2026 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

DSAssassin: Cross-VM Side-Channel Attacks by Exploiting Intel Data Streaming Accelerator

Ben Chen†, Kunlin Li† Shuwen Deng§, Dongsheng Wang§, Yun Chen†
†The Hong Kong University of Science and Technology (Guangzhou)
§Tsinghua University
chenb2022@mail.sustech.edu.cn, kli082@connect.hkust-gz.edu.cn, {shuwend, wds}@tsinghua.edu.cn,
yunchen@hkust-gz.edu.cn

***Abstract*—Modern datacenter infrastructures increasingly integrate bespoke accelerators to offload specific workloads from CPUs. Among them, Intel's Data Streaming Accelerator (DSA), supported by Intel VT-d, is deployed in the latest Xeon processors. However, its security implications remain unexplored. In this paper, we propose DSAssassin, a novel side-channel attack launched from DSA. Although Intel's scalable I/O virtualization mitigates device-based threats, we show that DSA introduces a new attack surface that bypasses such protections.**

By reverse-engineering the Device TLB (DevTLB) introduced by DSA, we find that it is not isolated by different processes. We further analyze DSA's shared work queue (SWQ) and uncover a timer-free side channel via DMWr transaction. Building on these insights, we develop two attack primitives: a DevTLB-based timing attack and an SWQ-based contention attack. These enable cross-VM covert- and side-channel attacks. First, the cross-VM covert-channel can achieve a true capability of 17.19 Kbps with a 4.64% error rate—5× faster and 4× more accurate than the latest work. Second, we present a website fingerprinting attack on the top 100 websites, and the classification accuracy can reach 85.7%. Finally, we demonstrate a cross-VM keystroke inference attack with F_1 scores of 92.0% (DevTLB) and 98.4% (SWQ)—outperforming the state-of-the-art.

I. Introduction

Datacenters, as memory-intensive infrastructures, consume a significant share of CPU cycles in data movement. A growing trend is to offload memory operations to on-chip or external accelerators to reduce CPU workload. To support this, Intel introduced the Data Streaming Accelerator (DSA) in its 4th Generation Xeon Scalable processors (Sapphire Rapids) [23], [28], enabling high-throughput memory transactions. Notably, DSA represents a novel user-programmable Data Processing Unit (DPU).

DSA operates as a PCIe device and leverages Intel's I/O virtualization (VT-d), which offers direct memory access with enhanced isolation. Prior to VT-d, I/O virtualization security was primarily enforced through the IOMMU, which restricts device access via I/O Virtual Addresses (IOVAs). However, IOMMUs still expose side-channel risks, particularly through IOTLB leakage [36], [60]. VT-d addresses this by introducing Process Address Space IDs (PASIDs) [22], enabling per-process IOTLB isolation and support for Shared Virtual Memory (SVM) [59], which allows devices to access userspace virtual addresses without additional address translation.

To the best of our knowledge, this work presents the first security analysis of Intel DSA. Despite the strong isolation guarantees introduced by VT-d, we propose DSAssassin, a novel side-channel attack vector originating from DSA that bypasses VT-d's protections deployed on virtual machine (VM) boundaries. In the absence of prior studies or official documentation on DSA's security implications, we develop a microbenchmark suite to perform the first microarchitectural-level reverse engineering of DSA.

The first key observation from the reverse-engineering enabling DSAssassin is the **unprotected DevTLB** feature. In addition to the IOTLB, DSA encapsulates device-side TLB (DevTLB) with Address Translation Service (ATS) to alleviate IOMMU pressure [50]. To dissect DSA's DevTLB, we reverse-engineer its capacity and policy. We find that DevTLB entries are indexed not by virtual address, but by **descriptor fields**, which we classify as source/source2 (`src/src2`), destination/destination2 (`dst/dst2`), and completion record address (`comp`). More importantly, we discover that the DevTLB **is not isolated by PASID** and is dedicated to engines. Finally, we benchmark the access latency variant of DevTLB with user-level `rdtsc` timer. Our in-depth characterizations of DevTLB reveal its potential as an unprivileged covert and side-channel attack vector within DSA, capable of bypassing virtual machine (VM) isolation.

Our second key observation targets shared work queues (SWQs). DSA supports synchronized work submissions to SWQs, ensured by Deferrable Memory Write (DMWr) transactions. DMWr is a non-posted MMIO write used to signal submission status. We find that a **failed DMWr indicates queue congestion**. Although initially DMWr exhibits constant-time behavior when benchmarking the timing differences between idle and full queues, we find that DMWr still leaks information: an attacker can flood an SWQ with descriptors, causing congestion, and infer victim submissions from different VMs, without relying on timing information.

To demonstrate the security impact across VMs, we develop two attack primitives. The first primitive introduces a Prime+Probe-style attack on DevTLB, requiring only a shared DSA engine between the victim and the attacker. The second exploits SWQ contention to detect victim submissions via DMWr, without timing measurements.

We evaluate both cross-VM covert-channel and side-channel attacks on multiple case studies. As no information from CPU cores is required for launching our attacks, the attacker and victim *do not necessarily share a CPU core*. For the cross-VM covert-channel attack, we achieve 17.19 Kbps with an error rate of 4.63%. In terms of side channels, we perform our attacks on Data Plane Development Kit (DPDK) and DSA Transparent Offload (DTO) workloads, two widely deployed libraries on Intel processors that leverage DSA. Our DevTLB-based attack fingerprints DPDK network traffic to classify 100 websites with 85.7% accuracy and captures keystrokes from a DTO-hooked SSH session with an F_1 score of 92.0%. The SWQ-based attack achieves an F_1 score of 98.4%. In addition to bypassing VT-d's protections, our attacks also demonstrate superior efficiency compared to state-of-the-art cross-VM side-channel techniques [36], [51], [65].

In summary, our contributions are as follows.

- We reverse-engineer the undocumented DevTLB and work queue microarchitecture in Intel DSA. To our knowledge, this is the first public analysis of DSA's microarchitecture details.
- We propose DSASSASSIN, a novel side-channel framework including two attack primitives: a timing attack on DevTLB and a timer-free contention attack via shared work queues using DMWr. These are the first attacks on DevTLB and DMWr, capable of breaking VT-d isolation.
- We show that DSASSASSIN achieves a covert-channel bandwidth of up to 17.19 Kbps with only 4.64% error in a cross-VM setting: 5× faster and 4× more accurate than the latest work.
- We present a cross-VM SSH keystroke attack with F_1 scores of 92.0% (DevTLB-based primitive) and 98.4% (SWQ-based primitive).
- We demonstrate that DSASSASSIN enables website fingerprinting attacks with 85.7% classification accuracy in a cross-VM scenario.

Responsible disclosure. We have disclosed our findings to Intel PSIRT, who has acknowledged the issue. Artifact is released at https://github.com/hkustgz-secarch/dsassassin.

II. BACKGROUND

A. Data Streaming Accelerator

Intel DSA is a high-throughput data mover designed to offload and accelerate memory operations such as `memset`, `memcpy`, `memcmp`, checksumming (e.g., CRC, DIF), and patching (e.g., delta record generation and merging). These operations are typically lightweight read/write memory requests with minimal computation, and are common in datacenter applications, storage, networking, deduplication, VM migration, and checkpointing workloads, necessitating dedicated hardware acceleration.

Figure 1 shows a simplified architecture of DSA within an Intel Xeon Scalable processor. DSA is integrated on-chip alongside PCIe and CPU cores and communicates via PCIe. It accepts two descriptor types from software: work descriptors and batch descriptors. To submit a job, software prepares **descriptors** and writes them to an MMIO portal, which forwards them to the appropriate work queue. Work queues buffer descriptors and dispatch them to available engines. Batch engines fetch work descriptors via pointers and store them in internal buffers. An arbiter then selects descriptors from either the batch buffer or the work queue.

Each processing unit decodes descriptors, reads from the source address, performs computation, writes results to the destination, and notifies the CPU if the job is finished via **completion records** or interrupts. During descriptor execution, memory accesses in stages 2, 3, and 5 of the pipeline conform to the Address Translation Services (ATS) protocol (see Section II-B).

B. Address Translation Services

Intel DSA enforces Address Translation Services (ATS) for I/O virtualization via two key capabilities: simplification and offloading. In terms of simplification, PASID-tagged Shared Virtual Memory (SVM) replaces traditional I/O Virtual Addresses (IOVAs). Whereas IOVA requires the OS to maintain separate page tables, SVM only requires the OS to assign a PASID when a process opens a device (i.e., maps a DSA portal into its address space).

Regarding the offloading, ATS grants devices the privilege to access memory via physical addresses (PAs). Unlike the (IO)MMU, ATS performs translation through the following steps: ❶ the device sends a virtual address (VA) translation request to the Translation Agent (TA); ❷ the TA uses the PASID and context to walk the process's page table, as in a typical MMU; ❸ the device receives the translated PA, caches it in its DevTLB, and accesses memory directly. To restrict malicious access, DevTLB is assured of read-only to devices. Additionally, the Page Request Service (PRS) enables devices to report major page faults without tenant intervention.

The ATS architecture for DSA is illustrated in the right portion of Figure 1. The Address Translation Cache (ATC), referred to as the Device TLB (DevTLB) in PCIe and DSA specifications [23], [29], [50], stores recent translations. A DevTLB hit allows the device to retrieve the target physical address directly from the DevTLB. In contrast, a DevTLB miss triggers the Translation Agent (TA) to intercept the request and perform virtual-to-physical address translation before memory access can proceed.

