Plug Your Volt: Protecting Intel Processors against Dynamic Voltage Frequency Scaling based Fault Attacks

Nimish Mishra
Indian Institute of Technology Kharagpur, India
nimish.mishra@kgpian.iitkgp.ac.in

Rahul Arvind Mool
Indian Institute of Technology Kharagpur, India
rahulmool@kgpian.iitkgp.ac.in

Anirban Chakraborty
Indian Institute of Technology Kharagpur, India
anirban.chakraborty@iitkgp.ac.in

Debdeep Mukhopadhyay
Indian Institute of Technology Kharagpur, India
debdeep@cse.iitkgp.ac.in

ABSTRACT

Existing countermeasures to DVFS based fault attacks are overly restrictive because (1) they prevent benign, non-SGX processes from utilizing DVFS, and (2) rely upon a *less* practical threat model than that of Intel SGX. Consequently, this work proposes a new countermeasure principle to defend against DVFS based fault attacks on modern Intel systems. First, we establish that the fundamental cause of DVFS fault attacks is the ability to *independently* control the frequency and voltage of a processor. Using this observation, we construct a partition of frequency-voltage tuples into `unsafe-safe` states based on whether a tuple causes timing violations according to switching circuit theoretic principles. Our countermeasure completely prevents DVFS faults on three Intel generation CPUs: Sky Lake, Kaby Lake R, and Comet Lake. Further, it can also be deployed both as microcode or as model-specific registers at the hardware level, unlike previous countermeasures. Our countermeasure incurs a slowdown of only 0.28% on overall system performance when benchmarked against `SPEC2017`.

CCS CONCEPTS

• **Security and privacy → Systems security**.

KEYWORDS

Dynamic Voltage Frequency Scaling (DVFS), Plundervolt, V0ltpwn

ACM Reference Format:
Nimish Mishra, Rahul Arvind Mool, Anirban Chakraborty, and Debdeep Mukhopadhyay. 2024. Plug Your Volt: Protecting Intel Processors against Dynamic Voltage Frequency Scaling based Fault Attacks. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658468

1 INTRODUCTION

Energy optimizations in modern systems are crucial since sub-par energy management decisions increase power consumption, and transitively, increase processor wear over a period of time. Consequently, most modern processors employ a number of sophisticated optimization techniques to maintain a balance between performance and energy consumption. While implementation aspects vary, all modern processor vendors tackle this problem by introducing a *spectrum* of processor energy consumption states, and introducing mechanisms to traverse this spectrum. At any point in time, the *state* of a processor is classified into either an *idle* state (otherwise generically named a **C**-state) or a *non-idle* state (otherwise generically named a **P**-state)[1].

To traverse the **P**-state spectrum, modern processors are shipped with a Dynamic Voltage Frequency Scaling (DVFS) interface. Ideally, a DVFS interface allows privileged software to throttle a CPU core's frequency through scaling drivers or through model-specific registers (MSRs). In light of this, a critical question arises: *what kind of attack vectors can be leveraged through **P**-state changes enforced by DVFS*? Prior works like [2, 5, 8] use the DVFS interface to introduce violations in the following required condition to stabilize output of a sequential circuitry [10]:

$$T_{src} + T_{prop} \leq T_{clk} - T_{setup} - T_{\epsilon} \tag{1}$$

where T_{src} denotes the time taken to produce unambiguous output for the *first* sequential element in the circuit. T_{prop} denotes the time taken for other combinational elements of the circuit to stabilize output. T_{setup} refers to the setup time of the sequential circuitry, while T_{ϵ} refers to small timing fluctuations in the system clock. Finally, T_{clk} refers to the time period of the synchronous clock pulse driving the circuitry. From an adversarial perspective, *undervolting* causes an increase in T_{src} and T_{prop} due to decreased voltage swings and slower transistor switching [1], while T_{clk}, T_{setup} and T_{ϵ} (being independent from any effects of voltage, and dependent solely on core frequency) remain unaffected. This causes a **timing violation** of the form $T_{src} + T_{prop} > T_{clk} - T_{setup} - T_{\epsilon}$, causing a digital circuit to produce incorrect output. All prior attacks [2, 5, 8] use such timing violations and subsequent incorrect outputs to influence critical operations, thereby successfully mounting purely software-based fault attacks.

