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

Nimish Mishra\textsuperscript{1}, Rahul Arvind Mool\textsuperscript{1}, Anirban Chakraborty\textsuperscript{1}, and Debdeep Mukhopadhyay\textsuperscript{1}

Indian Institute of Technology Kharagpur, India.
{nimish.mishra, rahulmool}@kgpian.iitkgp.ac.in, anirban.chakraborty@iitkgp.ac.in, debdeep@cse.iitkgp.ac.in

Abstract. The need for energy optimizations in modern systems forces CPU vendors to provide Dynamic Voltage Frequency Scaling (DVFS) interfaces that allow software to control the voltage and frequency of CPU cores. In recent years, the accessibility of such DVFS interfaces to adversaries has amounted to a plethora of fault attack vectors. In response, the current countermeasures involve either restricting access to DVFS interfaces or including additional compiler-based checks that let the DVFS fault occur but prevent an adversary from weaponizing it. However, such countermeasures are overly restrictive because (1) they prevent benign, non-SGX processes from utilizing DVFS, and (2) rely upon a less practical threat model than what is acceptable for Intel SGX. In this work, we hence put forth a new countermeasure perspective. We reason that all DVFS fault attacks are helped by system design decisions that allow an adversary to search through the entire space of frequency/voltage pairs which lead to DVFS faults on the victim system. Using this observation, we classify such frequency/voltage pairs causing DVFS faults as unsafe system states. We then develop a kernel module level countermeasure (in non-SGX execution context) that polls core frequency/voltage pairs to detect when the system is in an unsafe state, and force it back into a safe state. Our countermeasure completely prevents DVFS faults on three Intel generation CPUs: Sky Lake, Kaby Lake R, and Comet Lake, while allowing accessibility of DVFS features to benign non-SGX executions (something which prior works fail to achieve). Additionally, we also put forth the notion of maximal safe state, allowing our countermeasure to be implemented both as microcode (on the micro-architecture level) and as model-specific register (on the hardware level), as opposed to prior countermeasures which can not be implemented at the hardware level. Finally, we evaluate the overhead of our kernel module’s execution on SPEC2017, observing an minuscule overhead of 0.28%.

Keywords: Dynamic Voltage Frequency Scaling countermeasure · Software-based fault attack · Software Guard Extensions (SGX) · Plundervolt · VoltJockey · VoltPwn · Model-specific register
1 Introduction

Modern system design aims to maximize performance while optimizing resource usage. And one of the most important of such resources that need to be optimized is energy. Energy optimizations in modern systems are crucial since suboptimal energy management decisions increase power consumption, and transitively, increase processor wear over a period of time. Moreover, in case of laptops, insufficiently optimized energy management mechanisms have a direct impact on battery life as well. 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) [2, 11, 7, 1]. A processor core is said to be in a P-state when it is executing, implying that the core requires frequency throttling and dissipates energy (thereby increasing power consumption). On the other hand, a core is said to be in a C-state when it is idle, wherein several components of the core (like execution units) are switched to reduced power supply to save energy. While exact naming conventions vary across processor vendors (for instance, AMD calls a generic C-state as Cool-n-Quiet state), in the rest of the article, we consistently refer P-state and C-state to denote the idleness of processor cores.

Dynamic Voltage Frequency Scaling (DVFS) refers to an interface allowing traversing the spectrum of P-states of a core. Ideally, a DVFS interface allows privileged software to throttle a CPU core’s frequency. This could be achieved either through exposing a scaling driver (as in [7] or in [8]) or through exposure of specific model-specific registers or MSR (as 0x150 on Intel systems [11]). Such interfaces are exposed to privileged software like kernel, thereby allowing it to control the entire spectrum of P-states of the core. A straightforward downside of exposing DVFS interfaces to software is that it opens up arenas for newer attack surfaces to adversaries. But a critical question arises: what kind of attack vectors can be leveraged through P-state changes enforced by DVFS? Prior works like [19, 14, 6] use DVFS to introduce timing violations in a core’s internal circuitry. Ideally, the clocking of any processor core should ensure the following timing constraints (which in turn ensures sufficient time for the circuitry to stabilize output) [21]:

$$T_{src} + T_{prop} \leq T_{clk} - T_{setup} - T_{\epsilon}$$  \hspace{1cm} (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 $T_{\epsilon}$ 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, under-volting causes an increase in $T_{src}$ and $T_{prop}$ due to decreased voltage swings and slower transistor switching [3], while $T_{clk}$, $T_{setup}$ and $T_{\epsilon}$ (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 [19, 14, 6] use such timing violations and subsequent incorrect outputs to influence critical operations, thereby successfully mounting purely software-based fault attacks.

