (with the `clflush` instruction) and reload (or flush, in case of Flush+Flush attack) a particular cache line. Like any other cache attacks, flush based cache attacks rely on the calibration of cache latency. The calibration provides a threshold that can differentiate a cache hit from a cache miss. As the `clflush` instruction flushes (invalidates) the cache line from the entire cache hierarchy, a typical event that drives the threshold is the LLC miss latency.

The problem: One of the subtle problems with the flush based attacks is its low effectiveness in the presence of system noise in the form of compute noise, memory noise, and the noise from I/O. To understand the effect of these system noises on the effectiveness of flush based attacks, we perform simple side-channel, covert channel, and transient attacks that use clflush. On average, across eight possible combinations of compute, memory, and I/O noise, a single-character based key-logging attack using LLC as a side-channel show that Flush+Reload and Flush+Flush provide F_1 scores of 42.8% and 8.1%, respectively. In a covert channel attack, Flush+Reload and Flush+Flush attacks suffer from maximum error rates of 45% and 53%, respectively. In contrast, Flush+Reload and Flush+Flush provide high accuracy and F_1 scores in controlled environments where only the attacker and the victim run concurrently. We discuss more about these attacks in Section 7. One of the primary reasons for this

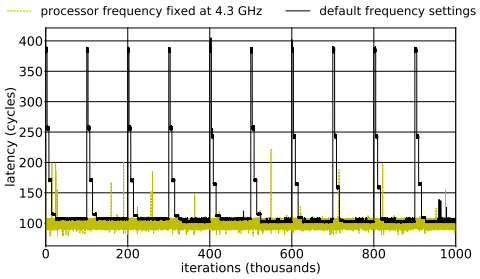


Figure 1: Variation in `reload` cache hit latency with `sleep()` system call invoked every 100 thousandth iteration.

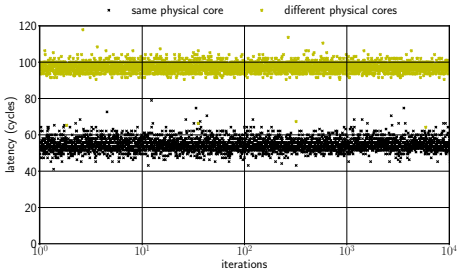


Figure 2: Variation in `reload` cache hit latency with relative placement of attacker and victim processes. All cores run at the (fixed) base frequency.

trend is that with the system noise, existing latency calibration mechanisms fail to provide a precise cache access time threshold. Prior works [6, 21] try to improve Flush+Reload attacks and noise in the covert channel attacks. Maurice et al. [21] tackle noise in covert-channel attacks only, which cannot be translated to side-channel attacks. Through this paper, we try to propose a generic approach to handle the system noise.

The details: To understand the subtleties, we perform the Flush+Reload attack in a highly controlled environment (with no noise from co-running threads). We perform the following steps: (i) *Flush* a cache line, (ii) *Wait* for the victim's access by yielding the processor (sleeping), and (iii) *Reload* the same cache line that is flushed in step (i). We perform these three steps for thousands of attack iterations, where one iteration involves the above mentioned three steps. Figure 1 shows the variation in execution latency of a reload cache hit with the `movl` instruction. For the rest of the paper, we refer to it as the `reload` instruction due to its utility in the reload step of Flush+Reload attack. We use the `rdtsc` instruction to measure the execution time of instructions. At every 100 thousandth iteration, we use `sleep()` function to sleep for 1 second, which results in the black curve. Note that in a real attack, an attacker will not sleep for one second. Next, we fix the processor frequency at 4.3 GHz (higher than the base frequency of 3.7 GHz but lower than the maximum Turbo Boost frequency of 4.5 GHz) and repeat the same experiment. The latency remains constant, mostly at around 100 cycles. We perform this extreme experiment to showcase the point which connects to the ineffectiveness of calibration techniques.

It is clear from Figure 1 that the `reload` latency increases drastically just after the `sleep()` system call. The increase in latency happens due to a change in processor frequency, which is triggered by the underlying dynamic voltage and frequency scaling (DVFS) [28] controller. If an attacker sets a latency threshold to distinguish a cache hit from a miss anywhere between 100 to 400 cycles, this results in false positives and reduces the effectiveness of flush based attacks. This frequency-oblivious latency threshold leads to low accuracy in flush based cache attacks. Moreover, even if we fix the frequency of all the cores, the latency of `reload` cache hit is still dependent on where the victim and attacker threads are located in the processor (refer to Figure 2). The `reload` hit latency when the two threads run on the same (multi-threaded) physical core is different from when they run on different physical cores.

This experiment sets the tone for the rest of the paper. In a noisy system with different applications running with different levels of compute, memory, and I/O load, the DVFS controller comes into the picture and throttles up and down the processor frequency accordingly. However, instructions such as `rdtsc` that measure the timing are unaffected by the change in the processor frequencies (DVFS controller). The consequence is, when the processor runs at a lower frequency, rdtsc still returns timing information based on a constant frequency that is unaffected by the DVFS controller, generating higher latency numbers even in case of a cache hit. This is further complicated by the relative placement of victim and attacker threads on the processor.

Our goal is to improve the effectiveness of flush based attacks and make them resilient to the effect of frequency, and thread placement changes so that flush based attacks will be effective even in the presence of extreme system noise.

Our approach: We propose a few refinements that make sure the cache access latency threshold remains consistent and resilient to system noise by improving the latency calibration technique and the attacker's waiting (sleeping) strategy. We name our refinements as DABANGG [1]. Overall, our key contributions are as follows:

- We motivate for noise resilient flush attacks (Section 4), identify and analyze the major shortcomings of existing flush based attacks (Section 5).
- We propose DABANGG refinements that makes the flush attacks resilient to processor frequency, thread placement, and therefore, system noise (Section 6).
- We evaluate flush based attacks in the presence of different levels of compute, memory, and I/O system noise (Section 7).

[1] DABANGG is a Hindi word that means "fearless". We envision DABNAGG refinements will make a flush based attacker fearless of the system noise.

2 Background

2.1 Cross-core Flush Based Cache Attacks

As per the Intel manual [14], a `clflush` instruction does the following: it *"Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location. The `clflush` instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and besides, a `clflush` instruction is allowed to flush a linear address in an execute-only segment"*.

Cross-core flush based attacks such as Flush+Reload and Flush+Flush use `clflush` instruction to invalidate cache block(s) from all levels of caches. The `clflush` instruction "Invalidates from every level of the cache hierarchy in the cache coherence domain, the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory." [14]. In a cross-core flush based attack, the attacker core flushes (using `clflush` instruction) a cache line address(es) from all levels of caches including remote cores' caches and the shared LLC. Later the attacker core reloads (Flush+Reload) or flushes (Flush+Flush) the same line address(es).

The three phases: Flush+Reload and Flush+Flush work in three phases: (i) flush phase, where the attacker core flushes (using `clflush` instruction) the cache line address(es) of interests. (ii) Wait phase, where the attacker waits for the victim to access the flushed address as the flushed cache line address is not present in the entire cache hierarchy. If the victim accesses the flushed address, then it loads the address into the shared LLC. (iii) Reload (Flush in case of Flush+Flush) phase, where the attacker reloads or flushes the flushed cache line address and measures the latency. If the victim accesses the cache line between phase I and III, then in case of Flush+Reload attack, the attacker core gets an LLC hit (LLC access latency), else an LLC miss (DRAM access latency). In case of Flush+Flush attack, the attacker core gets a `clflush` hit if the victim accesses the cache line between phase I and III, else a `clflush` miss. Since no memory accesses are performed in the case of Flush+Flush, Flush+Flush attack is harder to detect using performance counters which record cache references and misses, compared to Flush+Reload attack [10]. This makes the Flush+Flush attack stealthy.

Latency threshold and sleep time: The crux of flush based attacks lies in the difference in execution latency of `clflush` and `reload` instructions depending on whether they get a cache hit or a miss for the concerned address(es), and identifying the latency difference precisely. Additionally, the attacker waits (sleeps) in between phase I and phase III to provide adequate time for the victim to access the cache. Sleep time plays an important role in the overall effectiveness of flush based attacks. Usually, the three phases are executed step-by-step in an infinite loop, which we shall refer to as the *attack loop*. The attack may be synchronous or asynchronous, wherein the victim program runs synchronously or asynchronously with the spy program.

2.2 Dynamic Voltage & Frequency Scaling

Frequency and voltage are the two important run-time parameters managed through DVFS, as a function of perceived system usage. Specialized hardware and software components work cooperatively to realize this scheme.

Hardware support: A majority of modern processors are capable of operating in various clock frequency and voltage combinations referred to as the Operating Performance Points (OPPs) or Performance states (P-states) in Advanced Configuration and Power Interface (ACPI) terminology [24]. Conventionally, frequency is the variable which is actively manipulated by the software component. Therefore, performance scaling is sometimes referred to as *frequency scaling*. The P-states can be managed through kernel-level software, in which case power governors and scaling drivers are central to provide optimum efficiency. They can also be managed directly through a hardware-level subsystem, termed Hardware-managed P-states (HWP). Intel uses the Enhanced SpeedStep technology [4], and AMD uses Cool'n'Quiet and PowerNow! [3] technologies for HWP. In this case, the software driver usually relies on the processor to select P-states, although the driver can still provide hints to the hardware. The exact nature of these hints depends on the scaling algorithm (power governor). Another technology of interest is Intel's Turbo Boost [13] (and analogously, AMD's Turbo Core [5]) technology, which allows to temporarily boost the processor's frequency to values above the base frequency.

Depending on the processor model, Intel processor provides core-level granularity of frequency-scaling termed as the Per-Core P-State (PCPS), which independently optimizes frequency for each physical core [12]. This feature is available in higher-end models, one of which is our test processor, the *Xeon W-2145 Skylake* CPU. Another model, which is familiar in the consumer-level market (*Intel i7 6700 Skylake*) CPU, does not have PCPS support and all the cores in this processor change their frequencies simultaneously.

Software support: The software stack component responsible for coordinating frequency scaling is the `CPUFreq` subsystem in Linux, which is accessible by a privileged user via the `/sys/devices/system/cpu/` policy interface. In a compute-intensive workload without a lot of I/O requirements, the user might want to constrain the processor to higher (and possibly, the highest) P-states only. Servers, on the other hand, may benefit from not running at a high frequency for sustained performance [8] without exerting excess strain on the hardware. Fine-tuning of this interface is possible through the `sysfs` interface objects. Modern Intel processors come with `pstate`

drivers providing fine granularity of frequency scaling. It works at a logical CPU level, that is, a system with eight physical cores with hyper-threading enabled (two logical cores per one physical core) has 16 `CPUFreq` policy objects.