So far, to the best of our knowledge, there have been two philosophies in literature countering these attacks on Intel processors: ① restricting access to overclocking mailbox (OCM) DVFS interface and the MSR `0x150` to Intel SGX remote attestation reports [2] and ② using compiler-induced *traps* to prevent the adversary from taking any perceivable advantage of DVFS faults (if any) [6]. However,

[1]Intel 64 and IA-32 Architectures Software Developer Manuals
[2]Intel Processors Voltage Settings Modification Advisory: Intel-SA-00289

these countermeasure principles have their own set of drawbacks. For instance, the ① *access control* based countermeasure greatly restricts any benign non-SGX context from using the CPU's power management features to the fullest (when a SGX context is operational on a shared hyperthread), thereby greatly impacting system performance [6]. On the other hand, the ② compiler based approach is not self-sufficient since it cannot independently stop practical SGX attack vectors like instruction isolation using single stepping [13]. This raises the following question:

*Is it possible to design a countermeasure against DVFS-based fault attacks that does not restrict a benign, non-SGX context to take full advantage of the entire **P**-state spectrum available, while at the same time easy to implement and with minimal performance overhead?*

In this work, we propose an alternative countermeasure philosophy against DVFS fault attacks that fixes the issues with the previous countermeasures. First, we establish the root cause of DVFS based attacks- *independent manipulations of core frequency and core voltage are responsible for putting the system in* `unsafe` *states where faults occur.* We then develop a countermeasure around this observation and enforce a functional mapping between core frequency and core voltage, which forces the system into always being in a `safe` state. This countermeasure principle provides the required flexibility to demanding applications to undervolt and overclock the cores while protecting them from DVFS-styled fault attacks. To summarize, we make the following contributions:

- We put forth a new countermeasure philosophy for DVFS by characterizing a victim system into **`safe`** and **`unsafe`** states, depending on how DVFS faults occur. Unlike prior works, our countermeasure plugs in the root-cause of DVFS- *causal independence* in controlling frequency/voltage of a core.
- We develop a **software-only** deployment of our countermeasure that is based *outside* any SGX context, thereby adhering to a **more robust and practical threat model** than prior works [4, 6]. Our experiments show that our countermeasure is able to completely prevent DVFS induced faults, while showing an acceptable overhead of 0.28%.
- Finally, the flexibility of our countermeasure design allows identification of **maximal** `safe` state for a given system. This allows our countermeasure to be potentially deployed as a ① microcode assist, or ② model-specific register (MSR) by respective CPU vendors. Consequently, our countermeasure has the ability to be implemented at a more fundamental level in the micro-architecture, thereby incurring lesser overhead than prior works like [6] relying on explicit software checks.

Organization. In Sec. 2, we first summarize DVFS interface and Intel SGX. Then in Sec. 3, we capture the root-cause of prior DVFS fault attacks, and put forth our countermeasure philosophy. We first provide a software implementation of our countermeasure in Sec. 4, and then discuss deployment of the countermeasure at the hardware level in Sec. 5. Finally we conclude in Sec. 6.

2 BACKGROUND

In this section, we provide the necessary background on Intel SGX, DVFS, and attack methodology by undervolting.

2.1 Intel SGX

Intel Software Guard Extensions (SGX) is a Trusted Execution Environment solution providing hardware-based partitioning of system resources into *enclaves*, which are considered secure even if the kernel is compromised. SGX ensures that the enclave's execution state and memory are inaccessible to ① the non-enclave portion of the process, ② all other processes in the system as well as ③ the operating system (OS).

A number of attacks [9] have been proposed in literature that undermine the security guarantees of SGX through side-channels on resources that the enclave shares with the OS (like page tables, process scheduling, interrupt handling, and so on). Likewise, transient execution attacks like Foreshadow [14] utilize co-residency on micro-architectural artifacts to leak sensitive data from enclaves.

2.2 Power Management (DVFS) in Intel

In most modern Intel processors, Dynamic Frequency and Voltage Scaling (DVFS) interface allows control of dynamic power consumption and is usually employed to maintain a delicate balance between energy consumption and performance. DVFS interfaces usually constitute different *scaling governors* corresponding to various performance demands. The range of permissible frequency values [3] are set by the processor vendor for optimal usage with flexibility for dynamic scaling. DVFS interfaces are exposed to the userspace applications by the OS. In context of SGX applications, where compromised kernel is an acceptable threat model, DVFS interface is also available through MSRs `0x150` (for voltage) and `0x198` (for frequency). Prior works [8, 11] mount DVFS fault attacks by writing negative voltage offsets to `0x150`, which in turn uses the Overclocking Mailbox interface (OCM) to scale a core's frequency/voltage, leading to timing violations (as per Eq. 1).

