ARTIFACT EVALUATED
USENIX
AVAILABLE

UncoreBleed: AEX-Free, High-Resolution, and Low-Noise Side-Channel Attacks on SGX Enclaved Execution

Decheng Chen*
South China University of Technology

Zhi Zhang*
The University of Western Australia

Zhenkai Zhang†
Clemson University

Xin Zhang
Shandong University

Yansong Gao
Southeast University

Yi Zou†
South China University of Technology

Abstract

Trusted execution environments such as Intel SGX provide strong confidentiality and integrity guarantees by isolating enclaves from the OS and hypervisor. Prior works claim that SGX disables PMCs to mitigate side-channel attacks.

In this paper, we show that modern processors feature uncore PMCs whose behavior under SGX has not been fully evaluated. Leveraging this observation, we investigate the state of PMCs in production-mode SGX enclaves and overturn the long-held belief that performance monitoring is suppressed: uncore PMCs record events correlated with enclaved execution. We further identify a critical event in the mesh-to-memory uncore subsystem that allows address-based monitoring at 64 B granularity. Through reverse engineering, we uncover its filtering mechanism, programmability, availability, and address mapping across SGX-capable Xeon processors.

Building on the event, we present UncoreBleed, the first PMC-based, AEX-free, high-resolution, and low-noise side-channel attack against SGX. UncoreBleed can reconstruct pictures from enclaved Libjpeg and extract RSA private keys from a single decryption, in the presence of TLBlur with AEX-Notify, the most state-of-the-art software defense on off-the-shelf SGX platforms. Our findings demonstrate that active uncore PMCs pose a previously underestimated threat to enclave confidentiality, highlighting the need to reconsider SGX's security assumptions of performance monitoring.

1 Introduction

Trusted execution environments (TEEs) provide confidentiality and integrity guarantees, with Intel Software Guard Extensions (SGX) [7, 32] emerging as one of the widely adopted solutions. Introduced in 2015 with the Skylake microarchitecture, SGX allows applications to create hardware-isolated enclaves within their virtual address spaces, protecting them from unauthorized access including by the privileged OS and hypervisor. This makes SGX a key technology for confidential computing, offered by major commodity cloud providers [32, 44].

Given such strong security guarantees, SGX requires carefully designed safeguards to mitigate all plausible threats. Prior works [3, 10, 12, 33, 46] claim that Intel disables performance monitoring counters (PMCs) during enclave execution due to security concerns, as PMC events can be exploited as side channels to extract sensitive information. The importance of such countermeasures is underscored by recent PMC-based attacks against AMD SEV-SNP [10].

However, this claim warrants closer examination. First, Intel's documentation does not assert that all PMCs are disabled for production enclaves [17]. Second, while prior studies confirm that multiple core PMCs are suppressed during enclave execution [7, 13, 33], modern processors feature an expanded set of performance monitoring capabilities in uncore subsystems that have not been evaluated under the SGX threat model. This motivates our first question:

Q1: Are all PMCs truly disabled when executing an SGX enclave?

To answer it, we develop microbenchmarks targeting different performance monitoring subsystems and execute them within production-mode SGX enclaves. Consistent with prior works, sensitive core PMCs are indeed disabled. However, we find that *uncore PMCs continue to record events correlated with enclave execution*. This finding challenges the long-held belief that SGX completely suppresses performance monitoring during enclaved execution.

Clearly, the presence of active uncore PMCs does not, by itself, indicate a side channel. If such counters only provide coarse-grained or noisy measurements, they may pose minimal risk to enclave confidentiality. This observation motivates a second question:

Q2: Can active uncore PMCs be exploited to extract (fine-grained) secrets from SGX enclaves?

*Equal contributions.
†Corresponding authors.

We have identified the PKT_MATCH event from the Mesh to Memory (M2M) subsystem for fine-grained enclave monitoring because, unlike other uncore PMCs that aggregate system-wide events across all cores, it supports programmable address-based filtering, potentially enabling high-resolution observation of selected memory accesses. To understand and exploit this event, we performed comprehensive reverse engineering as Intel documentation provides limited, unparsed, or erroneous details.

First, we characterize its filtering mechanism, revealing that the event provides 64 B granularity. Next, we analyzed its availability and programmability on modern SGX-capable Xeon processors, identifying discrepancies between Intel manuals and actual hardware behavior. Last, we uncover the physical address-to-M2M mapping, enabling parallel monitoring of multiple sensitive paths.

Leveraging PKT_MATCH, we introduce UncoreBleed, the first PMC-based side channel against Intel SGX. It is *AEX-free*, *high-resolution*, and *low-noise*. Its non-invasive design relies solely on uncore PMCs to monitor DRAM accesses during enclave execution, without manipulating page faults or interrupts to trigger Asynchronous Enclave Exits (AEXs). High resolution is achieved by combining fine-grained temporal sampling of PKT_MATCH with parallel monitoring of multiple sensitive targets across M2M Performance Monitoring (PMON) boxes. It is inherently low-noise, as PKT_MATCH selectively counts memory accesses to monitored 64 B memory blocks, filtering unrelated memory activities.

As a case study, UncoreBleed has been demonstrated to successfully recover pictures from enclaved Libjpeg. Unlike prior controlled channels [26,51] that operate at the 4 KB page level and thus focus on sensitive stack pages, UncoreBleed operates at 64 B granularity and thus directly distinguishes sensitive paths within the same code page.

Last, Intel introduced AEX-Notify [6] on recent Xeon processors, a hardware extension enabling enclaves to become interrupt-aware, mitigating single-stepping attacks [41]. Building on this, TLBlur [44] provides state-of-the-art software-based protection. It prefetches sensitive pages into the TLB during enclave resumption to conceal control-flow patterns, mitigating prior side-channel attacks that leverage architectural interfaces [13, 25, 28, 36, 43, 51]. Since uncore PMCs also constitute an architectural interface, this raises the final question:

Q3: Can UncoreBleed bypass the state-of-the-art software defense on off-the-shelf SGX platforms?

Previous architectural-interface attacks [13, 25, 28, 36, 43, 51] exploit page faults and/or interrupts to trigger AEXs, deterministically enabling observations of chosen execution points at the page or instruction granularity. To mitigate the attacks, TLBlur [44] with AEX-Notify, pins sensitive pages into the TLB when resuming from an AEX, concealing control-flow patterns from page-table observations.

UncoreBleed differs fundamentally in that it performs passive monitoring through uncore PMCs without triggering AEXs. Besides, UncoreBleed monitors specific DRAM accesses directly, and remains effective even when all sensitive pages are pinned in the TLB. As the other case, we show that UncoreBleed successfully recovers RSA private keys from a single decryption, with TLBlur deployed, demonstrating that enclave secrets remain vulnerable despite the most recent defense.

In summary, the main contributions of this paper are:

- We demonstrate that, in contrast to the prevailing belief, SGX enclaves are not entirely excluded from performance monitoring: while core PMCs are disabled, uncore PMCs remain active and expose enclave-correlated events.
- We identify and reverse engineer the PKT_MATCH event in the M2M subsystem. We show that it supports programmable address filtering and achieves 64-byte granularity, enabling fine-grained monitoring of enclave memory accesses.
- Our reverse-engineering analysis establishes that PKT_MATCH is consistently available across multiple generations of SGX-capable Xeon processors, despite being undocumented or inaccurately described in Intel official manuals.
- We present UncoreBleed, the first PMC-based side channel against SGX, and demonstrate its effectiveness in two cases: recovering pictures during Libjpeg decompression, and extracting RSA private keys from a single decryption in production enclaves in the presence of TLBlur, the most recent software defense on off-the-shelf SGX platforms.

2 Background and Related Work

In this section, we first introduce SGX, then outline Intel's core/uncore domains, followed by hardware performance monitoring. Last, we summarize related work on software-only side-channel attacks against SGX.

2.1 Intel SGX

Intel Software Guard Extensions (SGX) allows a process to create hardware-isolated memory regions called *enclaves* within its own virtual address space. The virtual pages forming an enclave are backed by the *Enclave Page Cache* (EPC).

From an architectural perspective, the CPU package integrates the SGX Memory Encryption Engine (MEE) between the Last-Level Cache (LLC) and the memory controller. The MEE transparently encrypts and authenticates cache lines belonging to EPC pages before they leave the CPU die. Enclave execution begins with the EENTER instruction and suspends on interrupts via *Asynchronous Enclave Exit* (AEX), which saves execution context into a *State Save Area* (SSA) before transferring control to untrusted code. The ERESUME instruction later restores this context, allowing execution to continue inside the enclave.

Intel SGX was first introduced in client processors, and later deployed in certain Xeon server processors. Our work focuses on Xeon server platforms, providing both SGX capabilities and the server-grade (Un)core domains introduced below.

2.2 Core and Uncore Domains of Intel CPUs

Starting with the Nehalem microarchitecture, Intel split its modern processors into two largely independent power and clock domains: **Core** and **Uncore** [20].

Core domain contains the execution resources for user and kernel instructions, including private L1 and L2 caches, out-of-order execution engine, branch predictors, load/store units, and per-core *Performance Monitoring Units* (PMUs). It is optimized for high clock frequencies and low latency, maximizing single-thread performance.