2.3 Timekeeping mechanism

Most of the x86 based processors use the `IA32_TIME_STAMP_COUNTER` Model-Specific Register (MSR) to provide a timekeeping mechanism. Different processor families increment the counter differently. There are two modes of incrementing TSC: (i) to increment at the same rate as the processor clock and (ii) to increment at a rate independent of the processor clock. Modern Intel processors use the latter mode [14]. *The constant TSC behavior is unaffected even if the processor core changes its frequency.* Processors with x86 ISA provide several instructions like `rdtsc`, `rdtscp`, and `cpuid`, that can read TSC value accurately to provide a timestamp. The `rdtsc` instruction is not a serializing instruction. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed. `rdtsc` is also the most commonly used instruction to measure execution latency. However, for high precision, the attacker uses `mfence` followed by `lfence` for proper serialization and in-order execution. The `rdtscp` instruction reads the TSC value as well as the contents of the `IA32_TSC_AUX` register, which contains useful information such as the processor ID. It measures the execution latency in a similar way to the `rdtsc` instruction. The `cpuid` instruction is extremely versatile and can be used to return the TSC value. It is a serializing instruction. However, its implementation varies in virtual environments, where it may be emulated in software, and it also takes more time to commit, making it an unpopular choice due to portability issues [29].

3 Experimental Setup

Table 1 shows the system configuration that we use to conduct our experiments and mount attacks. Though we use an Intel machine, we perform our experiments and find our proposal is equally effective on AMD based x86 machines like AMD A6-9220 RADEON R4 and macOS X (Version: 10.15.4). We use the `stress` tool to generate compute-intensive and IO-intensive noise, and SPEC 2017 `mcf` [20] benchmark to generate memory-intensive noise. `mcf` is a famous benchmark used in the computer architecture community for memory systems research, with an LLC misses per kilo instructions (MPKI) of over 100. Table 2 shows eight possible combinations of noise levels (L-L-L to H-H-H) comprising compute, memory, and IO, where L refers to low noise level, and H refers to high noise level.

At the high noise level (H-H-H), eight CPU-intensive, eight IO-intensive and eight memory-intensive threads are running concurrently, pushing the core runtime-usage to 100% on all cores (observed using `htop`). Figure 3 shows core-wise run-time usage for 8 noise levels. High level of compute-intensive noise results in high core frequencies on which the relevant code executes. High level of memory-intensive noise causes a dip in the core frequencies. In contrast, a high level of IO-intensive noise does not result in sustained high core frequencies, because IO-intensive applications sleep and wake up on interrupts. Power governors take clues from IO-intensive behavior to not increase frequency due to repeated system-calls which subsequently yield the CPU. Repeatedly yielding the CPU also reduces the CPU utilization, since the application spends less time on the CPU and more time waiting for an interrupt. We perform the following attacks using Flush+Reload and Flush+Flush: (i) keylogging, (ii) AES secret key extraction, (iii) covert channel, and (iv) Spectre [16]. We use metrics like True Positive Rate (TPR), False Positive Rate (FPR), accuracy, and the F_1 score to evaluate the effectiveness of various flush based attacks.

Ubuntu 18.04.1 LTS, 8 Hyper-Threaded Intel Xeon W-2145 Skylake cores
Base (minimum) frequency: 3.7 (1.2) GHz and Turbo Frequency: Up to 4.5 GHz
L1-D and L1-I: 32KB, 8 way, L2: 1 MB, 16-way
Shared L3: 11MB, 11-way, DRAM: 16 GB

Table 1: System configuration for our experiments.

Noise Level **C-M-I**	`stress` #CPU hogs	`mcf` #Mem hogs	`stress` #IO hogs
L-L-L	0	0	0
L-L-H	0	0	8
L-H-L	0	8	0
L-H-H	0	8	8
H-L-L	8	0	0
H-L-H	8	0	8
H-H-L	8	8	0
H-H-H	8	8	8

Table 2: Eight combinations of system noise. C-M-I: compute-memory-io. CPU/Mem/IO hogs of eight corresponds to eight threads running compute/memory/IO intensive code.

4 Motivation

4.1 The Curious Case of Accuracy

The flush based attacks rely on the execution timing difference between a cache hit and a miss. Ideally, setting appropriate thresholds after calibration should be enough to distinguish a hit from a miss. Experiments, however, do not precisely agree. To emphasize this point, we perform a single character keylogging experiment where a synchronous spy process monitors a cache line accessed by the victim. The victim processes a few characters every second (refer Section 7.1 for details). Even at the L-L-L noise level, the standard Flush+Flush attack provides TPR of 4.4%. Flush+Reload attack performs appreciably with high TPR of more than 94% at all noise levels (refer to Table 7).

One might be tempted to write off the Flush+Flush attack as inferior to the Flush+Reload attack, but that is

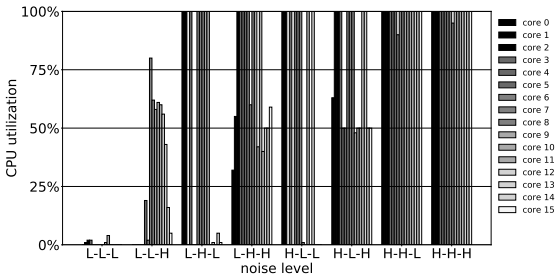


Figure 3: Logical core utilization at different noise levels.

not always the case. In our asynchronous AES secret key extraction experiment (refer Section 7.2 for details), both Flush+Flush and Flush+Reload attacks are capable of providing more than 90% average accuracy across noise levels. However, the number of encryptions (analogous to attack loop iterations) taken to achieve this is quite different for each attack. Clearly, given the right circumstances, Flush+Flush attack can deliver high accuracy even at high noise levels. Finally, the flush based attacks exhibit high variation in accuracy. In our Spectre attack experiment (refer Section 7.5 for details), the accuracy of Flush+Flush attack ranges from 25% to 91%, with an average accuracy of just 41%. Clearly, such an attack is not reliable. These considerations lead us to ask the following questions:

Question #1: Why is there a drastic difference in TPR in two representative attacks which rely on the cache-timing difference of ISA-level instructions?

Question #2: If the Flush+Flush attack is not inferior, what are the runtime changes that affect its accuracy?

Question #3: What is the root cause for varying accuracy, and can we do better than the standard attacks without using privileged operations?

We now uncover the root cause, which guides us to the shortcomings of the existing flush based attacks. We aim to produce uniformly high accuracy irrespective of system noise levels and without using any privileged operations.

4.2 The Root Cause

Flush+Flush attack: Figure 4(a) shows the latency of `clflush` instruction (in a system with L-L-L noise level) as a function attack-loop iterations. The frequency steps up slowly as the attacker code iterates through the attack loop. The processor does not step up to the maximum frequency (enabling the lowest latency in execution) if the code and data footprint of the program is minimal. The stepped increase in frequency gives rise to the distinct steps in the latency vs. iterations plot. An important point to be noted is that a single threshold value is not effective in distinguishing a `clflush` cache hit from a miss. For example, let us take 340 cycles as our threshold. Every `clflush` miss before 20,000 iterations are not identified correctly, driving the FPR up, thereby reducing the accuracy of the attack. It is an effective threshold after the cores have stabilized to a higher frequency, which is achieved after 75,000 iterations, taking up 335 million cycles (where one iteration is close to 4,500 cycles). 335 million cycles take about 84 milliseconds, which is comparable to the execution time of various cryptographic encryption suites.

Flush+Reload attack: Figure 4(b) shows the variation of `reload` latency over attack-loop iterations. If the threshold to distinguish a `reload` cache hit from a miss is set at 300 cycles, all of the `reload` cache hits after 5,000 iterations are accurately detected. The `reload` cache hit latency stagnates to 100 cycles within 15,000 iterations, while `clflush` takes close to 75,000 iterations to stagnate. *As a result, Flush+Reload attack is more resilient to frequency changes.*

In addition to frequency changes, both the attacks are equally susceptible to different placement of victim and attacker threads on the processor for each given frequency. Several attacks employ a warm-up period to remove residual microarchitectural states and, therefore, implicitly allow the processor to stabilize at a higher frequency. This increases the attack accuracy if (i) the placement of attacker and victim threads is unchanged after context switches by the OS, and (ii) the cores maintain the frequency at which the attack was calibrated for the duration of the attack. Both (i) and (ii) are not true in a noisy system since IO-intensive noise can lead to context switches of attacker and victim to other cores, and compute-intensive noise can ramp up the overall processor frequency.

5 Analysis & Insights

In this section, we first pinpoint the shortcomings of flush based attacks. Then we analyze the latency variation of `clflush` at different core frequencies and different relative placement of victim and attacker threads. We also look at multi-threaded victim process and its effect on the latency. Finally, we uncover the problems with cooperative yielding of processor in greater detail and suggest a simple alternative for phase (ii) (waiting phase) of the attack.

5.1 Shortcomings of State-of-the-Art attacks

Figure 4(c) illustrates the effect of different core frequencies on the `clflush` cache hit latency. If per-core P-states (Intel PCPS) is supported, the attacker and victim cores usually run at high frequencies while other cores run at lower frequencies. The cores running compute-intensive applications increase their frequency, while the idling cores do not save power, resulting in core-wise different frequencies. The power governors decide the core-wise frequency. If all cores run at a low frequency, then the measured `clflush` cache hit latency is high.