It is worthwhile to note that these attacks utilize a fundamental property of digital circuits and thus make it difficult to develop effective countermeasures. So far, to the best of our knowledge, there have been two countermeasure philosophies in literature. The first one relies on modifying \( 1 \) access control paths to disallow DVFS interface from being exposed to adversary. Intel’s microcode patches in response to [19, 14, 6] is a prime example [12]. Succinctly, under fixes to CVE-2019-11157, Intel added the disabled status of overclocking mailbox (OCM) interface and the MSR 0x150 to Intel SGX remote attestation reports (which are controlled by Intel Attestation Service), ensuring that the OCM is not accessible to a non-SGX context at a time when SGX context is in execution. The second philosophy is not to stop DVFS-enabled faults but rather to prevent an adversary from weaponizing them through \( 2 \) deflection. The work in [15] relies on an automated analysis of fault characteristics of x86 instructions to develop a compiler extension that deflects potentially faultable x86 instructions into traps. Such compiler-induced traps prevent the adversary from taking any perceivable advantage, even in case of successful injection of faults through DVFS.

While both the countermeasure philosophies are able to protect sensitive applications from fault attacks, they come with their own share of drawbacks. For instance, the access control based defenses greatly restrict benign non-SGX applications from using a CPU’s power management features to their fullest (when an SGX context is operational on a shared hyperthread). This greatly impacts system throughput and performance [15]. Moreover, implementing such access control checks dynamically at run-time uses complex microcode assists [15], further adding performance overhead. On the other hand, the deflection approach relies on SGX execution context and thus is not self-sufficient against practical attack vectors that utilize instruction isolation using single-stepping using [27]. This raises a core 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 answer this question in the affirmative! We present a distinct countermeasure design philosophy. we revisit the timing constraints mentioned in Eq. 1 to detect unsafe system states which can violate established timing constraints, and propose mechanisms to bring back the system into a safe state. First, we unearth the root-cause of DVFS fault attacks and put forth a fresh perspective that independent manipulations of core frequency and core voltage
are responsible for putting the system in unsafe states where fault attacks 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. Since our countermeasure allows the processor core voltage to tread freely into safe states, it provides the required flexibility to demanding applications to undervolt and overclock the cores while protecting them from DVFS-styled fault attacks.

1.1 Contributions

To summarize, we make the following contributions in this work.

– We put forth a new countermeasure philosophy for DVFS by characterizing a victim system into safe and unsafe states. Concretely, our countermeasure design attempts to use the fundamental causal property of DVFS fault attacks to develop the countermeasure, rather than using auxiliary methods like (1) access control checks, and (2) deflection, unlike prior works. We first root-cause DVFS attacks to note that modern system design allows causal independence in controlling a core’s frequency and voltage, and use this to define safe-unsafe system states with respect to system stability against DVFS attacks.

– We develop a software-only countermeasure that resides as a kernel module and uses our system characterization to prevent DVFS based fault attacks from being mounted. By relying on our definition of safe-unsafe states, we are able to base the countermeasure outside any SGX context, thereby allowing our countermeasure to work within a more robust and practical threat model than prior works [12, 15] (like not relying on third-party mechanisms to assume absence of single/zero-stepping of SGX enclaves). Our experiments show that our countermeasure is able to completely prevent DVFS induced faults, while showing an acceptable overhead of 0.28%.

– Our characterization of safe-unsafe system states allows identification of maximal safe state for a given system, allowing our countermeasure to be potentially deployable as a (1) microcode assist, or (2) model-specific register (MSR), by respective CPU vendors. As opposed to literature like [15] and [12], our countermeasure also has the ability to be implemented at a more fundamental level in the micro-architecture.

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 hardware-based security technology developed by Intel. It provides a secure execution environment, commonly
referred to as an enclave, where sensitive code and data can be protected from potentially malicious software and even privileged system software. Therefore, even if the kernel is compromised, the security of the programs and corresponding data processed inside the enclaves are supposedly guaranteed by the hardware-based isolation. This technology is particularly crucial in scenarios where confidentiality and integrity of computations are paramount.

Intel SGX logically partitions an application into a trusted and untrusted part where the trusted part containing source codes for sensitive computations runs inside the enclave and the untrusted part mostly consists of benign operations that do not involve secret information. The operating system initiates the execution of the process by launching the untrusted part, which in turn initiates the trusted part inside enclave as per the program logic. SGX ensures that the enclave’s execution state and memory are inaccessible to all other processes in the system as well as the operating system. The threat model of SGX only assumes the CPU to be trusted. Therefore, even in the presence of a compromised kernel and superuser adversary, the hardware provides isolation guarantee of enclaves.