C. Cache-Based and Contention-Based Side-Channel Attacks

Cache-based side-channel attacks are well-known sources of side-channel leakage. Vulnerable caches are typically shared resources, such as the Last-Level Cache (LLC) [44] and TLB [20]. However, cache-based channels alone suffer from noise and passive observability. To amplify leakage, attackers often abuse additional microarchitectural units, including branch predictors [5], [10], [37], [43], prefetchers [9], [11], [12], and internal buffers [47], [62]. By combining cache side-channel leakage with that from other vulnerable units, the adversaries can reveal the secret value (e.g., encryption key) or even the system state. For instance, adversaries can determine whether an address resides in kernel space—breaking KASLR [13], [31], or validate forged pointer authentication

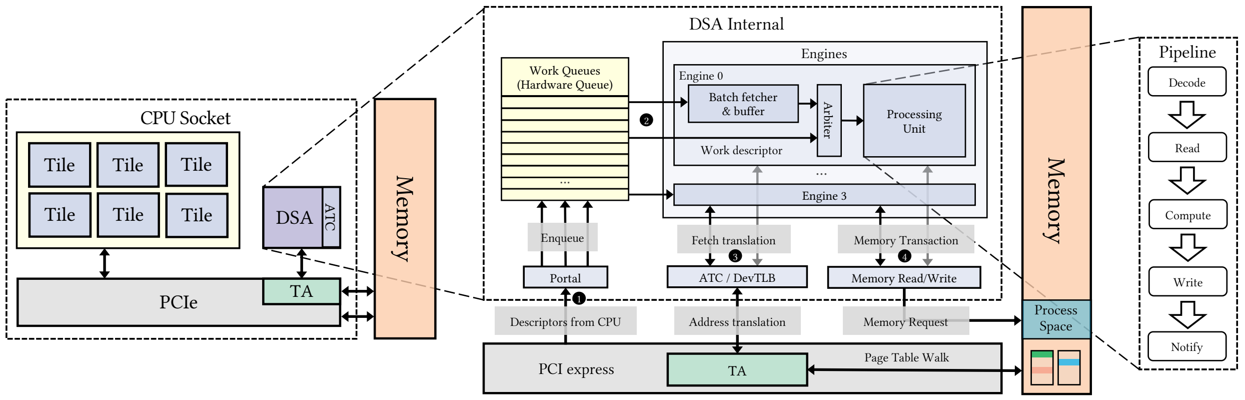


Fig. 1. Overview of DSA and ATS architecture. *Note that DSA accesses memory by sending a memory request to the root complex on the PCIe bus. The leftmost block is partially based on the die shot of Sapphire Rapids [48].

tags [35], [53], as successful validations trigger page walks and cache activity.

Contention-based side-channel attacks exploit microarchitectural components with limited capabilities, e.g., buses and buffers. To detect contention on system buses, adversaries normally initiate interrupts [15], [51], [66] or memory transactions [54] and observe timing variations. PCIe-level contention may also be triggered by peripherals such as GPUs and NICs (RDMA) [56]. Similarly, contention in internal buffers can be detected by saturating them and measuring latency increases [2], [18].

III. Motivation and Challenge

The rise of cloud computing has driven the widespread adoption of accelerators in datacenters to scale computational throughput. However, these accelerators may inadvertently introduce new security risks due to their architectural flaws [1], [19], [33], [67]. Such vulnerabilities are particularly concerning in virtualized environments, where accelerators often bypass the CPU's well-established virtualization protections. Moreover, optimization techniques in complex heterogeneous datacenter systems may unintentionally open side-channel leakage vectors, further compromising system security.

In this work, we conduct an in-depth security analysis of the Intel Data Streaming Accelerator (DSA)—a recent on-chip device designed to accelerate memory operations. Our investigation reveals two key observations that motivate this study: **insufficient DevTLB isolation** and **contention on shared work queues (SWQs)**.

Insufficient DevTLB isolation. Although the PCIe specification has long defined the Device TLB (DevTLB) standard [50], its deployment in commercial devices remains limited. Intel DSA's Address Translation Cache (ATC) is among the few commercial off-the-shelf implementations of DevTLB, yet no prior work has reverse-engineered or evaluated its security. DevTLB is designed to bypass the IOMMU for performance. However, we find that *the DevTLB of DSA lacks strict isolation guarantees*, potentially allowing malicious tenants to observe shared entries, even across PASID-tagged boundaries enforced by I/O virtualization.

Contention on SWQs. DSA supports SWQs to improve accessibility and Quality of Service (QoS). While beneficial for utilization, shared queues can introduce contention-based side channels. We observe that *an attacker can congest a SWQ with high-volume submissions and monitor the resulting delays* to infer co-tenant activity, leaking usage patterns across VMs.

Challenges. Although DevTLB and SWQ exhibit potential for side-channel leakage, practical exploitation requires overcoming significant challenges. Their internal characteristics, such as DevTLB capacity, replacement policies, and SWQ management logic, are undocumented and vendor-specific. Fully reverse-engineering these components is necessary to characterize their behavior, making such attacks non-trivial.

IV. Reverse Engineering DSA

To address the main challenges, we conduct an in-depth characterization of the DSA microarchitecture and uncover critical features that can be exploited for side-channel attacks. This section first outlines our key findings on Intel DSA's microarchitecture in Section IV-A. We then detail our reverse-engineering experiments targeting the DevTLB (Section IV-B) and the shared work queue (Section IV-C).

A. Overview of Reversed DSA Architecture

Figure 2 shows the detailed microarchitecture of Intel DSA after reverse-engineering. Our analysis focuses on two main components: the Device TLB (DevTLB) and the shared work queue (SWQ).

DevTLB. Unlike traditional TLB or IOTLB, the DevTLB does not use virtual or physical memory addresses to index entries. Instead, it indexes entries by **engine ID** and **field type** (e.g., `src`, `src2`). Within the engine, the processing unit retrieves DevTLB entries based on its engine ID, while accesses from the batch fetcher are not cached. As shown in color-coded slots in Figure 2, each engine's DevTLB consists of five sub-entries corresponding to different fields of the descriptor. As each sub-entry only maintains single slot, it is not isolated by PASID-tag.

SWQ. Although software splits different work queues as separate portals, they reside on a single hardware queue.

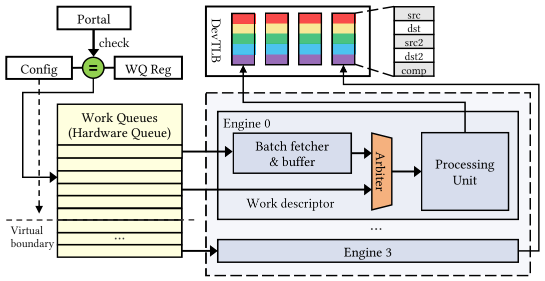


Fig. 2. Revealed detail of DSA implementation.

This hardware queue is partitioned into virtual queues via configuration registers. Each queue's metadata is maintained in per-queue registers and checked against the configuration registers at enqueue time. This design allows queue occupancy checks and congestion detection to occur in constant time.

Undocumented Arbiter. The arbiter highlighted in Figure 2 determines the dispatch order of descriptors within the engine. Although not documented publicly, we find that the arbiter prioritizes descriptors from the work queue over those from the batch buffer.

B. DevTLB Structure and Policy

Methodology and setup. All experiments were performed on two platforms equipped with Intel 4th Gen Xeon processors, each providing hardware support for VT-d and DSA: a local virtualized server with Intel Xeon Platinum 8468V processor and a public cloud virtualized instance based on an Intel Xeon Platinum 8475B processor (powered by Alibaba Cloud) [3]. This dual setup enables us to evaluate the DSA between local and public virtualized environments.

Prior reverse-engineering studies [31], [36] demonstrate that the Performance Monitoring Unit (PMU) is a reliable source of collecting hardware events. Thus, we leverage Intel `Perfmon` [23], [30] for reverse-engineering. `Perfmon` is a specialized, device-level performance counter. Similar to core-level PMUs, it is supported by DSA's internal monitoring logic and exposes metrics via memory-mapped registers.

Among the available `Perfmon` events, we identify three that reflect DevTLB behavior, summarized in Table I. `EV_ATC_HIT_PREV` counts DevTLB hits and serves as our primary signal. `EV_ATC_ALLOC` records the number of DevTLB translation requests, while `EV_ATC_NO_ALLOC` counts requests that do not trigger DevTLB replacing, ensuring visibility into updates on non-evicted entries.

However, `Perfmon` is typically accessed via the Linux `perf`, which requires a privileged user. Therefore, we only use `Perfmon` for benchmarking the DSA. The side-channel attacks are performed by only reading the CPU timer or non-privileged registers.

We first enable a work queue and bind it to a dedicated DSA engine. After that, we prepare four representative work descriptors to trigger DevTLB accesses: `noop`, `memcmp`, `memcpy`, and `dualcast`. Each descriptor exercises different memory behavior: `noop` only writes to the completion record, `memcmp` performs a memory read, and `memcpy` writes data

TABLE I
PERFMON EVENTS FOR EXPERIMENT.

Event Name	Category	Event	Description
`EV_ATC_ALLOC`	0x2	0x40	# requests to DevTLB
`EV_ATC_NO_ALLOC`	0x2	0x80	# not allocated entry
`EV_ATC_HIT_PREV`	0x2	0x100	# hit of entry

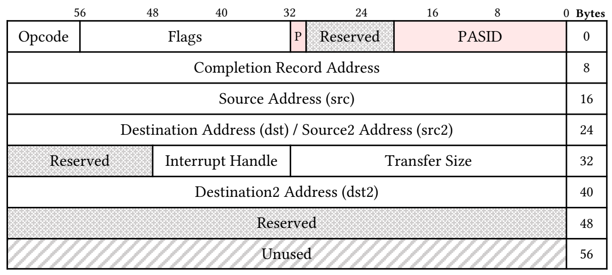


Fig. 3. DSA Descriptor format. Note that the descriptor is tagged with PASID to prevent other processes from executing it directly.

to a destination buffer. `dualcast` issues memory copy from `src` to two separate locations, `dst` and `dst2`. The descriptor's encoding format is shown in Figure 3. The `memcmp` descriptor probing primitive (`probe_memcmp`) is shown in Listing 1. For brevity, flag settings and address alignment are omitted, though these are required for correct handling of the completion record (e.g., Request Completion Record bits). Similarly, we also compose `probe_noop`, `probe_memcpy`, and `probe_dualcast`. Note that probing the `memcpy` requires an additional destination address, and probing the `noop` does not have the source address.

Capacity and replacement policy. We hypothesize that the DevTLB operates as a set-associative cache with W ways and S sets, which is common for modern processors. When $W = 1$, the structure is direct-mapped; when $S = 1$, it is fully associative. We developed a microbenchmark to explore DevTLB's set-associativity (see Listing 2).

The microbenchmark first accesses a base address `base`, followed by memory accesses at offset (`OFFSET`) less than, equal to, and greater than a page size (4 KiB). It then revisits `base`. By monitoring `EV_ATC_HIT_PREV`, we count DevTLB hits on `base`. Notably, when the offset remains within the same page (`OFFSET` $<$ 4 KiB), the event records two hits. In contrast, offsets that cross the page boundary

```c
uint64_t probe_memcmp(void* src, void* src2) {
    uint64_t start, end;
    struct dsa_hw_desc desc = {};
    desc.opcode = DSA_OPCODE_COMPVAL;
    desc.src_addr = src;
    desc.src2_addr = src2;
    enqcmd(wq_portal, &desc);
    // time polling for completion
    start = rdtsc();
    while (comp.status == 0);
    end = rdtsc();
    return end - start;
}
```

Listing 1. Microbenchmark primitive for probing DevTLB.

```
1  probe_noop(base); // base is 4KB aligned
2  probe_noop(base + OFFSET); // try evicting
3  probe_noop(base); // probe if base is present
```

Listing 2. Attempt to evict a DevTLB entry.