If all frequencies of all cores step up or step down simultaneously, all cores run at a particular frequency depending on overall CPU utilization. Moreover, the OS scheduling policy may decide to change the logical CPU

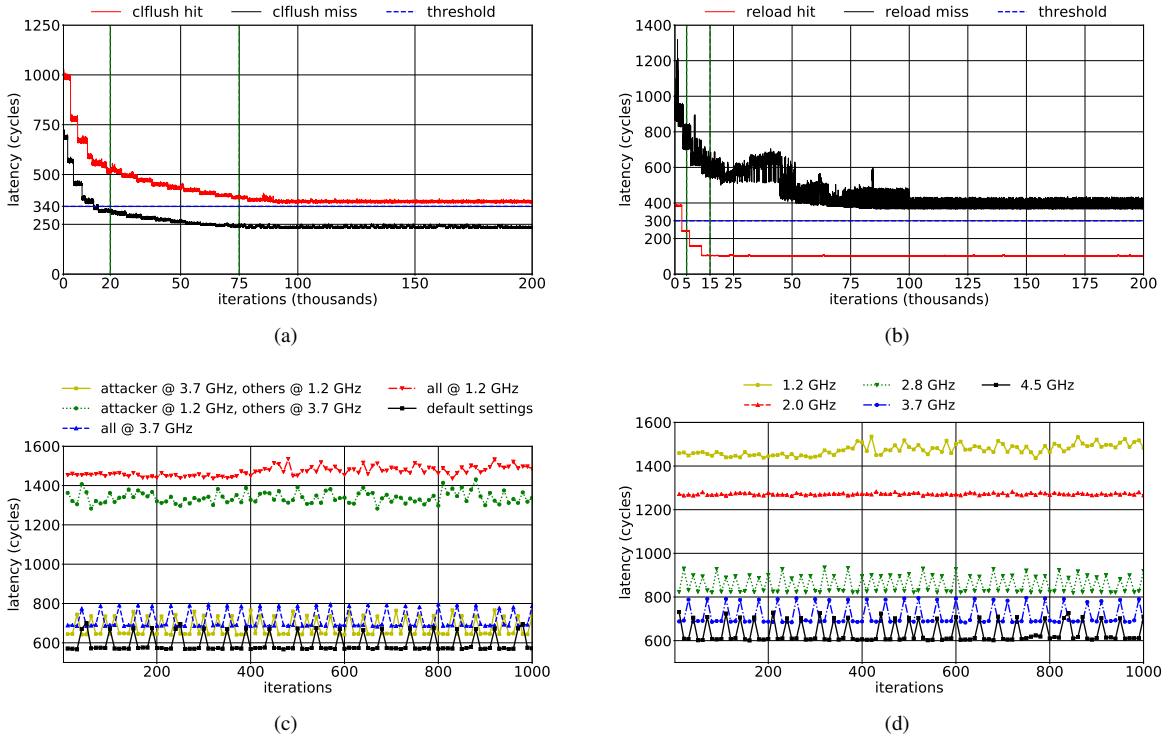


Figure 4: (a) and (b) show the variation of `clflush` and `reload` latency, respectively, with default power scaling settings. (c) shows variation of `clflush` hit latency with various relative core frequencies. (d) shows variation of `clflush` miss latency at different processor frequencies where all cores run at the mentioned frequency.

for the victim and attacker programs. Consequently, the corresponding frequencies change, depending on run-time usage and efficiency factors. This poses another fundamental issue, as we show that `clflush` takes different execution time even at a fixed frequency depending on whether the victim and attacker program run in the same physical core or not (refer Section 5.3 for details).

Moreover, the standard flush-based attacks simply yield the processor in phase II, using the `sched_yield()` function, which causes the attacker thread to relinquish the CPU, allowing a new thread to run. The attacks cooperatively yield as much as 30 times in-between two probes [10], which impedes the ability of the core to run at higher frequencies, making a reliable latency measurement difficult. We find the following shortcomings.

Shortcoming #1: Different cores may be at different frequencies at a given point of time, severely affecting `clflush` and `reload` cache hit latency.

Shortcoming #2: The attacker and victim threads may be part of the same process, or reside in same logical core, same physical core, or different physical cores. This has a measurable effect on `clflush` latency.

Shortcoming #3: The attacker cooperatively yields the CPU in-between probes, which can adversely affect the core frequency, ultimately degrading the attack accuracy.

5.2 Frequency Based Variation of `clflush`

In this section, we capture the variation of `clflush` latency as a function of processor frequency. We can capture the frequency directly by using the `/proc/cpuinfo` utility in Linux. However, this requires making a system call on every attack loop iteration. We, therefore, use an economical substitute. We plot the `clflush` latency at every attack loop iteration (same as Figure 4(a)).

The lowest available frequency on our processor under normal operating conditions is 1.2 GHz across all cores. The highest stable frequency is 4.3 GHz across all cores. Note that the maximum Turbo Boost frequency of 4.5 GHz is available for a short period, and the core then stabilizes at 4.3 GHz under sustained load. We pin the attacker to core #0 and the victim to core #1. In the Simultaneous Multi-Threading (SMT) processor that we use, pairs of logical cores #0 and #8, #1 and #9, and so on, reside in the same physical core. We pin the attacker and victim to different physical cores since this is usually how the Linux scheduler schedules the processes for maximum performance [19]. At 1.2 GHz, a `clflush` cache

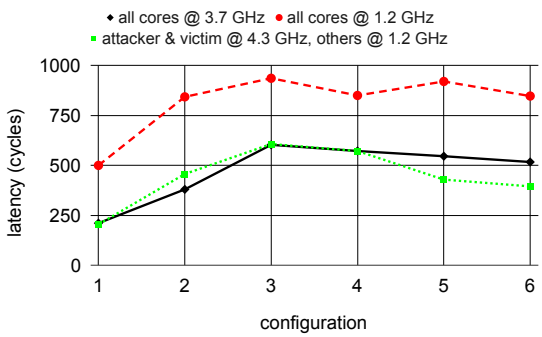


Figure 5: Variation of `clflush` latency with different configurations and frequencies.

miss has a latency of 1,500 cycles compared to 575 cycles when all the cores run at 4.5 GHz (refer to Figure 4(d)), a significant variation.

Explanation: A processor running at low frequency takes more time to execute the same instruction than a processor running at high frequency. Our processor increments the Time Stamp Counter (TSC) at a rate independent of the processor clock (refer Section 2.3 for details). Therefore, assuming the TSC is incremented every t nanoseconds (ns), the latency of `clflush` at 1.2 GHz is 1,500t ns and that at 4.5 GHz is 575t ns. The latency decreases by $\frac{1500\times t}{575\times t} \approx 2.6\times$. However, the frequency of the core increases by $\frac{4.5GHz}{1.2GHz} \approx 3.7\times$. Clearly, there is a non-linearity between the increase in frequency and decrease in latency. The non-linearity can be attributed to the multi-cycle path that is not exact divisible by newer clock period. We suspect that `clflush` being a complex instruction, cannot execute with lower latency as it encounters a multi-cycle path at higher frequencies. This multi-cycle path represents a lower bound to the latency of `clflush` instructions, and is unaffected by a further increase in frequency. Higher frequencies significantly decrease execution latency. The execution latency depends on both the critical path and frequency of the core. The latency varies non-linearly with frequency. The frequency-specific thresholds can be arranged in an array, indexed by a function of iteration number, to access the correct threshold for the given frequency. It is a relatively simple solution where the non-linearity is hidden behind heuristics-based indexing of an array of thresholds.

5.3 Core-based Variation of `clflush` Latency

It is crucial to understand the intricacies associated with the relative positioning of the victim and attacker processes on a physical processor. Figure 5 shows the variation in `clflush` latency at different relative frequencies, marked by their configurations on the horizontal axis. Table 3 shows various configurations for a four-core SMT system. In all the configurations, the attacker runs on core #0. Configurations 4, 5, and 6 correspond to a multi-threaded victim process which runs on multiple cores concurrently. For configurations 4 to 6, the victim accesses the cache line on all mentioned cores before a `clflush` hit by attacker, which means it is present in private caches of all those cores. Based on Figure 5, we can see at 1.2 GHz (red curve), if an attacker and the victim reside on the same physical core (configuration 2), it takes 843 cycles on average for `clflush` instruction to commit. This is considerably lower than when victim is also present on other cores (configuration 3 to 6), in which case it takes 936 cycles on average. Certainly, the scheduling of victim and attacker processes play an important role in determining the timing measurement.

Configuration	Victim	Attacker's phase (iii) clflush
1	no victim access	attacker misses on clflush
2	victim runs on C_0	attacker gets a clflush hit
3	victim runs on C_1	attacker gets a clflush hit
4	victim runs on C_0 and C_1	attacker gets a clflush hit
5	victim runs on C_1, C_2 and C_3	attacker gets a clflush hit
6	victim runs on C_0, C_1, C_2 C_3	attacker gets a clflush hit

Table 3: Different combinations of attacker and victim on a four-core SMT system. The attacker runs on core-0 (C_0).

Note that if the PCPS support is present on a system with low noise levels, the scheduler tries to place attacker and victim processes in different physical cores and steps up the frequency of these cores to provide maximum performance. Therefore, the most relevant point for our study is the curve of 4.3 GHz in Figure 5, where the victim and attacker cores run at 4.3 GHz on different physical cores, while the rest run at a common lower frequency which is fixed at 1.2 GHz.

Explanation: The number of cycles taken by an instruction to commit, increases when the cached memory address is not present in the attacker core and is rather present in another physical core. Note that `clflush` invalidates the cache line in the entire cache coherence domain (which includes cache lines present in remote core's caches too). If the attacker core gets a hit in its local caches, it invalidates the cache line from its L1/L2 without waiting for the other cores in the cache coherence domain to signal their respective hit or miss, and the instruction is committed [26]. However, in case of a miss in the calling core, the cache line is first searched in L1, then L2, and then the instruction waits for the invalidation signal (or lack thereof in case of a miss) from other cores for the particular cache line. The increase in latency due to explicit wait for other cores to register their invalidation signals is the reason for the increase in `clflush` latency.

5.4 Cooperative Yielding of Processor

We now focus on phase II of the attack loop, where the attacker yields the CPU and waits for the victim's access. Figure 6 shows the average cycles taken to yield the CPU 30 times, which is employed in standard flush attacks. The cycles taken go down with the increase in the processor frequency. We

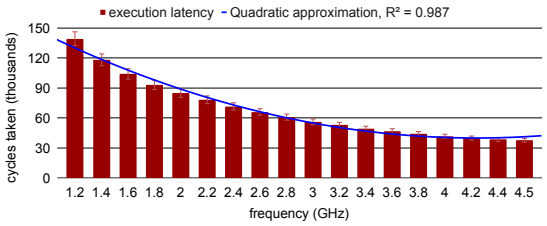


Figure 6: Execution timing of 30 `sched_yield()` function calls at various frequencies, averaged over 100 thousand data points at each frequency.

