TEEcorrelate: An Information-Preserving Defense against Performance-Counter Attacks on TEEs

Hannes Weissteiner[1], Fabian Rauscher[1], Robin Leander Schröder[3,4], Jonas Juffinger[1]
Stefan Gast[1], Jan Wichelmann[2], Thomas Eisenbarth[2], Daniel Gruss[1]
[1]*Graz University of Technology, Austria,* [2]*University Luebeck*
[3]*Fraunhofer SIT, Darmstadt, Germany,* [4]*Fraunhofer Austria, Vienna*

Abstract

Trusted-execution environments (TEEs) offer confidentiality in shared environments. While Intel restricts performance counter access, limiting load-balancing and anomaly detection on TEEs, AMD exposes performance counters to the host, leaving the TEE vulnerable to side-channel leakage.

In this paper, we propose TEEcorrelate, a lightweight information-preserving defense against performance-counter attacks on TEEs. TEEcorrelate reconciles monitoring capabilities of the host and confidentiality requirements of the TEE, by statistically decorrelating performance counters. TEEcorrelate combines two components, *temporal decorrelation* using counter aggregation windows, and *value decorrelation* using fuzzy performance counter increases. With default parameters, TEEcorrelate guarantees that the host can read performance counters hundreds of times per second, while the read value never deviates by more than 1024 from the actual value. Hence, the host can still use them for load-balancing, accounting, and detection of unusual or malicious activity. In state-of-the-art attacks on MbedTLS RSA 4096, a TOTP implementation, and the post-quantum HQC key-encapsulation mechanism, attack runtimes increase from 0.58–429 seconds to 1–775.6 days, even for a powerful, fully-informed attacker. We estimate that TEEcorrelate on AMD SEV-SNP has a negligible performance impact of 0.03 % for most context switches, and overall less than 0.09 %. Hence, TEEcorrelate is an effective low cost mitigation for all TEEs.

1 Introduction

Trusted-execution environments (TEEs) provide confidentiality and integrity for code execution and data even against privileged or physical access [4, 9, 32, 33]. The first generation of TEEs protected application parts from a malicious user or compromised system, e.g., Intel Software Guard Extensions (SGX) [32] and ARM TrustZone [10]. The current second generation focuses on the cloud, protecting virtual machines (VMs) from a malicious or compromised host, e.g., AMD Secure Encrypted Virtualization (SEV) [6], Intel Trust Domain Extensions (TDX) [29], and ARM Confidential Compute Architecture (CCA) [9], *i.e.*, running entire VMs (without modification) in a secure environment as *Confidential VMs*.

Numerous attacks target different TEEs [3,12,41–43,59,61, 62,66,70,73,75,76], often using side channels like processor caches [13,25,49,54,72], branch predictors [28,38], or power consumption [44]. Other attacks use page faults [78], timer interrupts [63], or speculative execution [59]. On AMD's SEV, many attacks focus on flawed implementations and design of SEV itself, e.g., vulnerable memory encryption [21,41,75], missing or insufficient register protection [27, 42, 73], and missing integrity protection [21,50,80]. Both for Intel SGX and for AMD SEV, single-stepping frameworks significantly advanced the security research on these TEEs [63,76].

Like most TEEs, SGX restricts the host's ability to analyze SGX enclaves: Code and data are inaccessible and encrypted and hardware-performance counters do not count TEE activity. Consequently, attacks can hide cache attacks, Rowhammer attacks, or zero-day exploits inside malicious SGX enclaves [26,54,56]. Cloudflare uses 7 hardware-performance counters to detect and prevent Spectre attacks in their Workers service [58,68]. While Cloudflare Workers are not confidential, it still shows that hardware-performance counters are used in practice to detect malicious behavior. Thus, while confidentiality is required for TEEs, the host also has a legitimate need to monitor activity for load-balancing, accounting, and detection of anomalies. This is even more true for TDX and SEV, with entire VMs running in TEEs, where malicious code or competitive cache sharing leading can result in degraded performance or security for all VMs on this host.

Performance counter side channels can recover secrets from an SEV-enabled VMs [24]. Lou et al. [46] proposed to inject noise for decorrelation, by executing gadgets affecting specific performance counters in parallel to the vulnerable code. This approach causes actual workload, reducing performance, biasing performance counters, and still rendering performance counters less useful for the cloud provider. Additionally, their approach is bypassed by single-stepping [24].

Preventing single-stepping also does not prevent leakage because interrupts cannot be delayed for a long period of time and the guest state can be rolled back. Even when stepping multiple instructions per interrupt, performance counters still leak information [74]. This leakage can be amplified if the guest state can be rolled back to replay vulnerable code paths. Intel TDX swaps performance counters so that the host cannot see any VM activity and AMD plans to adopt this approach as well [7], effectively removing the host's monitoring capabilities. *Consequently, no TEE offers unbiased performance-counter data to the host for load-balancing, accounting, and detection of unusual or malicious activity, while also maintaining security guarantees for the TEE.*

In this paper, we propose TEEcorrelate, an information-preserving principled defense against performance-counter attacks on confidential VMs. Existing TEEs are either vulnerable [24] or just disable performance counters entirely [32, 33], removing useful functionality. Instead, TEEcorrelate decorrelates performance counter data using two components:

First, TEEcorrelate uses *temporal decorrelation*. TEEcorrelate aggregates performance counter changes within non-overlapping aggregation windows, e.g., every 1 million retired instructions. TEEcorrelate introduces a second set of performance counter registers within the TEE's protected area, like Intel TDX already has. At the end of each window, the performance counter values are copied from the internal TEE-protected registers into the host's performance counter registers. Consequently, a malicious host has to let the victim VM (or enclave) make significant progress to observe any performance counter changes, effectively slowing down performance-counter-based attacks by a linear factor.

Second, TEEcorrelate uses *value decorrelation*. TEEcorrelate allows the reported performance counter values to deviate from the actual value based on a statistical distribution. Hence, performance counter values can lag behind and run ahead. To avoid too large deviations, the values are kept within a margin called deviation window. The deviation window size is negotiated between the host and the TEE during initialization. Consequently, TEEcorrelate preserves coarse-grained trend information for the host for load-balancing, accounting, and detection of unusual or malicious activity.

Both temporal and value decorrelation eliminate the main advantage of performance-counter attacks, *i.e.*, noise-free single-trace attacks. We provide a mathematical foundation for the security of TEEcorrelate and show computationally that the number of samples required to leak a single performance counter increment increases by 5 orders of magnitude (about 160 thousand) for our recommended deviation window size of 2 048[1], and by 7 orders of magnitude (about 40 million) for an even larger deviation window size of 32 768. We evaluate the security of TEEcorrelate using a deviation window size of 2 048 with a fully-informed attacker, aware of the working principle of TEEcorrelate, on the 4 case studies examined by Gast et al. [24]: The attack runtime of the MbedTLS RSA 4096 attack, increases to 824.6 days, which is about 160 thousand times slower than without TEEcorrelate. The TOTP verification attack is equally affected, as TEEcorrelate increases the attack runtime to 32.6 days to brute-force a TOTP token on average. The attack runtime for the TOTP key recovery increases to 285.4 days, 40 million times slower. For the HQC-KEM attack, TEEcorrelate increases the attack runtime to at least 108.59 days, compared to 800 ms previously. We conclude that all 4 attacks become impractical.

The architectural changes of TEEcorrelate are small, requiring only a few new configuration registers and some adjustments to the TEE/host context switch. We estimate the performance of a hypothetical implementation of TEEcorrelate on AMD SEV-SNP, and expect that TEEcorrelate only has a minimal overhead on context switches (0.09 %), which is too low to be visible in the full system performance. This confirms that TEEcorrelate is a lightweight defense.

Finally, we argue that TEEcorrelate also enables a victim TEE to detect ongoing attacks. The victim can then mitigate the attack before a sufficient amount of leakage accumulates. TEEcorrelate also reduces the reliability of SEV-Step [76] by eliminating the ability to count executed instructions. We conclude that given the negligible performance costs and the functional benefits of TEEcorrelate, it should be deployed for TEEs instead of simply disabling performance counters.

Contributions. Our main contributions are as follows:

- We present TEEcorrelate, a principled defense against performance-counter attacks on TEEs. TEEcorrelate uses windowed aggregation for temporal decorrelation and fuzzy increments for value decorrelation.
- We perform a statistical security analysis of TEEcorrelate with a fully-informed attacker, showing that it increases the required number of attack samples by 5 to 7 orders of magnitude (*i.e.*, from seconds to days).
- Counter values can be read by the host hundreds of times per second and never deviate more than 1 024. This balances the host's requirements for monitoring and resource management with the security needs of TEEs.
- Our performance analysis on AMD SEV-SNP estimates an overhead of only 0.09 % as we only have to perform actions on a small number of VM exits.

Outline. In Section 2, we provide background. In Section 3, we provide an overview of TEEcorrelate. In Section 4, we detail how TEEcorrelate can be implemented. In Section 5, we compute how TEEcorrelate increases the security. In Section 6, we evaluate case studies, showing that TEEcorrelate mitigates all of them. In Section 7, we discuss security implications and monitoring aspects. Section 8 concludes.

[1]The deviation window size refers to the maximum absolute deviation in the number returned by a specific performance counter, regardless of the meaning of that number. Hence, we consider the deviation window size as a numerical value without a specific unit. The unit would depend on the specific performance counter used, e.g., 2 048 branch mispredictions, cache misses, or retired instructions.