3 CHARACTERIZATION OF "SAFE" SYSTEM STATES

In this section, we develop the concept of `safe` states of a system. We first detail the different aspects of Eq. 1, and then use these aspects of violations of this inequality to define `safe`-`unsafe` states of the system. This classification of `safe`-`unsafe` states of the system is then used subsequently in the next section to develop and implement the countermeasure. Informally, we define what it means for a sequential element to be in a `safe` state.

`Safe` state of a sequential element. Informally, a sequential element i is defined to be in a `safe` state iff its output is stabilized by the time the subsequent sequential element $(i + 1)$ are driven by ① the clock and ② the output of i.

3.1 Establishing interplay of independent timing parameters

We first explain the different parameters involved in sequential digital circuitry (cf. Eq. 1) and their relative interplay that controls the output signal of such circuitry. In line with the prior works on DVFS fault attack [10, 12], we choose *flip-flops*, the basic sequential digital element, to define the context of `safe` and `unsafe` states. However, our definition of `safe`/`unsafe` states is directly extendable to more complex sequential units constructed as combinations of flip-flops. Referring to Fig. 1, the **objective** of tuning parameters of Eq. 1 is to *enforce flip-flop* `F1` *in a* `safe` *state.* As exemplified, we consider a circuit with a sequence of combinational logic, between

[3] Linux Kernel documentation: https://docs.kernel.org/cpu-freq/core.html

two sequential flip-flops F1 and F2 driven by the same clock of time period T_{clk} with **maximum** uncertainty T_ϵ. In this example, we use this over-arching parameter T_ϵ to denote the **maximum** of the immeasurable, transient variations in the arrival of the clock signal to the flip-flops. From a circuit design perspective, because of reasons like variations in the clock distribution network, spatial voltage and cycle-to-cycle variations in the loop distribution network and temporal/spatial jitter, the clock for F2 can arrive at any point in the closed time interval $[T_{clk} - T_\epsilon, T_{clk} + T_\epsilon]$. This leads us to make the first checkpoint observation:

O1: Handling unavoidable clock skewness. To ensure a safe flip-flop F1 state is to control the core frequency **f** such that the output of F1 is stable in time upper-bounded by $(\frac{1}{\mathbf{f}} - T_\epsilon)$. Evidently, $T_{clk} = \frac{1}{\mathbf{f}}$ in all subsequent discussions.

We now bring in another aspect: *setup* time of the sequential flip-flop F1, or the time for which D2 must be stabilized before the arrival of the clock edge (i.e. T_{setup} in cf. Fig. 1). In the worst case, the clock will arrive at F2 no later than $T_{clk} - T_\epsilon$, leading us to make the following observation atop observation **O1**.

O2: Handling setup delays atop clock skewness. To ensure a safe flip-flop F1 state when its output drives a sequential element F2, the core frequency **f** must be such that the output of F1 is stable in time upper-bounded by $(\frac{1}{\mathbf{f}} - T_\epsilon - T_{setup})$. Evidently, $T_{clk} = \frac{1}{\mathbf{f}}$ and T_{setup} is the setup time for F2.

Finally, from Fig. 1, we note that the input D2 is driven by application of combinational logic on output of Q1 (output of F1). According to Eq. 1 semantics, we refer to the time elapsed since driving of D1 to driving of D2 as summation of time taken for F1 to produce output Q2 (i.e. T_{src}) and the time taken for the combinational logic to operate on Q2 (i.e. T_{prop}).

3.2 Fundamental cause of DVFS fault attacks

We now use the observations made in Sec. 3.1 about safe state of F1 to establish the fundamental cause of DVFS based fault attack vectors. Concretely, a DVFS fault attack is successful when it forces the sequential flip-flop F1 (c.f. Fig. 1) into an unsafe state. We can define an unsafe state of F1 as:

$$T_{src} + T_{prop} > T_{clk} - T_{setup} - T_\epsilon \qquad (2)$$

Or the case where the input D2 is stable *after* the deadline for *setup* time of F2 has crossed, assuming the unavoidable clock skewness T_ϵ causes the clock to arrive *earlier* than expected (which is the worst case scenario). Our next observation summarizes the reasons which allow an adversary to force F1 into such unsafe states, which eventually becomes the foundation of the countermeasure:

O3: Root-causing DVFS fault attacks. The main cause of DVFS based fault attacks is the highly flexible DVFS interface design which inadvertently provides adversarial control over two *independent* system parameters: core frequency and core voltage. This implies that in Eq. 1, the LHS can be controlled *independently* of the RHS, allowing enumeration of frequency and voltage values leading to Eq. 2, i.e. unsafe states in sequential.