ALGORITHM 1: Compute_Heavy_Code[2]

```
1  Input: wait_gap
2  Initialization: (a, b) = (5, 6)
3  while (wait_gap) do // compute-intensive code
4      wait_gap -= 1
5      (a, b) = ( a×b / 4×a+1 , a / b+2.5 )
6      if (wait_gap%7 = 0) then
7          (a, b) = (0.11×b, 23×a)
8      else if (wait_gap%20 = 0 ) then
9          (a, b) = (b%3, a%14)
10     else
11         (a, b) = (2×b, 3.6×a)
```

explain the trend using a non-linear approximation (quadratic in the figure). We refer to yielding using `sched_yield()` function call as cooperative yielding because the attacker voluntarily relinquishes the CPU. The yield based waiting mechanism between two probes is subject to scaling governor as well. The `performance` power governor is aggressive in stepping up the frequency, while the `powersave` governor is more reserved. The waiting time-gap in phase II is deterministic for a given frequency. The function call to `sched_yield()` initiates a system call, which invokes the scaling driver to lower the processor frequency. The inadvertent behavior of `sched_yield()` further complicates the attack loop, since the core frequency can be different before the next iteration of the attack loop. Frequency changes in the hardware are not spontaneous. The apparent core frequency visible through `sysfs` interface is an approximation of current frequency, made by the Linux Kernel given the last known frequency reading and the hint for the next frequency value. The frequency can only be stepped up (or down) in increments of 100 MHz (in our system configuration), the apparent reading in between is merely an average calculated by the software. If the Linux scheduler is called proactively by the attacker, it may hint the power governor to lower the frequency, while also loading the next program's PCB onto the processor. The actual stepping down of the frequency is, as mentioned, non-spontaneous and can affect the frequency at which the next program operates. It is pragmatic to replace the `sched_yield()` based cooperative approach with a more aggressive compute-intensive approach. We run compute intensive operations in a busy-wait type loop, which steps up the processor frequency. It allows the execution latency of instructions to stagnate quickly. It also provides control over the waiting time-gaps for phase II of the attack loop. Next, we showcase that we can utilize a compute-intensive code segment as a replacement of `sched_yield()`, taking care not to make many memory references. Algorithm 1 shows an example code segment. The `wait_gap` deals with the number of attack loop iterations for the compute-intensive code segment. This code segment takes approximately 20,000 cycles at 1.2 GHz, and around 5,000 cycles at 3.4 GHz for `wait_gap` = 200. Crucially, the program execution time is known for a particular frequency. This provides finer granularity for the waiting time-gap control. Usage of compute-heavy code, therefore, can resolve two problems: (i) the attack loop is unambiguously reflected to power governors as a compute-intensive program, which helps in ramping up the core frequency quickly. (ii) Once the core reaches a stable frequency, the code segment provides excellent control over the waiting period as a function of `wait_gap`.

It is important to make sure our waiting time-gap remains approximately constant. If an address is accessed multiple times by the victim in a gap period, there is no way to ascertain one access from the other. On the other hand, if the attacker flushes the addresses in rapid succession, a true cache-hit may be missed due to overlap with phase-I of the attack. A suitable gap period is therefore, empirically derived. Existing literature [6] suggests that a gap period of 5,000 to 10,000 cycles is sufficient to detect individual cache accesses in many important flush based attacks. We can apply the mentioned analysis to the phase-II of synchronous attacks. In the case of asynchronous attacks, we don't need to wait a lot between probes. In that case, however, to eliminate the frequency-induced variation in latency, we run the compute-intensive loop for a few million cycles to stabilize the core at high frequency. We call the `Compute_Heavy_Code()` function once before going into the attack loop with `wait_gap` $\approx 10^5$. Note that this approach will not work for synchronous attacks since the victim can start the relevant execution at any moment. In a nutshell, the attack loop must be aware of the change in thresholds caused by variable frequency of the cores and relative core allocation of victim and attacker threads. We observe that the thresholds can be captured by the calibration tools in an array indexed by the iteration number of the attack loop. The waiting period (phase-II) requires a compute intensive approach. In the following Section, we outline these refinements and describe the modified attack loop.

6 DABANGG Attack Refinements

Taking into account the insights uncovered in previous sections, we outline three refinements over standard flush attacks. We call these refinements as the DABANGG refinements.

[2]The numbers in Algorithm 1 do not serve any special purpose. The aim is to use the CPU's compute units.

Parameters	Name	Description
Attacker-Specific	T_array	An array with each entry stores a tuple of lower and upper latency thresholds <TL,TH>.
	regular_gap	Regular waiting period of attacker in Phase II.
	step_width	Average width of a step in terms of number of attack loop iterations in latency vs #iterations plot.
Victim-Specific	acc_interval	Average number of attack loop iterations between two victim accesses without considering burst-mode accesses in between.
	burst_seq	In case of burst-mode access sequence by victim, number of victim accesses to target memory address in a single burst.
	burst_wait	Waiting time gap in terms of attack loop iterations before discarding an incomplete burst-mode access sequence as a false positive.
	burst_gap	Reduced waiting time gap to monitor burst-mode access sequence.
Runtime Variables in Algorithm 2	iter_num	A counter that counts the number of attack loop iterations.
	<TL,TH>	Pair of lower (TL) and upper (TH) latency threshold to detect cache hit.
	reload_latency	Execution latency of reload instruction in processor cycles.
	last_hit	Number of attack loop iterations since last true cache hit. A true cache hit is recorded by attacker when victim access interval (acc_interval) and victim burst-mode access sequence (burst_seq) criteria are satisfied, in addition to reload_latency $\in$ [TL,TH].
	potential_hit	Number of attack loop iterations since last potential cache hit. A potential hit may be either a false positive or a part of burst-mode access sequence by victim application.
	seq_id	Sequence identifier, stores the number of potential cache hits which, if it forms a burst-mode access sequence, implies a true cache hit.

Table 4: Specifications of parameters and runtime variables used by DABANGG attack loop (refer Algorithm 2).

Refinement #1: To capture the stepped frequency distribution of the processor while distinguishing a cache hit from a miss, we use comprehensive calibration tools.

Refinement #2: To identify the victim's memory access pattern in the attack loop, we use victim-specific parameters.

Refinement #3: To provide a better grip over waiting period of attack loop, we use compute-intensive function instead of cooperatively yielding the processor.

These refinements make the attacker frequency-aware and victim-aware. Note that even the standard flush based attacks are aware victim programs as the attacker flushes the critical memory addresses. With our refinements, we make the attacker aware of victim's behavior that will help in increasing the effectiveness of the attack.

6.1 Pre-Attack Steps

In the pre-attack step, the attacker calibrates for cache latency with DABANGG refinements to become frequency-aware. The attacker also profiles the victim application to identify the access pattern for the target memory address. Table 4 provides the details of all the parameters that DABANGG attack loop uses. We refer Table 4 throughout this section for different parameters of interest. We now explain the pre-attack steps briefly for the DABANGG+Flush+Reload attack.

Calibration: We derive attacker-specific parameters from the latency vs iterations behavior. We use Figure 7 for our reference (a fine-grained version of Figure 4(b)). The reload hit latency represents a stepped distribution. Multiple pairs of `<TL,TH>` are stored as tuples in `T_array`. `T_array` captures the frequency distribution in the attack loop. From Figure 7, four distinct steps are visible. The width of each step (that is, `step_width`) is 4000 attack loop iterations. For `iter_num` $\in$ [0,4000], we use TL = 375 and TH = 400.

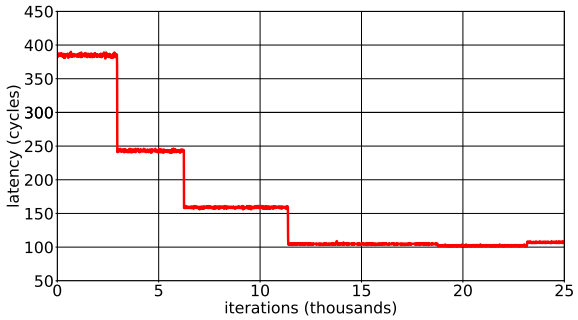


Figure 7: Variation of reload hit latency with attack iterations.

Therefore, `T_array[0]` = <375,400>. Similarly, we add three more tuples to `T_array`. These parameters are independent of victim applications. `regular_gap` parameter depends on the type of attack mounted (asynchronous or synchronous). `regular_gap` = 200 provides a waiting period of 5,000 to 10,000 cycles (refer Section 5.4 for details).

Profiling: We derive victim-specific parameters by observing the memory access pattern for target address of the victim application (for example, the `acc_interval` parameter). If the victim application accesses the critical memory address once in one million cycles on average, and an attack-loop iteration takes 10,000 cycles at low processor frequency, then `acc_interval` = $\frac{1{,}000{,}000}{10{,}000} = 100$.

A burst-mode access sequence occurs when the target address is inside a loop and gets accessed several times within a few `regular_gap` based attack loop iterations. Consider that the victim accesses the address 40 times within 20,000 cycles, for example. If we wait using the `regular_gap`, which takes 10,000 cycles at low frequency, we can only ascertain 2 cache hits. We utilize the burst-mode parameters to capture the burst-access pattern at finer granularity.

ALGORITHM 2: DABANGG+FLUSH+RELOAD

```
1  Initialization: last_hit, potential_hit, iter_num, seq_id = 0
2  while true do
3      iter_num += 1
4      <TL, TH> = T_array[ iter_num / step_width ] // update <TL,TH>
5      clflush(addr) // PHASE-I: Flush
       // PHASE-II: Wait
6      if (!rand()%400) then // branch taken 0.25% of time
7          Verify_Threshold(iter_num, addr) // Algorithm 3
8          sched_yield() // cooperatively yield the CPU
9      else if (seq_id > 0) then// burst sequence detected
10         Compute_Heavy_Code(burst_gap) // Algorithm 1
11     else
12         Compute_Heavy_Code(regular_gap) // Algorithm 1
       // PHASE-III: Reload
13     reload_latency = Measure_Reload_Latency(addr)[3]
14     if (reload_latency ∈ [TL,TH]) and ( last_hit > acc_interval)
         and (seq_id > burst_seq)) then // true hit
15         last_hit, seq_id = 0 // reset variables
16         print "low reload latency, it is a cache hit!"
17     else if (reload_latency ∈ [TL, TH]) then// potential hit
18         potential_hit = last_hit
19         seq_id += 1 // increment sequence identifier
20     else
21         last_hit += 1 // +1 iteration since last hit
22         print "high reload latency, it is a cache miss!"
23         if ((last_hit - potential_hit) > burst_wait) then
24             seq_id = 0 // discard seq as false positive
```

ALGORITHM 3: Verify_Threshold

```
1  Input: iter_num, addr
2  reload(addr)
3  Δ = Measure_Reload_Latency(addr)
4  if (Δ ∉ [TL,TH]) then
       // T_array[i].TL < Δ < T_array[i].TH
5      ∃<TL_new,TH_new> = T_array[i] : Δ ∈[TL_new,TH_new]
6      <TL,TH> = <TL_new,TH_new>
7      iter_num = step_width×i
8  end
```

`burst_seq` ≤ 40 (since we have 40 accesses by victim in burst-mode) and waiting period when a burst is detected should be $\leq \frac{20{,}000}{40} = 500$ cycles. This implies a `burst_gap` ≈ 10, which increases attack granularity. In practice, to reduce false negatives, we tolerate some missed cache-hits to determine the sequence, `burst_seq` $= \frac{40}{x}$ and `burst_wait` $= x$ where x is relatively small number compared to 40. For example, `burst_seq` = 20 for `burst_wait` = 2.