2 Background and Related Work

In this section, we discuss trusted-execution environments (TEE), including related work on the security of TEEs, and performance counters with their benign and malicious uses.

2.1 Intel SGX

Intel SGX is a TEE implementation that runs so-called enclaves [32], signed x86 applications. While SGX enclaves run on the same CPU core as regular applications, SGX restricts access to the enclave's encrypted memory and register state. Enclaves cannot use all x86 instructions, e.g., `CPUID` or `INT`. Hence, applications have to be adapted to run inside of an enclave. There is a wide range of scientific publications attacking Intel SGX. Cache attacks leak information through the detection of memory accesses of enclaves or the host system [18, 25, 57]. Controlled-channel attacks take advantage of the increased control the host has to mount side-channel attacks [63, 65, 70, 78]. Furthermore, a wide range of microarchitectural vulnerabilities have been used to leak enclave data [52,55,61,66,67]. Ahmad et al. [2] proposed a framework that also protects against some side-channel attacks.

2.2 AMD SEV

AMD *Secure Encrypted Virtualization (SEV)* builds on the concept of virtualization, where a host runs multiple VMs on the same physical host and assigns resources to them. VMs are isolated from the host and from each other. The host (the hypervisor) runs at a higher privilege level with access to all system resources, including VM states and memory. Confidential VMs are protected against access from the host. In contrast to traditional TEEs, confidential VMs allow running entire VMs of unmodified software and do not require rewriting parts of the application.

AMD SEV itself protects the memory contents of confidential VMs by transparently encrypting data in DRAM [35], using a unique, non-extractable key per VM. There are multiple attacks on SEV without extensions, targeting the unprotected guest state [27, 73] and protected memory [21, 27, 50, 75]. The *Encrypted State (ES)* and *Secure Nested Paging (SNP)* SEV extensions protect the guest state and introduce memory integrity protection. Despite these multiple protection mechanisms, attacks exploiting architectural bugs or physical properties can still lead to full compromises of the guest [71, 80].

2.3 Intel TDX

Like AMD SEV-SNP, Intel Trust Domain Extensions (TDX) allow VMs to run in a TEE [33]. With TDX, the guest memory and stored guest state are encrypted and managed by the TDX Module in the trust domain virtual processor state area (TDVPS). The TDX Module is a signed open-source software module, acting as a secure host, running in a new SEAM root execution mode, protected from the host. The host interacts with the TDX Module to, e.g., create VMs or map pages.

To protect the guest's memory while maintaining fast guest-host communication, there is a private encrypted part only accessible by guest and TDX module, and a shared, host-accessible part, distinguished by a guest physical address bit. Private memory uses an additional TD owner bit to prevent the host from reading even the ciphertext and optionally a MAC per cache line to protect against memory corruption [33].

2.4 Single-Stepping

The ability to single-step a TEE is the basis of many side-channel attacks targeting them [15,24,44,51,55,61,62,74,76, 80]. Single-stepping can be used to either increase the temporal resolution, or precisely target a specific instruction. TEEs do not provide single-stepping to the host as it is not necessary for regular operation. Van Bulck et al. [63] used the APIC timer to interrupt SGX enclaves after a single instruction, allowing attackers to single-step enclaves. Wilke et al. [76] implemented the same mechanism for SEV-based confidential VMs. Intel's TDX module includes a mitigation against single-stepping [33] that uses the timestamp counter and instruction pointer differences or, in a newer iteration, performance counters to determine if single-stepping occurred. In this case, it masks all external interrupts and runs a random number of instructions before returning back to the host. Wilke et al. [74] bypassed an early version of the Intel TDX single-stepping mitigation by reducing the CPU frequency and counting VM entries through a side channel. While Intel partially mitigated this attack using performance counters, it is still possible to determine the exact number of steps taken by the mitigation.

Intel TDX, SGX and AMD SEV directly run trusted code on the CPU cores, *i.e.*, the host still manages external hardware. Interrupts, in particular, are expected to being handled as soon as possible. Delaying them for too long can lead to usability issues in case of network or keyboard interrupts, crashes or even hardware damage in case of thermal event interrupts, *i.e.*, TEEs cannot delay them. Therefore, even with mitigations against single-stepping, a host can effectively interrupt a guest after executing a small number of instructions. Consequently, the goal of TEEcorrelate is to limit performance-counter leakage, even when the host has fine-grained control over the guest execution.

2.5 Performance Counters

Performance counters (also hardware performance counters or HPCs) are CPU registers that count hardware events, e.g., cache misses, branch mispredictions, and instructions executed. The specific events that can be tracked depend on the processor model [5, 31]. Kernel code can select multiple events to be counted via Model Specific Registers (MSRs).

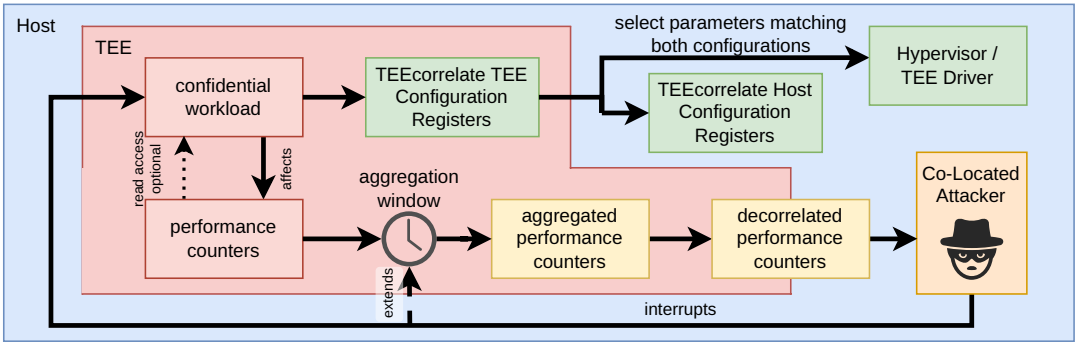


Figure 1: Host and TEE negotiate the TEEcorrelate configuration via newly introduced registers. Events in the TEE influence TEE performance counters, which are aggregated within long windows, that are additionally extended when external influences (e.g., interrupts) interfere with the execution. Attacks can only obtain aggregated counter data, substantially reducing leakage.

Performance counters are used to find application bottlenecks [34], identify resource-hogging applications or functions, and even detect some types of attacks, as shown by numerous works [11,14,16,19,39,40,82]. Prior attacks from malicious SGX enclaves [54,56] have shown that the host should try to detect attacks from TEEs, in addition to performing load-balancing and anomaly detection [34,47,60,69,77,81].

While performance counters enable hosts to optimize, load-balancing or protect their systems, fine-grained, real-time performance data, combined with single-stepping and page-tracking, has been exploited to break cryptographic code such as RSA and HQC, as well as TOTP implementations [24]. To prevent this, Intel CPUs do not allow the host to track performance events at all while an SGX or TDX TEE is actively running [30]. However, this approach also disallows hosts from using performance data for genuine applications.

Performance counters are not disabled when executing SEV VMs. Lou et al. [46] try to mitigate coarse-grained performance-counter attacks, e.g., website fingerprinting or keystroke timing, by injecting noisy code gadgets into the instruction stream. However, this does not mitigate active attacks from the host, e.g., by single-stepping or page-tracking, as the host can filter the performance counter increments caused by the gadgets. In addition, by artificially incrementing performance counters, the mitigation renders them useless for genuine applications. Adhering to constant-time standards would mitigate attacks from prior work [24]. However, doing this on full general-purpose systems, including kernel, libraries, and applications, is infeasible.

3 High-Level Overview of TEEcorrelate

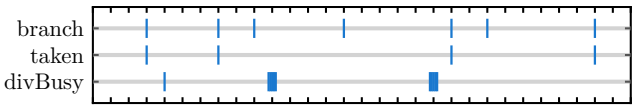
Unprotected performance counters leak information in two dimensions: First, malicious hosts can measure the exact timing of a performance counter increment by comparing counter values before and after each single-step of a TEE. Second, the amplitude, *i.e.*, the amount that the counter is incremented, can leak information as well, e.g., by leaking the size of a division operand, or by leaking the number of loop iterations in a `memcmp`-like function [24]. TEEcorrelate aims to mitigate leakage in both dimensions while keeping coarse-grained information intact and unbiased over longer timeframes.

As illustrated in Figure 1, TEEcorrelate mainly uses two components: a windowing mechanism aggregating performance counter data, and fuzzy performance counter increments. On a functional level, we introduce several changes to how TEEs and the host interact, e.g., new registers and negotiation mechanisms. This allows the host to still use performance-counter information for optimization, billing, and security purposes, e.g., anomaly detection. The security level of TEEcorrelate is configurable by changing the aggregation window size and performance counter deviation range to fit the needs of both the host and TEE.

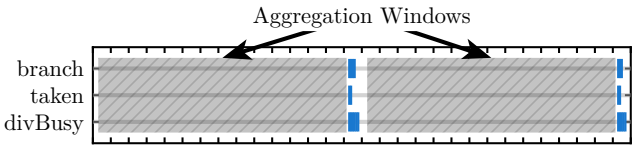
3.1 Temporal Decorrelation: Windowing

TEEcorrelate decorrelates the timings of performance counter increments using aggregation windows as shown in Figure 2. The window length is defined in retired instructions within the TEE in a configurable range of hundreds of thousands to millions of instructions. Hence, windows only pass with actual progress in the TEE. All performance counter changes within a window are aggregated and only updated at the end of the aggregation window. Similarly, activating, deactivating, or switching performance counters only takes effect at the end of an aggregation window. Consequently, the attacker does not know **at what exact time** within the window the performance counter was incremented. Furthermore, if multiple increments occur in the same window, the attacker cannot separate them.

