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 four 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 covert-channel, and a (iv) transient execution attack in the form the Spectre attack. For all the scenarios, DABANGG+Flush+Reload and DABANGG+Flush+Flush outperform the standard Flush+Reload and Flush+Flush attacks in terms of F₁ 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 [23] and Flush+Flush [6] and (ii) eviction based attacks [13], [7], [18]. 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 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 F₁ 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 F₁ 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 trend is that with the system noise, existing latency calibration mech-
different physical cores

To understand the subtleties, we perform the

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. We sleep for one second (using sleep() system call) on every 100 thousandth iteration resulting 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) [22] 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 reduce 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 Figure 2). The reload hit latency when the two threads run on the same 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 it 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).

---

1DABANGG is a Hindi word that means “fearless”. We envision DABANGG 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 [10], a `clflush` instruction does the following: “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.” [10]. 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 stealthier than Flush+Reload attack.

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 [19]. Conventionally, frequency is the variable which is actively manipulated by the hardware 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 [2], and AMD uses Cool’n’Quiet and PowerNow! [1] 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 [9] (and analogously, AMD’s Turbo Core [3]) 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 [8]. 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 [5] 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 phys-
tical 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) at increment at the same rate as the processor clock and (ii) at increment at a rate independent of the processor clock. Modern Intel processors use the latter mode. 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 [23].

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 [16] 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 memory-intensive, and eight memory-I/O 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 [12]. We use metrics like True Positive Rate (TPR), False Positive Rate (FPR), accuracy, and the F1 score to evaluate the effectiveness of various flush based attacks.

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. In contrast, the victim processes a character (refer Section 7.1 for details). Even at the L-L-L noise level, the standard Flush+Reload attack provides poor TPR of 4.4%. The TPR does not improve with an increase in noise levels. Flush+Reload attack performs appreciably with high TPR of more than 94% at all noise levels.

One might be tempted to write off the Flush+Flush attack as inferior to the Flush+Reload attack, but that is not always the case. In our asynchronous AES secret key

Table 1: System configuration for our experiments.

<table>
<thead>
<tr>
<th>Noise Level</th>
<th>stress</th>
<th>mcf</th>
<th>stress</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-M-I</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L-L-L</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L-L-H</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>L-H-L</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>L-H-H</td>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>H-L-L</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H-L-H</td>
<td>8</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>H-H-L</td>
<td>8</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>H-H-H</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

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/I0 intensive code.
We now uncover the root cause, which guides us to the enabling the lowest latency in execution) if the code and data (refer Section 7.4 for details), the accuracy of Flush+Flush. The processor does not step up to the maximum frequency variation in accuracy. In our Spectre attack experiment our threshold. Every clflush iterations plot. An important point to be noted is that a single in frequency gives rise to the distinct steps in the latency vs. footprint of the program is minimal. The stepped increase slowly as the attacker code iterates through the attack loop.

Figure 4(a) shows the latency of Flush+Flush 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.

5 Analysis & Insights

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 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 [6], 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.
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 [15]. At 1.2 GHz, a clflush cache miss has a latency of 1,500 cycles compared to 575 cycles when all the cores run at 4.5 GHz (refer 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 \( \frac{1,500t}{575t} \approx 2.6 \times \). However, the frequency of the core increases by \( \frac{4.5 \text{ GHz}}{1.2 \text{ GHz}} \approx 3.7 \times \). 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.
Victim attacker gets a reload hit victim runs on C attacker misses during reload attacker gets a reload hit attacker gets a reload hit attacker gets a reload hit attacker gets a reload hit No victim access Attacker’s Reload victim runs on C victim runs on C victim runs on C victim runs on C, C, C victim runs on C, C, C, C

1.6 3.2 2.2 2 2.4 4 4.5 3.8 4.2 2.6 3.4 3.6 4.4 1.4 3 2.8 1.8

Quadratic approximation, \( R^2 = 0.987 \)

Table 3: Different combinations of attacker and victim on a four-core SMT system. The attacker runs on core-0 (C₀).