6.2 Attack Loop

A self-contained Algorithm 2 explains the DABANGG refinements. Line 1 initializes the runtime variables of interest, refer Table 4 for details. Line 3 increments the iteration number. Line 4 updates <TL, TH> through a simple indexing mechanism. `iter_num` divided by `step_width` linearly indexes `T_array` to provide a single pair of thresholds per step. Line 5 starts the attack and flushes the shared memory address. Lines 6 to 12 represent the waiting phase of the attack. Approximately once every 400 iterations (0.25% of all iterations), the attack loop verifies the current value of <TL,TH>. The `Verify-Threshold()` function, given in Algorithm 3, checks if the current tuple of thresholds, <TL,TH> accurately detect a cache hit at the current frequency. Lines 2 and 3 of Algorithm 3 measure the accurate access latency for target memory address. If $\Delta \in$[TL,TH], the function returns without making any changes. However, if $\Delta \notin$ [TL,TH] (Line 4), then the tuple is updated. This is done by looking up `T_array` such that $\Delta \in$[TL_{new},TH_{new}] and `T_array[i]` = <TL_{new},TH_{new}> (Line 5). Lines 6 and 7 update the tuple and `iter_num`, respectively. Verification and feedback enables threshodls to dynamically adapt to frequency changes which differ from Figure 7 in extremely noisy environments.

After verifying thresholds, the control flow returns to Algorithm 2, Line 8. `sched_yield()` function yields the processor cooperatively (once in a while based on the condition in Line 6) to prevent detection of an attack loop based on continuous usage of computationally heavy code. Most of the time, however, the attacker runs a compute-heavy code (Lines 10 and 12). The `wait_gap` for Algorithm 1 is appropriately chosen. Line 9 checks if an active burst sequence is present (that is, `seq_id` > 0), and uses `burst_gap` to reduce the waiting period of the attack loop.

We now move to the third phase of the attack. Line 14 performs the reload and calculates its execution latency. Line 15 checks for a true cache hit. Here, in addition to `reload_latency` $\in$ [TL,TH], `last_hit` > `acc_interval`, checks if access interval since the last true cache hit is adequate and `seq_id` > `burst_seq` checks if the burst sequence pattern is identified. In this case, the variables are reset in Line 16 and a true cache hit is registered in Line 17. Line 18 deals with a potential cache-hit, wherein Line 20 increments the sequence identifier and `potential_hit` variable is updated.

Line 22 increments the `last_hit` variable if `reload_latency` $\notin$ [TL,TH]. Line 23 records a cache-miss for the current iteration of the loop. However, instead of resetting the sequence identifier (that is, `seq_id`) right away, awaiting window of `burst_wait` attack loop iterations exists (in Line 24). The waiting window allows us to account for cache-hits missed by the attack loop. A cache-hit missed by the attacker occurs due to overlapping in phase I (Flush phase) of the attack loop with access to monitored cache line by the victim, wherein the attack loop flushes the line right after the victim accesses it. Line 25 resets `seq_id` to zero if the waiting window is exceeded. This concludes an attack loop iteration, and the control switches back to Line 3 of the attack. Flush+Flush attack can similarly be extended to become DABANGG+Flush+Flush. *Note that in all the refinements, we do not use or demand privileged operations.*

[3] Measure_Reload_Latency(addr) is defined similar to code given in [29], Figure 4. We use the appropriate `lfence` and `mfence`, however we do not use `clflush` in our function.

Attack	Parameter	Value
D+F+F	T_array	{<650,675>,<530,560>,<440,460>, <340,370>,<230,255>}
	step_width	4000
D+F+R	T_array	{<375,400>,<235,255>,<150,165>, <95,105>}
	step_width	4000

Table 5: Attacker-specific parameters

Lookup Type	Parameter	D+F+F	D+F+R
Single & Multiple Characters	acc_interval	1000	1000
	burst_seq	15	20
	burst_wait	3	2
	burst_gap	40	30
Single Character	regular_gap	400	200
Multiple Characters		100	50

Table 6: Parameters for keylogging attack.

In the following section, we evaluate the DABANGG refined attacks in many real-world scenarios and compare the TPR, FPR, Accuracy, and F_1 Score with standard Flush+Flush and Flush+Reload attacks.

7 Evaluation of Flush based Attacks

We evaluate DABANGG refined flush based attacks in five experiments: (i) side-channel attack based on user input (keylogging), (ii) side-channel attacks on T-Tables based AES, (iii) side-channel attacks on Square-Multiple Exponentiation based RSA (iv) covert-channel attack, and (v) Spectre-based attack. We then discuss the feasibility of cross-VM variants of attacks (i) to (v). For all experiments, we use the attacker-specific parameters specified in Table 5. For the rest of the paper, all the tables use F+F, F+R, D+F+R, and D+F+F for Flush+Flush, Flush+Reload, DABANGG+Flush+Reload, DABANGG+Flush+Flush, respectively. The parameter `regular_gap` depends on the type of attack (synchronous/ asynchronous) and is therefore specified separately for each attack.

7.1 Side-channel Attack based on Keylogging

The objective of this attack is to infer specific or multiple characters (keys) processed by the victim program. We use an array of 1024 characters. The distribution of characters is uniform and random. The victim program takes as input a character from a set of accepted characters, and for each character, calls a unique function that runs a loop a few thousand times. The victim program processes multiple characters every second, with a waiting period between two characters to emulate the human typing speed.

Threat model: As all the flush based attacks demand page sharing between the victim and the attacker, the attacker maps the victim program's binary (using `mmap()` function) and disassembles the victim program's binary through `gdb` tool to find out the addresses of interest. The attacker then monitors the character(s) and infers if the specified character (or characters, each having unique addresses) is processed by the victim. The attacker tries Flush+Reload and Flush+Flush techniques to infer the keys.

We profile the victim to determine the average waiting period between two accesses and the number of accesses to the target address in burst-mode inside the loop. We derive the parameters specified in Table 6. We calculate the victim specific parameters are calculated as per the pre-attack steps (section 6.1).

The `regular_gap` parameter is reduced in multiple character lookup to monitor four target addresses iteratively. In general, if there are n addresses to be monitored, we set $\texttt{regular\_gap}_{\texttt{multiple\_chars}} = \frac{\texttt{regular\_gap}_{\texttt{single\_char}}}{n}$. The waiting period for a particular address, therefore, remains constant.

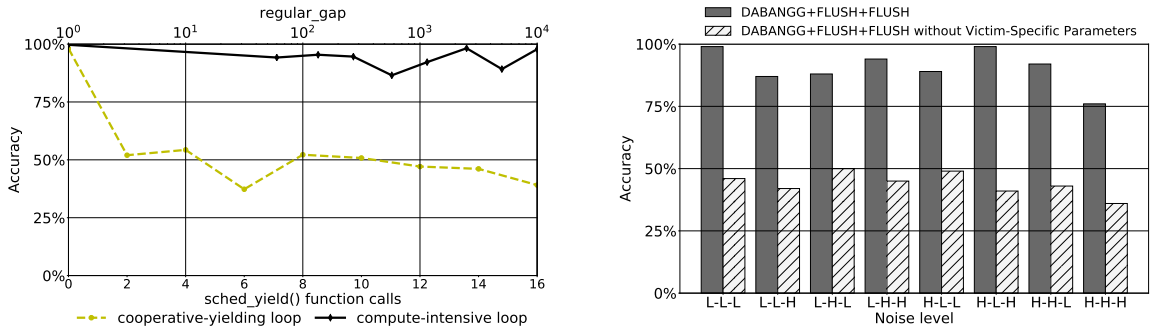
7.1.1 Single character lookup

We compare the DABANGG-enabled attacks (DABANGG+Flush+Flush and DABANGG+Flush+Reload) with the standard Flush+Flush and Flush+Reload attacks. The power-scaling settings are set to default state. We evaluate our attack as follows. The attacker outputs the timestamp, T, of the specific character processed by the victim. If T lies within N cycles of the timestamp obtained from the victim program, T_{real}, we conclude it to be a True Positive. The N cycle window takes into account the mechanism in DABANGG-enabled attacks, which makes it necessary for the attacker to take multiple observations to avoid false positives. Empirically, we choose $N = 150,000$ cycles to ascertain a real positive. The victim program waits for hundreds of millions of cycles after processing each character (to emulate human typing speed). Therefore the attacker must accurately determine T of the character input (that is, T must lie within N cycles of T_{real}) to register a true positive in our experiment.

Table 7 shows the results for different system noise levels. The DABANGG enabled attack outperforms both standard attacks in all three measurement criteria, namely the TPR, FPR and F_1 Score. The trend is visible at different noise levels. Refer Section 3 for CPU utilization at various noise levels.

Compute noise assists the standard attacks because it increases the core frequencies. The Flush+Flush attack, for example, improves its TPR from 4.4% at L-L-L noise level to 15.8% at the H-H-H noise level because the cores are already at high frequencies and the thresholds set for the attack are relatively accurate. However, IO-intensive noise, which is interrupt-driven, does not increase the processor frequency. The standard attacks struggle when IO-intensive noise is present. Lack of victim-specific parameters impacts the accuracy of Flush+Flush and Flush+Reload negatively, leading to high FPR in the presence of noise. For example, the FPR of Flush+Reload attack increases from 6.9% at L-L-L noise level to an average of 19% across all the other seven noise levels.

At all noise levels, the DABANGG+Flush+Flush attack accurately determines True Positives and maintains the



(a) (b)

Figure 8: Utility of (a) Compute-intensive code and (b) Victim-specific parameters on DABANGG+Flush+Flush attack.

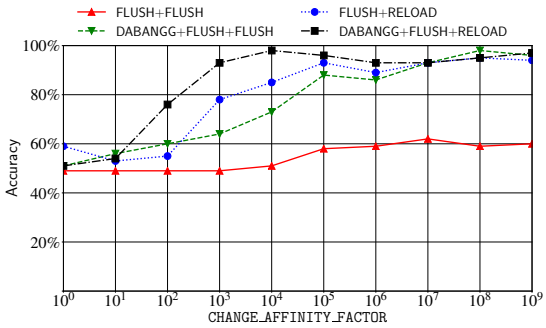


Figure 9: Effect of thread migration based on `CHANGE_AFFINITY_FACTOR`.

right balance between precision and recall. In contrast, the Flush+Flush attack fails to capture the true positives due to lack of comprehensive attacker-specific parameters, namely the dynamic thresholds. The DABANGG+Flush+Reload attack decreases the FPR (lower is better) over Flush+Reload due to victim-specific parameters. We now separately look at: (i) advantages of using compute-intensive loop and (ii) advantages of using victim-specific parameters, and (iii) To test our refinements in extreme conditions, we also showcase the effect of aggressive thread migration. This analysis provides a breakdown of the strengths of DABANGG-enabled attacks. We conduct (i) and (iii) at L-L-L noise level while we conduct (ii) at all eight noise levels.