Concretely, considering the ideal variations in core voltage result in decreased voltage swings and slower transistor switching, which in turn cause an increase in T_{src} and T_{prop}. In contrast, independent

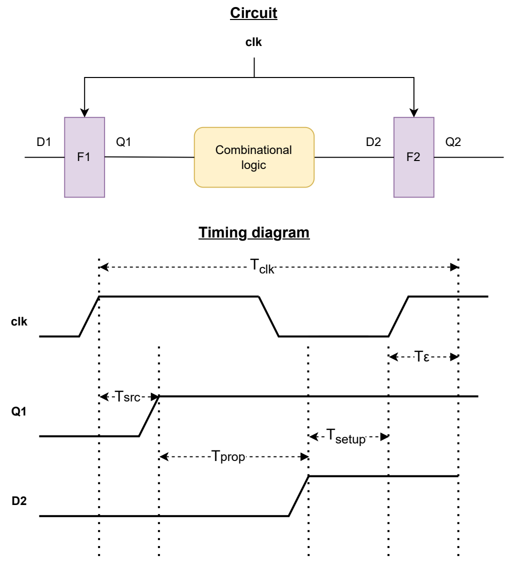

Figure 1: An example sequential circuit and associated timing diagram to visualize the relationships between different parameters of Eq. 1.

to changes in core voltage, variations in core frequency impact T_{clk}, and thereby influence the right-hand side of Equation $T_{src} + T_{prop} \leq T_{clk} - T_{setup} - T_\epsilon$. Consequently, an adversary is able to independently tweak core frequency as well as core voltage, causing inequality Eq. 2 to occur, thereby causing a system into unsafe state and eventually mounting a successful DVFS-based fault attack. We note that previous attacks like [2, 5, 8] focus on one aspect from the voltage-frequency pair while keeping the other constant.

3.3 Novel DVFS countermeasure philosophy: forcing safe states

In Sec. 3.2, we put forth a fresh perspective, missing from prior DVFS styled attacks (as in [2, 5, 8]), that by allowing independent adversarial control over frequency and voltage, modern systems have made themselves vulnerable. More precisely, the *independence* of control over core frequency and core voltage allows an adversary to find specific voltage-frequency pairs that force the system into unsafe states. Interestingly, from a defender's perspective, one can use this inquisitive observation to develop a countermeasure philosophy, as stated below.

Limiting causal independence of voltage-frequency. Based on root-causing DVFS (ref. Observation **O3**), our countermeasure philosophy relies on *limiting the independence* with which core frequency and voltage can be altered. This can be done by enforcing a relationship between allowed values of core frequency and core voltage, thereby preventing the system from entering into an unsafe state.

Concretely, by performing characterization of a system for safe-unsafe states, our countermeasure philosophy proposes to identify core voltage and core frequency relationships where the system enters unsafe state and deploy countermeasure mechanisms to prevent such unsafe states from occurring. We note that not all frequency/voltage tuples allow DVFS faults (i.e. unsafe system states). This fact allows us to search through the entire space of

Algorithm 1 Voltage offset computation

```
procedure OFFSET_VOLTAGE(OFFSET, PLANE)
    set val ← (offset*1024/1000)
    set val ← 0xFFE00000 and ((val and 0xFFF) left-shift 21)
    set val ← val or 0x8000001100000000
    set val ← val or (plane left-shift 40)
    return val
```

allowed frequency/voltage tuples, identify the ones responsible for DVFS faults, and then deploy a countermeasure to prevent occurrence of such frequency/voltage tuples. In the next section, we elaborate on the design of the countermeasure.

4 COUNTERMEASURE IMPLEMENTATION THROUGH `UNSAFE` STATE MANAGEMENT

In this section, we describe how we use the observations from Sec. 3.3 to develop and deploy a purely software-based countermeasure against DVFS-styled attacks. The countermeasure design proceeds in two steps:

- **S1**. Empirically creating core frequency and core voltage pairs that cause a system to enter into `unsafe` state.
- **S2**. Deploy a polling based mechanism on model-specific registers (MSRs) to *limit causal independence* of core frequency and core voltage to prevent occurrence of `unsafe` states.