(`OFFSET` ≥ 4 KiB) result in no hits.

This behavior confirms that the DevTLB caches translations at the page granularity, ignoring the lower 12 address bits. Furthermore, the lack of a hit after the second access suggests that the DevTLB maintains only a single entry—implying a direct-mapped configuration ($W = 1$, $S = 1$), as Line 2 accesses a new page (miss in the DevTLB) and evicts the cached entry for `base`, therefore causing a miss in the third memory access (Line 3). We also attempt to evict a DevTLB entry corresponding to a 4 KiB page using a conflicting address from a 2 MiB huge page and observe a successful eviction. This result suggests that the **DevTLB does not maintain dedicated entries for different page sizes**.

We design two microbenchmarks to investigate whether the DevTLB supports multiple entries (Listing 3) and to analyze its replacement policy (Listing 4). These experiments employ the previously introduced descriptors: `memcmp`, `memcpy`, and `dualcast`. To avoid indexing collisions, we ensure that no two addresses reside on the same page by varying the source and destination addresses across descriptors.

Listing 3 is designed to determine whether different **field types** (e.g., `src` vs. `dst`) are indexed into the same DevTLB entry. After priming the DevTLB at Line 3, we expect a DevTLB miss if the DevTLB is solely indexed by `src`, since the second access uses a different page. Surprisingly, we observe a **DevTLB hit**, which we attribute to the reuse of the same `dst` address. This indicates that `dst` is independently indexed and not combined with `src`. In other words, each field type is indexed into a dedicated sub-entry in the DevTLB.

To verify this, we then evaluate `dualcast`, which includes one source (`src`) and two destinations (`dst`, `dst2`). When we vary `src` while keeping `dst` and `dst2` unchanged, we observe two DevTLB hits at Line 6. This supports our hypothesis. Finally, we modify Listing 2 and find that each field type's sub-entry only holds a single slot.

We further test potential interference between fields that share encoding space in the descriptor. Specifically, `src2` and `dst` occupy the same bits in the instruction format and are distinguished only by the operation type (see Figure 3): interpreted as `src2` for `memcmp`, and `dst` for `memcpy`. In Listing 4, we map `src2` and `dst` to the same page but use different operations. If these two fields shared the same sub-entry, we would expect two hits at Line 4. However, we observe only a single hit (from the `src`-indexed sub-entry), suggesting no interference between `src2` and `dst`.

Takeaway 1. All five DevTLB **field types**, i.e., `src`, `dst`, `src2`, `dst2`, and `comp`, index into separate sub-entries. Each sub-entry is independently maintained and holds only one cached entry. No cross-field eviction or interference is observed. Besides, as each sub-entry only holds one slot, the new entry evicts the old one directly.

```
1  // different page
2  assert(PAGE_OF(src0) != PAGE_OF(dst0));
3  probe_memcpy(src0, dst0); // prime devtlb
4  // src1 is on a different page with the src
5  probe_memcpy(src1, dst0);
```

Listing 3. The `memcpy` example to discover additional entries.

```
1  // src2 overlaps dst
2  probe_memcmp(src, src2);
3  // dst and src2 in same page
4  probe_memcpy(src, dst);
```

Listing 4. Overlapping fields with different types of operation.

Cross-page behaviors. As physical pages are not necessarily contiguous, DSA splits cross-page memory transactions into per-page segments. To examine this behavior, we configure DSA descriptors with transfer sizes that exceed a single page. As a result, we observe an increase in `EV_ATC_ALLOC` but not in `EV_ATC_NO_ALLOC`, along with a reduction in DevTLB hit counts. Further, if a descriptor accesses an address on the same page as the last-accessed address from a previous descriptor—and both addresses correspond to the same field—we observe an additional DevTLB hit. This suggests that the DevTLB caches only the translation for the final page segment of a cross-page transaction.

We therefore conclude that DSA segments cross-page accesses internally and caches only the translation of the latest page touched by a descriptor field in the DevTLB.

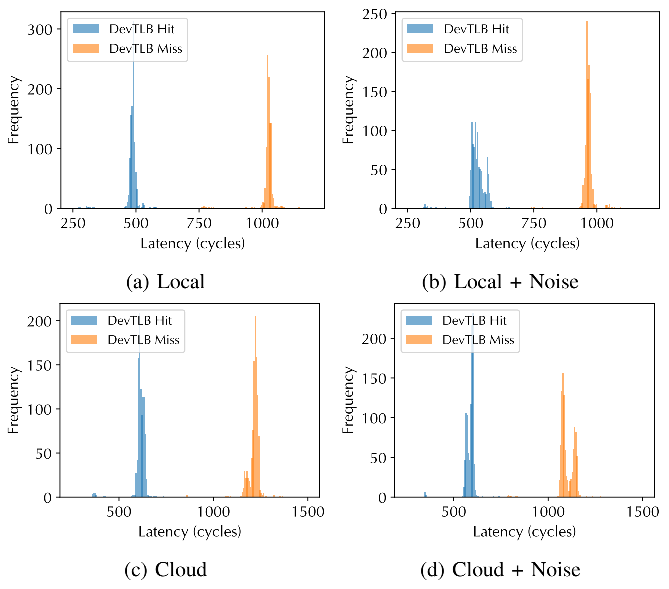
Distinguishable latency. An unprivileged adversary is limited to using user-space timers or registers, such as `rdtsc`, to infer microarchitectural behavior. To evaluate the feasibility of such timing-based inference, we re-run the benchmark from Listing 2, but collect latency measurements instead of `Perfmon` events. Prior to measurement, both the page table and IOTLB are warmed up to eliminate their effects.

Figure 4 presents the latency distribution for DevTLB hits (left) versus misses (right) across four distinct environment configurations: local quiet environment (`Local`), local environment with background noises under high workload pressure (`Local + Noise`), public cloud environment (`Cloud`) and with workload pressure (`Cloud + Noise`). Memory accesses that hit in the DevTLB (e.g., at `base`) consistently incur around 500 cycles, while DevTLB misses exhibit significantly higher latencies exceeding 1,000 cycles. As shown in Figure 4, the difference is large enough to allow a user-space program to distinguish between hits and misses by setting a threshold between 600 and 900 cycles.

As background noise may skew the DSA performance, we evaluate its behavior under deliberately noisy conditions by concurrently stressing both the PCIe and host memory. We generate PCIe pressure by rapidly reading and writing NVMe SSD with 2 GB/s traffic, and memory pressure with 10 GB/s core `memcpy` bandwidth. As shown in Figure 4, compared to the local (quiet) environment, the introduction of noise in both local and cloud VM contexts shifts the latency distribution, for instance, an average shift of 89 cycles in the cloud environment. Nevertheless, the threshold remains

```
1  enqcmd(wq_portal, &batch_desc);// (1)
2  _mm_sfence();
3  enqcmd(wq_portal, &desc); start = rdtsc();// (2)
4  while (desc_comp.status == 0);
5  latency = rdtsc() - start;
```

Listing 5. Microbenchmark for determining the adjudication strategy of the in-engine arbiter. Two positions (Line 1 and Line 3) are swapped for comparison.



(a) Local (b) Local + Noise

(c) Cloud (d) Cloud + Noise

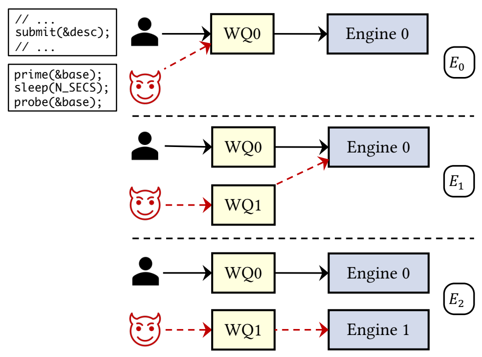
Fig. 4. Latency distribution between DevTLB hit and miss under four environments. We observe a high latency if `base + offset` is accessed (DevTLB Miss) and a low latency if `base` is accessed (DevTLB Hit).

clearly identifiable. This can be attributed to the latency boost provided by DevTLB. These findings indicate that DSA attacks without privileged access are indeed feasible, even amid system noise.

Indexing policy and absent PASID isolation. We design three experiments (E_0, E_1, and E_2), illustrated in Figure 5, each involving two processes running in different virtual machines (VMs): an attacker and a victim. The attacker first primes DevTLB entries with its own addresses and then keeps an idle state. Next, the victim process submits a descriptor intended to evict the attacker's DevTLB entry. Finally, the attacker probes the entries to detect eviction. The two processes are manually synchronized to ensure deterministic execution order. If DevTLB entries are not isolated by PASID, successful cross-VM eviction can be detected via increased access latency or reduced hit counts in `Perfmon` caused by DevTLB miss.

E_0 and E_1 assign both attacker and victim to the same DSA engine but differ in queue configuration: E_0 uses a shared work queue, while E_1 uses separate queues. In both cases, we observe a DevTLB miss in the attacker's probe phase, indicating that DevTLB entries are shared across PASIDs. The success of eviction in E_1 additionally confirms that DevTLB entries are not indexed by work queue identifiers.

E_2 assigns the attacker and victim to separate engines. In this case, the attacker's probe still registers a DevTLB hit, demonstrating that DevTLB entries are not shared across engines. This implies that DevTLB entries are indexed first by



```
// ...
submit(&desc);
// ...
```

```
prime(&base);
sleep(N_SECS);
probe(&base);
```

Fig. 5. Three experiments to distinguish the indexing policy of DevTLB and to identify the feasibility of a cross-VM attacker.

engine metadata before field-based lookup.

Takeaway 2. The DevTLB is indexed by engine ID and descriptor field type and is not isolated by PASIDs and virtual machines. As a result, processes from different VMs can share the same DevTLB entry.

The **batch fetcher** in the batch engine reads a group of descriptors and writes to a completion record. Using a similar method, we investigate whether DevTLB entries are indexed for memory requests issued by the batch fetcher. However, we find that neither the completion record write nor the descriptor reads by the batch fetcher are cached in the DevTLB, revealing that batch fetcher memory requests bypass DevTLB indexing.