Utility of compute-intensive loop: Figure 8(a) shows the advantage of compute-intensive loops over cooperative-yielding loops. The DABANGG+Flush+Flush attack utilizing `sched_yield()` suffers from excessively yielding the CPU, reducing the accuracy considerably. Note that the attacks corresponding to zero `sched_yield()` function calls and `regular_gap` $= 0$ are equivalent. It might be tempting to omit phase (ii) of the attack loop altogether given the high accuracy achieved in this experiment at `regular_gap` $= 0$. However, it must be noted that doing so increases the probability of overlap of victim cache line access with a phase (i) (`clflush`) of the attack loop. We avoid this possibility by keeping `regular_gap` $\approx 10^2$ in other experiments. The compute-intensive loop maintains reasonably high accuracy (>90%) over a comprehensive range of `regular_gap`, from 60 (≈1,000 cycles) to over 8,000 (≈120,000 cycles). This is achieved with increased accounting capabilities enabled by utilizing more thresholds, in conjunction with a tight grip over the waiting period.

Utility of victim-specific parameters: Figure 8(b) illustrates the importance of victim-specific parameters along with the compute-intensive loop. There are two issues with standard attacks: (i) a single cache hit in a victim where burst-mode access is present does not signify a true hit; it may be a false positive, and (ii) if we keep count of burst-mode accesses, a nearly correct sequence may be discarded by the attack loop due to a missed cache-hit. This reduces the accuracy of the attack. DABANGG refined attacks resolve these problems by (i) identifying burst-mode sequence (`seq_id` variable) and correlating it with victim-specific expected sequence (`burst_seq` parameter) and memory access interval (`acc_interval` parameter), and (ii) allowing missed cache-hits in the attack by keeping a waiting window of `burst_wait` iterations.

Effect of thread migration: Figure 9 corresponds to a thread migration analysis. An attack resilient to frequent core switches is desirable, as the latency changes based on the relative positioning of the victim and attacker programs on the processor cores. We artificially migrate the attacker core randomly, essentially de-scheduling the process from the current core and scheduling it on the intended core. We run a single character lookup experiment with all four attacks. DABANGG+Flush+Flush attack, whose accuracy is more dependent on processor frequency, is more affected by random core migrations compared to DABANGG+Flush+Reload attack. The number of attack loop iterations that are allowed to elapse before changing the core affinity is marked by the `CHANGE_AFFINITY_FACTOR`, which we vary and record the corresponding attack accuracy. The Linux scheduler may change the program core within a few 10s of milliseconds,

Noise	Attack	TPR	FPR	F_1 Score
L-L-L	F+F	4.4%	27.2%	9.1%
	D+F+F	**97.6%**	**0.3%**	**81.3%**
	F+R	99.6%	6.9%	54.3%
	D+F+R	**91.3%**	**2.8%**	**98.7%**
L-L-H	F+F	9.9%	29.4%	8.3%
	D+F+F	**72.6%**	**9%**	**72.8%**
	F+R	100%	25.8%	43.3%
	D+F+R	**100%**	**0%**	**100%**
L-H-L	F+F	7.9%	30.5%	6.9%
	D+F+F	**76.6%**	**7.6%**	**76.4%**
	F+R	99.6%	24.4%	42.1%
	D+F+R	**99.6%**	**0.01%**	**99.6%**
L-H-H	F+F	7.9%	30.1%	4.4%
	D+F+F	**88.5%**	**3.8%**	**88.8%**
	F+R	99.6%	21.6%	38.9%
	D+F+R	**100%**	**0%**	**100%**
H-L-L	F+F	12.3%	37.2%	7.6%
	D+F+F	**79.4%**	**8%**	**77.4%**
	F+R	99.6%	25.9%	45.8%
	D+F+R	**100%**	**0%**	**99.1%**
H-L-H	F+F	6.3%	28.1%	5.4%
	D+F+F	**99.2%**	**0.3%**	**80.5%**
	F+R	94%	6.6%	41.1%
	D+F+R	**99.6%**	**0.9%**	**98.9%**
H-H-L	F+F	8.7%	29.8%	8.2%
	D+F+F	**85.7%**	**4.4%**	**85.7%**
	F+R	99.2%	11.5%	36.2%
	D+F+R	**98.8%**	**0.4%**	**98.8%**
H-H-H	F+F	15.8%	27.5%	14.7%
	D+F+F	**51.6%**	**51%**	**51%**
	F+R	99.6%	17.5%	40.5%
	D+F+R	**98.2%**	**0.3%**	**99.2%**

Table 7: TPR, FPR, and F_1 scores of Flush+Flush, DABANGG+Flush+Flush, Flush+Reload, and DABANGG+Flush+Reload attacks for single character lookup.

which corresponds to `CHANGE_AFFINITY_FACTOR` of around 10^4. However, we test for `CHANGE_AFFINITY_FACTOR` ranging from 10^0 (≈10 microseconds) to 10^9 (≈few hours). We also experiment with hardware prefetchers ON/OFF at L1 and L2 levels, and we find it has a negligible effect on the DABANGG refinements.

The DABANGG refined attacks provide higher accuracy at each `CHANGE_AFFINITY_FACTOR`. The general trend obtained signifies that the accuracy increases with larger `CHANGE_AFFINITY_FACTOR`, which translates to more time available to stabilize the core frequency.

7.1.2 Multiple character lookup

In this experiment, we reproduce multiple characters processed by the victim program. We label each character and its identifying memory address A_M, and monitor all the addresses iteratively. A hit in the address is interpreted through its label. We utilize the Levenshtein distance (Lev) algorithm [18] to compare the accuracy of various attacks at all the system noise levels. The Lev algorithm compares the actual input sequence with the sequence observed by the attacker and computes accuracy based on the number of *insertion*, *substitution* and *deletion* operations.

DABANGG-refined attacks produce accurate results, as is evident from Table 8. The refined attacks are also more noise-tolerant than the standard attacks, especially the Flush+Flush attack, which suffers from yielding the CPU too often and highly variable `clflush` latency. DABANGG refined attacks produce more than 90% accuracy irrespective of the noise level, attesting to the robustness provided by DABANGG. The relative increase in attack accuracy with an increase in compute-intensive noise, especially compared to IO-intensive noise, exemplifies the effect of frequency.

The key takeaways from the keylogging experiments are: (i) DABANGG significantly improves the TPR, FPR, F_1 score, and accuracy, and (ii) DABANGG is robust to different noise levels. DABANGG+Flush+Flush uplifts the standard Flush+Flush attack to high accuracy levels of around 95% in single and multiple characters lookup, irrespective of noise levels, making the attacks highly feasible on a real system with high system noise. DABANGG+Flush+Reload also provides better accuracy than the standard Flush+Reload attack with a moderate boost in accuracy across noise levels. We now move on to a real cryptosystem to evaluate our attacks.

7.2 AES Key Extraction in OpenSSL

The OpenSSL [2] library no longer uses the T-Table based implementation of AES, as it is known to be susceptible to cache side-channel attacks. The T-Table based implementation exists to compare new and existing side-channel attacks. We build the library from source and enable this implementation through configuration options.

We briefly explain the T-Table based implementation of AES [27]. Eight pre-computed lookup tables exist in T-Table based implementation of AES, T_0 to T_3, and $T_0^{(10)}$ to $T_3^{(10)}$. Each lookup table contains 256 4-byte words. A 16-byte secret key k is expanded into 10 round keys, $K^{(r)}, \forall r \in [1,10]$, each of which is divided into 4 words of 4-bytes each. Given a 16-byte plaintext p, the encryption computes an intermediate state, $x^{(r)} = (x_0^{(r)},..,x_{15}^{(r)})$ at every r. The $x^{(r)}$ is initialized as $x_i^{(0)} = p_i \oplus k_i \forall i \in [0,15]$ The calculation of $x^{(r)}$ requires access to $T_i, i \in [0,3] \forall r \in [1,9]$, and that for $r = 10$ requires access to $T_i^{(10)}, i \in [0,3]$. The $x^{(10)}$ obtained at the end of 10^{th} round is the ciphertext, c.

Threat model: We mount an asynchronous attack, where the victim finishes execution before the attacker evaluates the memory addresses. The average execution time of `AES_Encrypt` is 750 cycles, too small a window for parallel execution of an attacker program. We monitor the first memory address of $T_i^{(10)}, i \in [0,3]$. Since this is a *known ciphertext attack*, we only need to flush one cache line before every encryption, without requiring the plaintext. This provides us with the reload-frequency of the ciphertext (c) bytes, $(c_0,..,c_{15})$. We then determine the correct secret key (k) bytes. The algorithm for ciphertext determination and consequent key determination is outlined by G. Irazoqui *et al* [15]. The parameters specific to this attack are specified in Table 9. We

Attack	L-L-L	L-L-H	L-H-L	L-H-H	H-L-L	H-L-H	H-H-L	H-H-H
F+F	37.2%	21.1%	31.4%	16.7%	36.4%	27.2%	19.7%	34.6%
D+F+F	**94.5%**	**92%**	**94.1%**	**92.2%**	**95.4%**	**94.6%**	**93.2%**	**96.7%**
F+R	84.2%	69.3%	74.9%	82.5%	85.1%	75.4%	71.6%	78.2%
D+F+R	**99.6%**	**91.2%**	**97.2%**	**96.5%**	**98.5%**	**97.2%**	**99.2%**	**98.1%**

Table 8: Accuracy of various flush based attacks on multiple character key-logging.

Parameter	D+F+F	D+F+R
acc_interval, burst_seq, and burst_wait	0	0
burst_gap and regular_gap	400	400

Table 9: Parameters for AES attack.

do not need to monitor any burst-mode sequences since this is an asynchronous attack. We aim to minimize the number of `AES_Encrypt` function calls, that perform the 10 AES rounds. We intend to recover the full 128 bit private key with a reasonably high prediction accuracy of $\geq$90%. We again use the Levenshtein distance to determine accuracy over 1000 runs of the standard attacks- Flush+Flush and Flush+Reload, and their DABANGG refinements. We vary the number of `AES_Encrypt` function calls, each on randomly generated plaintext and the same secret key, from 10^2 to 4×10^5 function calls for the attacks.

Figures 10(b) and 10(d) quantify time-domain boost achieved by integrating **refinements #1 and #2** (refer Section 6) to the standard Flush+Reload attack. The Flush+Reload attack achieves an average accuracy of $\geq$90% at the 100,000 encryptions. The same threshold of accuracy is met by DABANGG+Flush+Reload attack at 20,000 encryption mark. This is a 5$\times$ improvement. We achieve this due to the dynamic thresholds, which distinguish a `reload` cache-hit from a cache-miss accurately early on when the frequency isn't stable. The lower number of encryptions required also increases the stealth of Flush+Reload attack. If software countermeasures are implemented to flag concentrated calls to `AES_Encrypt` within a short period, DABANGG+Flush+Reload is much more likely to evade detection.