We first establish our attacker threat model largely based on prior works on DVFS-based fault attacks [5, 8, 10, 12].

4.1 Threat model

We follow a threat model reminiscent of confidential computing, wherein the attacker is assumed to be privileged (i.e. the attacker has control over the operating system and the BIOS). Additionally, our threat model does not require the overclocking mailbox (OCM) to be disabled (unlike Intel's countermeasure [4]). From the adversarial side, we assume the adversary mounts attacks directly on neither the SGX enclave management nor the code running within the enclave. The attacker, however, is assumed to have the capability of mounting DVFS attacks while the enclave is operational. The adversarial objective in this case is to use DVFS to fault instructions, whose results drive subsequent instruction execution.

Note on single-stepping and zero-stepping. We note that prior defences against DVFS fault attacks like [6] do not directly assume *single-stepping* in their threat model. This is a consequence of the countermeasure design choices: Since the *trap* instructions are placed inside SGX enclave *after* the instruction to be faulted, an adversary can simply use single-stepping to isolate the target instruction and inject the fault. Moreover, concepts like zero-stepping [7] allow an adversary unbounded time between injection of DVFS fault and occurrence of trap *deflections*. As such, [6] relies on non-DVFS related mechanisms like [3] to prevent any single-stepping or zero-stepping using Intel TSX instruction extension[4]. In contrast, we assume the adversary has capability for single/zero-stepping.

Note on adversarial control over unloading kernel modules. We assume the adversary can load/unload kernel modules. This raises an important question: *why can an adversary not simply unload the kernel module belonging to our polling countermeasure*? For this, we use Intel SGX's attestation: we propose that the *load/unload state of our countermeasure's kernel module be a part of SGX attestation report.* This change allows OCM access to all benign non-SGX processes even when SGX context is in execution, while still maintaining SGX security. We note that adding such software/micro-architectural optimization features into SGX attestation is a very normal security offering (similar to adding hyper-threading status into SGX attestation reports).

[4]Note that Intel TSX is no longer available on latest generation of Intel client processors.

Algorithm 2 DVFS thread

```
procedure DVFS_THREAD()
    set unsafe ← {}
    set F ← possible core frequencies (resolution of 0.1 GHz)
    set V ← {−1, −2, −3, ..., −300}
    set freq_volt_tuples ← F × V (× represents cartesian product)
    set original_freq ← Measure core frequency through MSR 0x198
    set original_voltage_offset ← Measure normal core voltage offset through MSR 0x150
    for (test_frequency, test_voltage_offset) in freq_volt_tuples do
        CPU_POWER(test_frequency) // set core frequency through the CPU Power linux utility
        offset_voltage_value ← offset_voltage(test_voltage_offset, 0)
        MSR_WRITE_0x150(offset_voltage_value) // write the offsetted voltage to 0x150
        // Allow EXECUTE thread to continue in a non-blocking way
        CPU_POWER(original_freq) // restore core frequency to normal
        MSR_WRITE_0x150(original_voltage_offset) // restore core voltage to normal
        if faults observed in victim thread execution then
            append (test_frequency, test_voltage_offset) to unsafe set
```

4.2 S1. Characterizing `unsafe` states

The first step of our countermeasure is to characterize a system-under-test into `safe` and `unsafe` states. For our experiments, we evaluated three generations of Intel processors: Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz (codename: Sky Lake, microcode version: `0xf0`), Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz (codename: Kaby Lake R, microcode version: `0xf4`), and Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz (codename: Comet Lake, microcode version: `0xf4`). Each system was configured to use a characterization framework consisting of two threads: ① DVFS thread and ② EXECUTE thread.

In the ① *DVFS thread*, we enumerate the entire search space of the independently controlled parameters: core frequency and core voltage, as summarized in Algo. 2. The *DVFS thread* then iterates over all possible voltage-frequency pairs in order to determine if the victim thread observed any faults. In line with undervolting mechanisms explored in works like [5, 8, 10], we use Algo. 1 to first compute the overall 64-bit value of MSR `0x150` that encapsulates appropriately chosen negative voltage offset **test_voltage_offset** and then uses Intel's MSR memory mapped I/O interface to write into MSR `0x150`. Then, the *DVFS thread* allows the victim thread to execute carefully selected arithmetic operations (which we discuss next) and observes occurrence of incorrect computation, implying successful fault injection. If a fault does indeed occur, then the *DVFS thread* considers the corresponding tuple **(test_frequency, test_voltage_offset)** as an `unsafe` state of the system.