To mitigate filling windows with unrelated activity by injecting interrupts or enclave calls, TEEcorrelate dynamically extends the aggregation window by a TEE-configurable amount for *host*-induced events. Hence, a malicious TEE cannot exploit the dynamic extension of aggregation windows, as TEE-controlled events do not extend the aggregation window.



(a) Points in time where performance counter values change without TEEcorrelate. Wider lines show multiple increments.



(b) Points in time where performance counter values change with TEEcorrelate, with a window size of 15.

Figure 2: TEEcorrelate reduces the temporal information of performance counters. The close temporal correlation between performance event and corresponding instruction is significantly reduced with TEEcorrelate as it accumulates data of the performance events over many instructions.

3.2 Decorrelation: Fuzzy Increments

We assume a powerful, fully-informed attacker, aware of our mitigation and the entire system state, except for the secret. If the attacker is only interested in the number of events that occurred during a specific time frame, they can pre-compute a prediction for how many events there are without the secret. The difference between the prediction and measurement is the number of events that depended on the secret. Thus, the attacker can still measure, e.g., the number of secret-dependent loop iterations within an aggregation window, even if the precise timestamps of the events are hidden by the temporal decorrelation. This can be combined with a sliding window attack (see Section 5.1), where the attacker performs measurements with differently aligned aggregation windows, to recover individual events. To prevent such attacks, we introduce a second primitive: *Fuzzy Increments*.

The idea behind fuzzy increments is that the host does not need extremely precise values for monitoring, load-balancing, or anomaly detection. In fact, performance counters typically do not even guarantee high precision, limiting their applicability for some security purposes [19]. Still, they generally have a high enough proximity to the actual number of corresponding events, and are generally monotonically increasing, if not manually reset [30]. This practical accuracy makes performance counters suitable for load-balancing and anomaly detection [34, 60, 69, 77]. We use this design space to let the performance counter value deviate further than they usually do, within a specified range, while preserving the monotonic increase. This allows the processor to let the reported values deviate from the real value to decorrelate effects of specific secret bits. To prevent fuzzy increments from saturating the size of the deviation window, we use a Gaussian curve or similar distribution to steer deviation away from the limits of the deviation window.

3.3 Configuration Options

TEEcorrelate provides two configuration options for the host and TEE to negotiate, affecting security and performance.

Aggregation Window Size. The aggregation window size determines how many instructions have to complete before performance counter values are updated. A larger window size linearly increases attack runtimes, as the attacker needs to wait longer to obtain a single performance counter measurement. A larger window size also aggregates more performance counter changes, making it harder to extract fine-grained information. Additionally, larger window sizes slightly decrease the performance impact of TEEcorrelate, as the value decorrelation is performed less frequently. However, larger window sizes decrease the temporal resolution of performance counters, making them less useful for legitimate applications.

In our experiments, we found that a window size of 1 million retired instructions is a good security-performance trade-off. With medium load, normal operation, our SEV-SNP CVM fills the aggregation window fast enough that performance counter data is available on most VM exits, thus not significantly affecting the temporal resolution for a benign host.

At the same time, a window size of 1 million instructions is large enough to make targeting specific events difficult. To break this decorrelation, the attacker needs to predict every single event within that window, except the secret-dependent event, which is unlikely for a window this large.

Deviation Window Size. The deviation window size determines how much the reported performance counter value can deviate from the actual value. The deviation window size is defined as a power of 2, where the minimum size is 64. The number of measurements to recover a single performance counter increment increases quadratically with the deviation window size, *i.e.*, by adding a single bit to the deviation window size, the number of measurements required increases by a factor of 4. Therefore, a larger deviation window size increases the security of TEEcorrelate.

However, a larger deviation window also decreases the coarse-grained accuracy of performance counters, rendering them less useful for benign use. In our experiments, a deviation window size of 2048 was a good trade-off between security and coarse-grained accuracy. The deviation window size does not impact the runtime performance of TEEcorrelate, as it does not introduce additional instructions, and the latency of the used operations is not affected by the operands.

4 Implementation of TEEcorrelate

Since TEEcorrelate touches hardware features like the performance monitoring unit and context switches between TEE and host, it has to be implemented in hardware, or at least in

Table 1: Summary of new fields with TEEcorrelate.[1]

Field	Bits	Location
PERF_MIN/MAX_WINDOW_SIZE	$2 \cdot 64$	TEE Configuration
PERF_WINDOW_EXTENSION_SIZE	64	TEE Configuration
PERF_DEVIATION_RANGE	64	TEE Configuration
PERF_CTR[0-N]	$N \cdot 64$	VM Save Area
PERF_CTR[0-N]	$N \cdot 64$	VM Save Area
PERF_LAST_REPORTED[0-N]	$N \cdot 64$	VM Save Area
PERF_AGGREGATION_COUNT	64	VM Save Area[2]

[1] Specific registers and their location depend on the TEE.
[2] This also requires an instruction-counting register.

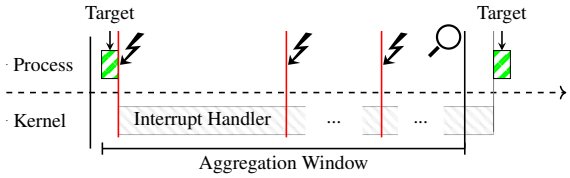


Figure 3: An attack on TEEcorrelate without aggregation window extension. After injecting interrupts to force the VM to execute interrupt handlers with predictable effects on performance counters, the host gets the target values by subtracting predicted from measured.

microcode. Regardless of the implementation approach, we overview the interface between TEEcorrelate and the TEE and host, and discuss how temporal and value decorrelation can be realized. We primarily focus on confidential VMs [6,9,33], provide concrete examples based on AMD SEV-SNP [6], and discuss how TEEcorrelate can be introduced into Intel SGX [32] and Intel TDX [33]. TEEcorrelate can also be used in full-system TEEs like ARM TrustZone [8,10].

4.1 Extending the TEE Interface

TEEcorrelate requires the cloud provider and customer to agree on suitable parameters, with a resolution and accuracy high enough for the cloud provider while still fulfilling the security requirements of the customer. Both the confidential VM and the host get new fields to configure ranges for parameters, specifically for the minimum and maximum window size, window extension size, and deviation range. These fields are stored in a secure area, e.g., the VM Control Block (VMCB) or an SGX configuration register. On Intel TDX, memory in the TDVPS can be used instead of MSRs.

Start and Attestation. Before starting the TEE, the host sets the parameters to overlap with the ranges of the customer such that the TEE can pass remote attestation. These fields are part of the attestation report and cannot be changed later.

TEE Execution. During TEE execution, the host cannot read the performance counters, and no overflow interrupts are sent to the host. TEEcorrelate uses the existing performance counter hardware registers and clears them before returning to the host. The performance counters may be exposed to the TEE, e.g., for confidential VMs, or not, e.g., for Intel SGX. Intel TDX already has a mechanism for switching performance counters upon confidential VM entry and exit [33].

VM Exits. On VM exits within an aggregation window, the performance counter values must be stored inaccessible to the host. For confidential VMs, we add fields to the VM Save Area (VMSA) on AMD SEV-SNP and in the TDVPS for Intel TDX. We require three 64 bit fields for each performance counter, to store the current configuration, the current value, and the previously reported value. The number of available performance counters N is dependent on the specific hardware platform: We measured 6 per logical core on a recent AMD EPYC Zen 4 machine, and 8 on a recent Intel Xeon Emerald Rapids machine. Furthermore, we require one additional 64 bit field PERF_AGGREGATION_COUNT for the temporal decorrelation implementation. We show an overview of the newly introduced fields and their memory usage in Table 1.

While saving and restoring the performance counter state for the VM is performed on VM entry and exit, the host is responsible for saving and restoring their own.

4.2 Temporal Decorrelation

TEEcorrelate only exposes performance counter values at VM exits if the aggregation window, a number of retired instructions, has passed. Within a TEE, the counters increment normally. Upon a VM exit within an aggregation window, all current values are saved and cleared before passing control to the host. Thus, performance counters always report values of 0 when exiting the TEE while the aggregation window is not yet filled. If the aggregation window is full, the host can read the aggregated and value-decorrelated counts.

Malicious Configurations. An attacker could gain a fine-grained read primitive by changing performance counter settings close to the start or end of an aggregation window. Thus, the processor only allows control register updates at the start of a new aggregation window. The processor also ignores the OsUserMode bits, to prevent the host from leaking fine-grained values by forcing the VM to kernel space.

Mitigating Active Sliding Window Attacks. For this attack, shown in Figure 3, the host forces the TEE to only execute a very small region of victim code by "filling" the rest of the aggregation window with interrupt handler code by repeatedly injecting interrupts. If the host knows how the interrupt handler code influences the performance counters, it can deduce the performance counter changes by the victim code.