Figures 10(a) and 10(c) illustrate the much quicker rise in accuracy as a function of the number of encryptions by integrating **refinement #3** to the standard Flush+Flush attack. The key reason behind the effectiveness of employing the compute-intensive loop is the extremely variable latency of `clflush` instruction. Instead of wasting compute cycles to determine a proper threshold corresponding to the encryption iteration number, we simply insert the compute-intensive code to increase the processor frequency sufficiently, thereby stabilizing the execution latency of `clflush` cache-hit and miss, thus improving the true-positive detection rate.

While the number of `AES_Encrypt` function calls is higher than Flush+Reload attack for both variants of Flush+Flush attack, the DABANGG+Flush+Flush attack achieves 90% accuracy in 200,000 encryptions, twice as quick than the 400,000 encryptions required for Flush+Flush. DABANGG+Flush+Flush attack also produces a decent accuracy of more than 50% on average at the 15,000 encryption mark, far lower than 100,000+ encryptions required by the Flush+Flush attack. We again see a stealthier attack that is more likely to evade detection due to a lesser number of calls to the encryption function.

7.3 RSA Private Key Extraction in GnuPG

The RSA implementation in GnuPG [1] library uses an asymmetric public and private key model. We attack the vulnerable Square-and-Multiply Exponentiation (SME) algorithm in RSA. The SME algorithm is no longer the default implementation, however it is useful to compare attacks and detection techniques. RSA generates a public key using an exponent e (=65537, chosen by GnuPG) and a product, n, of two large primes, say p and q. The private key consists of the p, q and a private exponent d, given by $d = e^{-1}(mod(p-1)(q-1))$. The encryption function E for plaintext p is $E(p) = p^e \ mod \ n$ and the decryption function D for ciphertext c is $D(c) = c^d \ mod \ n$.

Threat Model: We mount a synchronous attack to extract the private key by recovering the private exponent, d, during decryption[4]. The SME algorithm computes $c^d \ mod(n)$ by scanning the bits of d. It then uses a sequence of Square (S), Multiply (M), and Module Reduce (R) operations. A sequence of Square-Reduce-Multiple-Reduce (S-R-M-R) denotes a set bit (1) and occurrence of Square-Reduce-Square (S-R-S) denotes a clear bit (0) of the private exponent d. The details of the threat model are same as [29]. We monitor the memory addresses in `mpih_sqr_n()`, `mul_n()`, and `mpihelp_divrem()` functions that respectively carry out S, M, and R operations. We then compare the extracted bits against the actual bit sequence. We make use of the Levenshtein distance to determine the accuracy of the attacks. We monitor 3 memory addresses in this attack, the parameters for addresses corresponding to S and M operations are same as AES attack (Table 9) except for `burst_gap` and `regular_gap`, which are set to 20. The parameters for R operation is same as that for covert channel attack (Table 10) except for `acc_interval`, which is set to 0.

Figure 11 illustrates the more than 6$\times$ accuracy improve-

[4]The CRT-RSA optimization allows us to only extract $d_p = d \ mod(p-1)$ and $d_q = d \ mod(q-1)$. However, d_p and d_q are sufficient to factor n and break the encryption [7].

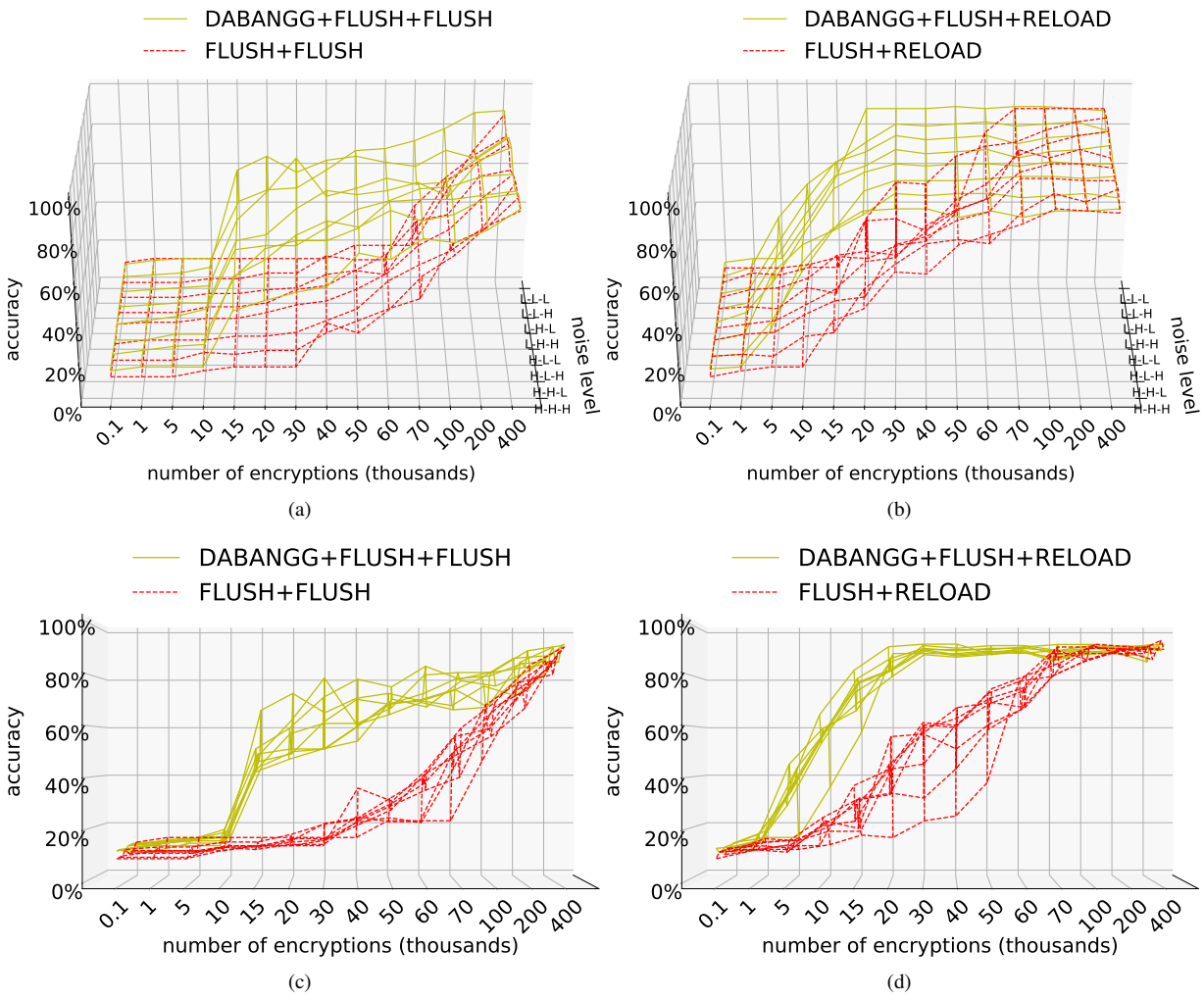


Figure 10: Accuracy comparison of Flush+Reload, Flush+Flush, DABANGG+Flush+Reload, and DABANGG+Flush+Flush attacks. (a) and (b) show the accuracy for different number of encryptions at various noise levels. (c) and (d) show the accuracy with different number of encryptions and vertical spread of curves of a particular attack gives the range of accuracy at the given number of encryptions.

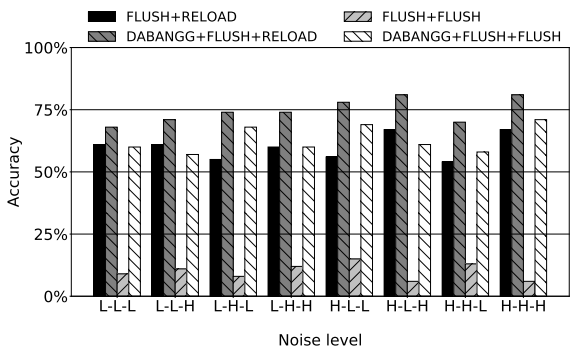


Figure 11: Accuracy of different attacks on RSA private key extraction in GnuPG at various noise levels.

Parameter	acc_interval	burst_seq	burst_wait	burst_gap	regular_gap
D+F+F	10	2	1	5	20
D+F+R	10	2	1	5	20

Table 10: Parameters for covert channel attack.

ment in D+F+F attack over the F+F attack which has an average accuracy of 10%. Practically, DABANGG optimizations make the F+F attack possible as the private key extraction isn't feasible with 10% average accuracy. F+R maintains respectable accuracy of over 60% throughout the tests. D+F+R attack performs 5% better on average compared to F+R. Flush+Flush attack improves significantly due to feedback mechanism of Algorithm 3 and **refinement #1**, which captures the `clflush` latency variation to allow accurate cache hit/ miss determination. The major gain for Flush+Reload attack is through **refinement #3** which allows better control over the waiting period of the attack loop.

7.4 Covert Channel Attack

Threat model: We use a sender-receiver model where the sender core sends a bit-stream through a socket, which is monitored by the receiver using a flush-based covert channel. The presence of the cache line corresponding to the memory address of the socket is interpreted as a set bit by the receiver, while the lack of such a cache line is interpreted as a reset bit. Thus, a covert communication channel is established without any explicit link between the programs. It must be noted that the socket does not establish any direct connection between the programs, and is used by the sender to send the bit-stream. The size of the bit-stream is fixed at 1000 bytes for our experiment. Table 10 shows the parameters of interest.

Figure 12 illustrates the error rate of these attacks at various noise levels. We also plot the bandwidth of different attacks in Figure 13. The bandwidth increases as the average core frequency (that is, compute or memory-intensive noise level) increases. We obtain a peak bandwidth of 217 KBps using the DABANGG+Flush+Reload attack, with an overall error-rate of 0.01%. This is enough to transfer a decently large image file within a second. While bandwidth increases

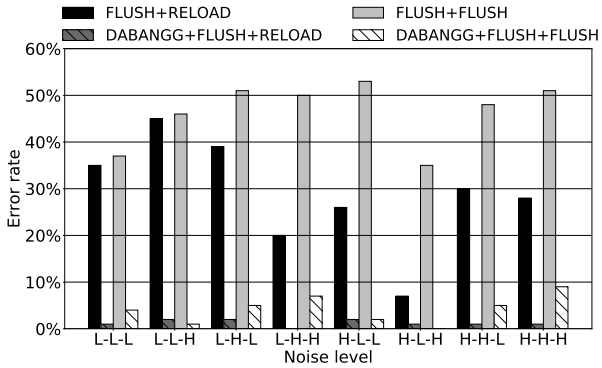


Figure 12: Error rates of different attacks in covert channel scenario at various noise levels.