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Victim</th>
<th>Attacker’s Reload</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No victim access</td>
<td>attacker misses during reload</td>
</tr>
<tr>
<td>2</td>
<td>victim runs on C₀</td>
<td>attacker gets a reload hit</td>
</tr>
<tr>
<td>3</td>
<td>victim runs on C₁</td>
<td>attacker gets a reload hit</td>
</tr>
<tr>
<td>4</td>
<td>victim runs on C₀, C₁</td>
<td>attacker gets a reload hit</td>
</tr>
<tr>
<td>5</td>
<td>victim runs on C₁, C₂ and C₃</td>
<td>attacker gets a reload hit</td>
</tr>
<tr>
<td>6</td>
<td>victim runs on C₀, C₁, C₂, C₃</td>
<td>attacker gets a reload hit</td>
</tr>
</tbody>
</table>

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. Based on Figure 5, we can see at 1.2 GHz. 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 and attacker reside on different physical 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.

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 [20]. 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 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 [4] 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 ≈ 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.

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 attacked 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

---

**Algorithm 1: Compute_Heavy_Code**

```
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) = (\frac{a+b}{2}, \frac{a+b}{5})
6       if (wait_gap \equiv 0) then
7          (a, b) = (0.11 \times b, 23 \times a)
8       else if (wait_gap \equiv 0) then
9          (a, b) = (b \% 3, a \% 14)
10      else
11         (a, b) = (2 \times b, 3.6 \times a)
```

---

2The numbers in Algorithm 1 do not serve any special purpose. The aim is to use the CPU’s compute units.
### Table 4: Specifications of parameters and runtime variables used by DABANGG attack loop (refer Algorithm 2).

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attacker-Specific</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>f_array</td>
<td>An array with each entry stores a tuple of lower and upper latency thresholds &lt;TL, TH&gt;.</td>
</tr>
<tr>
<td></td>
<td>regular_gap</td>
<td>Regular waiting period of attacker in Phase II.</td>
</tr>
<tr>
<td></td>
<td>step_width</td>
<td>Average width of a step in terms of number of attack loop iterations in latency vs #iterations plot.</td>
</tr>
<tr>
<td><strong>Victim-Specific</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>acc_interval</td>
<td>Average number of attack loop iterations between two victim accesses without considering burst-mode accesses in between.</td>
</tr>
<tr>
<td></td>
<td>burst_seq</td>
<td>In case of burst-mode access sequence by victim, number of victim accesses to target memory address in a single burst.</td>
</tr>
<tr>
<td></td>
<td>burst_wait</td>
<td>Waiting time gap in terms of attack loop iterations before discarding an incomplete burst-mode access sequence as a false positive.</td>
</tr>
<tr>
<td></td>
<td>burst_gap</td>
<td>Reduced waiting time gap to monitor burst-mode access sequence.</td>
</tr>
<tr>
<td><strong>Runtime Variables in Algorithm 2</strong></td>
<td>iter_num</td>
<td>A counter that counts the number of attack loop iterations.</td>
</tr>
<tr>
<td></td>
<td>&lt;TL, TH&gt;</td>
<td>Pair of lower (TL) and upper (TH) latency threshold to detect cache hit.</td>
</tr>
<tr>
<td></td>
<td>reload_latency</td>
<td>Execution latency of reload instruction in processor cycles.</td>
</tr>
<tr>
<td></td>
<td>last_hit</td>
<td>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 ∈ [TL, TH].</td>
</tr>
<tr>
<td></td>
<td>potential_hit</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>seq_id</td>
<td>Sequence identifier, stores the number of potential cache hits which, if it forms a burst-mode access sequence, implies a true cache hit.</td>
</tr>
</tbody>
</table>

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

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 ∈ [0,4000), we use TL = 375 and TH = 400. 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. regular_gap = 200 provides a waiting period of 5,000 to 10,000 cycles (refer to 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 20,000 cycles at low processor frequency, then acc_interval = \( \frac{1,000,000}{20,000} = 50 \). A burst-mode access sequence occurs when the target address is inside a loop and gets accessed several times within a small interval. Consider that the victim accesses the address 50 times within 40,000 cycles, for example. Then, burst_seq 50 and waiting period when a burst is detected should be \( \leq \frac{40,000}{50} = 800 \) cycles. This implies a burst_gap 20, which increases attack granularity to identify the burst-access sequence. In practice, to reduce false negatives, we tolerate some missed cache-hits to determine the sequence, burst_seq = \( \frac{50}{20} \) and burst_wait = x where x is relatively small number compared to 50. For example, burst_seq = 25 for burst_wait = 2.