In spite of such hardware-based isolation, a number of side channel attacks have raised serious question on the security guarantees of Intel SGX. While the enclave memory and execution states are protected, other important features that interact with programs such as page table management, scheduling, interrupt handling, etc. are managed by the OS. A number of attacks [20] have been proposed in literature that undermine the security guarantees of SGX. Due to the permissible threat model of SGX, adversary can manipulate critical OS features like APIC timer interrupts to precisely control the execution flow of the processes running inside the enclave. In addition, transient attacks like Foreshadow [26], Zombieload [22], etc. leak information from enclaves.

2.2 Power Management (DVFS) in Intel

Modern computing systems have different power and energy requirements which vary across form factors and their usage. Specifically, mobile devices such as laptops, tablets and smartphones require constant balancing between power consumption and performance. The amount of energy consumed by the processor in the integral of the instantaneous power over a certain period of time. The instantaneous power consists of two components - dynamic and static power. While the static power is independent of the operations being performed in the system, the dynamic power is dependent on the switching activities of the digital circuits in the processor. More specifically, the dynamic power is directly proportional to the clock frequency and voltage. In consequence, low frequency and voltage help in reducing energy dissipation.

In most modern processors, Dynamic Frequency and Voltage Scaling (DVFS) is employed to maintain a delicate balance between energy consumption and performance. Linux-based operating systems provide a DVFS driver to dynamically manage core frequencies and voltage using different interfaces that vary across
Table 1: Description of different bits of MSR 0x150. 0 indicates the least significant bit (LSB).

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 20</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>21 - 31</td>
<td>offset</td>
<td>Voltage offset (in milli-volts) relative to base core voltage</td>
</tr>
<tr>
<td>32</td>
<td>write-enable</td>
<td>Enable bit to allow read/write functionality</td>
</tr>
<tr>
<td>33 - 39</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>40 - 42</td>
<td>Plane select</td>
<td>Domain whose voltage needs to be scaled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = CPU core; 1 = GPU; 2 = cache; 3 = uncore; 4 = analog I/O</td>
</tr>
<tr>
<td>43 - 63</td>
<td>-</td>
<td>Reversed</td>
</tr>
</tbody>
</table>

different processor vendors. The driver generally provides different scaling governors for different performance demands. The operational (allowable) frequency of a processor is limited to a range of independent values, called frequency table [9]. The range of permissible frequency values are set by the processor vendor for optimal usage with flexibility for dynamic scaling without damaging it. The OS provides interface for applications to configure frequencies through userspace using suitable scaling governors. However, the operating voltage of the processors is not allowed to be changed through these governors.

### 2.3 Frequency and Voltage Manipulation through MSRs

Overclocking and undervolting features help system owners to extract optimal performance from the system, such as for gaming applications (overclocking) and power-saving state (undervolting). To provide computer enthusiasts and interested end-users flexibility to customise their machines for optimal performance, Intel processors expose traditional BIOS features to perform real-time overclocking of the processor cores. DVFS allows changing the voltage and frequency from privileged software using Model Specific Registers (MSR). Recent works [25, 19] have reverse-engineered the use of Over-Clocking Mailbox (OCM) to reveal that writes to MSR 0x150 allows to change the alignment between voltage and frequency, i.e, deviate from the specified voltage-frequency table mappings.

As reported in previous works that reverse-engineered and exploited the OCM, the MSR 0x150 has the structure as depicted in Tab. 1. The bit 63 is fixed and must be set to ‘1’ for writes to happen successfully in the MSR. Bits [42:40] represent the plane index that denote which CPU component to be affected for the undervolting. The scaling voltage is denoted by the 11-bit value of the register bits [31:21]. This value is expressed in units of 1/1024 V (about 1 mV). Once the MSR 0x150 has been written to, the system takes some time for the scaled voltage to apply. The current operating voltage can be queried from the MSR 0x198.
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. 2, 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 $i$.

Over the course of subsequent subsections, we formalize this definition of safe state in terms of timing parameters used in Eq. 2.

3.1 Establishing interplay of independent timing parameters

We first explain the different parameters involved in sequential digital circuitry and their relative interplay that controls the output signal of such circuitry. Re-iterating Eq. 1, we note the following constraint:

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

(2)

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 sequential/combinational elements of the circuit to stabilize output. $T_{setup}$ refers to the setup time of the sequential circuitry, while $T_{\epsilon}$ refers to small timing fluctuations. Finally, $T_{clk}$ refers to the time period of the synchronous clock pulse driving the circuitry.