Summary and security implications. Our experiments reveal that each DSA engine maintains five DevTLB entries. The DevTLB first identifies the engine by its ID, and then indexes entries based on the field types in the descriptor (e.g., `src`, `dst`). These entries are **not** partitioned by PASID and can therefore be evicted by processes or virtual machines belonging to different tenants. Despite the timing noise introduced by the PCIe bus, the latency difference between DevTLB hits and misses is clearly distinguishable from the CPU's perspective. Moreover, the per-engine organization of the DevTLB helps mitigate interference from other engines. Consequently, an unprivileged adversary can monitor DevTLB status at engine-level granularity using only `noop` descriptors.

C. Shared Work Queue Implementation

In this section, we investigate the undocumented behavior of Shared Work Queues (SWQs), which is essential for constructing contention-based side-channel attacks.

Latency Benchmark. Before generating congestion, we first measure how long a work descriptor occupies a DSA engine. We use the `memcpy` descriptor in this experiment, as it involves both memory read and write operations, thus maximizing execution latency. Alongside the execution cost (i.e., completion latency), we also measure submission latency. As illustrated in Figure 6, the completion latency increases linearly (on a logarithmic scale) with the transfer size, whereas submission latency remains constant at approximately 700 cycles. This observation enables fine-grained control over congestion windows during subsequent experiments.

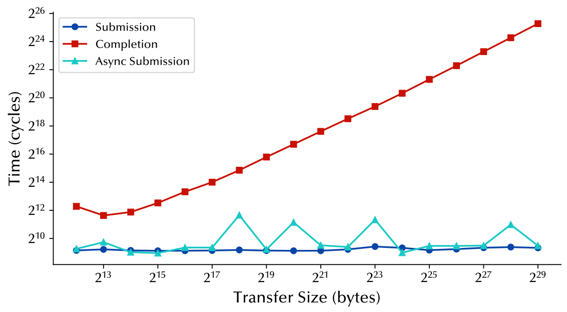


Fig. 6. Latency of submission and completion for a `memcpy` descriptor. Asynchronous submissions are not polled for completion.

QoS for Descriptors. As discussed in Section II-C, constructing congestion-based attacks requires submitting high-cost descriptors to saturate the queue. While batch descriptors are seemingly ideal for this purpose, they may be de-prioritized by the in-engine arbiter for Quality-of-Service (QoS) reasons. To clarify the arbiter's scheduling policy, we reverse-engineer its behavior.

We design a test scenario where a work descriptor and a batch descriptor are submitted concurrently to the same work queue, as illustrated in Listing 5. If swapping the submission order of the two results in significantly different latencies, the arbiter likely follows a FIFO policy. Otherwise, the arbiter exhibits prioritization based on descriptor type or arrival time.

Our experiments show that latencies remain nearly identical across all tested permutations, whether the work descriptor is submitted alone, concurrently with a batch descriptor, or the two are reordered. This indicates that **the arbiter consistently prioritizes the work descriptor, even if the batch descriptor arrives first**. Consequently, batch descriptors cannot be reliably used for congestion-based attacks, limiting their utility in our adversarial model.

Exploitable DMWr. Deferrable Memory Write (DMWr) serves as a handshake mechanism between the MMIO portal and the software submitter. A DMWr operation waits for acknowledgment from the DSA hardware. If the MMIO portal cannot enqueue a descriptor due to work queue saturation, the DSA communicates failure to the submitter by setting the `EFLAGS.ZF` flag [23]. This makes DMWr an ideal side-channel primitive, as it exposes internal queue state without requiring explicit timing measurements.

Since deferred MMIO may expose timing signals regarding submission status, we rerun the latency benchmark with asynchronous submissions, i.e., issuing descriptors sequentially without awaiting prior completions. To induce congestion, we minimize the interval between submissions (e.g. by reusing prior descriptors with minimal modification). During each submission, we detect contention via the `EFLAGS.ZF` flag. Contention is observed only when the transfer size reaches or exceeds 2^{25} bytes. However, as shown in Figure 6, submission latencies remain indistinguishable.

This result implies that submission latency is not feasible for constructing timing side-channel attacks and the `EFLAGS.ZF`

```
struct dsa_hw_desc desc_buf[] = {...};

void congest(int wq_size) {
    enqcmd(wq_portal, &mass_desc);
    for(int i = 1; i < wq_size - 1; i++)
        enqcmd(wq_portal, &desc_buf[i]);
}

int probe() { return enqcmd(wq_portal, &desc); }
```

Listing 6. Primitives of `congest` and `probe` in the attack on SWQ. The preparation of the descriptors is omitted.

flag remains the only viable channel for side-channel leakage. To demonstrate the exploitability of DMWr, we revisit the experiment E_0 in Figure 5, replacing the `prime` phase with `congest`, and removing completion polling from `probe`, as shown in Listing 6. During the attack phase, the attacker fills the SWQ with `wq_size`-1 descriptors and idles, awaiting the victim's submission. The SWQ size (`wq_size`) can be read without root privileges using `accel-config` [26]. Finally, the attacker probes the SWQ and reads the submission result via `enqcmd`. A non-zero value indicates contention when the victim has submitted a descriptor and the SWQ is full.

Takeaway 3. While the submission latency for all descriptors in SWQs remains constant, an attacker can infer whether the queue is full by checking the `EFLAGS.ZF` flag without requiring the root privilege.

V. Attack Primitives

In this section, we formalize the threat model and present the workflow of DSAssassin, which comprises two attack primitives: one exploiting the DevTLB (DSA_{DevTLB}) and the other targeting the shared work queue (DSA_{SWQ}).

A. Threat Model

We assume an unprivileged adversary who can execute arbitrary user-space code on a system running a recent Linux kernel (version `6.11.0-25-generic` in our case) with the Intel Data Accelerator (IDXD) driver enabled. The platform is powered by Intel 4th Generation or newer Xeon Scalable processors featuring the Data Streaming Accelerator (DSA). Since the system is configured with Intel VT-d in scalable mode, traditional IOTLB-based attacks are mitigated, and PASID-tagged isolation between processes and VMs is enforced. The attacker is not assumed to be capable of configuring engine affinity and work queue settings, since this access to driver requires root privilege [27]; the attacker only needs the `CAP_SYS_RAWIO` capability to submit descriptors to DSA as a normal user.

Both attack primitives are capable of performing cross-VM attacks. The attacker and victim reside in isolated virtual machines with no shared memory or direct communication channels. As DSAssassin does not require any CPU-side information, the attacker and the victim can run on different CPU cores. The victim uses a DSA instance to accelerate production workloads. The unprivileged attacker aims to bypass VT-d's PASID-based protections and extract sensitive information by exploiting microarchitectural side channels exposed by DSA.

In the DevTLB-based primitive (DSA_{DevTLB}), the attacker is assumed to have access to a work queue bound to the same DSA engine as the victim's. In the SWQ-based primitive (DSA_{SWQ}), the attacker and victim share the same SWQ but use distinct logical queue entries. Note that, setting DSA affinity to a specific engine or queue does not require root access, making both primitives feasible for unprivileged attackers. These assumptions are prevalent among DSA applications [24], [25], [42]

B. *Thrashing DevTLB (DSA_{DevTLB})*

This attack primitive targets the DevTLB and follows a classic Prime+Probe pattern involving three main phases: ❶ priming the DevTLB, ❷ idling to allow potential victim activity, and ❸ probing the evictions caused by the victim.

Preparation. The attacker first binds to a work queue that shares the same engine with the victim, as illustrated in configurations E_0 and E_1 in Figure 5. Based on our earlier analysis, the `noop` descriptor is ideal for this attack due to its low execution cost and distinguishable latency between DevTLB hits and misses.

Step 1: Priming. The attacker submits a `noop` descriptor that writes to a chosen completion record address, effectively loading this address into a DevTLB sub-entry. The attacker actively polls for completion to confirm that the entry is cached.

Step 2: Idling. The attacker idles for a time interval longer than the execution latency of a typical DSA operation. During this window, the victim (or sender, in the case of a covert channel) may execute a memory operation that evicts the attacker's DevTLB entry. This binary condition encodes the transmitted secret: if the victim submits a job, a DevTLB eviction occurs (bit 1); otherwise, the DevTLB state remains unchanged (bit 0).

Step 3: Probing. To detect eviction, the attacker submits another `noop` descriptor targeting the same completion record address. The attacker then records its completion time using `rdtsc`. Comparing the observed latency against the threshold reveals whether the DevTLB entry was evicted, thus enabling the attacker to infer the victim's activity.

C. *Congesting SWQ (DSA_{SWQ})*

This attack primitive targets the shared work queue (SWQ) by exploiting contention to leak activity from a co-resident process. The attack follows three major steps: ❶ congesting the SWQ with descriptors, ❷ idling for potential victim submission, and ❸ probing to detect whether the SWQ is full.

Preparation. In contrast to the DevTLB-based primitive, this attack requires the attacker to be mapped to the *same* SWQ as the victim process. In this paper, we use the `memcpy` descriptor to occupy the engine to make sure the congestion window is long enough for observation.

Step 1: Congesting. The attacker begins by submitting one large `memcpy` descriptor to anchor the head of the SWQ, followed by `wq_size` - 2 additional simple descriptors to fully occupy the queue. No polling or completion checks are required at this stage.

Step 2: Idling. The attacker idles for a fixed interval to allow the victim (or covert sender) an opportunity to submit a job to the shared queue. During this time, the attacker ensures that the head descriptor is still executing and that the SWQ remains nearly full—this is critical to maintaining contention. In other words, the idle period should be less than the execution time of the `memcpy` descriptor.

Step 3: Probing. The attacker attempts to submit a new descriptor using `enqcmd` and inspects the resulting `EFLAGS.ZF` flag. If `ZF` is set (i.e., `ZF` = 1), it indicates that the submission failed due to a full queue, implying that the victim has also issued a submission during the idle window. Otherwise, the victim does not submit any descriptors.

By iteratively performing these three steps, the attacker can infer activity on the shared queue and thereby establish a cross-VM side channel. As we will demonstrate in Section VI, both DevTLB and SWQ-based primitives can be leveraged to mount practical attacks on real-world applications.

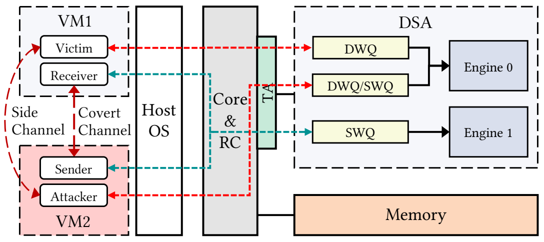


Fig. 7. Setup of cross-VM attacks on DSA. The red line represents DSA_{DevTLB} attack flow, and the teal line shows DSA_{SWQ} attack flow.