To prevent this, we extend the aggregation window by a configurable amount, the *window extension size*, whenever an interrupt is injected into the VM. The value is chosen such that a significant amount of non-host-controllable code is executed in the extended window. Choosing a value too large resets the aggregation window more often, which is only a functional limitation for the host when interrupting

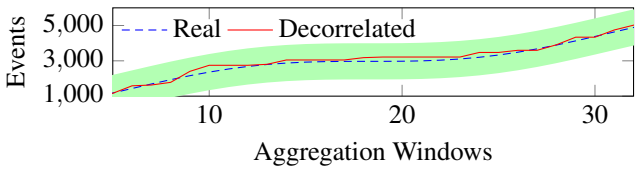


Figure 4: Performance counters are decorrelated from actual events but follow the trend within the deviation window.

the victim too frequently. Choosing a value too low does not properly mitigate the attack. In practice, we measured an average of one million instructions between normal interrupt injections for regular usage like web-browsing, three million when compiling the Linux kernel, and 80 000 when idle, on a VM running Xubuntu 24.04.1 Minimal.

Performance Counter Hiding by a Malicious VM. A malicious VM might try to hide performance counter increments by increasing the timer interrupt frequency to an interval shorter than the *window extension size*, always resetting the aggregation count to 0. However, since windows are only extended on host-injected events, the host can detect this and act accordingly, e.g., by delaying interrupt injections.

4.3 Value Decorrelation

To prevent differential attacks, where the difference to a predicted performance counter change is measured, we introduce fuzzy increments. The TEE and host negotiate a maximum deviation from the actual value, the *deviation range* (see Figure 4). To ensure monotonically increasing counters, TEEcorrelate only shows new values to the host when they are larger than previously reported values and old values otherwise.

Randomness. We need to introduce a sufficient amount of randomness into performance counter values to prevent the recovery of the original trace, even across numerous measurements. The randomness must fulfill the following two requirements: **First**, the randomness should almost never reach the maximum deviation. Otherwise, the attacker could see the change from probability zero to non-zero and infer a performance counter value change. To prevent this, the deviation of the exposed value from the real value follows an approximated normal distribution. **Second**, the overhead for context switching, *i.e.*, VM entry and VM exit, should be minimal, ruling out computationally heavy instructions like divisions or cryptographic functions.

Fast Normal Distribution. The random offset between real and reported performance counter value is an integer. A binomial distribution with a success probability of 0.5 is a discrete approximation of a normal distribution. To generate a random number with such a distribution, we can count the number of '1's in a random number. However, to cover a deviation range of n values, we need to generate n random bits, which is slow. Thus, we split our distribution into binomial-distributed sub-ranges *i.e.*, *buckets*. The final performance counter offset Δ is constructed programmatically by concatenating the binomially generated bucket index B with the linear random bucket offset O. We only obtain the 6 bit bucket index B from the binomial distribution, by counting the number of '1's in a 64 bit random number. We then generate the sub-offset O within the bucket as a linear random value. This distribution has similar properties to a binomial distribution, *i.e.*, it is symmetric around a center, and the probability of reaching the edges of the deviation range is very low (e.g., a bucket index greater than 60 is chosen only once every 27×10^{12} times on average.). However, it requires significantly fewer random bits to generate. Finally, we shift Δ by half the deviation range, to center the distribution around zero (cf. Algorithm 1.).

Running our value decorrelation on a real performance counter trace shows that the difference between the real and decorrelated value follows a normal distribution, with the mean offset being close to 0, *i.e.*, the decorrelated value is close to the real value on average (see Figure 5). For very-slowly incrementing (e.g., once in a few windows) performance counters, the offset drifts upwards in the deviation range. However, with 64-bit numbers used for bucket selection, it is infeasible to completely saturate the deviation range.

Algorithm 1: The value decorrelation algorithm. We get a binomial random value by counting the number of ones in a 64-bit number, and use it as the 6 most significant bits of our offset. We fill up the remaining bits with linear randomness.

Input: The old reported HPC value P_o
Input: The current real HPC value P_r
Result: The new reported HPC value P_n

```
begin
    B ← POPCNT(RNG(64));
    S_b ← TZCNT(BUCKET_SIZE);
    O ← RNG(S_b);
    X ← ((B ≪ S_b) ∨ O) − (SPEC_RANGE ≫ 1);
    P_n ← X + P_r;
    if P_n < P_o then
        P_n ← P_o;
    end
    return P_n
end
```

Table 2: Estimated overhead of TEEcorrelate per performance counter on AMD SEV-SNP. This overhead only occurs once at the end of each aggregation window.

Instructions	Cycles	Usages Each	Total
`RDRAND`	75	2	150
`TZCNT`	2	1	2
`POPCNT`, `OR`, `CMP+JCC`, `SHL`, `SHR`, `ADD`, `SUB`	1	1	7
Total Cycles			159

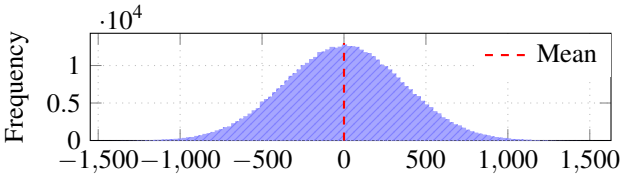


Figure 5: Histogram of deviations of the TEEcorrelate-reported HPC value from the real value with window size 4096 on a real trace of the `cpu_core/l2_request.miss` event. The mean of the distribution is 0.001 16 , and the offsets form a binomial distribution around the mean.

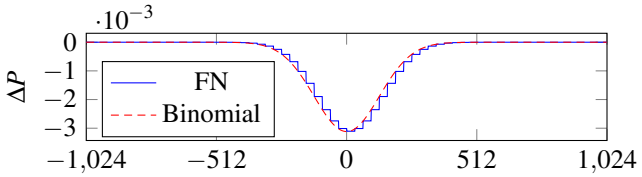


Figure 6: The probability differential $\Delta P = P(X) - P(X-1)$ depending on the offset between current real performance counter value and last reported counter value, for a deviation window size of 2048. This graph shows that the maximum probability differential is equal between an ideal binomial distribution and our fast normal (FN) distribution.

4.4 Performance Overhead

For estimating the performance impact of TEEcorrelate upon entering and exiting the VM, we do not consider swapping the performance counter data as overhead, as other mitigations against performance-counter attacks, like PMC virtualization [7, 33] do the same. The only overhead is caused by Algorithm 1. Therefore, we sum up the worst-case latencies of the used instructions for a Zen 3 machine [22], see Table 2.

With 159 cycles per performance counter, we arrive at 954 cycles of overhead in total, for full aggregation windows. It is likely that processors supporting SEV-SNP have more efficient ways to generate randomness than the `RDRAND` instruction, due to the high amount of randomness required for *VMSA Register protection*. However, as we do not know the performance of any potential internal optimizations, we still compute our estimate using `RDRAND`.

Wilke et al. [76] show that entering a SEV-SNP VM, executing a single `NOP` instruction, and returning to the host, requires a median of 6 061 cycles. We use this number as a baseline to estimate the number of cycles required to context-switch between a SEV-SNP VM and the host. Adding 954 cycles to this baseline cycle count results in an overhead of 16 % per context switch. However, this overhead only occurs when the aggregation window is filled. In all other cases, the overhead is only 2 cycles (0.03 %) per context switch, for adding the number of instructions executed in the current TEE timeslice to the `PERF_AGGREGATION_COUNT` register, and comparing the result against `PERF_WINDOW_SIZE`. Thus, for an aggregation window of 1 000 000 instructions in the VM (assuming single-cycle instructions), and 6061 cycles to context switch, the maximum overhead is 0.09 %.

4.5 Reduced Number of Buckets

As shown in Table 2, the performance overhead is dominated by the 2 `RDRAND` instructions. We can reduce the performance overhead by using a bucket size of 32 bits, allowing us to use half of the random bits for the bucket index, and the other half for the linear random value. This reduces the number of `RDRAND` instructions to one, saving 75 cycles. Due to extra required instructions to split the 64-bit random value into two 32-bit values, the total overhead is 89 cycles per performance counter, for a total of 534 cycles per aggregation window.

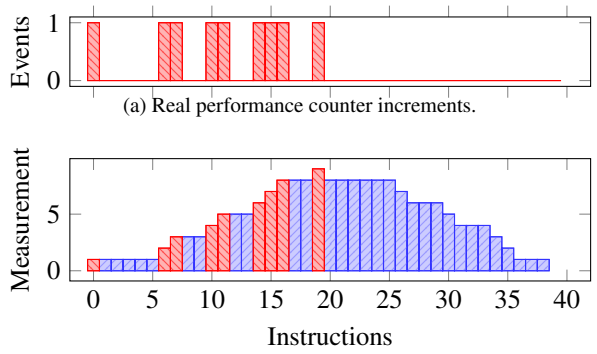
This does not reduce the security of TEEcorrelate in the attacks from Section 5. Instead, it flattens the distribution, *increasing* the number of samples required to recover a bit at the ideal location, while decreasing the required samples on other locations. However, with only 32 bits of randomness, the probability of selecting the highest bucket index (*i.e.*, generating a random value consisting of 32 ones), is 1 in 2^{32}. This probability is likely enought that it will occur multiple times over a long runtime of weeks or months.

While we did not find any attacks exploiting deviation window saturation, the increased probability is a potential attack vector for future work. Thus, the number of bits used in the bucket index is a trade-off between performance and security. As the performance improvement of a reduced number of buckets is rather small compared to the overall performance overhead, we recommend using 64 bits for the bucket index generation, and will use this value for the rest of this paper.