We show the interplay of these parameters in Fig. 1. In line with discussions on timing violations done in previous works on DVFS fault attacks like [24, 21], we also restrict our definitions of safe/unsafe states on the most basic sequential unit: flip-flops. Our observations naturally extend to more complex sequential units as well since flip-flops are the foundation blocks for all sequential unit designs. Referring to Fig. 1, the objective of tuning parameters of Eq. 2 is to enforce flip-flop F1 in a safe state. In other words, the flip-flop F1 can be claimed to be in a safe state iff its output is stable before flip-flop F2 is driven by the clock and by input D2.

We now begin to establish the interplay of different parameters from Eq. 2, which play an important role in achieving the aforementioned objective. As exemplified, we consider a circuit with a sequence of combinational logic, between two sequential flip-flops F1 and F2 driven by the same clock of time period $T_{clk}$ with maximum uncertainty $T_{\epsilon}$. In this example, we use this over-arching parameter $T_{\epsilon}$ 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, the clock for $F_2$ can arrive at any point in the closed time interval $[T_{clk} - T_e, T_{clk} + T_e]$. On a real circuit, these variations can arise from a variety of sources like variations in the clock distribution network, spatial voltage and cycle-to-cycle variations in the loop distribution network, and temporal/spatial jitter. Since these variations are immeasurable and unavoidable, a circuit should not be configured to deliver a stabilized output beyond a time upper-bounded by $(T_{clk} - T_e)$. This is the worst case scenario when unavoidable variations cause the clock to arrive earlier than expected. This leads us to make the first checkpoint observation:

**O1: Handling unavoidable clock skewness.** To ensure a safe flip-flop $F_1$ state is to control the core frequency $f$ such that the output of $F1$ is stable in time upper-bounded by $(\frac{1}{f} - T_e)$. Evidently, $T_{clk} = \frac{1}{f}$. 

---

Fig. 1: An example sequential circuit and associated timing diagram to visualize the relationships between different parameters of Eq. 1.
We now bring in another aspect important to defining whether \( F_1 \) is indeed in a **safe** state. Note that the circuit exemplified in Fig. 1 is such that the overall output of \( F_1 \) manipulated by the combinational logic is driven as input \( D_2 \) of \( F_2 \), which is itself a sequential execution unit driven by synchronous system clock. This raises another important concern with respect to the **setup** time needed for \( F_2 \). Recall that setup time for any sequential element is the amount of time the input line (\( D_2 \) in this case) needs to be stable before the arrival of the clock edge. In Fig. 1, this is denoted by \( T_{\text{setup}} \). In the worst case, the clock will arrive at \( F_2 \) no later than \( T_{\text{clk}} - T_{\epsilon} \), leading us to make the following observation atop observation **O1**.

**O2: Handling setup delays atop clock skewness.** To ensure a **safe** flip-flop \( F_1 \) state when its output drives a sequential element \( F_2 \), the core frequency \( f \) must be such that the output of \( F_1 \) is stable in time upper-bounded by \((\frac{1}{f} - T_{\epsilon} - T_{\text{setup}})\). Evidently, \( T_{\text{clk}} = \frac{1}{f} \) and \( T_{\text{setup}} \) is the setup time for \( F_2 \).

Finally, from Fig. 1, we note that the input \( D_2 \) is driven by application of combinational logic on output of \( Q_1 \) (output of \( F_1 \)). According to Eq. 2 semantics, we refer to the time elapsed since driving of \( D_1 \) to driving of \( D_2 \) as summation of time taken for \( F_1 \) to produce output \( Q_2 \) (i.e. \( T_{\text{src}} \)) and the time taken for the combinational logic to operate on \( Q_2 \) (i.e. \( T_{\text{prop}} \)).

### 3.2 Fundamental cause of DVFS fault attack vectors

We now use the observations made in Sec. 3.1 about **safe** state of \( F_1 \) 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 \( F_1 \) (c.f. Fig. 1) into an **unsafe** state. We can define an **unsafe** state of \( F_1 \) as:

\[
T_{\text{src}} + T_{\text{prop}} > T_{\text{clk}} - T_{\text{setup}} - T_{\epsilon}
\]  

Informally stating, in context of Fig. 1, this equation implies that flip-flop \( F_1 \) is **unsafe** iff the input \( D_2 \) is stable after the deadline for setup time of \( F_2 \) has crossed, assuming the unavoidable clock skewness \( T_{\epsilon} \) causes the clock to arrive earlier than expected (which is the worst case scenario, as discussed in Sec. 3.1). Our next observation summarizes the reasons which allow an adversary to force \( F_1 \) into such **unsafe** states.

**O3: Root-causing DVFS fault attacks.** The main cause of DVFS based fault attacks is the tendency of modern systems to provide adversarial control over two **independent** system parameters: core frequency and core voltage. This implies that in Eq. 2, the LHS can be controlled independently of the RHS, allowing enumeration of frequency and voltage values leading to Eq. 3, i.e. **unsafe** states in sequential.