Uncore domain comprises the shared resources and interconnect fabric that coordinate communication between cores, last-level caches, and I/O/memory subsystems. Key uncore components include:

- Last-Level Cache (LLC) is divided into slices, each coupled with a Caching and Home Agent (CHA).
- CHA performs coherence directory lookups, arbitration, and queuing of outstanding memory requests in the Table of Requests (TOR).
- Mesh Network is a bidirectional on-die interconnect used to transport cache lines and coherence messages between cores, CHAs, and memory controllers.
- Integrated Memory Controllers (iMCs) interface with DDR4 or DDR5 channels via the Scalable Memory Interconnect (SMI3).
- M2M Bridges are mesh-to-memory agents that translate mesh flits into SMI3 packets.

For example, Figure 1 illustrates this division, showing both the intra-core hierarchy and the uncore interconnect paths.

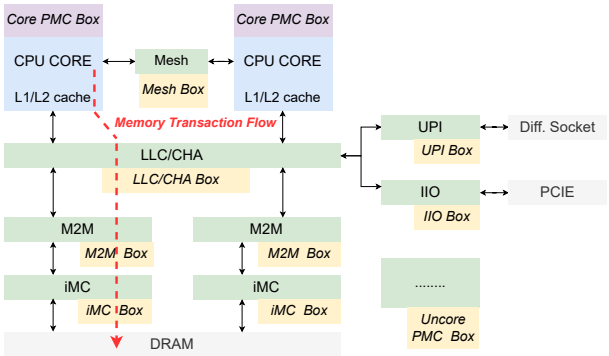


Figure 1: The hierarchical architecture of the Core and Uncore domains.

Memory Transaction Flow in Xeon Servers. As shown in Figure 1, a memory request (illustrated as an arrowed line) in Intel Xeon server processors traverses a sequence of hardware stages before it reaches DRAM, as described below:

1. CPU Core issues load/store instruction to private L1/L2 caches.
2. Upon L1/L2 miss, the request probes the LLC cache slice via the CHA.
3. On L3 miss, the CHA checks the coherence directory, queues transactions in the TOR, and encapsulates the request into mesh flits.
4. Mesh Network routes the request through the on-die interconnect.
5. M2M Bridge converts mesh protocol to memory controller protocol.
6. iMC translates SMI3 packets into DDR commands.
7. DRAM completes a desired read/write operation.

In multi-socket systems, if the address resides in a remote NUMA node, the UPI (Ultra Path Interconnect) forwards the request to the corresponding socket's CHA.

2.3 Hardware Performance Monitoring

Hardware performance monitoring counters (PMCs) are special-purpose registers that track microarchitectural events such as cache misses, branch mispredictions, and instruction retirements. Modern Intel CPUs expose two main categories:

- Core PMCs, located in each core's PMU, track events local to that core, e.g., `IA32_FIXED_CTRn` for fixed-function counters and `IA32_PERFEVTSELn`/`IA32_PMCn` for programmable counters).
- Uncore PMCs, located in shared hardware components (e.g., LLC slices/CHAs, iMCs, M2M bridges, interconnects), capture events not tied to any single core.

Core PMCs only observe activity before a request leaves the L2 cache, so they cannot capture package-level contention or traffic. To address this, Intel equips the uncore domain with dedicated Performance Monitoring (PMON) boxes, marked in yellow in Figure 1. Each PMON box corresponds to a shared hardware component and typically provides:

- Four 48-bit counters (`*_PMON_CTR{0-3}`).
- Matching control registers (`*_PMON_CTL{0-3}`).
- Global MSRs for synchronized start/stop, freeze, and clear operations.

In the 5th-generation Xeon Emerald Rapids, more than 2000 uncore events are supported [19], enabling fine-grained monitoring across different hardware units. For instance, Mesh boxes track on-die network congestion and traffic, while LLC/CHA boxes capture LLC cache hit/miss and snoop responses; iMC boxes expose DDR command patterns and read/write distributions.

Table 1: System configurations. All listed motherboards are different models from Supermicro.

CPU	Microprocessor	DRAM	Motherboard	SGX	AEX-Notify
Gold 6130	Skylake-SP	DDR4 (4 × 8 GB)	X11SPL-F	✗	✗
Gold 6230	Cascade Lake-SP	DDR4 (4 × 8 GB)	X11SPL-F	✗	✗
Silver 4314	Ice Lake-SP	DDR4 (8 × 8 GB)	X12SPL-F	SGX1&2	✓
Bronze 3408U	Sapphire Rapids-SP	DDR5 (8 × 16 GB)	X13SEI-F	SGX1&2	✓
Silver 4514Y	Emerald Rapids-SP	DDR5 (8 × 16 GB)	X13SEI-F	SGX1&2	✓

2.4 Software-only Side-Channel Attacks Against Intel SGX

A large number of software-only side-channel attacks against SGX have been demonstrated, which broadly fall into two main categories.

Microarchitectural Contention. These attacks exploit shared resources between enclaves and attackers [5, 24, 26, 27, 42]. They rely on timing measurements to infer resource contention. However, timing channels are inherently noisy and nondeterministic, making them vulnerable to system noise [44]. Moreover, they can be effectively mitigated by system-level defenses such as partitioning or flushing shared microarchitectural resources [8, 9, 11].

Architectural Interfaces. Other attacks leverage CPU interfaces that are accessible to privileged software. The exploited interfaces include page tables, interrupts and the Running Average Power Limit (RAPL) interface [13, 25, 28, 36, 41, 43, 51]. These attacks have been shown to reconstruct highly sensitive data, including plaintext documents, JPEG images, and cryptographic keys. They all rely on manipulating page faults and/or interrupts (thus causing AEXs) to compromise enclaves. We note that while Intel RAPL allows monitoring CPU/DRAM power consumption, it is susceptible to noise and requires SGX-Step [41] to single-step a running enclave.

Distinguishing UncoreBleed. Unlike existing architectural-interface attacks, UncoreBleed is the first to exploit an uncore PMC interface to compromise SGX confidentiality, without page faults or interrupts. The most closely related work is Gast et al. [10], who used core PMC events to attack AMD SEV-SNP. Still, their exploited interface remains noisy and requires interrupts to single-step SEV-SNP VMs.

3 Revisiting PMC Behaviors in SGX Enclaves

The security community has long accepted that Intel SGX disables PMCs during enclave execution, as prior works have confirmed the suppression of many core PMC events [3, 10, 12, 33, 46]. However, we re-examine this for three reasons.

First, Intel's documentation does not claim that all PMCs are disabled in production enclaves. It explicitly suppresses core PMCs, noting that fixed-function counters (e.g., `IA32_FIXED_CTR0`) and general-purpose counters (`IA32_PMCx`) stop accumulating counts upon enclave entry [17]. Yet, it makes no mention of uncore PMCs.

Second, modern Intel processors have significantly expanded their performance monitoring capabilities beyond core PMCs, introducing uncore monitoring subsystems. Many uncore PMCs operate via distinct hardware interfaces (e.g., PCIe configuration space rather than MSRs) and serve different architectural roles. To date, no study has examined whether SGX enclaves suppress these uncore PMCs.

Last, while core PMCs reside within individual CPU cores and can be suppressed during enclave execution, uncore PMCs monitor shared system resources that span multiple cores. Blocking all system-wide performance monitoring whenever a single core enters an enclave might hinder legitimate observability of runtime system performance, such as for performance analysis, debugging, or optimization.

To this end, we investigate whether any PMCs remain active during enclave execution by designing controlled experiments consisting of the following four steps. All experiments are performed on Xeon Silver 4314 (Ice Lake-SP) in Table 1.

1. Select representative events from core and uncore PMCs.
2. Develop microbenchmarks to stress selected events.
3. Run each microbenchmark both inside a production-mode SGX enclave and in a non-enclave environment.
4. Compare event counter growths across both environments.

Each microbenchmark is implemented as a finite loop that repeatedly triggers a specific event. We place high-precision timestamps (e.g., `rdtsc()`) at the start and end of the loop to align the execution interval with a privileged `perf` sampling process, which records events at 1 ms intervals. To reduce scheduling interference, we pin the microbenchmark and `perf` to separate physical cores, with `perf` configured to sample only the core executing the microbenchmark.

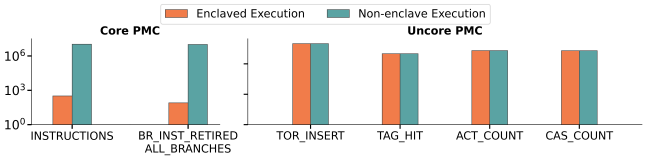


Figure 2: Core PMC and uncore PMC event counts when running microbenchmarks both in a production-mode enclave and in a non-enclave environment.

3.1 Core PMC Behaviors

Prior works have shown that core PMCs are disabled during SGX enclaved execution [3, 10, 12, 33]. We confirm it using two representative events: `INSTRUCTIONS` and `BR_INST_RETIRED.ALL_BRANCHES`, both previously exploited in attacks on AMD SEV-SNP [10]. These events serve to establish a baseline of disabled PMC behaviors for comparison with uncore PMCs.

`INSTRUCTIONS` counts retired instructions per logical core via the fixed-function counter `IA32_FIXED_CTR0`. We stress

it with a microbenchmark that executes 10^7 `nop` instructions in a loop.

`BR_INST_RETIRED.ALL_BRANCHES` counts retired branch instructions per logical core, configured by setting the `EventSel` field of `IA32_PERFEVTSELx` to 0xC4. We stress this event also in a loop of 10^7 iterations. The loop control itself will generate a high rate of retired branch instructions.

As Figure 2 shows, the counts for the `INSTRUCTIONS` and `BR_INST_RETIRED.ALL_BRANCHES` events in non-enclaved execution reach the expected order of 10^7 whereas in enclaved execution the counts are only a few hundreds, confirming that these core events are effectively suppressed in the enclave.