The EXECUTE thread, on the other hand, is similar to the faulting instruction sets from [5, 6, 8]. The EXECUTE thread runs a tight

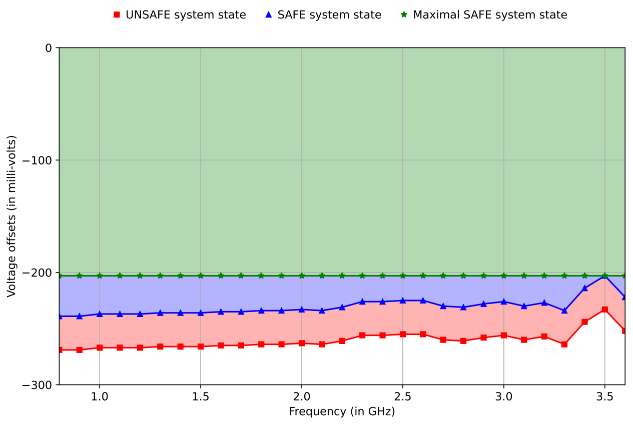

Figure 2: Characterization of `unsafe`/`safe` system states for Sky Lake, microcode version: `0xf0`.

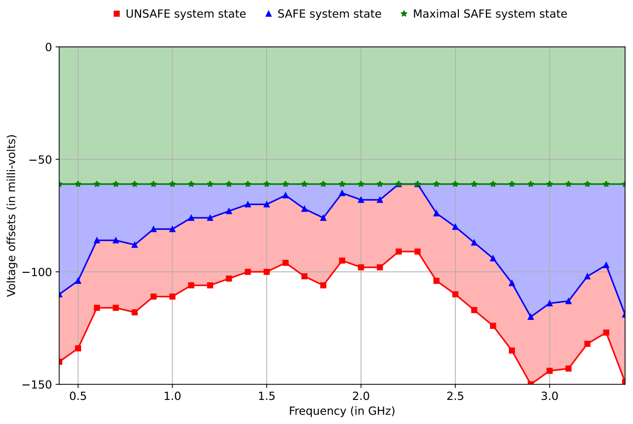

Figure 3: Characterization of `unsafe`/`safe` system states for Kaby Lake R, microcode version: `0xf4`.

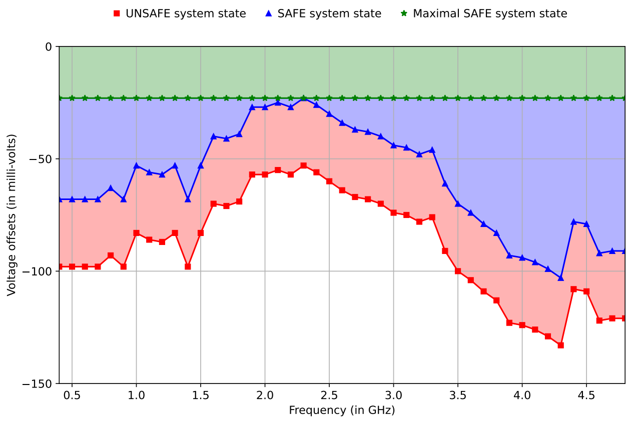

Figure 4: Characterization of `unsafe`/`safe` system states for Comet Lake, microcode version: `0xf4`.

loop of one million iterations of *imul* instructions with varying 64-bit operands. A fault is said to occur if the output of some *imul* instruction (while *DVFS thread* is operational) is different from the actual output of the *imul* instruction (under normal operational frequency/voltage settings). As evident from Algo. 2, the EXECUTE thread continues in *parallel* to the *DVFS* thread.

We now detail the characterizations of `safe`/`unsafe` states across three generations on Intel processors, depicted in Fig. 2, Fig. 3, and Fig. 4. For any given frequency on all three systems, after a certain undervolt offset, we start to observe a region of interest where faults begin to manifest. This is exactly the point in execution where the system is no longer in a `safe` state. For each frequency, we keep characterizing the *width* of the `unsafe` region (i.e. the range of undervolting offsets where the system continues to be in `unsafe` states) until we observe a system crash[5]. Once we have characterized the entire frequency spectrum, we have the tuples of voltage-frequency values for `unsafe` system states. We note that this characterization is system specific and needs to be carried out for each system. However it's a one time process and can be easily automated without requiring any human expertise or intervention.