Concretely, variations in core voltage result in decreased voltage swings and slower transistor switching [3], which in turn causes an increase in \( T_{\text{src}} \) and \( T_{\text{prop}} \) (i.e. the left-hand side of Eq. 2). In contrast, independent to changes in
core voltage, variations in core frequency impact $T_{dk}$, and thereby influence the right-hand side of Eq. 2. Consequently, an adversary is able to independently tweak core frequency as well as core voltage, causing inequality Eq. 3 to occur, thereby sending the system into unsafe state and eventually mounting a successful DVFS-based fault attack. It is worth mentioning that previous attacks like [19, 14, 6] 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 [19, 14, 6]), 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. 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 the system from entering into unsafe states.

Before moving forward, we first establish the attacker threat model based on the publicly available works that propose DVFS-based fault attacks.
4.1 Threat model

For developing the countermeasure, we rely on the threat models used in prior works like [19, 14, 21, 24]. Our countermeasure works on the threat model of trusted computing, wherein the attacker is assumed to be privileged (i.e. the attacker has controller over the operating system and the BIOS). Since our countermeasure aims to prevent the system from entering into unsafe states, our threat model does not require the overclocking mailbox (OCM) to be disabled. Concretely, this implies that Intel’s countermeasure [12] of adding the disabled status of OCM to SGX attestation reports is no longer applicable. 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 [15] do not directly assume single-stepping in their threat model. That is, the countermeasure in [15] does not assume an adversary which has the capability of DVFS faulting as well as interrupting SGX enclaves post a single instruction execution (which can be achieved using tools like [27]). 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 isolation the target instruction and inject the fault. Moreover, concepts like zero-stepping [17] allow an adversary unbounded time between injection of DVFS fault and occurrence of trap deflections. As such, [15] relies on non-DVFS related mechanisms like [23, 5, 10] to prevent any single-stepping or zero-stepping.

It is worthwhile to note here that our countermeasure does not depend on SGX enclave execution at all, rather depends on managing safe states through limiting causal independence of core voltage and core frequency. Alternatively stating, the countermeasure kicks in as soon as the DVFS fault occurs. As such, in contrast to the threat model [15], we assume the adversary has capability for single/zero-stepping. Hence, we do not rely on any third-party mechanisms like [23, 5, 10] to provide completeness to our countermeasure’s operation.

Note on adversarial control over unloading kernel modules. We note that the threat model we have assumed in this work allows an adversary to load/unload kernel modules. This raises an important question: why can an adversary not simply unload the kernel module belonging to our polling countermeasure? This is where Intel SGX’s attestation comes into picture. We propose that the load/unload state of our countermeasure’s kernel module be a part of SGX attestation report provided to the client. We state that this has not downgraded the generality of our countermeasure. We have simply removed the overclocking mailbox (OCM) from Intel SGX attestation report, and added our countermeasure kernel module to the report. This change allows OCM access to all benign
Algorithm 1 Voltage offset computation

```plaintext
1: procedure offset_voltage(offset, plane)
2:   set val ← (offset*1024/1000)
3:   set val ← 0xFFF00000 and ((val and 0xFFF) left-shift 21)
4:   set val ← val or 0x8000001100000000
5:   set val ← val or (plane left-shift 40)
6: return val
```

Algorithm 2 DVFS thread

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

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 [29]).

With the threat model re-instated, we now proceed to elaborate on our proposed two-step countermeasure against DVFS-based fault attacks.

4.2 S1. Empirical characterization of unsafe system 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: Skylake, 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...
thread, we enumerate the entire search space of the independently controlled parameters: core frequency and core voltage. In order to control core voltage, the DVFS thread uses the same mechanism as in [19, 14, 21]. Concretely, the DVFS thread chooses a negative voltage offset $x$ milli-volts, offsets the baseline voltage by $x$ milli-volts, and writes the corresponding value into MSR 0x150. In order to compute the actual value to be written in line with semantics of MSR 0x150 (c.f. Sec. 2.3), Algo. 1 is used. Overall, the DVFS thread is executed as depicted in Algo. 2. We first initialize an empty set referring to all unsafe states of the system. Likewise, set $F$ is initialized to contain all possible frequency values a system core can support (with a resolution of 0.1 GHz), while the set $V$ is initialized to contain negative voltage offsets. The latter choice is by design, since all prior works [19, 14, 21] observed DVFS faults through undervolting only (i.e. through consideration of negative voltage offsets while modifying core voltage).

The DVFS thread then iterates over all possible voltage-frequency pairs in order to determine if the victim thread observed any faults. As evident from Algo. 2, we use the cpupower Linux utility [18] to modify the core frequency. Likewise, 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 [13] (abstracted in Algo. 2 as MSR WRITE 0x150) 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.