5 Security

In the following, we analyze the security of TEEcorrelate. As the security layers of TEEcorrelate are independent of each other, we analyze their security properties separately and afterward compute the overall lower bound security improvement.

Threat Model. We design TEEcorrelate against a fully-informed attacker knowing everything about the VM except the secret and having fine-grained control over the execution of the TEE. While no prior work achieved this, we assume they are capable of 100 % reliable single-stepping, without any unexpected zero- or multi-steps. There is no noise or other disturbance. While for some attacks the specific number of increases or the exact timestamps of increases may be required in practice, for our analysis, we assume that the attacker already wins if they can detect *whether* a performance counter has increased, *i.e.*, they do not need to recover the value of the performance counter increment.



(a) Real performance counter increments.

(b) Events measured with sliding aggregation window of size 10.

Figure 7: The attacker can still recover secret performance counter increments by sliding the aggregation window over the secret. Every peak in Figure 7a corresponds to an increment in the aggregated measurements in Figure 7b.

5.1 Temporal Decorrelation

TEEcorrelate aggregates performance counter measurements over a configurable instruction window. In our threat model, the attacker knows everything about the system state except the secret. This means that they can subtract the number of expected performance counter increments from the measured value, to get only the secret-dependent increments. With a large window size, multiple key bits likely get aggregated together, preventing the identification of individual bits.

Active Sliding Window Attacks. We assume the attacker has fine-grained control over the execution of the TEE. Consequently, they can control where an aggregation window starts and ends. The attacker fills the aggregation window with unrelated operations with a known effect on the performance counters, e.g., an interrupt handler. Using page-tracking to prevent the execution of any code besides the chosen interrupt handler, they can ensure that all performance counter updates after the target instructions are predictable. By placing the aggregation window so that only the first secret bit is included in the aggregation window, the attacker can successfully leak that bit. Then, sliding the aggregation window over the entire secret leaks it, see Figure 7.

Security of TEEcorrelate against Active Sliding Window Attacks. However, instead of a single trace like in previous attacks, the attacker requires one trace per secret bit. Consequently, temporal decorrelation increases the attack time in proportion to the number of secret bits. Such an attack incurs an additional significant slow-down, depending on the aggregation window size, as it defines the minimum number of executed instructions for each measured secret bit. In addition, TEEcorrelate mitigates sliding window attacks by dynamically extending the aggregation window whenever the host injects an interrupt into a TEE. With this dynamic extension of the aggregation window, the aggregation window cannot be filled with well-known interrupt handlers anymore.

5.2 Value Decorrelation

To compute the security gain from our value decorrelation, we again assume a strong attacker with the same capabilities as in Section 5.1. The attacker can target the aggregation window such that only a single unknown secret bit is measured at a time. For simplicity, we assume that the secret-dependent performance event is the only event in the aggregation window, e.g., when only using temporal decorrelation, the attacker would measure either 0 or 1 events, depending on the secret bit. With value decorrelation, this correlation disappears.

In the following, we introduce foundational properties of the binomial distribution, which we use to model the properties of our fast normal distribution. We then construct formulas modelling those properties for our fast normal distribution. We use these formulas to accurately compute the probability of performance counter increments with our distribution, based on the previous performance counter offset. We then explain statistical sampling and how to compute the number of samples required to recover a single bit of the secret. Finally, we use these probabilities to compute how many samples the attacker requires to recover a single bit of the secret.

Binomial Distribution. An important property of probability distributions is the *probability density function* (PDF) for continuous distributions, and the *probability mass function* (PMF) for discrete distributions. This function describes the probability of a random variable taking on a specific value k. For a binomial distribution, the PMF is given by

$$f(n,k) = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}, \tag{1}$$

where n is the number of independent experiments, k is the number of successful experiments , and p, the success probability of each event (*i.e.*, 0.5 for a uniformly random number).

The cumulative distribution function (CDF) is the probability of a random variable being less than or equal to k. In discrete distributions, this is the sum of the PMF for all values less than or equal to k:

$$F(n,k) = \sum_{i=0}^{k} \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i}. \tag{2}$$

If our distribution were a binomial distribution, we could use the CDF to compute the probability of a performance counter increment, depending on the distance between the last reported performance counter value and the current real performance counter value. However, because our fast normal distribution is based on a binomial distribution, we can use these formulas as a basis to model the cumulative distribution function for our fast normal distribution.

Modelling the Fast Normal Distribution. As described in Section 4, the most significant bits of our random number are binomially distributed. This is achieved by counting the number of '1's in a 64 bit random number. The remaining bits are filled with a linear random value. We express this construction as $X = B||O$, where X is the final random number, B is the binomially distributed bucket index, and O is the linear

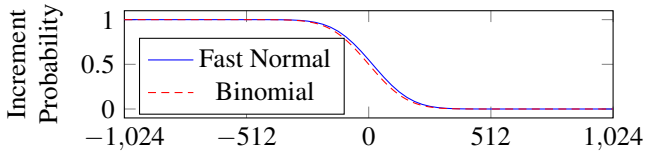


Figure 8: The probability of measuring a performance counter increment, depending on the offset (x-axis) between current real performance counter value and last reported performance counter value, for a deviation window size of 2048.

random offset within the bucket. To model the cumulative distribution function of our fast normal distribution, we need to combine the CDF of the binomial distribution with the CDF of a linear distribution. We start with the probability of a randomly generated bucket index B' being *less* than our current bucket index B. As the CDF expresses the probability of a random variable being *less than or equal to k*, the probability to be *less than* is $F(64, B-1)$. The probability of a randomly generated bucket index B' being *equal to B* can be expressed via the PMF, *i.e.*, as $f(64, B)$. To account for the linear random offset O within the same bucket, we add the probability of a linear random value O' being *less than or equal to O*, which is $O/\texttt{BUCKET\_SIZE}$. Adding these probabilities, we arrive at the final formula for the CDF of our fast normal distribution:

$$P'(X) = F(64, B-1) + f(64, B) \cdot \left(\frac{O}{\texttt{BUCKET\_SIZE}} \right).$$

For our security analysis, we need to compute the probability of a performance counter increment, which is the probability of a random variable X being *greater than* X'. Thus, we can compute the probability of a performance counter increment as $P(X) = 1 - P'(X)$.

Modelling a Value Distinguishing Attack. To evaluate the security level of our defense, we consider a basic case, namely that the attacker distinguishes one value from a neighboring value. In this attack, the attacker needs to determine whether the real performance counter increased within an aggregation window. They can do this by observing whether the reported performance counter value increases or not. In our attack model, we assume that, across the entire measurement, there is only a single, secret-dependent performance event that increments by either 0 or 1, and no other changes to the real counter. Depending on whether the real performance counter value increased, the offset between the real and reported performance counter value changes. This causes the probability of a performance counter increment $P(X)$ to change. We define the difference between the two probabilities as $\Delta P(X) = P(X) - P(X-1)$. To account for a worst-case scenario, we assume that the attacker has control over X, *i.e.*, the attacker can control the deviation to maximize ΔP at the start of the measurement. We argue that, in such a strong threat model, the difference between a real binomial distribution and our optimized version is irrelevant, because the adversary has precise enough control over X that it can choose an optimal window for ΔP in any case.

Figure 6 shows the probability differentials of $\Delta P(X) = P(X) - P(X-1)$ for any X in the parameter space for a binomial distribution and our fast normal distribution. For a deviation window size of 2048, the highest possible probability differential $\Delta P = |P(X) - P(X-1)|$ across the parameter space is 0.0031. Even though our fast normal distribution has more possible values for X where ΔP is maximized, the maximum value of ΔP is equal between binomial and fast normal distributions.

Computing Sample Sizes. To recover data when TEEcorrelate is enabled, the adversary needs to collect enough samples such that they can distinguish the two distributions $P(X)$ from $P(X-1)$. The standard deviation of a random binary value with a specific probability can be estimated as

$$\sigma = \sqrt{\frac{p \cdot (p-1)}{n}},$$

where p is the specific probability of the event (e.g., $P(X)$), and n is the number of measured samples.

To distinguish the two events with probabilities $P(X)$ and $P(X-1)$, the attacker needs to keep collecting samples until the standard deviation shrinks enough to separate the two events up to a sufficient probability. Due to the large parameter space of TEEcorrelate, we compute the required sample size for a distinguisher using a z-test. We want to distinguish two cases: First, that our measured probability $P(M)$ is larger than $P(X)$. Second, that our measured probability $P(M)$ is smaller than $P(X-1)$. The *z-score* is a standardized way to express the distance of a measured value from the real value, based on the standard deviation. For example, the z-score for a confidence of 0.9 is 1.28, which means that 90 % of measurements are below $\mu + \sigma \cdot 1.28$, where μ is the mean of a distribution, and σ is the standard deviation.

Number of Samples Required by the Attacker. To distinguish $P(X)$ and $P(X-1)$, the attacker wants to reduce the likelihood of mistaking $P(X)$ for $P(X-1)$ and vice-versa below a confidence level α. Therefore, the distance $Z_\alpha \cdot \sigma$ must be smaller than $|P(X) - P(X-1)|$, where Z_α is the z-score for our chosen confidence. As we consider both failure cases (*i.e.*, mistaking $P(X)$ for $P(X-1)$ and vice-versa), the probability α accumulates multiplicatively, *i.e.*, for $\alpha = 0.9$, we achieve a total confidence level of 0.81.[2] The attacker can compute the difference in values as

$$p_1 = P(X), \qquad p_2 = P(X-1),$$
$$|p_1 - p_2| = z_\alpha \cdot \sqrt{\frac{p_1 \cdot (p_1-1)}{n}} + z_\alpha \cdot \sqrt{\frac{p_2 \cdot (p_2-1)}{n}}.$$

Finally, we compute the number of samples needed as

[2] For cases where the performance counter increments by more than 1 depending on the secret, we can replace $P(X-1)$ with $P(X-k)$, where k is the expected performance counter increment. This reduces the number of samples required to distinguish the two cases.