4.3 S2. Countermeasure deployment: *Polling* kernel module

Our polling countermeasure is described in Algo. 3. The deployed kernel module will *poll* MSR `0x198` for core frequency and MSR `0x150` for core voltage. Based on the characterization already done in Sec. 4.2, in the time duration between when the system enters in an `unsafe` state and when the fault occurs, the countermeasure updates `0x150` to force the system back into a `safe` state. In our experiments, this countermeasure was able to **completely eliminate DVFS faults** on EXECUTE thread (c.f. Sec. 4.2) when operational. To substantiate the overhead of our *polling* countermeasure with respect to system performance, we analyse perf scores from `SPEC2017` benchmark suite with/without the deployment of the kernel module housing our polling countermeasure. The results are depicted in Fig. 5, which depicts the normalized slowdown of SPEC rate benchmarks when the countermeasure is operational. Overall, our countermeasure incurs an overhead of 0.28% (geometric mean of all SPEC rate slowdowns). [6]

5 MAXIMAL SAFE STATE: REDUCING COUNTERMEASURE TURNAROUND TIME

The countermeasure discussed in the previous section resides as a kernel module and offers greater flexibility wrt. `safe`/`unsafe` state checks. However, we note that the characterization of `safe`/`unsafe` system states is made in a way that it allows the countermeasure to be implemented at a *deeper* level than a kernel module to reduce the performance overhead even further. To do so, we first define a **maximal** `safe` state of the system. Intuitively, as depicted in Fig. 2, Fig. 3, and Fig. 4, the **maximal** `safe` state is the maximum negative voltage offset for which DVFS cannot be mounted for *any* frequency in the *entire* frequency spectrum available on a system. We now describe different levels of deploying our countermeasure, and note that *only CPU vendors can deploy the countermeasure at these deeper levels in practice*. Hence, we leave the actual deployment of our countermeasure at these levels as out-of-scope for this work. We note that by choosing to deploy the countermeasure through the microcode sequencer or through a model-specific register allows ignoring unsafe writes to `0x150` altogether, preventing the system from even entering into an `unsafe` state.

[5]Concretely, the system reboots. A system crash is essentially useless to an adversary as it reveals no SGX secrets [8] even through kernel crashdumps.
[6]Note that the overhead is for the entire system and not only the kernel module.

Algorithm 3 Polling countermeasure implemented as a kernel module

```
1: procedure POLLING_COUNTERMEASURE()
2:     while True do
3:         for each CPU core do
4:             set core_frequency ← MSR_READ(0x198)
5:             set core_voltage_offset ← MSR_READ(0x150)
6:             if (core_frequency, core_voltage_offset) ∈ unsafe system state then
7:                 // write to 0x150 to force the system into safe state
```

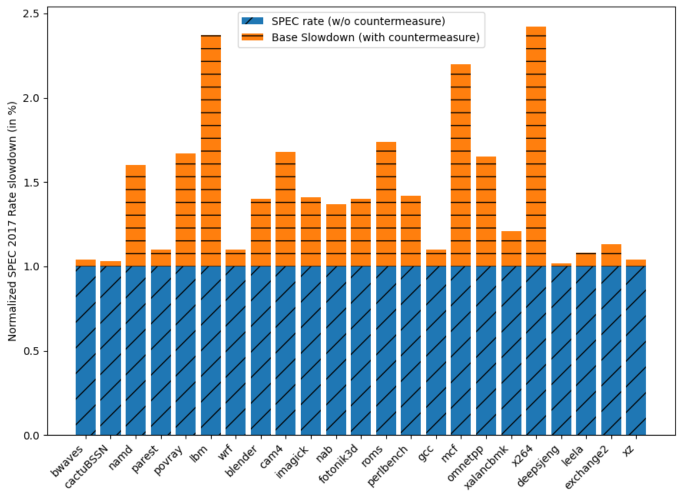

Figure 5: Normalized slowdown (in %) of SPEC2017 for Algo. 3.

5.1 Microcode Sequencer

Microcode allows a layer atop a CPU to allow a mechanism to patch CPU execution in-place without requiring any special hardware. At the time when an event takes place for which microcode intervention is needed, a microcode sequencer kicks in and operates the entire decoding process for subsequent micro-operations. The microcode sequencer is capable of handling conditional microcode branches as well, making it an ideal choice for implementing our countermeasure. Concretely, the microcode read-only memory (ROM) stores the value of the **maximal** `safe` state and the microcode sequencer kicks in a microcode conditional branch whenever a `wrmsr` (x86 instruction to write to MSR) is executed on MSR `0x150`. If the `wrmsr` instruction puts the system into an `unsafe` state (by violating the **maximal** `safe` state boundary), the conditional microcode branch simply *ignores* the write to `0x150`.