3.2 Uncore PMC Behaviors

As introduced in Section 2.3, uncore PMCs monitor system-wide activities across multiple functional units, called "boxes". To evaluate whether SGX suppresses these counters, we examine three critical uncore boxes along the memory hierarchy: CHA, M2M, and iMC. Guided by Intel's performance monitoring manual [16], we select four representative events covering a memory transaction flow from cache coherency to DRAM transactions.

CHA: `TOR_INSERT`. This event counts requests that are successfully inserted into the Table of Requests (ToR). ToR is a queue tracking address-carrying transactions in the uncore interconnect. To test this event, we sequentially access a large memory of 8 GB with a 64 B stride for 10^8 iterations, generating frequent requests that produce numerous ToR insertions.

M2M: `TAG_HIT`. This event increments when the iMC locates the requested address and returns a matching response tag. To test this event, we use `flush+reload` [53] to repeatedly access a memory block for 10^7 iterations, which generates frequent requests visible in the event counter.

iMC: `ACT_COUNT` and `CAS_COUNT`. The `ACT_COUNT` event tracks DRAM row activations, while `CAS_COUNT` counts column access strobes for actual DRAM read/write operations. This time, we use `flush+reload` to alternately access two page-aligned addresses for 10^7 iterations. Because each iteration alternately touches two rows, it triggers two row activations and two column accesses per iteration, leading to frequent memory operations. To stress these two events, the two addresses must be mapped to different DRAM rows within the same DRAM bank, inducing row buffer conflicts. Thus, alternating accesses to the two addresses will generate frequent row activations recorded by `ACT_COUNT` and column accesses recorded by `CAS_COUNT`. To find such address pairs, we allocate approximately 50% of the physical memory (32 GB out of 64 GB), randomly select two page-aligned addresses, and perform rapid alternating accesses. If the accesses result in a significant increase in `ACT_COUNT`, we record their corresponding physical addresses.

Results. Figure 2 shows that the counts of all the four uncore events are almost identical for enclaved and non-enclaved execution, confirming that SGX's suppression of core PMCs does not extend to uncore PMCs. This indicates that uncore performance monitoring remains operational inside the enclave, potentially serving as a side channel.

4 Reverse Engineering M2M Filtering

While the aforementioned uncore PMCs remain active during SGX enclaved execution, their potential for exploitation is limited as they are low-resolution in that they accumulate all occurrences of target events unconditionally across the system and aggregate behaviors from all cores. As a result, their measurements are inherently noisy and cannot be directly exploited to recover fine-grained enclave secrets.

To address this limitation, we investigate filterable uncore events introduced since Skylake-SP. These events, supported by certain uncore PMC boxes, are paired with dedicated configuration registers that enable hardware to selectively count only samples matching user-defined criteria. Such filtering provides the potential for high-resolution side-channel observation.

Among these events, `PKT_MATCH` from the M2M subsystem has particularly potential. By design, it monitors packet-level memory traffic on the mesh interconnect and provides programmable filters to match physical addresses [15, 16, 18, 19]. In principle, such functionality makes `PKT_MATCH` a promising candidate for monitoring fine-grained enclave memory access patterns. However, Intel's manual provides only limited information about its semantics and programming interface, leaving critical aspects of its mechanism unknown.

To bridge this gap, we conduct a systematic study of `PKT_MATCH` across multiple Intel server platforms in Table 1. We begin by characterizing its filtering mechanism on Skylake-SP in Section 4.1. Next, we examine its availability on recent SGX-capable Xeon processors in Section 4.2. Last, we reverse engineer the address-to-M2M mapping in Section 4.3, a prerequisite for targeting specific enclave addresses. Together, this study lays the foundation for exploiting `PKT_MATCH` as a practical side channel against SGX enclaves in Section 5.

4.1 Characterizing `PKT_MATCH` in M2M

According to Intel's Skylake-SP manual [15], `PKT_MATCH` is programmed through five 32-bit registers, i.e., `ADDRMATCH0`, `ADDRMATCH1`, `ADDRMASK0`, `ADDRMASK1`, and `OPCODE_MM`. The first four registers form two 64-bit pairs: {`ADDRMATCH0`, `ADDRMATCH1`} is one pair, where all 32 bits of `ADDRMATCH0` specify physical address bits `[31:0]` and the lower 15 bits of `ADDRMATCH1` specify physical address bits `[46:32]`; The other pair of {`ADDRMASK0`, `ADDRMASK1`} follows the same structure to define a corresponding address mask. In this study, we focus on these two pairs, omitting `OPCODE_MM` as it controls filtering by memory transaction types rather than addresses.

However, the manual provides only limited details about how these registers operate together, leaving the main characteristics of PKT_MATCH undocumented. To understand it, we conducted experiments on the Xeon Gold 6130 platform with Skylake-SP. While SGX is not supported on Skylake-SP[1], this platform remains suitable because its PKT_MATCH documentation, while sparse, is accurate compared to the erroneous and incomplete PKT_MATCH documentation in newer Xeon manuals (detailed in Section 4.2).

We began by evaluating the default configuration, where all the four registers were set to zero. In this setting, the PKT_MATCH event counter incremented on memory accesses at runtime, indicating that no filtering was in place. We next configured the two ADDRMATCH registers with a chosen physical address while keeping the ADDRMASK registers at zero, and probed the corresponding virtual address using flush+reload [53]. In this case, the counter incremented not only for accesses to the specified address but also for adjacent addresses. By varying the tested offsets, we established that the matching granularity is one 64-byte cache line: all accesses within the same cache line triggered the event, whereas accesses beyond did not (Figure 3).

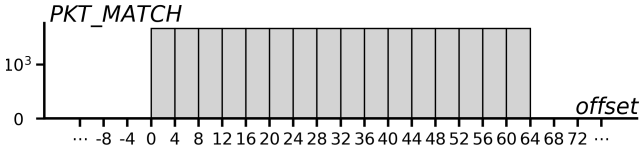


Figure 3: PKT_MATCH increases for accesses within the same 64B cache line of the configured ADDRMATCH.

We then analyzed the effect of the ADDRMASK registers. With all mask bits set to one, we found that accesses to any address triggered the PKT_MATCH event, regardless of the value in ADDRMATCH. To characterize this behavior, we performed a bit-wise analysis.

Specifically, we configured ADDRMATCH with a physical address $\mathcal{A}$ and derived another valid address $\mathcal{B}$ by flipping a few bits in $\mathcal{A}$. Since the address matching granularity is one 64-byte cache line, we randomly selected three bits from bit 6 to bit 34 of $\mathcal{A}$, flipped them to generate $\mathcal{B}$, and repeated this procedure ten times to obtain different test addresses.

Initially, all bits in ADDRMASK were set to 1, causing PKT_MATCH to count all memory accesses in the system. In an idle system, the counter remained low and stable, providing a baseline. Repeated accesses to $\mathcal{A}$ or $\mathcal{B}$ using flush+reload [53] triggered a spike in the counter by about two orders of magnitude.

To reveal the semantics of ADDRMASK, we iteratively cleared its bits from least to most significant. After clearing each bit, we repeatedly accessed $\mathcal{B}$ and monitored PKT_MATCH: if the counter spikes, the cleared bit is retained; otherwise, it is restored. The bits that remained set to 1 corresponded precisely to the positions where $\mathcal{A}$ and $\mathcal{B}$ differed, demonstrating the "don't care" behavior of ADDRMASK: a bit set to 1 is ignored for address matching, whereas a bit set to 0 enforces exact matching. Thus, given an address $\mathcal{X}$, accesses to $\mathcal{X}$ trigger PKT_MATCH if

$$\begin{aligned}
\texttt{ADDRMASK} &= (\texttt{ADDRMASK1} \ll 32) \,|\, \texttt{ADDRMASK0}, \\
\texttt{ADDRMATCH} &= (\texttt{ADDRMATCH1} \ll 32) \,|\, \texttt{ADDRMATCH0}, \\
\mathcal{X} \,\&\, {\sim}\texttt{ADDRMASK} &= \texttt{ADDRMATCH} \,\&\, {\sim}\texttt{ADDRMASK},
\end{aligned}$$

Where the six lowest-order bits of ADDRMATCH and ADDRMASK are ignored for address matching, since the matching granularity is 64 B.

4.2 Recovering PKT_MATCH on SGX-Capable Xeon Processors

Having established the filtering characteristics of PKT_MATCH on Skylake-SP, we next investigate whether this functionality persists on modern Xeon platforms that support SGX. Since Intel SGX was first introduced to Xeon Scalable processors with the Ice Lake-SP generation, understanding the availability and programmability of PKT_MATCH on these platforms is critical to assessing its exploitability under the SGX threat model.

However, when we attempted to follow Intel manual for Ice Lake-SP [16], we found that the documented filter registers could not be programmed as in Section 4.1. Even more concerning, in the manuals for Sapphire Rapids-SP [18] and Emerald Rapids-SP [19], the PKT_MATCH event was removed from the M2M event list. This inconsistency raises a question:

> Is PKT_MATCH still available on SGX-capable Xeon processors, and if so, how can it be correctly accessed?

To address this question, Section 4.2 locates the correct filter register addresses on Ice Lake-SP, where Intel manual appears incorrect. Section 4.2.2 identifies the undocumented event code on Sapphire Rapids-SP and Emerald Rapids-SP, where PKT_MATCH has been intentionally or mistakenly omitted from the published manuals.