We now detail the operations of the EXECUTE thread. From [15, 14, 19], the imul instruction has the maximum probability of being faulted by DVFS styled attacks. Hence, in our characterization also, we use the imul instruction. Concretely, the EXECUTE thread runs a tight 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 without blocking the latter’s execution, thereby posing no problems of ensuring synchronization.

We now detail the characterizations of safe/unsafe states across three generations on Intel processors, depicted in Fig. 2, Fig. 3, and Fig. 3. As evident, across the entire frequency spectrum of each system, we observe a range of under-volted offsets where no DVFS related faults are observed. Additionally, 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, but rather has entered into an unsafe 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.
Fig. 2: Characterization of unsafe/safe system states for Sky Lake, microcode version: 0xf0.

Fig. 3: Characterization of unsafe/safe system states for Kaby Lake R, microcode version: 0xf4.
Once we have characterized the entire frequency spectrum, we have the tuples of voltage-frequency values for which the target system is in an unsafe state.

### 4.3 S2. Countermeasure deployment: Polling kernel module

With the characterization of safe/unsafe system states done in Sec. 4.2, we are ready to describe details of the deployment of our countermeasure as a kernel module. This is depicted in Algo. 3. Basically, 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, should the system be in an unsafe state, 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.

Algorithm 3 Polling countermeasure implemented as a kernel module

```plaintext
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
```
Table 2: Experimental evaluation of the overhead incurred by polling countermeasure on Comet Lake, microcode version: 0xf4.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base rate (w/o polling)</th>
<th>Base rate (with polling)</th>
<th>Slowdown (%)</th>
<th>Peak rate (w/o polling)</th>
<th>Peak rate (with polling)</th>
<th>Slowdown (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>503.bwaves</td>
<td>628.59</td>
<td>628.9</td>
<td>-0.04%</td>
<td>604.21</td>
<td>606.84</td>
<td>-0.43%</td>
</tr>
<tr>
<td>507.cactuBSSN</td>
<td>222.95</td>
<td>223.03</td>
<td>-0.03%</td>
<td>202.87</td>
<td>203.15</td>
<td>-0.13%</td>
</tr>
<tr>
<td>508.namd_r</td>
<td>175.96</td>
<td>177.03</td>
<td>-0.06%</td>
<td>179.55</td>
<td>182.51</td>
<td>-1.64%</td>
</tr>
<tr>
<td>510.parest_r</td>
<td>387.96</td>
<td>388.41</td>
<td>-0.1%</td>
<td>324.46</td>
<td>326.05</td>
<td>-0.49%</td>
</tr>
<tr>
<td>511.povray_r</td>
<td>328.67</td>
<td>330.89</td>
<td>-0.67%</td>
<td>267.29</td>
<td>268.05</td>
<td>-0.28%</td>
</tr>
<tr>
<td>519.fbm_r</td>
<td>224.08</td>
<td>227.17</td>
<td>-1.37%</td>
<td>176.56</td>
<td>176.72</td>
<td>-0.09%</td>
</tr>
<tr>
<td>521.wrf_r</td>
<td>404.21</td>
<td>406.84</td>
<td>-0.43%</td>
<td>324.46</td>
<td>326.05</td>
<td>-0.49%</td>
</tr>
<tr>
<td>526.blender_r</td>
<td>256.54</td>
<td>257.71</td>
<td>-0.4%</td>
<td>239.52</td>
<td>239.62</td>
<td>-0.04%</td>
</tr>
<tr>
<td>527.cam4_r</td>
<td>315.77</td>
<td>317.94</td>
<td>-0.68%</td>
<td>324.12</td>
<td>328.14</td>
<td>-1.24%</td>
</tr>
<tr>
<td>538.imagick_r</td>
<td>401.88</td>
<td>403.56</td>
<td>-0.41%</td>
<td>318.06</td>
<td>321.89</td>
<td>-1.2%</td>
</tr>
<tr>
<td>544.nab_r</td>
<td>315.25</td>
<td>316.44</td>
<td>-0.37%</td>
<td>282.02</td>
<td>282.47</td>
<td>-0.15%</td>