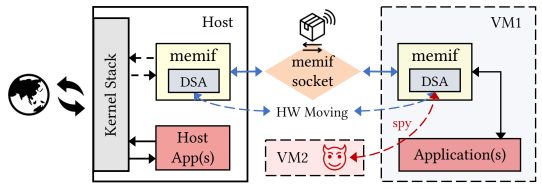
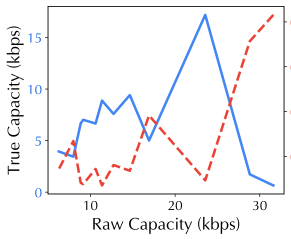


Fig. 8. VPP memif architecture with attack scenario.

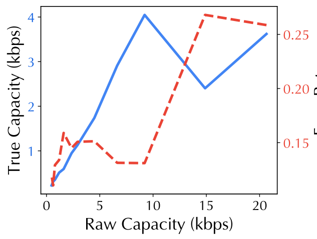
VI. Cross-VM Attacks

In this section, we demonstrate the effectiveness of our attack primitives through practical cross-VM attack scenarios.

Attack Setup. All attacks are conducted using the identical setups as outlined in Section IV. In local server and public cloud environments, DSA is virtualized using Single-Root IOV (SR-IOV), where descriptors submitted to the virtual WQ will be directly passed into physical WQ with near native performance [21], [23], [24]. Figure 7 illustrates the overall setup of our attack scenarios following the threat model defined in Section V. We consider two general configurations: (a) the attacker and the victim use different portals (i.e., work queues) that are bound to the same DSA engine (DSA_{DevTLB}), and (b)



(a) DevTLB



(b) SWQ

Fig. 9. Raw capacity with corresponding true capacity and error rate of the covert-channel built upon DSA_{DevTLB} and DSA_{SWQ} attack primitive.

the attacker and the victim share the same SWQ (DSA_{SWQ}). In the first scenario, the attacker's queue configuration is unrestricted. The corresponding attack flow is depicted by the red line in Figure 7. In the second scenario, multiple VMs share an SWQ through Scalable I/O Virtualization (IOV) [23], [59]; the attack flow is shown in teal.

Attack Scenarios. DSA is widely adopted in two application frameworks: the Data Plane Development Kit (DPDK) [42] and the DSA Transparent Offload (DTO) Library [25]. Since DPDK is not a standalone system, we deploy the Vector Packet Processor (VPP), a high-performance user-space network stack built on DPDK. Its architecture is shown in Figure 8. In VPP, DSA accelerates packet transfers within the shared memory interface (memif) [21], [24].

DTO, on the contrary, provides transparent DSA acceleration for unmodified applications. It relies on the Linux Runtime Linker to intercept and offload standard memory functions (e.g., `memset`, `memcpy`) to DSA. Thus, we consider any application that employs DPDK or DTO for acceleration as a potential victim of DSAssassin.

A. Constructing Covert-Channel Attacks

Covert channels are a widely adopted evaluation method in side-channel research, particularly for demonstrating the feasibility of cross-domain leakage [20], [36], [44], [51]. In our context, we establish a covert channel between two isolated virtual machines (VMs) that are unable to communicate through any legitimate interface. Leveraging either DSA_{DevTLB} or DSA_{SWQ}, we construct covert channels based on the attack setup illustrated in Figure 7.

Transmitting Secrets. We employ an asynchronous time-slicing protocol to enable message transmission. Similar to the previous cross-VM covert-channel [51], the initial synchronization occurs through the sender transmitting an initialization sequence via DevTLB or SWQ. More concretely, we transmit a preamble of several consecutive '1' bits to synchronize both processes. Once synchronization is established, the receiver first primes the DevTLB or congests the SWQ. After that, the sender encodes a bit '1' by evicting DevTLB entries (for DSA_{DevTLB}) or issuing a submission to a congested SWQ (for DSA_{SWQ}); bit '0' is encoded by remaining idle. The receiver finally probes the DevTLB's access latency or SWQ's congestion state to decode the data.

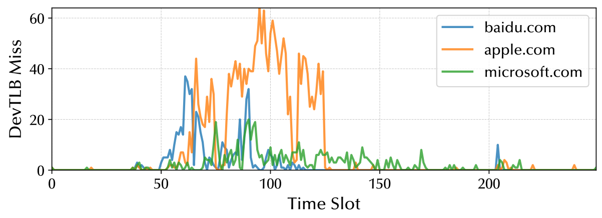


Fig. 10. DevTLB misses across 250 slots. Each slot contains 400 samples collected at 10 μs intervals.

Evaluation. Figure 9 presents the performance of our covert channels. The DSA_{DevTLB} channel achieves a peak true capacity of 17.19 Kbps with an average bit error rate of 4.63%. In contrast, the DSA_{SWQ} channel achieves 4.02 Kbps with an error rate of 13.11%. These results confirm that both primitives can support high-bandwidth covert communication across VMs without requiring privileged access.

B. Website Fingerprinting

We evaluate the feasibility of website fingerprinting in a realistic cloud-native scenario where the victim VM (VM1) uses a VPP `memif` interface as its sole communication channel, as illustrated in Figure 8.

Collecting Traces. The attacker aims to infer network activity by monitoring DSA usage during packet transfers over the `memif` interface. To do so, the attacker binds to a work queue that shares the same DSA engine with the victim and applies the DSA_{DevTLB} primitive to monitor DevTLB activity. The attacker samples the DevTLB state every 10 μs and aggregates every 400 samples into a single **time slot** (approximately 4 ms per slot). Within each slot, the attacker counts the number of DevTLB misses.

An increased number of DevTLB misses within a slot correlates with greater packet arrival activity. Over time, the temporal variation in miss patterns forms distinguishable behavioral traces corresponding to specific websites. Figure 10 presents DevTLB miss distributions across 250 time slots for three example websites. These traces exhibit unique characteristics, which can be leveraged for classification.

Classifying Websites. To classify these time-series traces, we adopt an Attention-based Bidirectional LSTM (BiLSTM) model, which is well-suited for sequential temporal data and widely used in prior website fingerprinting works [36], [51], [56], [65]. Our model architecture includes two BiLSTM layers, followed by an attention mechanism to prioritize informative time steps and a softmax output layer for classification. Dropout layers are incorporated between components to mitigate overfitting and improve generalization.

Evaluation. We evaluate our fingerprinting attack using headless Google Chrome (version 138.0) to access the top 100 websites from the Alexa Top Sites list [17]. For each website, we collect 200 traces and divide the dataset into 80% for training and 20% for testing.

Figure 11 presents the confusion matrix for a representative subset of 15 websites. Our classifier achieves an accuracy of

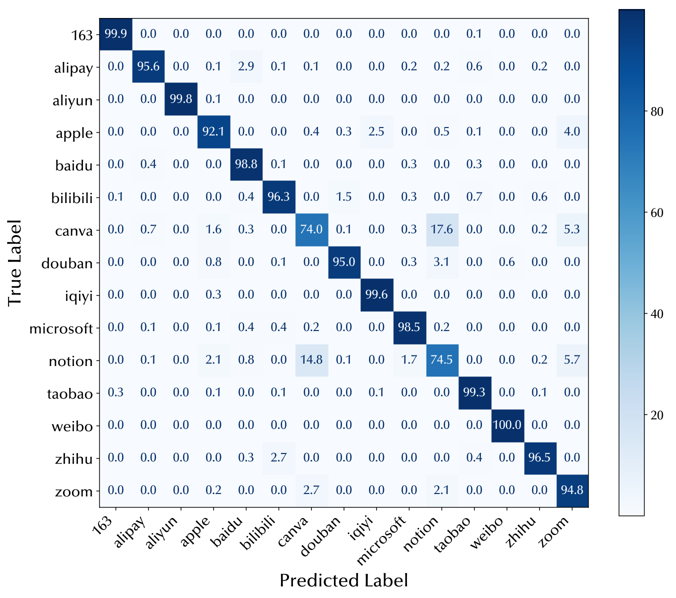


Fig. 11. Confusion matrix for 15 selected websites. Each cell denotes the predicted probability for the corresponding class.

96.5% on this subset. Each matrix cell indicates the model's predicted probability distribution over the target classes. On the full 100-website dataset, our model reaches a top-1 classification accuracy of 85.73%, significantly outperforming random guessing (1%). Relatively lower accuracy for certain domains (e.g., `canva.com`, `notion.com`) suggests overlapping traffic patterns (but still the first choice of our classifier), which we leave as future work to refine.

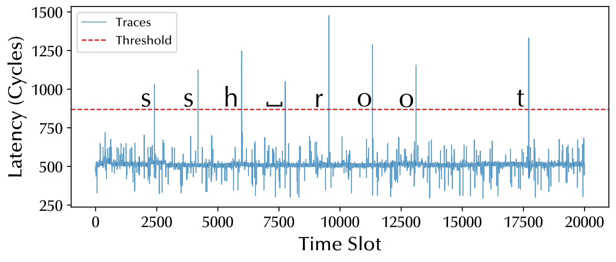
C. Keystroke Detection

This section demonstrates the SSH keystroke timing attacks using DSA_{DevTLB} and DSA_{SWQ}. Once the attacker gets the accurate keystroke timing information, it can infer the text by feeding the dataset to a machine learning model [40].

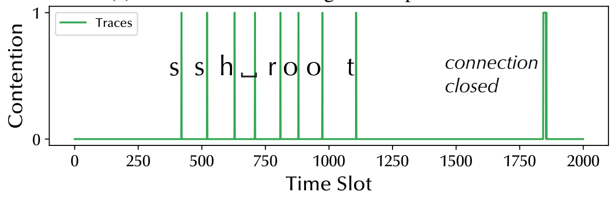
SSH Keystrokes. SSH keystroke timing has been a recurring target in side-channel research, typically using CPU caches [36], [40] or interrupt timing [51], [52]. The core observation is that the timing of inbound SSH packets correlates with keystroke events, allowing attackers to infer the timing—and potentially the content—of user input. While prior works [40], [55] focus on reconstructing the original keystrokes from packet patterns, our attack targets an earlier stage: inferring keystroke timings based on DSA activity induced by memory operations.

Specifically, when a user types a command over SSH, the OpenSSH client and daemon invoke `mem*` routines (e.g., `memcpy`) to handle packets. When DTO is enabled, these memory operations are transparently offloaded to DSA, producing measurable side effects in DevTLB or SWQ.

Tracking Keystrokes. To mount the attack, the adversary first identifies the SWQ or engine used by the victim. One approach is to initiate a temporary SSH connection while concurrently probing candidate SWQs from a separate process.



(a) DevTLB traces during SSH input events.