4.2.1 Identifying Correct Register Addresses

As in Table 2, the PKT_MATCH event on Skylake-SP has an event ID of 0x4C, with its filter registers (ADDRMATCH0/1 and ADDRMASK0/1) located at offsets 0x268-0x274 in the PCI configuration space of device 8086:2066. Intel manuals for Cascade Lake-SP (2nd generation of Xeon Scalable processors) retain the same event ID and register addresses. However, while these documented addresses work correctly on Skylake-SP and Cascade Lake-SP, programming them on Ice Lake-SP

[1] SGX was first introduced in consumer-grade Skylake processors but was not supported in most server-grade counterparts.

fails[2]. This suggested that Intel manual contains erroneous information regarding Ice Lake-SP.

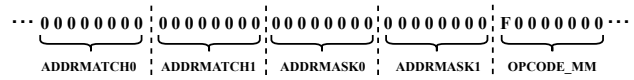


Figure 4: The relative positions and default values of ADDRMATCH0, ADDRMATCH1, ADDRMASK0, ADDRMASK1, and OPCODE_MM in the PCI configuration space on Skylake-SP and Cascade Lake-SP.

A naive brute-force search across all PCI configuration spaces could, in principle, recover the correct register locations, but the search space is prohibitively large. To reduce the complexity, we sought distinguishing characteristics. By comparing Skylake-SP and Cascade Lake-SP, we discovered that while the four ADDRMATCH and ADDRMASK registers reset to zero, the OPCODE_MM register consistently initialized to 0x000000F0[3]. As illustrated in Figure 4, this distinctive pattern, combined with the relative positioning of the registers, forms a fingerprint to locate the filter registers.

Using this heuristic, we scanned Ice Lake-SP's PCI configuration space and identified device 8086:344A, where registers at offsets 0x490–0x4A0 matched the expected pattern. Programming these registers yielded the expected PKT_MATCH behavior, thereby confirming that the Intel manual was incorrect.

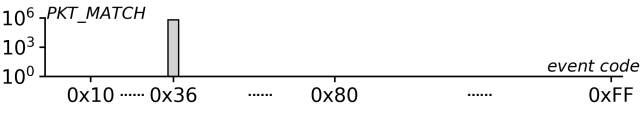


Figure 5: Results of brute-force search of the M2M event code space (0x00–0xFF) on Sapphire Rapids-SP. Among all tested codes, only event code 0x36 showed counter growth corresponding to PKT_MATCH behavior.

4.2.2 Exposing Undocumented Event Code

With the correct filter registers identified on Ice Lake-SP, we turned our attention to the recent Xeon generations of Sapphire Rapids-SP and Emerald Rapids-SP. However, Intel manuals present a contradictory picture: while the filter registers are still described in the manuals, the PKT_MATCH event itself has been entirely removed from the list of available M2M events. This raises the possibility that the event is still implemented in hardware but omitted or hidden by Intel.

To test this hypothesis, we first verified that the default values of the filter registers on Sapphire Rapids-SP matched the characteristic fingerprint identified in Figure 4, confirming that the register addresses are correct and still present.

Since the event code for PKT_MATCH is 0x4C across the first three generations (Skylake-SP through Ice Lake-SP), we initially assumed this code would remain unchanged. To test this, we configured the filter registers on Sapphire Rapids-SP to monitor a specific 64 B memory block and repeatedly accessed the block using flush+reload [53] to generate memory traffic. However, the PKT_MATCH event counter did not increase, suggesting that Intel had reassigned the event code.

To identify the correct code for PKT_MATCH, we employed a brute-force search. This approach was feasible because the M2M event code space is limited to 256 values (0x00–0xFF). We iterated through the entire event code space on Sapphire Rapids-SP, using our configured filter registers and monitoring event counter behavior. Figure 5 shows only event code 0x36 produced the expected counter growth, responding precisely to our memory accesses in the same manner as PKT_MATCH on earlier generations.

Table 2: Configuration details of PKT_MATCH for five Xeon Scalable generations. Erroneous and incomplete documented values are in *italics*, and verified values are in **bold**.

Microprocessor	Device ID	ADDRMASK	ADDRMATCH	Event Code
Skylake-SP (Xeon Scalable)	0x2066	0x270,0x274	0x268,0x26C	0x4c
Cascade Lake-SP (2nd Xeon Scalable)	0x2066	0x270,0x274	0x268,0x26C	0x4c
Ice Lake-SP (3rd Xeon Scalable)	*0x2066* **0x344a**	*0x270,0x274* **0x498,0x49C**	*0x268,0x26C* **0x490,0x494**	0x4c
Sapphire Rapids-SP (4th Xeon Scalable)	0x324a	0x498,0x49C	0x490,0x494	*Undocumented* **0x36**
Emerald Rapids-SP (5th Xeon Scalable)	0x324a	0x498,0x49C	0x490,0x494	*Undocumented* **0x36**

We repeated this experiment on Emerald Rapids-SP and confirmed that event code 0x36 also corresponds to PKT_MATCH, consistent with Sapphire Rapids-SP. This indicates that Intel changed the event code for PKT_MATCH from 0x4C to 0x36 beginning with Sapphire Rapids. Table 2 shows the PCI device IDs, register addresses, and event codes for PKT_MATCH across all five Xeon Scalable generations, including both documented and reverse-engineered values.

4.3 Uncovering Address-to-M2M Mapping

The M2M subsystem is not a single monolithic unit but consists of multiple independent boxes distributed throughout the processor. On Xeon Scalable processors, there are typically four M2M boxes, each paired with an iMC. Since each box observes a subset of system memory traffic and maintains its own PKT_MATCH event, monitoring a specific physical address requires knowing which M2M box the address is routed

[2]When values are written to these register addresses, the values read back do not match the original

[3]The default value 0xF0 causes all SMI3 opcodes to be treated as matches, i.e., opcode-based filtering in PKT_MATCH is disabled.

through. Unfortunately, Intel does not disclose the address-to-M2M mapping, making reverse engineering necessary.

Prior works on DRAM bank indexing [31, 45, 50] have shown that Intel applies linear hash functions to distribute memory traffic across DRAM banks. In this scheme, selected physical-address bits are grouped, XORed together, and combined to form the bank index. Because the number of banks is often a power of two, each bank bit corresponds to the XOR of a small set of physical-address bits. Flipping a participating bit may change the XOR output and can redirect the memory access to a different bank. This observation suggests that the M2M routing logic may be constructed in a similar manner, using XOR-based functions over specific address bits.

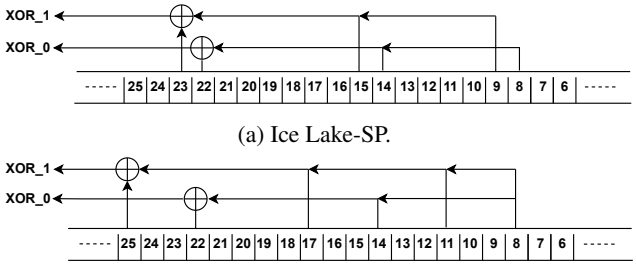
With this hypothesis, we aim to recover the physical address-to-M2M mapping. Since there are four M2M boxes, the routing likely involves two independent XOR functions, each producing one output bit to index the boxes. To uncover these functions, we proceed as follows.

Starting from a physical address X, we generate test addresses by flipping exactly one bit at a time (excluding the cache-line offset bits below bit 6). For each test address, we induce memory accesses and monitor the `PKT_MATCH` counters across all M2M boxes to determine which box observes the access. If they route to different boxes, the flipped bit participates in the M2M hash function; otherwise, it does not.

After identifying the participating bits, we determine how they are combined. We start by pairwise testing because of the algebraic properties of XOR. If flipping bits i and j individually changes the routing, but flipping them together restores the original routing, then i and j are XORed within the same hash function. By exhaustively applying this test, we uncover entire XOR chains. For example, in Figure 6b, bits 8, 17, and 25 all XORed together to form bit 1 of the M2M box index. This emerges from observing that the pairs $(8,17)$, $(8,25)$, and $(17,25)$ all cancel each other, revealing a three-bit XOR group. More generally, larger XOR groups exhibit the same cancellation behavior in pairwise tests, allowing them to be uncovered.

Some address bits may participate in different XOR chains. To confirm such cases, we flip a candidate "shared" bit together with a known participant from each chain. If both flips restore the original routing, the bit contributes to both XOR functions.

Applying this method, we uncover the address-to-M2M mapping on SGX-capable Xeon processors, shown in Figure 6. In these processors, bit 8 of the physical address is the lowest-order bit involved in the mapping. As a result, two addresses that differ only in bit 8 will be routed to different M2M boxes. Within a single 4KB page, this enables the use of multiple `PKT_MATCH` events from different M2M boxes to simultaneously monitor accesses to distinct 256 B regions.



(a) Ice Lake-SP.

(b) Sapphire Rapids-SP and Emerald Rapids-SP.

Figure 6: Uncovered address-to-M2M mapping.

5 UncoreBleed

Building on the reverse-engineered efforts in Section 4, we present UncoreBleed in this section, the first performance-counter–based side channel against SGX enclaves, that is AEX-free, high-resolution, and low-noise. In the following sections, we first describe the threat model and provide an overview of UncoreBleed, followed by its technical details.

5.1 Threat Model

Following prior work on software-based side-channel attacks against TEE platforms [3, 28, 33, 40, 43, 51], we use the same privileged-software adversary model, where the attacker has full control over the system software stack beneath the enclave, including the operating system and the hypervisor.