5.2 Model Specific Register

To implement our countermeasure at MSR level, we propose to follow the same MSR semantics as followed by the MSRs `0x618` (`MSR_DRAM_POWER_LIMIT`) and `0x61C` (`MSR_DRAM_POWER_INFO`). The MSR `0x618` allows software to set power limits for DRAM domain (this is analogous to writes to MSR `0x150` in the context of our countermeasure). However, MSR `0x61C` allows to set a value `DRAM_MIN_PWR` which is the *minimal* power setting allowed for DRAM power throttling. As such, any value *lower* than `DRAM_MIN_PWR` is *clamped* to `DRAM_MIN_PWR`, which preventing any prospect of undervoltage induced faults in the DRAM. In our context, the CPU vendors can use an additional MSR (hypothetically referred here as `MSR_VOLTAGE_OFFSET_LIMIT`) which puts a *clamp* on `0x150` based on the **maximal** `safe` state characterization performed for a given CPU generation. This allows `MSR_VOLTAGE_OFFSET_LIMIT` to behave as a hardware gatekeeper against any attempts to put the system into `unsafe` states, thereby providing a hardware level countermeasure to DVFS fault attacks.

6 CONCLUSION

In this work, we root-cause DVFS fault attacks and propose a new countermeasure philosophy: by not allowing complete independence in controlling core frequency and voltage, our countermeasure completely prevents DVFS faults. More importantly, unlike prior countermeasures, it also allows access to DVFS features to benign non-SGX executions even when SGX enclaves are executing, and has the ability to be deployed at any of the software, microcode, or hardware levels. Therefore, we conclude that this countermeasure design allows for complete protection against DVFS attacks while allowing availability and flexibility of DVFS features to non-SGX contexts within the purview of `safe` system state, thereby not compromising majorly on the performance of a CPU core.

ACKNOWLEDGMENT

The authors would like to thank the Department of Science and Technology (DST), Govt of India, IHUB NTIHAC Foundation, C3i Building, Indian Institute of Technology Kanpur, and Centre on Hardware-Security Entrepreneurship Research and Development, Meity, India, for partially funding this work.

REFERENCES

[1] Mark Balch. 2003. *Complete digital design: a comprehensive guide to digital electronics and computer system architecture.* McGraw-Hill Education.
[2] Zitai Chen et. al. 2021. {VoltPillager}: Hardware-based fault injection attacks against Intel {SGX} Enclaves using the {SVID} voltage scaling interface. In *30th USENIX Security Symposium (USENIX Security 21).*
[3] Daniel Gruss et. al. 2017. Strong and efficient cache {Side-Channel} protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17).* 217–233.
[4] Intel. 2019. Intel Processors Voltage Settings Modification Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html.
[5] Zijo Kenjar et. al. 2020. {V0LTpwn}: Attacking x86 processor integrity from software. In *29th USENIX Security Symposium (USENIX Security 20).* 1445–1461.
[6] Andreas Kogler et. al. 2022. Minefield: A Software-only Protection for {SGX} Enclaves against {DVFS} Attacks. In *31st USENIX Security Symposium (USENIX Security 22).* 4147–4164.
[7] Moritz Lipp et. al. 2021. PLATYPUS: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 355–371.
[8] Kit Murdock et. al. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP).* IEEE.
[9] Alexander Nilsson et. al. 2020. A survey of published attacks on Intel SGX. *arXiv preprint arXiv:2006.13598* (2020).
[10] Pengfei Qiu et. al. 2019. VoltJockey: Breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* 195–209.
[11] Beng Chiew Adrian Tang. 2018. *Security Engineering of Hardware-Software Interfaces.* Columbia University.
[12] Adrian Tang et. al. 2017. {CLKSCREW}: Exposing the perils of {Security-Oblivious} energy management. In *26th USENIX Security Symposium (USENIX Security 17).* 1057–1074.
[13] Jo Van Bulck et. al. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution.* 1–6.
[14] Jo Van Bulck et. al. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18).* 991–1008.

Received 13 November 2023; accepted 27 February 2024