(b) SWQ traces during SSH input events.

Fig. 12. Traces of DSA activity (via DevTLB and SWQ) during SSH keystrokes. Filtering is applied based on probe thresholds and DTO byte thresholds.

The attacker then continuously monitors DSA activity via either the DSA_{DevTLB} or DSA_{SWQ} primitive. In the DevTLB case, the attacker performs a Prime+Probe routine on the completion record address; in the SWQ case, the attacker performs Congest+Probe using the `EFLAGS.ZF` side-channel.

Evaluation. We simulate the attack scenario using the setup shown in Figure 7, where the attacker VM monitors DSA activity without direct interaction with the victim. DTO acceleration is enabled on the victim VM.

Figure 12 illustrates filtered traces collected during the transmission of the command `ssh root`, using both attack variants. The filter removes the data from our traces if it is labeled as `DTO_MIN_BYTES` (for both attacks) or probed latency $\geq$ 2,000 cycles (for DevTLB attack).

We evaluate the precision and recall of our attack using the F_1 score and timestamp deviation as metrics. Across 512 ground truth keystrokes, the DevTLB variant captures 515 events with 15 false positives and 61 false negatives (i.e., we have 500 true positive keystroke detections), yielding a recall of 87.8% and precision of 97.6%. The SWQ variant records 511 events with 7 false positives and only 9 false negatives (i.e., 507 true positive keystroke detections), achieving 98.2% recall and 98.6% precision.

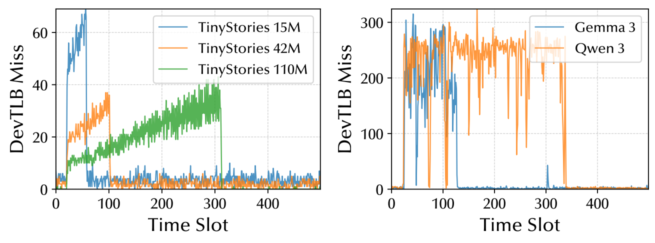
In terms of timing accuracy, the standard deviation of timestamp difference is 5.29 ms for DevTLB and only 1.21 ms for SWQ. These results outperform prior software-based attacks in both robustness and timing fidelity. We attribute this performance to our reverse-engineered understanding of DSA's microarchitecture, which allows us to precisely tune probing window sizes and eliminate external noise sources.

D. Classifying LLMs

With the widespread deployment of large language models (LLMs) in cloud-hosted AI services, inference workloads increasingly involve data-intensive operations, such as moving model weights between CPU and GPU or across memory hierarchies. Although these operations are transparent to users,

TABLE II
TESTED LLM MODELS.

Model	Parameters	Description
TinyStories	15M/42M/110M	Tiny model in LLaMA 2
Meta LLaMA 2	7B	Official LLaMA 2
Gemma 3	1B/4B	Model on single GPU
Qwen3	1.7B/4B	Dense and MoE model



(a) TinyStories 15M/42M/110M (b) Gemma 3 and Qwen 3 (4B)

Fig. 13. DevTLB miss traces of LLM inference workloads: (a) the same model with different parameter sizes and (b) different models with the same size. Each slot aggregates 800 samples (8 ms).

they trigger observable DSA activity when offloaded via DTO. This allows the attacker to leak the LLM fingerprinting across VM boundaries via DSASSASSIN.

Snooping and Classification. We adopt the same trace collection and classification methodology described in Section VI-B, with adjusted hyperparameters to accommodate the longer duration of LLM inference. Specifically, we configure the sampling **time slot** to 800 intervals (i.e., 8 ms per slot) to align with the timing of model weight transfers.

Different LLM architectures and backend configurations (e.g., CPU-only vs. CPU-GPU) exhibit distinct patterns of DSA usage. These patterns could be used to infer internal model structure, such as transformer layer depth or model size.

Evaluation. We evaluate our attack on 8 representative LLMs with diverse sizes and architectural characteristics, executed on two common inference runtimes: `llama.c` (CPU-only) [34] and `ollama` (CPU-GPU hybrid) [49]. A full list of models is provided in Table II. For each model, we collect 50 traces under identical prompts and split 80% of the dataset for training and 20% of the dataset for validation.

Figure 13 illustrates DevTLB miss traces collected during inference, revealing fine-grained DSA activity patterns that are distinguishable across models. Our classifier, based on the Attention-BiLSTM architecture, achieves a validation accuracy of 98.6% in predicting the correct model from a given trace. These results demonstrate that LLM architectures can be reliably fingerprinted based on DSA activities, despite PASID isolation and VM boundaries.

E. Noise Impact

To evaluate the noise impact on these attacks, we replicate the experiments on noisy environments mentioned in Section IV. Previous attacks, Covert Channel (CC), Website Fingerprinting (WF), SSH Keystrokes (SSHK) and LLM Classification (LLMC), are repeated 50 times to obtain confidence

TABLE III
ATTACK EVALUATION ON NOISY ENVIRONMENTS.

	Noisy Local	Cloud	Noisy Cloud	Local + CI
CC[1†]	16.81 / 4.73%	16.97 / 4.69%	16.52 / 5.13%	17.19 ± 0.78
CC[2†]	4.08 / 12.91%	3.96 / 13.3%	3.80 / 13.9%	4.02 ± 0.44
WF	86.0%	85.5%	85.1%	85.7 ± 2.8%
SSHK[1*]	90.5% / 5.41	93.4% / 5.39	93.0% / 5.35	5.29 ± 0.14
SSHK[2*]	98.2% / 1.25	99.1% / 1.27	98.9% / 1.28	1.21 ± 0.09
LLMC	98.6%	97.7%	98.0%	98.6 ± 1.0%

[1] DSA_{DevTLB}. [2] DSA_{SWQ}.
[†] format: true capacity (kbps) / bit error rate. CI is in terms of capacity.
[*] format: F_1 score / standard deviation (ms). CI is in terms of std.

intervals (CI). Table III summarizes the results of the attack experiments repeated under noisy conditions. CI h at a confidence level of 95% is expressed in terms of $\overline{x} \pm h$. The interval $[\overline{x} - h, \overline{x} + h]$ represents that 95% of the sampled measurements (e.g., attack success rate, covert-channel error rate, etc.) fall within this range. In this paper, the $\overline{x}$ denotes the geometric mean of the samples collected from the local server without noise (`Local`) (e.g., the true capability of the covert-channel attack). Across all evaluated attack targets (i.e., CC, WF, SSHK, and LLMC), the CI of `Local` is capable of containing 95% samples in all attack scenarios (i.e., `Local`, `Local + Noise`, `Cloud`, and `Cloud + Noise`), which indicates that system noise and PCIe noise negligibly impact the attack performance. Moreover, the observed h values are small, demonstrating DSASSASSIN's stability and high accuracy.

F. Comparison to Prior Cross-Core and Cross-VM Attacks

All evaluation results are summarized in Table IV, alongside representative cross-core and cross-VM side-channel attacks for comparison. Blank entries indicate experiments not conducted or reported by the corresponding work. All listed attacks operate under comparable settings and assumptions.

For capability, DSASSASSIN achieves up to 5× higher throughput and 4× lower bit error rate compared to the state-of-the-art IPI-based channel [51]. In website fingerprinting, DSASSASSIN ranks among the top two in classification accuracy (the top-performing method has since been mitigated by PASID isolation introduced in VT-d). For keystroke timing attacks, DSASSASSIN outperforms all prior approaches, attaining the highest F_1 score and a 5× reduction in timing deviation, demonstrating superior robustness.

Most notably, despite the PASID-based isolation enforced by Intel VT-d, DSASSASSIN successfully bypasses inter-VM boundaries, highlighting the effectiveness of DSASSASSIN.

VII. POTENTIAL MITIGATION

Hardware-Level Defenses. The fundamental vulnerability exploited in DSA_{DevTLB} stems from the lack of isolation in the DevTLB. We propose introducing DevTLB partitioning based on PASID, analogous to the existing protections applied to IOTLB. While this hardware change may incur performance and area overhead, it would provide strong isolation guarantees between tenants [64]. For DSA_{SWQ}, we suggest redesigning the

TABLE IV
COMPARISON TO RECENT CROSS-CORE AND CROSS-VM ATTACKS.

		Website Fingerprinting		**Keystroke Detection**		**Covert Channel**		**Mitigation**
Work	**Co-location**	Accuracy	F_1 Score	F_1 Score	Std. Deviation	True Capacity	Bit Error Rate	PASID
IPI [51]	CPU	N/A	80.4%	97.9%	6.15 ms	3.45 kbps	18.9%	N/A
DEVIOUS [36]	Device	98.9%[1]	N/A	N/A	10.08 ms	2.16 kbps	2.18%	✓
(M)WAIT [65]	CPU	78%	78%			697 bps	0%	N/A
This work (DevTLB)	Device	85.7%	N/A	92.0%	5.29 ms	17.19 kbps	4.63%	✗
This work (SWQ)	Device			98.4%	1.21 ms	4.02 kbps	13.11%	✗

[1] The trace collected in DEVIOUS includes GPU fingerprints.

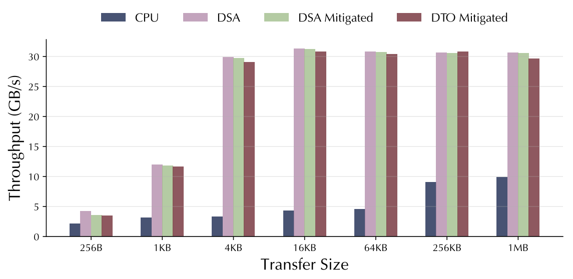


Fig. 14. Mitigation overhead in `dsa-perf-micros` / DTO.

DWMr completion register as a privileged register, preventing unprivileged eavesdropping of queue state.

Software-Level Defenses. Given the challenges of hardware redesign, software-based mitigation offers a more practical defense path [7]. Since both attack primitives rely on access to shared engines or shared work queues, tenant workloads handling sensitive data should avoid using DSA configurations that permit resource sharing. In practice, this can be achieved by configuring the `accel-config`. Moreover, if exclusive access cannot be guaranteed, tenants may disable DSA acceleration for critical paths to prevent this attack.

Since no existing mitigation against DSASSASSIN, we implement a software-based DevTLB mitigation to illustrate the associated overhead [14]. The *partitioning* algorithm periodically inserts random entries into DevTLB to mislead attackers. We measure the overhead for such mitigation using `dsa-perf-micros` [30] and DTO [25] test suites, in line with prior analysis of DSA [39]. Throughput of equivalent instructions on CPU and DSA without mitigation is referred as the baseline for comparison. We execute the test suite 1,000 iterations per configuration and use the mean value for analysis.