### 6.2 Attack Loop
A self-contained Algorithm 2 explains the DABANGG refinement. 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, 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 ∆ ∈ [TL,TH], the function returns without making any changes. However, if ∆ ∉ [TL,TH] (Line 4), then the tuple is updated. This is done by looking up T_array such that ∆ ∈ [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. 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 pre-
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

---

**Table 5: Attacker-specific parameters**

<table>
<thead>
<tr>
<th>Attack</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F+F+R</td>
<td>T_array</td>
<td>[&lt;375,400&gt;,&lt;235,255&gt;,&lt;235,255&gt;,&lt;150,165&gt;,&lt;95,105&gt;]</td>
</tr>
<tr>
<td></td>
<td>step_width</td>
<td>4000</td>
</tr>
<tr>
<td>D+F+R</td>
<td>T_array</td>
<td>[&lt;375,400&gt;,&lt;235,255&gt;,&lt;150,165&gt;,&lt;95,105&gt;]</td>
</tr>
<tr>
<td></td>
<td>step_width</td>
<td>4000</td>
</tr>
</tbody>
</table>

---

**ALGORITHM 2: DABANGG+FLUSH+RELOAD**

1. **Initialization:** last_hit, potential_hit, iter_num, seq_id = 0
2. **while** true **do**
3.     iter_num += 1
5.     clflush(addr) // PHASE-I: Flush
6.     // PHASE-II: Wait
7.     if ((rand()%400) then // branch taken 0.25% of time
8.         Verify_Threshold(iter_num, addr) // Algorithm 3
9.         sched_yield(); // cooperatively yield the CPU
10.    else if (seq_id > 0) then // burst sequence detected
11.        Compute_Heavy_Code(burst_gap) // Algorithm 1
12.    else
13.        Compute_Heavy_Code(regular_gap) // Algorithm 1
14.    end
15.    // PHASE-III: Reload
16.    reload_latency = Measure_Reload_Latency(addr)
17.    if (reload_latency ∈ [TL, TH]) and (last_hit > acc_interval)
18.        and (seq_id > burst_seq) then // true hit
19.        last_hit, seq_id = 0 // reset variables
20.    print "low reload latency, it is a cache hit!"
21.    else if (reload_latency ∈ [TL, TH]) then // potential hit
22.        potential_hit = last_hit
23.        seq_id += 1 // increment sequence identifier
24.    else
25.        last_hit += 1 // +1 iteration since last hit
26.        print "high reload latency, it is a cache miss!"
27.    if ((last_hit - potential_hit) > burst_wait) then
28.        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
5.     // T_array[i].TL < Δ < T_array[i].TH
7.     <TL, TH> = <TLnew, THnew>
8. iter_num = step_width × i
9. end
Table 6: Parameters for keylogging attack.

<table>
<thead>
<tr>
<th>Lookup Type</th>
<th>Parameter</th>
<th>D+F+F</th>
<th>D+F+R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single &amp; Multiple</td>
<td>acc_interval</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Characters</td>
<td>burst_seq</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>burst_wait</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>burst_gap</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>Single Character</td>
<td>regular_gap</td>
<td>400</td>
<td>200</td>
</tr>
<tr>
<td>Multiple Characters</td>
<td>regular_gap</td>
<td>100</td>
<td>50</td>
</tr>
</tbody>
</table>

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 $\text{regular_gap}_{\text{multiple_chars}} = \frac{\text{regular_gap}}{n}$. 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, $T_{\text{real}}$, 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 $T_{\text{real}}$) 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 to Section 3 for CPU utilization at various noise levels.