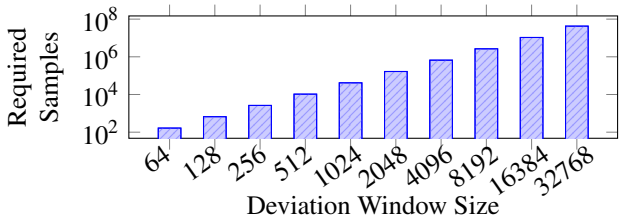


Figure 9: The required samples to reach $\alpha = 0.9$ for each sliding window size. While a Binomial distribution performs slightly better than the bucket-based approximation, the approximation reduces the required number of samples by less than 4 % on average.

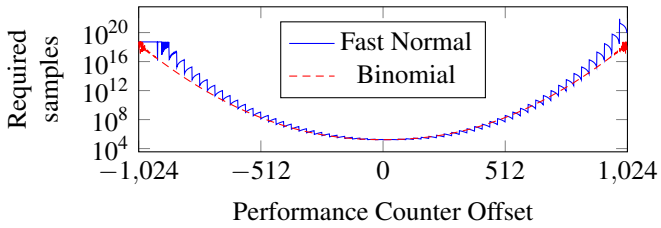


Figure 10: The required samples to distinguish $P(X)$ and $P(X-1)$ with $\alpha = 0.9$, with a deviation window size of 2048. The discretization of the probability distribution introduces non-linearities. Sample size is smallest towards the center where ΔP is the largest.

$$\sqrt{n} = \frac{z_\alpha \cdot \left(\sqrt{p_1 \cdot (p_1 - 1)} + \sqrt{p_2 \cdot (p_2 - 1)}\right)}{|p_1 - p_2|}$$

$$n = \frac{z_\alpha^2 \cdot \left(\sqrt{p_1 \cdot (p_1 - 1)} + \sqrt{p_2 \cdot (p_2 - 1)}\right)^2}{|p_1 - p_2|^2}.$$

We can see that the number of samples is inversely proportional to the probability differential in a given point, *i.e.*, the larger $|p_1 - p_2|$, the lower the number of samples required to distinguish the distributions. Thus, with an ideal binomial distribution, the optimal offset for measuring these probabilities is at $X = 0$, *i.e.*, when the reported performance counter value is equal to the real performance counter value. Due to the nonlinear properties of our cumulative distribution function $P(X)$, the minimum sample size is smaller, and is located at the bucket transition nearest to offset 0 due to the lower standard deviation $\sigma = p \cdot (1 - p)$ away from the center of the probability distribution, while the probability differential stays the same for the entire bucket. However, this only reduces the number of required samples by less than 4 %, while allowing us to implement TEEcorrelate using only bitwise operations, optimizing performance and side-channel resistance. We show a graphical representation of the number of required samples in Figure 9.

Results. In practice, the sample size scales as the square of the deviation window size, *i.e.*, for each additional bit in the maximum offset (doubling the deviation window size), the sample size increases by a factor of 4. To reach a confidence level of 0.81 ($\alpha = 0.9$), which still incurs a 19 % chance of detecting the wrong value, we need 166 077 samples. Instead of leaking the entire secret in a **single** trace, we require over **150 thousand traces** to recover *a single bit*.

Table 3: Estimated number of samples required to distinguish probabilities at different deviation window sizes.

Win. Size	Min. Samples	Win. Size	Min. Samples
64	165	2 048	166 077
128	663	4 096	663 679
256	2 627	8 192	2 653 452
512	10 437	16 384	10 611 269
1 024	41 597	32 768	42 439 986

Table 4: Summary of case study runtime results.

Case study	No mitigation	TEEcorrelate	Overhead
String Comparison	18.14 s	34.7 days	× 160 thousand
Lookup Table	0.58 s	285.4 days	× 40 million
RSA	7.15 min	824.6 days	× 160 thousand
HQC[1]	6.13 min	27.0 days	× 6 thousand

Numbers based on our recommended deviation window size of 2 048.

[1] Estimated runtime based on number of required single-steps and number of oracle calls required with an ideal oracle.

6 Case Studies

Prior work presented four attack case studies using performance counters on AMD SEV-SNP confidential VMs [24]. In this section, we analyze the same case studies to show how much slower attacks become with TEEcorrelate. Like in Section 5, we assume a fully-informed attacker that knows everything about the system except the secret and has strong primitives to control the aggregation window and performance counter offset, as well as perfect single-stepping. We compute the number of traces required to mount the four attacks with and without TEEcorrelate. We evaluate different deviation window sizes from 64 to 32 768, with 2 048 as our recommended default. Table 4 shows the overall results for 2 048, showing that all attacks are mitigated.

6.1 Mitigating String Comparison Leakage

Simple string comparison functions iterate over the two input strings, comparing them character by character, returning as early as a mismatch occurs. The early return can be measured with the *Retired Taken Branches* performance counter. Gast et al. [24] attacked a TOTP library with this primitive, recovering a 6-digit TOTP token by guessing character by character. With an average of 31.1 guesses and 0.58 s per guess, the brute-force duration then is 18.14 s on average, staying within the 30 s window required before a token change.

With TEEcorrelate, an attacker needs a large number of traces to reliably measure a single performance counter increment. However, the attacker does not need to measure every single increment of every unknown branch instruction. Instead, the attacker can measure the entire string comparison, and distinguish between increments from 0 to 6. While distinguishing increments that are not at the sweet spot increases the theoretically required number of samples for a confident measurement, we still use the minimum number of required samples to simplify our computation. In reality, our results are the lower bound of required samples for our given setup.

Even for the smallest possible deviation window size of 64 (cf. Table 3), the attacker needs approximately 165 traces to measure the outcome of a single brute-force iteration with a confidence level of 81 %. Consequently, with an average of 0.58 s per trace, and 31.1 guesses for a successful attack, the average TOTP token attack time is 49.6 min. Hence, even with the smallest possible window size, TEEcorrelate raises the runtime of brute-force attacks beyond any reasonable validity window of a TOTP token.

With larger deviation window sizes, the runtime increases further: Using our recommended size of 2048, the attacker needs approximately 160 thousand (166 077) samples to distinguish the two cases. This increases the required time for a single iteration of the brute-force attack to 26.76 h. Thus, to recover the full TOTP token, the attacker would need 34.7 days on average. For a large deviation window of 32 768, a single iteration takes 284.9 days, while a full brute-force with 31.1 attempts takes 24.3 years. Hence, we consider this attack fully mitigated with all window sizes.

6.2 Mitigating Lookup Table Leakage

The second case study presented by Gast et al. [24] recovered the entire TOTP secret key using the *Retired Taken Branches* performance counter. Gast et al. [24] found the root cause to be an insecure Base32 decoding process, iterating through a 32 character lookup table to find the index for the value to decode. Similar to the string comparison, the reverse lookup has an early exit, visible through performance counters. Without a defense, Gast et al. [24] recover the secret in a single trace.

With our temporal decorrelation, the attacker needs to measure all bits of the secret individually. The secret is the number of iterations during the reverse lookup, for each character. Thus, we need 1 to 32 measurements per base32-encoded character, *i.e.*, 16 iterations on average. In contrast to Section 6.1, the attacker cannot measure the entire code at once, as this would overlap measurements of multiple independent characters in a non-reversible way. Additionally, the attacker does not know when the code switches to the next character. Thus, every secret branch outcome must be leaked individually.

With a commonly used 16-character key [24], and 16 lookups to find the correct character on average, we need 256 measurements to recover the entire secret. With the smallest possible window size of 64, we again need 165 traces for an 81 % confidence in our measurement, resulting in a total of 42 240 measurements for a full key recovery on average. The secret key can be extracted from the same traces generated in Section 6.1. With the same runtime of 0.58 s for a single trace, we obtain 6.8 h on average, or 13.6 h in the worst case. While this slows down the attack by over 40 000, TEEcorrelate does not provide sufficient security with such small parameters.

However, with a deviation window size of 2 048, the attacker requires approximately 40 million measurements (computed as 42 515 712) on average to recover the secret key. With an average runtime of 0.58 s per trace, this results in a runtime of approximately 285.4 days. A large deviation window of 32 768 increases the average number of required traces to over 10 billion, increasing the runtime to approximately 200 years.

6.3 Mitigating RSA Secret Key Leakage

RSA square-and-multiply exponentiation executes different functions depending on the secret exponent bit processed. It is vulnerable to, e.g., timing [36], cache [45, 79], branch prediction [1], hardware- and software-based power [37, 44, 48], and performance counter [24] side channels.

To estimate the impact of TEEcorrelate on such an attack, we assume the worst case: The attacker manages to align aggregation windows to have only one secret event per aggregation window. This means that the temporal decorrelation is effectively bypassed. Gast et al. [24] state that their attack on RSA had an average runtime of 7.15 min. Again, we compute the expected runtime with TEEcorrelate for a small (64), medium (2 048), and large (32 768) deviation window size.