Figure 14 presents the throughput overhead compared with the baseline of unprotected functions. Performance degradation reaches up to 15.7% with native DSA acceleration and 17.9% with DTO, observed at the minimum tested transfer size (256 B).

VIII. RELATED WORKS

Data Streaming Accelerators. Kuper et al. [39] introduced the programming model and performance characterization of Data Streaming Accelerators (DSAs). Subsequent work has adopted DSAs for in-memory computation [6], page table migration [4], [45], and in-kernel memory deduplication [32]. However, these studies primarily emphasize performance benefits, with limited attention to their security implications.

Device-Driven Side Channels. Shared peripheral devices pose cross-VM attack surfaces. BUSted [54] demonstrated side-channel leakage via bus interconnects in MCUs with malicious peripherals. In datacenter environments, prior attacks have exploited GPUs [1], [8], [16], [33], [41], [63], [67], accelerators [60], and RDMA [40], [56], [57], [61]. To our knowledge, we are the first to exploit DSA as side-channel vector.

Attacks on (IO)TLBs. TLB-based side channels are commonly leveraged to amplify speculative execution attacks [20], [31], [38], [46], [53], [58]. The IOTLB has also been shown to leak sensitive information in device-driven attacks [36]. While recent VT-d extensions introduce PASID-based isolation to mitigate such threats, our attack demonstrates a method to bypass this isolation mechanism.

IX. CONCLUSION

This work exposes critical microarchitectural vulnerabilities in Intel's Data Streaming Accelerator (DSA), revealing that DevTLB and shared work queues (SWQ) are not isolated by PASID despite VT-d protections. We reverse-engineer these structures and develop DSASSASSIN, a set of unprivileged attack primitives exploiting DevTLB timing behavior and SWQ congestion feedback. DSASSASSIN enables cross-core and cross-VM side- and covert-channel attacks without privileged access. We demonstrate a covert channel achieving 17.19 Kbps across VMs, a website fingerprinting attack with 85.7% accuracy, the SSH keystroke timing attack with a 98.4% F_1 score and 1.21 ms deviation, and an LLM inference classification attack at 98.6% accuracy. These attacks underscore that accelerators represent a growing security blind spot in cloud systems. Our findings show the requirement of stronger isolation mechanisms and new hardware-software defenses tailored to heterogeneous architectures.

ACKNOWLEDGMENT

This work was generously supported by the National Key Research and Development Program of China under Grant (2024YFB4405400), Guangdong Municipal Science and Technology Project (Nos. 2024QN11X196), NSFC (No. U24A6009), BNRist. We would like to also thank the anonymous reviewers for their helpful feedback.

References

[1] J. Ahn, J. Kim, H. Kasan, L. Delshadtehrani, W. Song, A. Joshi, and J. Kim, "Network-on-Chip Microarchitecture-based Covert Channel in GPUs," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 565–577. [Online]. Available: https://doi.org/10.1145/3466752.3480093

[2] P. Aimoniotis, C. Sakalis, M. Själander, and S. Kaxiras, "Reorder Buffer Contention: A Forward Speculative Interference Attack for Speculation Invariant Instructions," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 162–165, 2021.

[3] Alibaba Cloud, "Elastic Compute Service: General-purpose instance families (g series)," https://www.alibabacloud.com/help/en/ecs/user-guide/general-purpose-instance-families#g8i, 2025. [Online]. Available: https://www.alibabacloud.com/help/en/ecs/user-guide/general-purpose-instance-families#g8i

[4] J. Baik, J. Kim, C. H. Park, and J. Ahn, "Accelerating Page Migrations in Operating Systems With Intel DSA," *IEEE Comput. Archit. Lett.*, vol. 24, no. 1, p. 37–40, Jan. 2025. [Online]. Available: https://doi.org/10.1109/LCA.2025.3530093

[5] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 971–988. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/barberis

[6] A. Berthold, C. Fürst, A. Obersteiner, L. Schmidt, D. Habich, W. Lehner, and H. Schirmeier, "Demystifying Intel Data Streaming Accelerator for In-Memory Data Processing," in *Proceedings of the 2nd Workshop on Disruptive Memory Systems*, ser. DIMES '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 9–16. [Online]. Available: https://doi.org/10.1145/3698783.3699383

[7] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 715–733. [Online]. Available: https://doi.org/10.1145/3460120.3484583

[8] B. Brezillon, "PanCSF: A new DRM driver for Mali CSF-based GPUs," https://www.collabora.com/news-and-blog/news-and-events/pancsf-a-new-drm-driver-for-mali-csf-based-gpus.html, Collabora, Feb. 2023.

[9] B. Chen, Y. Wang, P. Shome, C. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, "GoFetch: Breaking Constant-Time cryptographic implementations using data Memory-Dependent prefetchers," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1117–1134. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/chen-boru

[10] Y. Chen, A. Hajiabadi, and T. E. Carlson, "GADGETSPINNER: A New Transient Execution Primitive Using the Loop Stream Detector," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Edinburgh, United Kingdom: IEEE, 2024, pp. 15–30.

[11] Y. Chen, A. Hajiabadi, L. Pei, and T. E. Carlson, "PREFETCHX: Cross-Core Cache-Agnostic Prefetcher-based Side-Channel Attacks," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Edinburgh, United Kingdom: IEEE, 2024, pp. 395–408.

[12] Y. Chen, L. Pei, and T. E. Carlson, "AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 16–32. [Online]. Available: https://doi.org/10.1145/3575693.3575719

[13] D. Davoli, M. Avanzini, and T. Rezk, "On Kernel's Safety in the Spectre Era (And KASLR is Formally Dead)," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1091–1105. [Online]. Available: https://doi.org/10.1145/3658644.3670332

[14] S. Deng, W. Xiong, and J. Szefer, "Secure TLBs," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 346–359. [Online]. Available: https://doi.org/10.1145/3307650.3322238

[15] W. Diao, X. Liu, Z. Li, and K. Zhang, "No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA: IEEE, 2016, pp. 414–432.

[16] S. B. Dutta, H. Naghibijouybari, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Leaky Buddies: Cross-Component Covert Channels on Integrated CPU-GPU Systems," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. Virtual Event, Spain: IEEE, 2021, pp. 972–984.

[17] ExpiredDomains.net, "Alexa Top Websites - ExpiredDomains.net," https://www.expireddomains.net/alexa-top-websites/, 2025.

[18] S. Gast, J. Juffinger, M. Schwarzl, G. Saileshwar, A. Kogler, S. Franza, M. Köstl, and D. Gruss, "SQUIP: Exploiting the Scheduler Queue Contention Side Channel," in *2023 IEEE Symposium on Security and Privacy (SP)*. USA: IEEE, 2023, pp. 2256–2272.

[19] C. Gongye, Y. Luo, X. Xu, and Y. Fei, "Side-Channel-Assisted Reverse-Engineering of Encrypted DNN Hardware Accelerator IP and Attack Surface Exploration," in *2024 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2024, pp. 4678–4695.

[20] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: defeating cache side-channel protections with TLB attacks," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USA: USENIX Association, 2018, p. 955–972.

[21] Intel, "Intel Data Streaming Accelerator (DSA) – Offloading Memory Copying in VPP by Shared Memory Packet Interface (memif) Plugin with Intel DSA on 4th Gen Intel Xeon Scalable Processors," https://builders.intel.com/solutionslibrary/intel-data-streaming-accelerator-dsa-offloading-memory-copying-in-vpp-by-shared-memory-packet-interface-memif-plugin-with-intel-dsa-on-4th-gen-intel-xeon-scalable-processors, 2023.

[22] Intel, "Intel Virtualization Technology for Directed I/O Architecture Specification," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/774206/intel-virtualization-technology-for-directed-i-o-architecture-specification.html

[23] Intel, "Intel Data Streaming Accelerator Architecture Specification," 2024. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/671116/intel-data-streaming-accelerator-architecture-specification.html

[24] Intel, "Intel Data Streaming Accelerator (Intel DSA) - Calico VPP Multinet with Intel DSA on 4th Gen Intel Xeon Scalable Processor Technology Guide," https://builders.intel.com/solutionslibrary/intel-data-streaming-accelerator-intel-dsa-calico-vpp-multinet-with-intel-dsa-on-4th-gen-intel-xeon-scalable-processor-technology-guide, 2024.

[25] Intel, "Data Streaming Accelerator Transparent Offload (DTO)," https://github.com/intel/DTO, 2025.

[26] Intel, "idxd-config: Accel-config / libaccel-config," 2025. [Online]. Available: https://github.com/intel/idxd-config

[27] Intel, "idxd-config: Utility library for configuring intel dsa and iaa," 2025. [Online]. Available: https://github.com/intel/idxd-config

[28] Intel, "Intel Data Streaming Accelerator (Intel DSA)," https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/data-streaming-accelerator.html, 2025.

[29] Intel, "Intel Data Streaming Accelerator User Guide," https://www.intel.com/content/www/us/en/content-details/759709/intel-data-streaming-accelerator-user-guide.html, 2025.

[30] Intel, "Intel DSA Performance Micros." 2025. [Online]. Available: https://github.com/intel/dsa-perf-micros

[31] H. Jang, T. Kim, and Y. Shin, "SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 64–78. [Online]. Available: https://doi.org/10.1145/3658644.3690189

[32] H. Ji, M. Kim, S. Oh, D. Kim, and N. S. Kim, "Cooperative Memory Deduplication With Intel Data Streaming Accelerator," *IEEE Computer Architecture Letters*, vol. 24, no. 1, pp. 29–32, 2025.

[33] Z. Jin, C. Rocca, J. Kim, H. Kasan, M. Rhu, A. Bakhoda, T. M. Aamodt, and J. Kim, "Uncovering Real GPU NoC Characteristics: Implications on Interconnect Architecture," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Austin, TX, USA: IEEE, 2024, pp. 885–898.

[34] A. Karpathy, “llama2.c: Train and infer Llama 2 in pure C,” https://github.com/karpathy/llama2.c, 2023.

[35] J. Kim, J. Park, S. Roh, J. Chung, Y. Lee, T. Kim, and B. Lee, “TikTag: Breaking ARM’s Memory Tagging Extension with Speculative Execution,” 2024. [Online]. Available: https://arxiv.org/abs/2406.08719

[36] T. Kim, H. Park, S. Lee, S. Shin, J. Hur, and Y. Shin, “DevIOus: Device-Driven Side-Channel Attacks on the IOMMU,” in *2023 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2023, pp. 2288–2305. [Online]. Available: https://ieeexplore.ieee.org/document/10179283/

[37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: exploiting speculative execution,” *Commun. ACM*, vol. 63, no. 7, p. 93–101, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3399742

[38] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi, “TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs,” in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020, pp. 309–321.