We assume the target platform is Intel Xeon processors, which are widely deployed in data centers and continue to support SGX (Intel discontinued SGX beginning with its 11th Generation Core processors). The system can include the latest microcode updates, and enclaves run in production mode with debugging features disabled. We also assume Intel HyperThreading is disabled, aligned with the state-of-the-art defense [44]. We do not assume any physical access to the machine or software vulnerabilities inside the enclave.

Aligned with prior SGX controlled channel attacks [13, 28, 43, 51], the attacker is assumed to know the victim enclave's compiled code, which allows reconstructing the enclave's memory layout and identifying sensitive code regions. The unknowns to the attacker are the enclave's run-time secrets (e.g., cryptographic keys), which are the target of the attack.

Fundamentally different from existing microarchitectural contention-based side channels [5, 24, 26, 27, 42] such as cache or TLB contention, which rely on detecting subtle timing differences and are inherently nondeterministic and noisy, our attack, similar to controlled-channel attacks [13, 28, 43, 51], exploits architecturally defined CPU interfaces. Also, our approach can extract information from a single enclave execution, avoiding detection by remote attestation schemes.

More importantly, unlike existing SGX controlled-channel attacks that exploit page faults (PF) or interrupts to trigger

AEXs, our attack is both PF-free and interrupt-free. It leverages architectural performance monitoring counters, specifically uncore events, to observe specific memory accesses.

5.2 Overview

As shown in Figure 7, UncoreBleed works in three main steps.

❶ **Locating Sensitive Targets.** UncoreBleed locates secret-dependent virtual addresses within a target enclave program, which we term *sensitive targets*. Prior page-fault–based attacks [43, 51] require distinct targets to be on different pages (i.e., page-granularity). Unlike them, UncoreBleed achieves 64 B resolution.

❷ **Runtime Monitoring.** During enclaved execution, UncoreBleed resolves the sensitive virtual addresses to their physical counterparts via page-table walks and maps them to the appropriate M2M boxes using the reverse-engineered address-to-M2M mapping. By configuring the `PKT_MATCH` events in the corresponding M2M boxes, UncoreBleed passively collects frequent accesses to the targets without triggering page faults or interrupts.

❸ **Secrets Recovery.** UncoreBleed correlates the collected temporal traces of sensitive accesses with program knowledge to infer program secrets.

To exploit `PKT_MATCH`, it is essential that sensitive targets be accessed directly from DRAM. To enforce this, UncoreBleed ensures that every access to a sensitive location bypasses the CPU cache (Section 5.3). UncoreBleed further designs a high-resolution monitoring primitive that not only captures accesses to a single sensitive target with fine-grained temporal granularity, but also extends to monitoring multiple sensitive targets in parallel (Section 5.4).

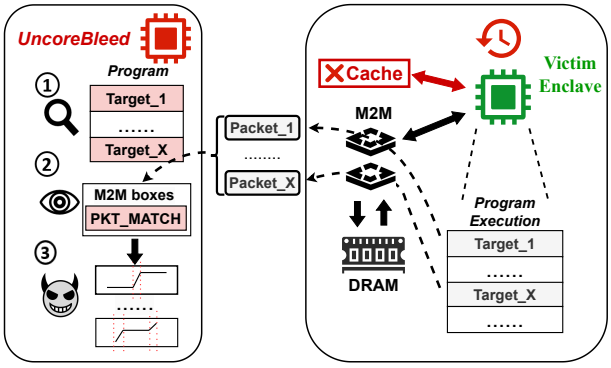


Figure 7: Overview of UncoreBleed.

5.3 Forcing DRAM Accesses

In normal execution, frequently accessed data is served by the CPU caches. In Intel SGX, EPC memory is hardware-enforced as write-back cacheable, and its caching policy cannot be disabled through page-table attributes [21]. To guarantee that every access to a sensitive target is serviced from DRAM, one naive approach would be to employ the `WBINVD` instruction. `WBINVD` flushes the entire last-level cache (LLC), ensuring that subsequent accesses are routed to DRAM. However, each flush consumes tens of millions of CPU cycles, during which enclave accesses may still hit in the cache. This makes `WBINVD` impractical for reliably capturing all sensitive memory accesses.

Instead, UncoreBleed adopts a more efficient approach: it forces the enclave's execution core to perform uncacheable (UC) accesses by setting the `CR0.CD` bit in kernel mode [6]. This configuration enforces UC-like behavior for the core's memory operations, preventing cache allocation and fills without modifying the underlying memory type. Consequently, while EPC memory remains write-back cacheable at the hardware level, all accesses from this core bypass the caches and reach DRAM. These DRAM requests are then observable via `PKT_MATCH`.

5.4 High-Resolution Monitoring Primitive

A natural first candidate for implementing a high-temporal-resolution primitive is SGX-Step [41], which provides instruction-level single-stepping for enclaves by forcing frequent AEXs. This allows detecting the completion of each instruction. However, SGX-Step inevitably introduces numerous enclave exits. To avoid incurring AEXs, UncoreBleed develops a monitoring primitive that combines high-frequency sampling, execution slowdown, and parallel monitoring to achieve a high-resolution and low-noise side-channel.

Fine-Grained Sampling. Linux `perf` utility provides sampling intervals no shorter than 1 ms, which is coarse for UncoreBleed. To achieve higher resolution, UncoreBleed leverages the `M2M` PCIe registers exposed by the uncore. Intel supports two modes. The first is *Interrupt-driven sampling* counters, which trigger performance-monitoring interrupts (PMIs) when pre-defined thresholds are reached. The second is *Polling-based sampling* counters that are directly read without interrupts.

Obviously, we adopt the second mode. Specifically, we developed a lightweight Linux kernel module that repeatedly polls the `M2M_PCI_BAR` register corresponding to the `PKT_MATCH` event. Each polling iteration executes a sequence of "read timestamp; read the event" on a thread pinned to a dedicated CPU core, while `mlock` prevents the sampling process's pages from being swapped out. With this setup, we achieve sampling intervals as short as 650 ns on the Bronze3408U and Silver4514Y platforms.

To assess whether this sampling granularity suffices for real attacks, we first consider traditional RSA. During RSA decryption, the execution of a Montgomery multiplication depends on the current key bit (see Figure 8b), allowing an attacker to distinguish between bit values by monitoring iterations of the `for` loop. To extract all bits, the sampling interval

$t_{interval}$ must be shorter than the execution time $t_{execution}$ of one loop iteration.

Without enforcing DRAM accesses in Section 5.3, our measurements show that one loop iteration takes approximately 5500 ns for a 2048-bit key and 12000 ns for a 4096-bit key. Since our sampling interval of 650 ns is an order of magnitude smaller, UncoreBleed can reliably capture every iteration and hence every key bit.

Stretching Enclaved Execution. Unfortunately, modern cryptographic libraries including WolfSSL [49] and MbedTLS [39] have deployed advanced techniques (e.g., sliding-window exponentiation) that substantially reduce the loop execution times. In our measurements, a single loop iteration in these libraries (WolfSSL v5.8.0 and MbedTLS v2.6.0) often completes in only tens of nanoseconds, well below the 650 ns sampling interval.

To address this, UncoreBleed deliberately slows down enclave execution so that $t_{execution}$ is expanded into a range accessible by our sampling resolution. First, as described in Section 5.3, setting the `CR0.CD` bit enforces uncacheable execution, inflating per-instruction cost by roughly `1200`× compared to cached execution. Beyond this, UncoreBleed can apply a set of core slowdown techniques: (1) lock a core's base frequency to its lowest supported setting; (2) reduce the duty cycle to 12.5% using `IA32_CLOCK_MODULATION`; and (3) disable turbo boost while forcing `intel_pstate` into `powersave` mode, thereby locking the core into its lowest P-state.

While the exact slowdown varies across platforms, our measurements indicate that these techniques reduce instruction throughput to approximately one-half to one-third of the uncached baseline, corresponding to a total magnification of roughly 2500–3600×. For modern cryptographic libraries, where a single RSA loop iteration typically completes in tens of nanoseconds, these combined slowdowns inflate iteration times into the tens of microseconds. Under these conditions, our high-resolution sampling primitive reliably satisfies $t_{interval} < t_{execution}$.

Parallel Monitoring. Monitoring a single sensitive target is often insufficient to recover secrets from applications with multiple sensitive targets. A joint analysis of sampling multiple targets in parallel yields finer control-flow resolution. To achieve this, UncoreBleed exploits the address-to-M2M mapping (Section 4.3) to determine the corresponding M2M box for each sensitive target. Since each M2M box exposes its own `PKT_MATCH` counter, UncoreBleed can monitor multiple sensitive targets in parallel (e.g., 4 in our platforms), with the concurrency bounded by the number of M2M devices.

6 Security Implications of UncoreBleed

To demonstrate the security implications of UncoreBleed, we present two case studies. The first targets Libjpeg [14], a widely used library for JPEG decompression. Here, UncoreBleed captures input-dependent execution with 64 B granularity, while prior controlled channels [51] operate only at the 4-KB page level and cannot distinguish branches within the same page. Instead, they infer execution indirectly by page-faulting targeted stack pages.

The second is to recover keys from RSA protected TLBlur [44], a state-of-the-art software defense deployed on off-the-shelf SGX platforms.

```
1  jpeg_idct_islow() {
2    ......
3    for each column in DCT block {
4      if (all AC coefficients == 0) {
5        // Fast path: only use DC term
6        fill column with DC_value;
7        ......
8        continue;
9      }
10
11     // Normal path: full inverse DCT
12     compute_even_part();
13     compute_odd_part();
14     combine_results();
15     ......
16   }
17 }
```

(a) `jpeg_idct_islow()`

```
1  /* square() and multiply() resides
2   on a different page than modpow() */
3
4  square(res, n) {......}
5  multiply(res, a, n){......}
6
7  int modpow(base, d, n) {
8    ......
9    //d represents the private key
10   for (i < length(d)) {
11     res = square(n);
12     if (d & mask)
13         res = multiply(base, n);
14     mask >>= 1;
15   }
16   ......
17 }
```

(b) `modpow()`.