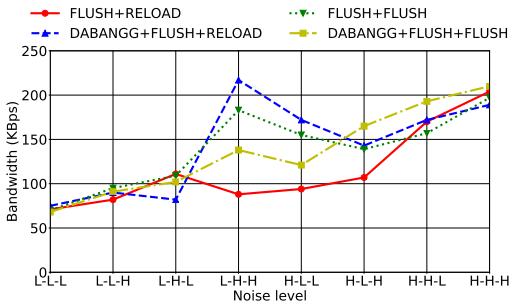


Figure 13: Bandwidth of different attacks in covert channel scenario at various noise levels.

across the board as noise levels increase, a consistent low error rate is crucial for the practical feasibility of the covert channel, which is provided by the DABANGG refinements. The bandwidth increases at higher noise levels (that is, L-H-H, H-L-H, H-H-L, and H-H-H levels) because all core of our PCPS-enabled processor run at high frequency at these noise levels (refer Section 3 for details). This allows the programs to send and receive more bits per second due to decreased execution latency (refer to 5.2 for details).

7.5 Transient Execution Attack

Spectre [16] is a transient-execution attack that relies on the microarchitectural covert-channels and exploits speculative execution.

```
1  if(index < array_size):
2      access(array[index])
```

Listing 1: Target code segment for Spectre attack.

Consider the code segment in Listing 1. If an `index` > `arr_size`, we expect the program to not execute line no. 2, since the branch at line no. 1 resolves to not be taken. However, modern processors may speculatively execute instruction 2, resulting in the data element `array[index]` being cached. In Spectre attack, the cache is usually profiled us-

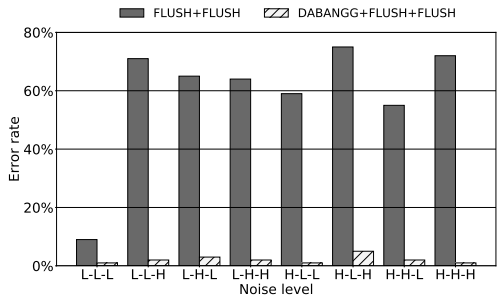


Figure 14: Error rates of Flush+Flush and DABANGG+Flush+Flush attacks for Spectre attack at various noise levels.

ing the Flush+Reload attack. Flush+Flush attack is rarely employed to mount the Spectre attack in particular and transient execution attacks in general due to its low accuracy. We, therefore, focus on the Flush+Flush attack in this experiment. While we omit the details of performing Flush+Reload attack for brevity, the Flush+Reload attack produces an average error rate of 7.4% (ranging from 13% to 2%), and the DABANGG+Flush+Reload attack pushes it to an average error-rate of 1.9% (ranging from 3% to 1%).

Threat model: We mount an asynchronous attack. The data segment of the program stores a 160 bytes long secret-character array. We maintain an attacker array in the data segment. We manipulate the index of attacker array to bring the secret array's data speculatively to the cache. We access an out-of-bounds index of the attacker array, once in five legal accesses. We also flush all elements of the attacker array, and the variable containing its size, to increase the transient execution window. We infer secret array's data by profiling the cache for hits after each instance of speculative execution using Flush+Flush attack. If a data element in the secret array is already present in the cache, it registers a `clflush` instruction hit when we speculatively access the location using attacker array, indicating the presence of secret data at the element's address. The base code is optimized to provide the most likely outcome for each secret character. The parameters for this experiment are same as AES attack (Table 9).

We conduct 1000 runs of each experiment, the result of which is the inferred secret array of characters. We then use Levenshtein distance to determine the accuracy of the attack by comparing it against the real secret character array. The profiling phase is done using both the standard Flush+Flush attack and the DABANGG+Flush+Flush attack. The principle refinement for this attack is **refinement #3**. It steps up the processor frequency.

Figure 14 shows the error rates of the attack at various noise levels. The DABANGG refinements significantly improve the error rate by stabilizing the core at high frequency and eliminating the false positives. As a result, at very high noise levels (HHH), the error-rate drops significantly from 72% in the standard Flush+Flush attack to 1% in DABANGG+Flush+Flush attack. Relatively high accuracy is achieved using Flush+Flush attack at noise levels, which ramp up the processor frequency. Error rate suffers at noise levels that do not let the processor frequency stabilize, like I/O intensive noise. DABANGG+Flush+Flush eliminates the difference in processor frequency using compute-intensive code segment, thereby producing a uniformly low error rate of less than 10%.

7.6 Cross-VM Attacks

VM environment introduces some challenges that are not related to system noise. For example, memory de-duplication needs to be enabled to carry out cross-VM flush-based attacks. Nonetheless, to check the effectiveness of DABANGG optimizations (with memory de-duplication enabled), we mount a cross-VM AES attack using a TCP-based server-client model outlined in [15]. We use Windows 10 (v. 1903) as host OS and 2 VirtualBox (v. 6.1) VMs running Ubuntu 18.04 as guests. The parameters for this attack are listed in Table 9. On average, D+F+R attack requires 31% less encryptions than F+R attack, which requires 10^6 encryptions, to achieve $\geq$ 90% accuracy. Despite our best efforts, F+F and D+F+F attacks stagnate at 27% and 51% accuracy, respectively, after 10^7 encryptions. To the best of our knowledge, cross-VM Flush+Flush attack hasn't been successfully mounted yet and exploring Flush+Flush attack on a cross-VM setup is a promising future direction.

A cross-VM keylogging attack is feasible by implementing a shared library instead of a victim program which is used by both victim and attacker. The effectiveness of DABANGG enhancements remain equally effective with cross-VM covert channel. An attack on GnuPG library is possible using a technique demonstrated in [29] or by using the client-server model similar to our cross-VM AES attack. A cross-core Spectre attack was recently demonstrated in [25]. Demonstrating practical cross-VM transient execution attacks remains an attractive research area in the security community.

8 Mitigation & Detection Techniques

As DABANGG refined flush attacks are fundamentally flush based attacks, all the mitigation and detection techniques discussed in Flush+Reload [29] and Flush+Flush [10] that are applicable to flush based attacks [9,23,30], are also applicable to DABANGG refined attacks. From the Operating System's view, the only significant difference in DABANGG-enabled attacks compared to standard attacks is the increased CPU utilization of the attacker thread since we don't yield the CPU often. Many workloads have high CPU utilization, but the OS can potentially use other indicators (such as performance counters used to detect Flush+Reload attack) to pinpoint the attacker thread with more confidence.

9 Conclusion

In this paper, we uncovered the dependence of the accuracy of flush based attacks on the threshold set to distinguish

a cache hit from a miss. We showcase that dynamic core frequencies, induced by system noise due to Dynamic Voltage and Frequency Scaling (DVFS) Power Governors, result in varying `clflush` and `reload` instruction latencies. We also reveal the change in latency due to the relative positioning of attacker and victim programs on CPU cores. To make flush based attacks resilient to frequency changes and therefore system noise, we proposed a set of three refinements, and named it as DABANGG refinements, over existing Flush+Flush and Flush+Reload attacks. We outline the algorithm so that the attack loop can dynamically change the thresholds and employ a dynamic busy-waiting period. We also take into account the victim-specific parameters in our algorithm. We tested DABANGG-enabled attacks in five experiments: (i) side-channel based keylogging, (ii) AES key extraction, (iii) RSA private key extraction, (iv) covert channel, and (v) Spectre attack, and showed the effectiveness across different system noise levels. The improved, noise resilient DABANGG-enabled attacks pose a significant challenge to the micro-architectural security community. DABANGG-enabled attacks have all of the perks of flush based attacks while being significantly more accurate and precise, making the flush based attacks more practical.

Acknowledgments

We would like to thank members of CAR3S Group especially Aditya Rohan, Chavhan Sujeet Yashavant, and Dixit Kumar. We also would like to thank Vinod Ganesan, Rahul Bodduna, and Clementine Maurice for their feedback on the draft. This work is supported by the SRC grant SRC-2853.001.

Availability

Upon public disclosure of DABANGG attack, Intel coordinated with us and has acknowledged the vulnerability. A GitHub repository with the source code is available at https://github.com/DABANGG-Attack/Source-Code.

References

[1] Gnupg. https://gnupg.org/.

[2] Openssl. http://www.openssl.org.

[3] Amd powernow! technology- informational white paper, 2000. https://www.amd.com/system/files/TechDocs/24404a.pdf.

[4] Frequently asked questions about enhanced intel speedstep technology for intel processors, 2019. https://www.intel.in/content/www/in/en/support/articles/000007073/processors.html.

[5] Amd turbo core technology, 2020. https://www.amd.com/en/technologies/turbo-core.

[6] 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.

[7] Matthew Campagna and Amit Sethi. Key recovery method for crt implementation of rsa. *IACR Cryptology ePrint Archive*, 2004:147, 01 2004.

[8] Avinash Goud Chekkilla. Monitoring and analysis of cpu utilization, disk throughput and latency in servers running cassandra database: An experimental investigation, 2017.

[9] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *IACR Cryptology ePrint Archive*, 2015:1034, 2015.

[10] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[11] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 897–912, 2015.

[12] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IPDPSW '15, pages 896–904, Washington, DC, USA, 2015. IEEE Computer Society.

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018. April 2018.

[14] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. March 2018.

[15] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In *International Workshop on Recent Advances in Intrusion Detection*, pages 299–319. Springer, 2014.

[16] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre

attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[17] Nate Lawson. Side-channel attacks on cryptographic software. *IEEE Security & Privacy*, 7(6):65–68, 2009.

[18] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady. Vol. 10. No. 8*, pages 707–710, 1966.

[19] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[20] Dr. Andreas Löbel. Spec 2017 benchmark description, 2019. https://www.spec.org/cpu2017/Docs/benchmarks/505.mcf_r.html.

[21] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[22] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[23] Mathias Payer. Hexpads: A platform to detect "stealth" attacks. In *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, pages 138–154, 2016.

[24] Rafael J. Wysocki. Cpu performance scaling - the linux kernel, 2017. https://www.kernel.org/doc/html/v4.15/admin-guide/pm/cpufreq.html.

[25] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*, May 2021. Intel Bounty Reward.

[26] Subramaniam M. Salvador P., Stephen A. F. Clflush micro-architectural implementation method and system, 1999. https://patents.google.com/patent/US6546462B1.

[27] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[28] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[29] Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[30] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE Symposium on Security and Privacy*, pages 313–328, 2011.