</tr>
<tr>
<td>549.fotonik3d_r</td>
<td>418.76</td>
<td>420.44</td>
<td>-0.40%</td>
<td>415.46</td>
<td>419.79</td>
<td>-1.04%</td>
</tr>
<tr>
<td>550.perlbench</td>
<td>222.95</td>
<td>223.03</td>
<td>-0.03%</td>
<td>202.87</td>
<td>203.15</td>
<td>-0.13%</td>
</tr>
<tr>
<td>554.gcc_r</td>
<td>221.4159</td>
<td>221.64</td>
<td>-0.10 %</td>
<td>218.91</td>
<td>220.74</td>
<td>-0.83%</td>
</tr>
<tr>
<td>505.mcf_r</td>
<td>339.97</td>
<td>344.05</td>
<td>-1.20 %</td>
<td>297.68</td>
<td>298.72</td>
<td>-0.34 %</td>
</tr>
<tr>
<td>520.omnetpp_r</td>
<td>509.805</td>
<td>513.139</td>
<td>-0.65 %</td>
<td>479.08</td>
<td>484.51</td>
<td>-1.13 %</td>
</tr>
<tr>
<td>523.xalancbmk_r</td>
<td>295.87211</td>
<td>297.122</td>
<td>-0.42 %</td>
<td>253.71</td>
<td>264.47</td>
<td>-4.24 %</td>
</tr>
<tr>
<td>525.x264_r</td>
<td>234.729</td>
<td>235.44</td>
<td>-0.30 %</td>
<td>197.52</td>
<td>201.23</td>
<td>-1.80 %</td>
</tr>
<tr>
<td>527.cam4_r</td>
<td>315.77</td>
<td>317.94</td>
<td>-0.68%</td>
<td>324.12</td>
<td>328.14</td>
<td>-1.24%</td>
</tr>
<tr>
<td>538.imagick_r</td>
<td>401.88</td>
<td>403.56</td>
<td>-0.41%</td>
<td>318.06</td>
<td>321.89</td>
<td>-1.2%</td>
</tr>
<tr>
<td>544.nab_r</td>
<td>315.25</td>
<td>316.44</td>
<td>-0.37%</td>
<td>282.02</td>
<td>282.47</td>
<td>-0.15%</td>
</tr>
<tr>
<td>549.fotonik3d_r</td>
<td>418.76</td>
<td>420.44</td>
<td>-0.40%</td>
<td>415.46</td>
<td>419.79</td>
<td>-1.04%</td>
</tr>
<tr>
<td>550.perlbench</td>
<td>222.95</td>
<td>223.03</td>
<td>-0.03%</td>
<td>202.87</td>
<td>203.15</td>
<td>-0.13%</td>
</tr>
<tr>
<td>554.gcc_r</td>
<td>221.4159</td>
<td>221.64</td>
<td>-0.10 %</td>
<td>218.91</td>
<td>220.74</td>
<td>-0.83%</td>
</tr>
<tr>
<td>505.mcf_r</td>
<td>339.97</td>
<td>344.05</td>
<td>-1.20 %</td>
<td>297.68</td>
<td>298.72</td>
<td>-0.34 %</td>
</tr>
<tr>
<td>520.omnetpp_r</td>
<td>509.805</td>
<td>513.139</td>
<td>-0.65 %</td>
<td>479.08</td>
<td>484.51</td>
<td>-1.13 %</td>
</tr>
<tr>
<td>523.xalancbmk_r</td>
<td>295.87211</td>
<td>297.122</td>
<td>-0.42 %</td>
<td>253.71</td>
<td>264.47</td>
<td>-4.24 %</td>
</tr>
<tr>
<td>525.x264_r</td>
<td>234.729</td>
<td>235.44</td>
<td>-0.30 %</td>
<td>197.52</td>
<td>201.23</td>
<td>-1.80 %</td>
</tr>
<tr>
<td>527.cam4_r</td>
<td>315.77</td>
<td>317.94</td>
<td>-0.68%</td>
<td>324.12</td>
<td>328.14</td>
<td>-1.24%</td>
</tr>
<tr>
<td>538.imagick_r</td>
<td>401.88</td>
<td>403.56</td>
<td>-0.41%</td>
<td>318.06</td>
<td>321.89</td>
<td>-1.2%</td>
</tr>
<tr>
<td>544.nab_r</td>
<td>315.25</td>
<td>316.44</td>
<td>-0.37%</td>
<td>282.02</td>
<td>282.47</td>
<td>-0.15%</td>
</tr>
<tr>
<td>549.fotonik3d_r</td>
<td>418.76</td>
<td>420.44</td>
<td>-0.40%</td>
<td>415.46</td>
<td>419.79</td>
<td>-1.04%</td>
</tr>
<tr>
<td>550.perlbench_r</td>
<td>222.95</td>
<td>223.03</td>
<td>-0.03%</td>
<td>202.87</td>
<td>203.15</td>
<td>-0.13%</td>
</tr>
<tr>
<td>554.gcc_r</td>
<td>221.4159</td>
<td>221.64</td>
<td>-0.10 %</td>
<td>218.91</td>
<td>220.74</td>
<td>-0.83%</td>
</tr>
<tr>
<td>505.mcf_r</td>
<td>339.97</td>
<td>344.05</td>
<td>-1.20 %</td>
<td>297.68</td>
<td>298.72</td>
<td>-0.34 %</td>
</tr>
<tr>
<td>520.omnetpp_r</td>
<td>509.805</td>
<td>513.139</td>
<td>-0.65 %</td>
<td>479.08</td>
<td>484.51</td>
<td>-1.13 %</td>
</tr>
<tr>
<td>523.xalancbmk_r</td>
<td>295.87211</td>
<td>297.122</td>
<td>-0.42 %</td>
<td>253.71</td>
<td>264.47</td>
<td>-4.24 %</td>
</tr>
<tr>
<td>525.x264_r</td>
<td>234.729</td>
<td>235.44</td>
<td>-0.30 %</td>
<td>197.52</td>
<td>201.23</td>
<td>-1.80 %</td>
</tr>
</tbody>
</table>