Figure 8: Both JPEG inverse DCT and RSA modular exponentiation functions have secret-dependent paths.

6.1 Recover Pictures from Libjpeg

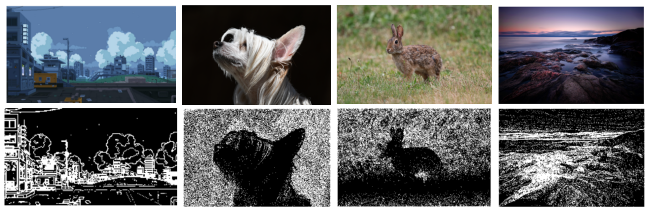


Figure 9: Original JPEG pictures (top) and their corresponding reconstructions from UncoreBleed (bottom). From left to right, the image resolutions are (1) 900 × 600, (2) 2160 × 1280, (3) 4016 × 2016, and (4) 6016 × 4016 pixels.

Libjpeg [14] partitions input images into 8×8 discrete cosine transform (DCT) blocks during decoding. Each block contains a DC coefficient in the top-left entry and 63 AC coefficients in the remaining entries.

In the inverse transform stage, the `jpeg_idct` family of functions (e.g., `jpeg_idct_islow()` in Figure 8a) perform a two-dimensional inverse DCT to reconstruct pixel values using a data-dependent optimization: if all AC coefficients in a row or column are zero, the decoder takes a fast path; otherwise, it executes the normal path with full multiply–accumulate computations.

Thus, the control path taken leaks information about the presence of non-zero AC coefficients, and monitoring this behavior across multiple blocks can reveal image contours [26,

51]. However, since the `jpeg_idct` family function resides entirely within a single code page and makes no further calls, prior controlled-channel attacks cannot distinguish the fast and normal paths at the page level. Instead, they indirectly infer differences by comparing the number of page faults on stack pages accessed in each path.

Unlike them, UncoreBleed operates at 64 B granularity, allowing it to directly distinguish between the two paths within the same code page. In the following, we demonstrate UncoreBleed against `jpeg_idct_islow()` in Libjpeg-turbo v3.0.3 on Xeon Bronze 3408U, following the three main steps in Section 5.2 to recover the input pictures.

Locating Sensitive Target. The leakage stems from the `for` loop in `jpeg_idct_islow()`. As shown in Figure 8a, Line 3 introduces the `for` loop, followed by an `if` statement at Line 4 that checks the AC coefficients. Lines 6–8 implement the fast path when the condition is true. Lines 12–15 execute the normal path. To distinguish between the two, the attacker must reliably detect the start of the loop and decide whether execution subsequently enters the fast path.

Since the loop entry and the fast-path code reside adjacently in memory, and the loop entry occupies only 10 B, aligning the filter to the loop entry ensures that both share a single 64 B memory block. At runtime, counter updates would exhibit distinct signatures: a sharp spike followed by a brief pause indicates the fast path since the spike means the execution of the monitored fast path, and the short pause reflects the time until the next iteration begins. A normal path is characterized by a small counter increase followed by a much longer pause, since the small rise comes from the loop entry, while the extended period of no counter growth occurs because execution of the normal path that is not monitored.

Runtime Monitoring. After the loop entry is identified, UncoreBleed extracts the target's physical address from OS-controlled page tables, maps it to the responsible M2M box, and programs the `PKT_MATCH` filter accordingly. Monitoring then begins on a dedicated core with high-frequency sampling. The enclave execution is deliberately stretched as described in Section 5.4.

Picture Recovery. The collected traces indicate whether each loop iteration followed the fast or normal path. Figure 9 shows an example sequence of "F, F, N, N, N, F," where a fast-path iteration is denoted as "F" and a normal-path iteration as "N". This sequence reflects the decoder applying different paths across consecutive DCT blocks. The two types of iterations are distinguishable: "F" is indicated by a sharp counter spike within a very short interval, followed by a brief pause, whereas "N" exhibits a small counter increase followed by a much longer pause.

By aggregating these traces across all DCT blocks, we can check the presence or absence of non-zero AC coefficients, thereby recovering block structures and image contours. Figure 9 presents recovered pictures of varying resolutions, demonstrating that UncoreBleed successfully extracts visual information from Libjpeg.

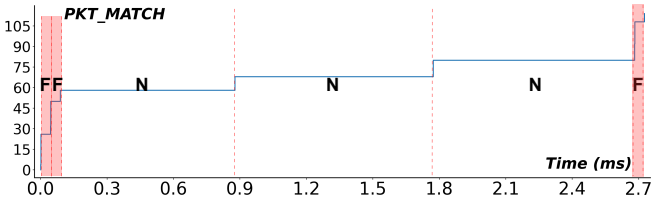


Figure 10: Sampled traces of monitoring the loop entry and fast path. "F" denote when the function takes the fast path, while "N" means the normal path.

6.2 Recover TLBlur-Protected RSA Keys

TLBlur [44] is the most practical software-only defense against controlled-channel attacks on commodity SGX platforms. It leverages AEX-Notify [6], a recent hardware extension in Intel SGX processors, to prefetch sensitive pages into the TLB upon enclave resumption after interrupts or page faults. This mechanism conceals subsequent memory accesses from page-table walks. However, since page tables are still under OS control, TLBlur cannot defend against an attacker that directly monitors physical memory activities. UncoreBleed achieves this: by observing specific memory accesses without triggering AEXs, it bypasses both TLBlur and AEX-Notify.

To evaluate the effectiveness of UncoreBleed against TLBlur, we examined its open-sourced repository[4], which uses RSA as a representative cryptographic workload. This repository also includes a page-fault-based attack against RSA, which we confirmed is fully mitigated by its defense. We then applied UncoreBleed, following the three steps in Section 5.2, to recover secret keys from TLBlur-protected RSA on an off-the-shelf Xeon Silver 4514Y platform.

Locating Sensitive Target. In the TLBlur example, the critical secret-dependent computation resides in the function `modpow()`, shown in Figure 8b. As introduced in prior works [3, 51], the key bit can be inferred by whether `multiply()` in Line 13 is executed: if only `square()` (Line 11) runs, the bit is '0'; if both `square()` and `multiply()` execute, the bit is '1'. Since these two functions occupy different code pages, prior controlled-channel attacks relied on page-faulting them to reconstruct keys.

UncoreBleed targets these two functions. However, a careful selection of monitoring specific physical locations is needed as the locations may be mapped to the same M2M device. As described in Section 4.3, the mapping function depends on multiple address bits, with bit 8 being the lowest relevant. It means that physical addresses differing in bit 8 (e.g., 256 B) are assigned to different M2M devices. Thus,

[4] https://github.com/TLBlur-SGX/tlblur

we identify a suitable target address within `multiply()` that maps differently from the chosen one in `square()`, enabling independent monitoring of the two functions.

Runtime Monitoring. After the two monitoring locations are located, we deploy TLBlur-protected RSA example at runtime, where `ecall_rsa_decode()` is protected using `tlblur_enable()` and `tlblur_disable()`. This function encapsulates the critical `modpow()` function. UncoreBleed only passively monitors `square()` or `multiply()` and does not trigger TLBlur's prefetching. While `square()` and `multiply()` reside on different pages in this case, UncoreBleed allows both functions to reside within the same page, due to its 64 B granularity.

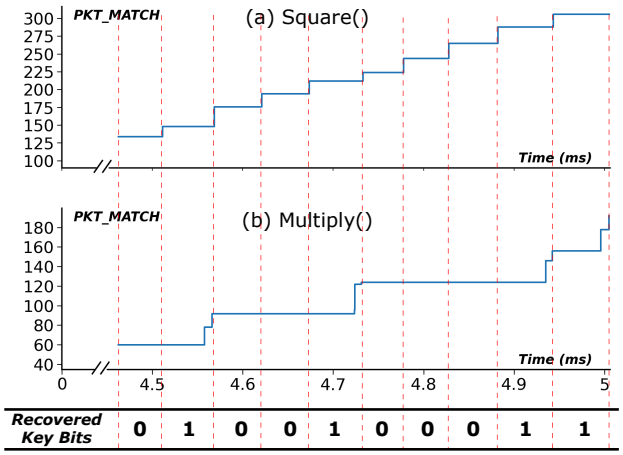


Figure 11: Part of sampled traces during TLBlur-protected `modpow()`. The trace from `square()` (top) marks the iteration boundary. The trace from `multiply()` recovers the secret bits.

Secret Key Recovery. Figure 11 shows part of sampled traces during TLBlur-protected `modpow()`. The top trace denotes `square()`, while the bottom represents `multiply()`.

Since `square()` executes in every loop iteration, the segments of solid line separated by dashed lines denote the duration of individual iterations. For each iteration, we examine the trace of `multiply()`: if its counter increases, `multiply()` has executed and thus the key bit is '1'; if not, the bit is '0'. From the shown trace, the recovered sequence of key bits is '0100100011'.

In this experiment, we respectively evaluated three key lengths with 64, 128, and 256-bit private exponents. For each length, we repeated the attack above 3 times. In all runs, we successfully recovered every key bit.