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.

At all noise levels, the DABANGG+Flush+Flush attack accurately determines True Positives and maintains the 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
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 ≈ 10^3` 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, which corresponds to `CHANGE_AFFINITY_FACTOR` of around 10^4. However, we test for `CHANGE_AFFINITY_FACTOR` ranging from 10^0 (≈10 microseconds) to 10^8 (=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 $A_i$, and monitor all the addresses iteratively. A hit in the address is interpreted through its label. We utilize the Levenshtein distance (Lev) algorithm [14] 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 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 [21]. 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, $x(r) = (x_0(r), \ldots, x_{15}(r))$ at every $r$. The $x(r)$ is initialized as $x_0(r) = p_i \oplus k_i, \forall i \in [0, 15]$. 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 10th 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 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 ($c_i, \ldots, c_{15}$). We then determine the correct secret key ($k_i$) bytes. The algorithm for ciphertext determination and consequent key determination is outlined by G. Irazoqui et al [11]. The parameters specific to this attack are specified in Table 9. We 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 $\geq90\%$. 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 $4 \times 10^5$ 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 $\geq90\%$ 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. DA-
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.
Table 8: Accuracy of various flush based attacks on multiple character key-logging.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F+F</td>
<td>37.2%</td>
<td>21.1%</td>
<td>31.4%</td>
<td>16.7%</td>
<td>36.4%</td>
<td>37.2%</td>
<td>19.7%</td>
<td>34.6%</td>
</tr>
<tr>
<td>D+F+F</td>
<td>94.5%</td>
<td>92%</td>
<td>94.1%</td>
<td>92.2%</td>
<td>95.4%</td>
<td>94.6%</td>
<td>93.2%</td>
<td>96.7%</td>
</tr>
<tr>
<td>F+R</td>
<td>84.2%</td>
<td>69.3%</td>
<td>74.9%</td>
<td>82.5%</td>
<td>85.1%</td>
<td>75.4%</td>
<td>71.6%</td>
<td>78.2%</td>
</tr>
<tr>
<td>D+F+R</td>
<td>99.6%</td>
<td>91.2%</td>
<td>97.2%</td>
<td>96.5%</td>
<td>98.3%</td>
<td>97.2%</td>
<td>99.2%</td>
<td>98.1%</td>
</tr>
</tbody>
</table>

Table 9: Parameters for AES attack.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>D+F+F</th>
<th>D+F+R</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc_interval, burst_seq, and burst_wait</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>burst_gap and regular_gap</td>
<td>400</td>
<td>400</td>
</tr>
</tbody>
</table>

Table 10: Parameters for covert channel attack.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>acc_interval</th>
<th>burst_seq</th>
<th>burst_wait</th>
<th>burst_gap</th>
<th>regular_gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>D+F+F</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>D+F+R</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>

BANGG+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 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 11 illustrates the error rate of these attacks at various noise levels. We also plot the bandwidth of different attacks in Figure 12. 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 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 to 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).

7.4 Transient Execution Attack

Spectre [12] is a transient-execution attack that relies on the microarchitectural covert-channels and exploits speculative execution.

```java
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
We, therefore, focus on the Flush+Flush attack in this experiment which is the inferred secret array of characters. We then use various noise levels to improve the error rate by stabilizing the core at high frequencies 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%.

8 Mitigation Techniques

As DABANGG refined flush attacks are fundamentally flush based attacks. All the mitigation techniques discussed in Flush+Reload \[23\] and Flush+Flush \[6\] that are applicable to flush based attacks, are also applicable to DABANGG refined attacks.

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 on four experiments: (i) side-channel based keylogging, (ii) AES secret key extraction, (iii) covert channel, and (iv) 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 CARS research group especially Aditya Rohan, Chavhan Sujeet Yashavant, 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.

References


Figure 13: Error rates of Flush+Flush and DABANGG+Flush+Flush attacks for Spectre attack at various noise levels.