When configuring TEEcorrelate with a deviation window size of 64, the attacker needs 165 measurements to reach 81 % confidence in the result, this takes 19.7 h on average. While this is clearly not secure enough, it degrades performance-counter attacks from single-trace to multi-trace already, opening up time for detection of the attack. With a medium-sized deviation window size, the attacker needs about 160 thousand traces to distinguish whether a branch was taken. This attack requires 824.6 days on average of single-stepping to get the required number of traces. While this could be feasible in an offline attack, it is unlikely that single-stepping a TEE for such a long time would not trigger any suspicion or monitoring alerts. For large deviation windows of 32 768, the attacker needs about 40 million traces to recover the key, resulting in a total runtime of over 500 years.

6.4 Mitigating HQC-KEM Key Recovery

Schröder et al. [53] presented an attack on the HQC KEM, that works by distinguishing a chosen message from modified messages. They use a timing side channel in the `DIV` instructions on Zen 2, used by the seed-dependent random number generator to build an oracle. Gast et al. [24] implemented a

similar attack using the *Div Cycles Busy* performance counter. Dong et al. [20] show an improved attack method, claiming as few as 460 required plaintext-checking (PC) oracle calls.

The HQC attack exploits the input-dependent cycle count of the `DIV` instruction. Larger quotients yield higher cycle counts, measurable using performance counters. Measuring only the total division latency is sufficient to building the PC-oracle [24]. Thus, we can measure all divisions in one aggregation window, and target the total difference in division latencies to build a distinguisher. This eliminates the need to measure individual bits, and even reduces the number of required samples, as the difference in probabilities ΔP increases with a bigger difference of performance counter increments.

The attack by Schröder et al. [53] is split into an offline brute-force and an online trace generation phase. For Zen 3 we approximate the probability of a fast division as a constant $p_0 = 512 \cdot (17\,669 - 37)/2^{32} = 9\,027\,584/2^{32} \approx 0.002$ [24]. There are $75 \cdot 3$ divisions in total, and the attacker must brute-force a fast seed to build the PC-oracle. Using the binomial distribution CDF we can estimate the number of attempts required to generate a seed with at least k fast divisions using a brute-force rate of 65 000 MH/s, which is the hash rate of an Nvidia RTX 3090 on MD5 [17]. Gast et al. [24] report that the tracing process requires approximately 800 single-steps. We assume an optimistic 1 ms per single-step [76], which means that the attacker requires 800 ms per trace.

We do not consider the extra runtime required to fill the aggregation window, since this part of the code does not need to be single-stepped. The number of required samples to distinguish cases depends on the probability differential $\Delta P = P(X) - P(X-k)$, where k is the difference in performance counter increments for both cases. Therefore, k is equal to the number of fast divisions resulting from the brute-forced seed. Thus, the total runtime of the attack is

$$T(k) = \frac{1}{F(225,k) \cdot R} + \frac{N(k,\alpha) \cdot Q_\alpha}{T_t},$$

where R is the hash rate (65 000 MH/s), T_t, the attack runtime per trace (800 ms), $F(n,k)$, the CDF of the binomial distribution, $N(k,\alpha)$, the minimum number of required samples for a confidence of α for a given performance counter increment, and Q_α, the number of oracle calls required by the attack at confidence α. For all window sizes, we found the minimum runtime to be reached at an oracle reliability of 0.95, where the attack [20] requires 664 calls.

The number of fast divisions for the fastest possible attack runtime is different between window sizes. For the different window sizes, these are $k = 10, 12$, and 13, respectively. The total attack runtime is estimated to be 43.4 min, 27.0 days, and 15.1 years, respectively. Thus, even with a very powerful attacker and worst-case assumptions, TEEcorrelate can protect secret data from leaking through performance counters when using the appropriate deviation window size.

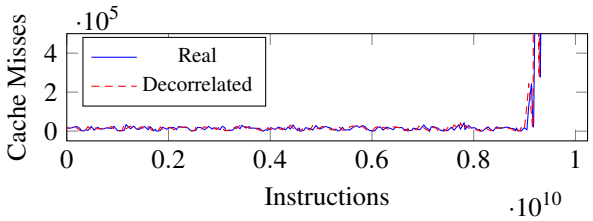


Figure 11: Real and decorrelated HPC to detect resource exhaustion or anomalies. When the system starts cache-thrashing due to the running workloads, both the real and decorrelated performance counters jump up extremely.

7 Discussion

The goal of TEEcorrelate is to mitigate fine-grained performance-counter attacks, while preserving the course-grained trend data for genuine use. In contrast to the mitigation presented by Lou et al. [46], performance counter data collected with TEEcorrelate closely follows the real performance counter values, with a configurable maximum deviation.

Limits. Naturally, the defensive capabilities of TEEcorrelate are significantly reduced when a secret-dependent event increments the target performance counter by a large amount. In extreme cases (e.g., a secret-dependent branch that executes thousands of performance-counter incrementing instructions), TEEcorrelate is unable to provide any protection at all. However, code written in such a way is likely vulnerable to a multitude of other side channels as well [23, 41, 42, 44, 64, 70]. TEEcorrelate does not fully eliminate performance counter leakage, but is a lightweight defense for large amounts of real-world code, for which it raises the runtime of any performance-counter-based attacks far beyond any realistic time frame. In addition, it completely eliminates single-trace attacks.

Temporal Decorrelation. Temporal Decorrelation reduces the resolution of the performance counter data by aggregating it until the aggregation window is filled. In our tests, the TEE executed roughly 1 million instructions between injected interrupts during regular usage (e.g., web browsing). During heavy load (e.g., compiling), we measured an average of 3 million instructions between interrupts. Thus, under regular conditions, with our recommended aggregation window size of 1 million, the aggregation window is filled on most control transfers to the host. Therefore, temporal aggregation is unlikely to affect genuine performance counter applications.

Value Decorrelation. Value Decorrelation allows the reported performance counter value to deviate from the real value within a configurable window size. With our recommended size of 2048, the reported value can deviate by up to 1024. The deviation is chosen such that the reported value is more likely to be close and slightly above the real value. Value decorrelation can cause the reported value to increment by more than the real value. Over a single aggregation win-

dow, this behavior may lead to erratic jumps that can affect unmodified performance-counter-based detection algorithms.

Performance Counters for Cloud Scheduling. Load-balancing and anomaly detection [34,47,60,69,77,81] are crucial in cloud scheduling to prevent multiple VMs from combinedly bottlenecking a hardware resource, e.g., the last-level cache. Zhang et al. [81] investigated a scenario with a perpetrator and a victim and Mars et al. [47] on competitive sharing. Both use performance counters similar to Wang et al. [69]. All of these works detect high contention levels or bottlenecks, and allow the cloud provider to reschedule VMS or applications accordingly. Mars et al. [47] use either dynamic or fixed thresholds (e.g., $\geq$1 500 cache misses per 1 ms). Such thresholds are far from the subtle differences we protect with TEEcorrelate, as they focus on avoiding exhaustion of a hardware resource. Figure 11 shows how the decorrelated performance counters on a large scale of changes (*i.e.*, $\geq$40 000 cache misses) stay close to the actual values and respond quickly to these extreme increases, e.g., an application thrashing the cache. We observe that with TEEcorrelate, in the worst case, the threshold is exceed exactly one aggregation window later, *i.e.*, less than 1 ms after this rapid increase.

Anomaly Detection. Prior work showed that attacks from within a TEE can compromise the host or co-located VMs [54, 56]. Performance counters can be used to detect attacks from within a malicious TEE. Li et al. [40] detect Rowhammer attacks by sampling performance counters every 100 ms. Cloudflare uses performance counters to detect Spectre attacks in their Workers service [58, 68]. They sample the performance counters every 50 ms, achieving no false negatives and only 0.61 % false positives. With such a low sampling rate, temporal decorrelation does not affect measurements at all. Additionally, over such a long time-frame, the noise introduced by value decorrelation is negligible compared to the extremely large number of cache misses caused by Rowhammer, branch mispredictions caused by Spectre attacks or similar resource-exhausting attacks. The presented examples show that coarse-grained algorithms are not affected by TEEcorrelate. Thus, TEEcorrelate indeed allows cloud providers to optimize their server usage and protect against malicious guests, while mitigating fine-grained leakage of guest data.

Preventing Single-Stepping. The attacks presented by Gast et al. [24] rely on single-stepping the TEE to measure performance counter data. Thus, by mitigating single-stepping, as in Intel TDX, the reliability of their attacks is greatly reduced. However, Wilke et al. [74] bypassed to TDX's single-stepping mitigation, re-enabling the attack. Additionally, the mitigation forces the TEE to progress for only up to 32 instructions before returning to the host. Wilke et al. [74] show that this induced multi-stepping is fine-grained enough to still leak secrets. Therefore, with enabled performance counters, this single-stepping mitigation alone does not prevent fine-grained performance-counter attacks. In general, single-stepping mitigations are a trade-off between security of the guest and reliability of the host, as the hardware cannot defer events like timer interrupts indefinitely without impacting functionality. TEEcorrelate forces the host to fill a large aggregation window before exposing any performance counter data. Thus, an attacker does not gain any advantage from single-stepping the TEE. However, other attack vectors on TEEs may rely on single-stepping as a primitive. Therefore, we believe that TEEcorrelate and single-stepping mitigations solve different problems, and should be used in conjunction.