7 Discussion and Mitigation

UncoreBleed on Dual-Socket Platforms. As discussed in Section 4.3, Xeon Scalable processors contain four M2M boxes per socket. To evaluate UncoreBleed's scalability on multi-socket systems, we tested a Supermicro X13DEI motherboard with one or two Xeon Silver 4514Y processors, representing single- and dual-socket configurations. Linux enumerates M2M components via `sysfs` and we observed that the number of physical M2M boxes scales with the number of sockets. In the single-socket configuration, querying the Device IDs in Table 2 for Xeon Silver 4514Y revealed one group of four M2M boxes (`fe:0c.0-fe:0f.0`). In the dual-socket configuration, two groups of four M2M boxes were enumerated (`fe:0c.0-fe:0f.0` and `7e:0c.0-7e:0f.0`), confirming that each socket maintains its set of four M2M boxes. Across sockets, the Device:Function numbering consistently maps to specific M2M components. For example, `0c:00` to `M2M_0` and `0d:00` to `M2M_1`. Thus, M2M boxes with the same Device:Function numbers in different PCI domains are mapped to the same M2M components within their sockets.

To understand cross-socket behavior, we designed an experiment where all `ADDRMASK` registers of the eight M2M boxes were cleared, a pre-allocated physical address was programmed into the `ADDRMATCH` registers, and repeated `clflush+reload` [53] accesses were performed sequentially on each CPU core. The results revealed a clear socket-local effect: accesses from cores on socket 0 triggered `PKT_MATCH` increments only in the first group of M2M boxes (`fe:0c.0-fe:0f.0`), while the second group remained inactive. For accesses from cores on socket 1 triggered increments only in the second group (`7e:0c.0-7e:0f.0`). This demonstrates that the address-to-M2M mapping still holds in the dual-socket configuration and that each socket maintains a private set of M2M boxes for performance monitoring, effectively isolating the monitoring capabilities for each socket.

These observations indicate that UncoreBleed can still perform targeted side-channel monitoring in dual-socket settings. By mapping the M2M box group to the socket where targeted enclave executes, UncoreBleed leverages socket-local `PKT_MATCH` counters to track memory accesses. While these results confirm effective monitoring on dual-socket settings, investigating larger multi-socket systems remains future work.

Other Potential Uncore Events. By design, the `PKT_MATCH` event in the M2M unit was intended for performance analysis. However, its capability for programmable filtering of fine-grained memory traffic also enables runtime enclave monitoring. Therefore, we will explore two other uncore subsystems, Ultra Path Interconnect (UPI) and Compute Express Link (CXL), that expose similar programmable filtering events.

The UPI fabric provides link-layer monitoring events such as `TxL_BASIC_HDR_MATCH` and `RxL_BASIC_HDR_MATCH`. These allow fine-grained filtering of memory packets traversing the inter-socket interconnect. Filters can be configured along multiple dimensions, including transaction type (request, snoop, and response), packet category (data and non-data), and source locality (local and remote).

We performed a preliminary evaluation of `RxL_BASIC_`-

HDR_MATCH on the Xeon Gold 6230 platform where a thread runs on a core on socket 0 and another on socket 1. The socket-1 thread repeatedly writes to a physical address P, with RxL_BASIC_HDR_MATCH configured to count only data-write packets. When the socket-0 thread conditionally writes to P, we observe clear spikes in the event counter from socket 1, while accesses to other addresses cause no such change.

Starting with 5th Generation Xeon processors, Intel integrated CXL into uncore PMCs. The CXL unit supports packet match/mask filters that allow selective observation of memory transactions flowing through shared CXL.mem devices. By programming filters on packet-header fields (e.g., request type or address region), aggregate CXL traffic can be decomposed into enclave-specific subflows, enabling fine-grained inference of memory-access patterns inside an enclave.

UncoreBleed on Intel TDX. Intel Trust Domain Extensions (TDX) support hardware-isolated VMs (i.e., "Trust Domains", TDs) by protecting TD memory from the host OS/VMM, using TDX-module-managed SEPT (Secure Extended Page Table) mappings. We evaluated Intel TDX on the Xeon SILVER 4514Y platform using Intel's default TDX configuration, and confirmed that PKT_MATCH remains capable of observing targeted memory traffic from a running TD, consistent with Intel's reply to our responsible disclosure that SGX/TDX do not aim to prevent host access to uncore performance monitoring.

Implementing UncoreBleed against TD requires the host physical address (HPA) of the sensitive target inside the TD, which involves resolving two mappings. The first is GPA-to-HPA mapping. While SEPT is protected from host inspection, the host provisions TD pages and requests the TDX module to establish SEPT mappings (e.g., via TDH.MEM.SEPT.ADD). Thus, it can learn the mapping. The other is GVA-to-GPA mapping. While the host cannot directly observe the TD's guest page tables, prior work (e.g., TDXdown [48]) suggests this mapping can be inferred via templating. We leave a full UncoreBleed implementation on TDX as future work.

7.1 Mitigation

To effectively mitigate UncoreBleed, a straightforward approach is to disable uncore PMCs (as some other uncore events may also be exploitable), at least the PKT_MATCH event, during enclave execution, similar to the suppression of core PMCs in SGX enclaves. However, this measure would hinder runtime performance analysis. Based on Intel's feedback, they do not plan to disable uncore PMCs capabilities.

To enable legitimate use of PMCs while mitigating CounterSEVeillance [10], TEEcorrelate [46] reduces fine-grained leakage by aggregating counter updates into coarse time windows and introducing controlled noise to event counts. However, it requires hardware modifications and is not available on Xeon processors.

Besides PMC-focused defenses, controlled-channel mitigation is also relevant. Following TLBlur [44], we classify existing countermeasures into four categories. The first two target attacks that rely on controlling AEXs, making them ineffective against UncoreBleed. The last two aim to either randomize enclave execution or eliminate secret-dependent memory access patterns. While effective, these approaches incur high performance overhead and are deemed impractical [44]. Specifically:

Interrupt Detection monitors the rate or pattern of asynchronous enclave exits (AEX) or interrupts to identify potential controlled-channel attacks [4, 29, 35]. These approaches fundamentally assume that a successful attack must trigger AEXs or interrupts, making the detection feasible. However, this reliance leaves them blind to leakage channels that operate entirely without inducing such events. UncoreBleed works by observing physical memory accesses without generating any AEXs or interrupts, thereby evading this class of defenses by design.

Page pinning focuses on keeping sensitive pages resident in the TLB [6, 30, 38, 44, 52], thereby reducing and even preventing exploitable interrupts, page faults or page-table walks. However, they cannot address attacks that bypass the page-table entirely. Since UncoreBleed monitors specific physical memory accesses directly, it remains effective even when all sensitive pages are pinned in the TLB.

Randomization mitigates controlled-channel attacks by unpredictably relocating code and data in memory [2, 34, 54], making it difficult for an attacker to correlate observed memory accesses or page faults with actual secret-dependent operations. However, randomization cannot completely eliminate information leaks, as residual patterns such as page-fault sequences, stack accesses, and control-flow traces may still reveal secrets [47], and frequent re-randomization introduces significant performance overhead. Clearly, strong runtime randomization could prevent controlled-channel attacks including UncoreBleed from pre-identifying physical pages containing sensitive functions.

Memory access obliviousness aims to remove any secret-dependent differences in memory access patterns [1, 37, 47]. This is achieved by techniques such as Oblivious RAM (ORAM)-based execution or compiler-enforced deterministic memory accesses, hiding sensitive information from memory access sequences. However, these approaches generally incur very high performance overhead, require invasive code modifications or manual annotations, and are impractical for complex or legacy enclaves. While complete obliviousness could neutralize UncoreBleed, its deployment cost means real-world enclaves remain without this level of protection.

8 Conclusion

In this work, we demonstrated that, contrary to prior beliefs, not all PMCs were disabled during enclave execution: in particular, uncore PMCs are still active. Further, we iden-

tified the PKT_MATCH event in the M2M uncore subsystem, which enabled fine-grained monitoring of enclave memory accesses with 64 B granularity. Through comprehensive reverse engineering, we uncovered the event's filtering mechanism, programmability, availability, and address mapping across multiple generations of SGX-capable Xeon processors, despite incomplete or inaccurate Intel documentation.

Leveraging this, we developed UncoreBleed, the first PMC-based side-channel attack against SGX that operated without triggering AEXs, maintained low noise, and achieved high resolution. Using UncoreBleed, we successfully reconstructed visual content from JPEG images and extracted RSA private keys from a single decryption operation, even in the presence of TLBlur. As a result, our study overturned the long-held assumption that SGX completely suppressed performance monitoring and highlighted that active uncore PMCs could be exploited to compromise enclave confidentiality.

Ethical Considerations

All experiments in this work were conducted by us on machines under our exclusive control, and no experiments were performed on systems without the owner's consent. Potential stakeholders include Intel (SGX), cloud providers, enclave developers (especially those implementing SGX-based cryptographic algorithms), and the academic security community.

Before submission, we reviewed Intel's SGX guidelines, which consistently emphasize constant-time/data-oblivious enclave code [22, 23]. Our preliminary assessment was that UncoreBleed monitors specific enclave physical memory accesses, which could be neutralized by data-oblivious enclave code. We expected that UncoreBleed would likely be considered outside Intel's scope. Therefore, we thought that Intel would benefit most from a complete package of the paper and code, including the reverse-engineered details of the M2M filtering mechanism across multiple generations of Xeon platforms, which could help Intel validate our findings and update their inaccurate and incomplete documents.