[39] R. Kuper, I. Jeong, Y. Yuan, R. Wang, N. Ranganathan, N. Rao, J. Hu, S. Kumar, P. Lantz, and N. S. Kim, “A Quantitative Analysis and Guidelines of Data Streaming Accelerator in Modern Intel Xeon Scalable Processors,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. La Jolla CA USA: ACM, Apr. 2024, pp. 37–54. [Online]. Available: https://dl.acm.org/doi/10.1145/3620665.3640401

[40] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, “NetCAT: Practical Cache Attacks from the Network,” in *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2020, pp. 20–38.

[41] C. S. Lin, J. Qu, and G. Saileshwar, “Gpuhammer: Rowhammer attacks on gpu memories are practical,” in *Proceedings of the 34th USENIX Conference on Security Symposium*, ser. SEC ’25. USA: USENIX Association, 2025.

[42] Linux Foundation, “Data Plane Development Kit (DPDK),” 2025. [Online]. Available: http://www.dpdk.org

[43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading Kernel Memory from User Space,” in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[44] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *2015 IEEE symposium on security and privacy*. San Jose, CA, USA: IEEE, 2015, pp. 605–622.

[45] R. Liu, T. Ma, M. Zhang, J. Huang, Y. Shan, Z. Liu, L. Xiang, Z. Lin, H. Lu, , J. Rao, K. Chen, and Y. Wu, “DSA-2LM: A CPU-Free Tiered Memory Architecture with Intel DSA,” in *2025 USENIX Annual Technical Conference*. USA: USENIX Association, 2025.

[46] L. Maar, L. Giner, D. Gruss, and S. Mangard, “WHEN GOOD KERNEL DEFENSES GO BAD: Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks,” in *34rd USENIX Security Symposium: USENIX Security 2024*. USENIX Association, 2025.

[47] D. Moghimi, “Downfall: Exploiting Speculative Data Gathering,” in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 7179–7193.

[48] D. Mulnix, “Technical Overview of the 4th Gen Intel Xeon Scalable Processor Family,” https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html, Intel Corporation, Technical Report ID 766082, Jul. 2022.

[49] Ollama Dev Team, “ollama: A lightweight, extensible framework for running large language models locally,” https://github.com/ollama/ollama, 2025.

[50] PCI-SIG, “PCI Express Base Specification Revision 4.0, Version 1.0,” 2017. [Online]. Available: https://pcisig.com/specifications/pciexpress

[51] F. Rauscher and D. Gruss, “Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 94–108. [Online]. Available: https://doi.org/10.1145/3658644.3690242

[52] F. Rauscher, A. Kogler, J. Juffinger, and D. Gruss, “”IdleLeak: Exploiting Idle State Side Effects for Information Leakage”,” in *Network and Distributed System Security Symposium (NDSS) 2024*, Feb. 2024, network and Distributed System Security Symposium 2024 : NDSS 2024, NDSS 2024 ; Conference date: 26-02-2024 Through 01-03-2024. [Online]. Available: https://www.ndss-symposium.org/ndss2024/

[53] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, “PACMAN: Attacking ARM Pointer Authentication with Speculative Execution,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 685–698. [Online]. Available: https://doi.org/10.1145/3470496.3527429

[54] C. Rodrigues, D. Oliveira, and S. Pinto, “ BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect ,” in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2024, pp. 3679–3696. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00062

[55] D. X. Song, D. Wagner, and X. Tian, “Timing analysis of keystrokes and timing attacks on SSH,” in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM’01. USA: USENIX Association, 2001.

[56] M. Tan, J. Wan, Z. Zhou, and Z. Li, “Invisible Probe: Timing Attacks with PCIe Congestion Side-channel,” in *2021 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2021, pp. 322–338.

[57] M. Taram, A. Venkat, and D. Tullsen, “Packet Chasing: Spying on Network Packets over a Cache Side-Channel,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, 2020, pp. 721–734.

[58] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, “TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 989–1007. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/tatar

[59] The Linux Foundation, “SVA (Scalable Virtualization Architecture) on x86,” 2023. [Online]. Available: https://www.kernel.org/doc/html/v6.13/arch/x86/sva.html

[60] T. Tiemann, Z. Weissman, T. Eisenbarth, and B. Sunar, “IOTLB-SC: An Accelerator-Independent Leakage Source in Modern Cloud Systems,” in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 827–840. [Online]. Available: https://doi.org/10.1145/3579856.3582838

[61] S.-Y. Tsai, M. Payer, and Y. Zhang, “Pythia: Remote Oracles for the Masses,” in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019, pp. 693–710. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/tsai

[62] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “RIDL: Rogue In-Flight Data Load,” in *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2019, pp. 88–105.

[63] X. Wu, D. J. Tian, and C. H. Kim, “Building GPU TEEs using CPU Secure Enclaves with GEVisor,” in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 249–264. [Online]. Available: https://doi.org/10.1145/3620678.3624659

[64] L. Yin, H. Wang, Y. Lyu, C. Hu, and D. Wang, “VeriCache: Formally Verified Fine-Grained Partitioned Cache for Side-Channel-Secure Enclaves,” *IEEE Transactions on Dependable and Secure Computing*, pp. 1–12, 2025.

[65] R. Zhang, T. Kim, D. Weber, and M. Schwarz, “(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels,” in *USENIX Security*, 2023.

[66] X. Zhang, Z. Zhang, Q. Shen, W. Wang, Y. Gao, Z. Yang, and J. Zhang, “SegScope: Probing Fine-grained Interrupts via Architectural Footprints,” in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Edinburgh, United Kingdom: IEEE, 2024, pp. 424–438.

[67] Z. Zhang, K. Cai, Y. Guo, F. Yao, and X. Gao, “Invalidate+Compare: A Timer-Free GPU cache attack primitive,” in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 2101–2118. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-zhenkai

Appendix A
Artifact Evaluation

A. Abstract

DSAssassin is a hardware side-channel attack on the Intel Data Streaming Accelerator of the Intel Xeon 4th generation and above Scalable processor family. In this artifact, we provide the necessary information to reproduce the main results and claims presented in this paper. These experiments involve four real-world attacks.

B. Artifact check-list (meta-information)

- **Compilation:** GCC 13.3.0 or above and Python 3.12
- **Run-time environment: Ubuntu 24.04**
- **Hardware:** Intel Xeon Platinum 8468V & Alibaba Cloud ecs.ebmc8i.48xlarge
- **Execution:** Since website fingerprinting collets thousands of traces (each takes several seconds), it may require hours of completion. Meanwhile, the rest of attacks takes about thirty minutes.
- **Metrics:** Memory latency (for attacks, indicating DevTLB hit or miss) and `EFLAGS` register.
- **Output:** Terminal text outputs and visualized figures of the collected data.
- **Experiments:** Scripts and documents provided for detailed instruction and explanation.
- **How much disk space is required (approximately)?:** 2 GiB
- **How much time is needed to prepare workflow (approximately)?:** 10 mins
- **How much time is needed to complete experiments (approximately)?:** 5 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT License
- **Archived (provide DOI)?:** 10.5281/zenodo.17799273

C. Description

1) How to access: We leverage the lab server and Alibaba Cloud ECS ecs.ebmc8i.48xlarge instance to launch attacks. Thus, we recommend using them to reproduce this artifact.

2) Hardware dependencies: DSAssassin requires to be run on Intel 4th or 5th Xeon processors with DSA.

3) Software dependencies: `accel-config` library and Docker are necessary for setting up the environment.

D. Installation

Reviewer should be able to access (with privilege) a server with Intel DSA support, and check that the driver is installed and DSA is enabled properly. Then, download the artifact from https://zenodo.org/records/17799274 and run the setup script.

```
dsa-reversing $ sudo dmesg | grep "idxd "
dsa-reversing $ ./setup/install.sh
```

Refer to `README.md` and `ISSUES.md` for troubleshooting known issues.

E. Experiment workflow

Reviewers can replicate these attacks on servers with DSA (Alibaba Cloud ECS ecs.ebmc8i.48xlarge as our recommendation). Next, we will reproduce the covert channel attack (Section VI-A, website fingerprinting attack (Section VI-B), keystroke attack (Section VI-C and LLM fingerprinting (Section VI-D. All of these attacks can be launched independently. To process the traces collected by the attacks, we provide scripts to automatically analyze them.

F. Evaluation and expected results

In this section, we will reproduce four of the attacks mentioned in VI and evaluate them with visualization.

1) Covert Channel: After compiling with `make`, the user executes to launch covert channel and evaluate

```
attack $ ./run_cc.sh atc
evaluate $ python3 covert.py -a atc -f eval
```

Expected result will be output to the terminal, indicating the statistic of the launched covert channel. Additionally, the user can alter the parameters in `cc.h` and recompile to compare various configurations. We also provide script to plot the covert channel results showed in Figure 9.

2) Website Fingerprinting: To launch a website fingerprinting attack on DSA, the user must prepare a DSA-accelerated network stack which is VPP/DPDK as our setup. In `setup` directory, execute the following commands (note the directory).

```
setup $ sudo ./setup_dsa.sh config/dpdk.conf
setup $ ./run_vpp.sh
```

In another terminal session, the user executes the attacker process. This process will take 4 hours to finish. Complete dataset (top-100 websites) will cost approximately a day to collect.

```
attack $ ./run_wf.sh
evaluate $ python3 websites.py
evaluate $ python3 classify.py
```

After collecting the traces, the user can classify the them with provided script in `evaluate` directory. The first command will produce Figure 10 at `data/wf-traces.pdf`, and the latter one outputs confusion matrix (Figure 11) at `data/confusion-matrix.pdf` and classification accuracy.

3) SSH Keystroke: The user can launch the keystroke attack by executing the script. Next, the user evaluates the timestamps of keystrokes detected by the attacker.

```
attack $ ./run_keys.sh
evaluate $ python3 keystroke.py
```

The script will generate evaluation statistic based on the timestamps, such as standard deviation and F_1 score which is consistent with the data in Section VI-C

4) LLM Fingerprinting: To spy on DSA-accelerated LLM traces, the user executes the script at `attack` directory.

```
attack $ ./run_llm.sh
evaluate $ python3 models.py
evaluate $ python3 classify-ml.py
```

Evaluation of LLM traces can be done by running the plot script. This will depict the accuracy and Figure 13.