We now substantiate the overhead of our polling countermeasure with respect to system performance when the system is under stress. For this, we analyse perf scores from SPEC2017 benchmark suite with and without the deployment of the kernel module housing our polling countermeasure. The results are depicted in Tab. 2. As evident, our countermeasure incurs an overhead of 0.28%.

5 Maximal safe state: reducing countermeasure turnaround time

The countermeasure discussed in the previous section resides as a kernel module that polls MSRs 0x198/0x150 and takes appropriate decision that forces the system into safe state. However, being a kernel module, there are two sources of delays in the turnaround time of the countermeasure (i.e. the time elapsed when between the moment the kernel module issues a write to MSR 0x150 and the moment when the system actually comes into an unsafe state). There are two major contributors to this non-negligible turnaround time:

1. The ioctl calls invoked in the kernel module that drives the MSR read/write functionality [13].
2. The delay between a successful write to MSR 0x150 and the actual change in voltage by the voltage regulator [19].
We note that this turnaround time posed no empirically observable failures in preventing DVFS styled faults. Notwithstanding, 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 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.

5.1 Deployment at Micro-architectural level: microcode sequencer

Microcode [4, 16, 28] allows a layer atop a CPU to allow a mechanism to patch CPU execution in-place without requiring any special hardware. Microcodes are the prime carriers of patches that CPU vendors push in response to vulnerabilities arising as a result of hardware optimizations (c.f. Sec. 2.1). Such microcode updates are loaded through BIOS/UEFI and need to be loaded once the processor resets. 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. This write-ignore behaviour is implemented upon several other MSRs as well [11].

5.2 Deployment at hardware level: model-specific register

The insight about using maximal safe state also allows for implementing the countermeasure at the hardware level- as a Model Specific register. We propose to follow the same MSR semantics as followed by MSRs 0x618 (MSR_DRAM_POWER_LIMIT) and 0x61C (MSR_DRAM_POWER_INFO) [11]. The MSR MSR_DRAM_POWER_LIMIT 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 MSR_DRAM_POWER_INFO 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. As evident, this kind of MSR is exactly where our countermeasure can reside, incurring minimal hardware overhead of an additional MSR. The CPU vendors can use an additional MSR (hypothetically referred here as
MSR\_VOLTAGE\_OFFSET\_LIMIT) which puts a \textit{clamp} on 0x150 based on the \textbf{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 \textbf{unsafe} states, thereby providing a hardware level countermeasure to DVFS fault attacks.

6 Conclusion

In this work, we take an orthogonal route from existing approaches to protect DVFS based fault attacks. Instead of preventing access to DVFS interface or relying on compiler extensions, we focus on the root-cause of such DVFS-style attacks and build a countermeasure around it. Along these lines, we first put forth the perspective that modern system design allows \textit{independent control} over a CPU core’s frequency and voltage. Since core frequency and core voltage control different aspects of a CPU’s digital circuitry, there exist certain voltage-frequency configurations that make a particular system more susceptible to DVFS fault attacks. We enumerate such configurations for three Intel generations, and introduce the concept of \textbf{safe-unsafe} system states with respect to DVFS fault attacks around such configurations.

Our countermeasure is then constructed around \textbf{safe-unsafe} characterization of the system and implemented as a kernel module incurring an acceptable overhead of 0.28%. Moreover, we also characterize \textbf{maximal safe} state of a system, and discuss how our countermeasure (unlike previous works) has the potential to be deployed at both the microcode level as well as the hardware level (as a model-specific register).

From a countermeasure design perspective, 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. Therefore, we conclude that this countermeasure design (utilizing the characterization of \textbf{safe-unsafe} system states) allows for complete protection against DVFS attacks while allowing availability and flexibility of DVFS features to non-SGX contexts within the purview of \textbf{safe} system state, thereby not compromising majorly on the performance of a CPU core.

References