Disabling Performance Counters. Disabling performance counter increments while executing a TEE or providing two sets of performance counter registers, one for the host and one for the TEE, are undeniably easy solutions against performance-counter attacks. However, we argue that simply turning off every feature that opens up an attack surface is not how we should handle this and similar issues (e.g., SMT). Instead, we show that mitigating performance-counter attacks against TEEs is practical as it requires only minor hardware changes, and causes no performance degradation. There are no disadvantages in comparison to disabling performance counters, while having the big advantage that they can still be used for anomaly detection [40, 58, 68] or load balancing [47, 69, 81].

Cross-Core Performance Counters. In addition to per-core performance counters, some architectures also provide cross-core performance counter registers, e.g., to count events on the L3 cache [7, 30]. Neither SEV nor TDX disable cross-core performance counters during CVM execution. TEEcorrelate also does not defend against cross-core performance counter leakage, as we cannot defend against multiple cores (host and TEE) accessing shared counters simultaneously. Defending against potential leakage through these counters is subject to future work.

8 Conclusion

TEEcorrelate is a new information-preserving principled defense against performance-counter attacks on confidential VMs. TEEcorrelate decorrelates host-visible performance counter data from secrets using temporal and value-base decorrelation. Our evaluation shows that even with a powerful, fully-informed attacker, aware of the working principle of TEEcorrelate as well as the entire machine state except for the secret, state-of-the-art attacks are slowed down by 5 to 7 orders of magnitude. We practically evaluate the slowdowns in 4 attack case studies and show that attack runtimes increase from the range of 0.58 to 429 seconds to 1 to 824.6 days. We estimate the performance impact of TEEcorrelate on AMD SEV-SNP to be 0.09 % when using an aggregation window size of 1 000 000 retired instructions. We conclude that TEEcorrelate is a lightweight defense that should be deployed in all TEE contexts.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable feedback. We furthermore thank Marcell Haritopoulos and Maria Eichlseder. This research is supported in part by the European Research Council (ERC project FSSec 101076409), the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85), Deutsche Forschungsgemeinschaft (project ReTEE) and the National Research Center for Applied Cybersecurity ATHENE as part of the PORTUNUS project in the research area Crypto. Additional funding was provided by generous gifts from Red Hat, Google, and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

Ethics Considerations

TEEcorrelate mitigates an attack presented in prior work. As our work does not describe a new vulnerability, no responsible disclosure process is required. All traces were collected on lab machines with no other users connected.

Open Science

We published all code used to simulate the mitigation on Zenodo: https://doi.org/10.5281/zenodo.15592842

References

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *AsiaCCS*, 2007.

[2] Adil Ahmad, Alex Schultz, Byoungyoung Lee, and Pedro Fonseca. An Extensible Orchestration and Protection Framework for Confidential Cloud Computing. In *OSDI*, 2023.

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *S&P*, 2019.

[4] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, 2020. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[5] AMD. Processor Programming Reference (PPR) for AMD Family 19h Model 21h, Revision B0 Processors, 2021.

[6] AMD. AMD Secure Encrypted Virtualization (SEV), 2024. URL: https://developer.amd.com/sev/.

[7] AMD. AMD64 Architecture Programmer's Manual, 2024.

[8] ARM. Security technology building a secure system using trustzone technology, 2009. URL: https://developer.arm.com/documentation/PRD29-GENC-009492/c/TrustZone-Hardware-Architecture.

[9] ARM. Arm Confidential Compute Architecture, 2024. URL: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[10] ARM. TrustZone for Arm Cortex-M Processors, 2024. URL: https://www.arm.com/technologies/trustzone-for-cortex-a.

[11] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation Rowhammer attacks. *ACM SIGPLAN Notices*, 2016.

[12] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security*, 2022.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[14] Stefano Carnà, Serena Ferracci, Francesco Quaglia, and Alessandro Pellegrini. Fight Hardware with Hardware: Systemwide Detection and Mitigation of Side-channel Attacks Using Performance Counters. *Digital Threats: Research and Practice (DTRAP)*, 4(1):1–24, 2023.

[15] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P*, 2019.

[16] Jonghyeon Cho, Taehun Kim, Soojin Kim, Miok Im, Taehyun Kim, and Youngjoo Shin. Real-time detection for cache side channel attack using performance counter monitor. *Applied Sciences*, 10(3):984, 2020.

[17] Sam Croley. Hashcat v6.1.1 benchmark on the Nvidia RTX 3090, 2020. URL: https://gist.github.com/Chick3nman/e4fcee00cb6d82874dace72106d73fef/.

[18] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. In *CHES*, 2018.

[19] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. SoK: The challenges, pitfalls, and perils of using hardware performance counters for security. In *S&P*, 2019.

[20] Haiyue Dong and Qian Guo. OT-PCA: New Key-Recovery Plaintext-Checking Oracle Based Side-Channel Attacks on HQC with Offline Templates. *Cryptology ePrint Archive*, 2024.

[21] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. *arXiv:1712.05090*, 2017.

[22] Agner Fog. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs, 2022. URL: https://www.agner.org/optimize/instruction_tables.pdf.

[23] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In *S&P*, 2023.

[24] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In *NDSS*, 2025.

[25] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *EuroSec*, 2017.

[26] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

[27] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. *ACM SIGPLAN Notices*, 52(7):129–142, 2017.

[28] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. In *CHES*, 2020.

[29] Intel. Intel Trust Domain Extensions, 2021. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf.

[30] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2024.

[31] Intel. Intel Performance Monitoring Events, 2024. URL: https://perfmon-events.intel.com/.

[32] Intel. Intel Software Guard Extensions (Intel SGX), 2024. URL: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html.

[33] Intel. Intel Trust Domain Extensions Module Base Architecture Specification, 2024. URL: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html.

[34] Alexandre Kandalintsev, Renato Lo Cigno, Dzmitry Kliazovich, and Pascal Bouvry. Profiling Cloud Applications with Hardware Performance Counters. In *International Conference on Information Networking (ICOIN)*, 2014.

[35] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption, 2016.

[36] Paul Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

[37] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, 1999.

[38] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*, 2017.

[39] Congmiao Li and Jean-Luc Gaudiot. Online detection of spectre attacks using microarchitectural traces from performance counters. In *Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018.

[40] Congmiao Li and Jean-Luc Gaudiot. Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters. In *COMPSAC*, 2019.

[41] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A systematic look at ciphertext side channels on AMD SEV-SNP. In *S&P*, 2022.

[42] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security*, 2021.

[43] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *ACSAC*, 2021.

[44] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*, 2021.

[45] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[46] Xiaoxuan Lou, Kangjie Chen, Guowen Xu, Han Qiu, Guo Shangwei, and Tianwei Zhang. Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels. In *DSN*, 2024.

[47] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention Aware Execution: online contention detection and response. In *CGO*, 2010.

[48] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. Power Analysis Attacks of Modular Exponentiation in Smartcards. In *CHES*, 1999.

[49] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[50] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting AMD's virtual machine encryption. In *EuroSec*, 2018.

[51] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*, 2020.

[52] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CROSSTALK: Speculative Data Leaks Across Cores Are Real. In *S&P*, 2021.

[53] Robin Leander Schröder, Stefan Gast, and Qian Guo. Divide and Surrender: Exploiting Variable Division Instruction Timing in HQC Key Recovery Attacks. In *USENIX Security*, 2024.

[54] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[55] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[56] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In *DIMVA*, 2019.

[57] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. *Cybersecurity*, 3(1):2, 2020.

[58] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Daniel Gruss, and Michael Schwarz. Dynamic Process Isolation. In *ESORICS*, 2021.

[59] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. MicroScope: Enabling Microarchitectural Replay Attacks. In *ISCA*, 2019.

[60] Gildo Torres and Chen Liu. Adaptive Virtual Machine Management in the Cloud: A Performance-Counter-Driven Approach. *International Journal of Systems and Service-Oriented Engineering (IJSSOE)*, 4(2):28–43, 2014.

[61] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[62] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[63] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Workshop on System Software for Trusted Execution*, 2017.

[64] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS*, 2018.

[65] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*, 2017.

[66] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, 2019.

[67] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *S&P*, 2021.

[68] Kenton Varda. Dynamic Process Isolation: Research by Cloudflare and TU Graz, 2021. URL: https://blog.cloudflare.com/spectre-research-with-tu-graz/.

[69] Sa Wang, Wenbo Zhang, Tao Wang, Chunyang Ye, and Tao Huang. VMon: Monitoring and Quantifying Virtual Machine Interference via Hardware Performance Counter. In *ACSAC*, 2015.

[70] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, 2017.

[71] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV. In *DIMVA*, 2023.

[72] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *AsiaCCS*, 2018.

[73] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In *AsiaCCS*, 2019.

[74] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In *CCS*, 2024.

[75] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity–Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *S&P*, 2020.

[76] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A Single-Stepping Framework for AMD-SEV. In *CHES*, 2024.

[77] Lai Leng Woo, Mark Zwolinski, and Basel Halak. Early Detection of System-Level Anomalous Behaviour using Hardware Performance Counters. In *DATE*, 2018.

[78] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, 2015.

[79] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[80] Ruiyi Zhang, CISPA Helmholtz Center, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security*, 2024.

[81] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *EuroSys*, 2013.

[82] Hongwei Zhou, Xin Wu, Wenchang Shi, Jinhui Yuan, and Bin Liang. HDROP: Detecting ROP Attacks Using Performance Monitoring Counters. In *ISPEC*, 2014.