Due to the conference deadline, we prioritized finishing the reverse-engineered results and assembling a complete package so that both reviewers and Intel could accurately evaluate the attack in depth. As a result, we disclosed UncoreBleed to Intel shortly after submission. In retrospect, we recognize that earlier disclosure would have been preferable, and we are committed to initiating contact earlier in future work.

Shortly after submission, we reported UncoreBleed to Intel PSIRT, providing the full paper and code and informing them of the submission. After reviewing our report, Intel classified the vulnerability as out of scope under their threat model, reiterating their recommendation that enclave code follow constant-time or data-oblivious coding practices [22, 23]. Intel indicated that they did not plan a mitigation specific to UncoreBleed and noted that SGX/TDX do not aim to prevent OS/VMM access to general performance monitoring capabilities. Their only request was that we provide advance notice of public disclosure and artifact release so that they could prepare customer messaging. We updated Intel with the planned publication dates for the final version of the paper and the artifact, which they acknowledged.

We believe that the benefits of being able to explore, understand, and quantify new side-channel leakage in SGX outweigh the potential harms of making the leakage openly available (e.g., UncoreBleed enables an attacker to extract secrets from legacy enclave programs that exhibit secret-dependent memory-access patterns). Publishing this work with an open-source and reproducible artifact, together with explicit mitigations, maximizes defensive benefit, corrects previous misunderstandings about PMUs during SGX enclaved execution, empowers a broad set of stakeholders to harden systems, and advances SGX vulnerability research while minimizing potential security threats.

Open Science

All of our software artifacts, including the case studies, are available in a public repository at https://doi.org/10.5281/zenodo.17902416. The repository includes 6 subdirectories. Each subdirectory contains its own README.md with build, run, and usage instructions. Specifically, MicroBenchmark/ contains a microbenchmark suite to compare core and uncore PMCs inside a production-mode SGX enclave and in a non-enclave environment; EventAnalysis/ includes test programs and analysis scripts to study the PKT_MATCH behavior; ReverseM2MMap/ provides the address-to-M2M reverse-engineering details; tools/ collects low-level infrastructure used by the attacks (e.g., VA→PA translation, PA→M2M mapping helpers, PKT_MATCH configuration, high-frequency sampling, and kernel modules); Demo/ and RecordAnalysis/ contain the end-to-end case studies and offline trace-processing scripts.

Acknowledgments

We would like to thank the anonymous reviewers for their advice and comments, which helped improve this paper.

This research was supported partly by the National Science Foundation of China Fund No. U24B20151, and the Shenzhen Bureau of Science and Technology Research Fund No. KJZD20240903102708012.

References

[1] Shaizeen Aga and Satish Narayanasamy. Invisipage: Oblivious demand paging for secure enclaves. In *International Symposium on Computer Architecture*, pages 372–384, 2019.

[2] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. Dr. sgx: Automated and adjustable side-channel protection for sgx using data location randomization. In *Annual Computer Security Applications Conference*, pages 788–800, 2019.

[3] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:{SGX} cache attacks are practical. In *USENIX Workshop on Offensive Technologies*, 2017.

[4] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Asia Conference on Computer and Communications Security*, pages 7–18, 2017.

[5] Yun Chen, Lingfeng Pei, and Trevor E Carlson. Afterimage: Leaking control flow data and tracking load operations via the hardware prefetcher. In *Architectural Support for Programming Languages and Operating Systems*, pages 16–32, 2023.

[6] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. {AEX-Notify}: Thwarting precise {Single-Stepping} attacks through interrupt awareness for intel {SGX} enclaves. In *USENIX Security Symposium*, pages 4051–4068, 2023.

[7] Victor Costan and Srinivas Devadas. Intel SGX explained. *Cryptology ePrint Archive*, 2016.

[8] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[9] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient caches for trusted execution environments. In *USENIX Security Symposium*, pages 451–468, 2020.

[10] Stefan Gast, Hannah Weissteiner, Robert L. Schröder, and Daniel Gruss. Counterseveillance: Performance counter attacks on AMD SEV-SNP. In *Network and Distributed System Security Symposium*, 2025.

[11] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing os abstraction. In *Proceedings of the Fourteenth EuroSys Conference*, 2019.

[12] 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 *IEEE Symposium on Security and Privacy*, pages 245–261, 2018.

[13] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution side channels for untrusted operating systems. In *USENIX Annual Technical Conference*, pages 299–312, 2017.

[14] Independent JPEG Group. libjpeg. http://ijg.org/. Accessed: 2025-08-18.

[15] Intel Corporation. Intel xeon processor scalable memory family uncore performance monitoring reference manual. Technical Report 336274-001US, Intel Corporation, July 2017.

[16] Intel Corporation. *3rd Gen Intel® Xeon® Processor Scalable Family, Codename Ice Lake, Uncore Performance Monitoring Reference Manual*, revision 1.00 edition, May 2021. Reference Manual.

[17] Intel Corporation. *Intel® Software Guard Extensions (Intel® SGX) Programming Reference*, revision 2.0 edition, 2022. Available at https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf.

[18] Intel Corporation. *4th Gen Intel® Xeon® Scalable Processor XCC (Codename Sapphire Rapids) Uncore Performance Monitoring Guide*, revision 001 edition, April 2024. Monitoring Guide.

[19] Intel Corporation. *5th Gen Intel® Xeon® Scalable Processor XCC (Codename Emerald Rapids) Uncore Performance Monitoring Guide*, revision 001 edition, August 2024. Monitoring Guide.

[20] Intel, Inc. Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c. October 2011.

[21] Intel, Inc. Intel 64 and ia-32 architectures software developer's manual, September 2016.

[22] Intel, Inc. Intel software guard extensions developer guide: Protection from side-channel attacks, 2017.

[23] Intel, Inc. Guidelines for mitigating timing side channels against cryptographic implementations, 2022.

[24] 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 Symposium*, pages 557–574, 2017.

[25] 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 *IEEE Symposium on Security and Privacy*, 2021.

[26] Chang Liu, Shuaihu Feng, Yuan Li, Dongsheng Wang, Wenjian He, Yongqiang Lyu, and Trevor E. Carlson. *MDPeek: Breaking Balanced Branches in SGX with Memory Disambiguation Unit Side Channels*, page 622–638. 2025.

[27] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.

[28] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled Instruction-Level attacks on enclaves. In *USENIX Security Symposium*, pages 469–486, 2020.

[29] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting {SGX} enclaves from practical {Side-Channel} attacks. In *USENIX Annual Technical Conference*, pages 227–240, 2018.

[30] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing controlled channels with self-paging enclaves. pages 1–16, 2020.

[31] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security Symposium*, pages 565–581, 2016.

[32] Mark Russinovich. Confidential computing: Elevating cloud security and privacy. *Communications of the ACM*, 67(1):52–53, 2023.

[33] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[34] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *Network and Distributed System Security Symposium*, 2017.

[35] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Network and Distributed System Security Symposium*, pages 15–43, 2017.

[36] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Asia Conference on Computer and Communications Security*, page 317–328, 2016.

[37] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Asia Conference on Computer and Communications Security*, pages 317–328, 2016.

[38] Raoul Strackx and Frank Piessens. The heisenberg defense: Proactively defending sgx enclaves against page-table-based side-channel attacks. *arXiv preprint arXiv:1712.08519*, 2017.

[39] TrustedFirmware.org. Mbed TLS. https://www.trustedfirmware.org/projects/mbed-tls/, 2024. Accessed May 13, 2025.

[40] 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 Symposium*, pages 991–1008, 2018.

[41] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX@SOSP*, pages 4:1–4:6, 2017.

[42] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *ACM SIGSAC Conference on Computer and Communications Security*, page 178–195, 2018.

[43] 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 Symposium*, pages 1041–1056, 2017.

[44] Daan Vanoverloop, Andres Sanchez, Flavio Toffalini, Frank Piessens, Mathias Payer, and Jo Van Bulck. Tl-blur: Compiler-assisted automated hardening against controlled channels on off-the-shelf intel sgx platforms. In *USENIX Security Symposium*, 2025.

[45] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. Dramdig: A knowledge-assisted tool to uncover dram address mapping. In *Design Automation Conference*, 2020.

[46] Hannes Weissteiner, Fabian Rauscher, Robin Leander Schröder, Jonas Juffinger, Stefan Gast, Jan Wichelmann, Thomas Eisenbarth, Daniel Gruss, SIT Fraunhofer, and Vienna Fraunhofer Austria. Teecorrelate: An information-preserving defense against performance-counter attacks on tees. In *USENIX Security Symposium*, 2025.

[47] Jan Wichelmann, Anja Rabich, Anna Pätschke, and Thomas Eisenbarth. Obelix: Mitigating side-channels through dynamic obfuscation. In *IEEE Symposium on Security and Privacy*, pages 4182–4199, 2024.

[48] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. Tdx-down: Single-stepping and instruction counting attacks against intel tdx. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 79–93, 2024.

[49] wolfSSL Inc. wolfSSL Lightweight SSL/TLS Library. https://www.wolfssl.com/, 2024. Accessed May 13, 2025.

[50] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation. In *USENIX Security Symposium*, pages 19–35, 2016.

[51] Yi Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[52] Sujay Yadalam, Vinod Ganapathy, and Arkaprava Basu. SG^{XL}: Security and performance for enclaves using large pages. *ACM Transactions on Architecture and Code Optimization*, 18(1):1–25, 2020.

[53] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *USENIX Security Symposium*, pages 719–732, 2014.

[54] Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. Klotski: Efficient obfuscated execution against controlled-channel attacks. In *Architectural Support for Programming Languages and Operating Systems*, pages 1263–1276, 2020.