Compute eXpress Leak: New Cache Attacks Using CXL Devices

Ali Aqdas[†], Zheng Zhang[†], Yue Pan[‡], Zixuan Wang[‡], Tajana Rosing[‡], Jishen Zhao[‡], Kazem Taram[†]
[†]Purdue University
[‡]University of California San Diego
{aaqdas, zhan5394, kazem}@purdue.edu, {yup014, zxwang, tajana, jzhao}@ucsd.edu

***Abstract*—This paper presents the first systematic study of timing channels introduced by Compute eXpress Link (CXL) protocols and implementations. We show that CXL coherence operations expose new attack surfaces that allow untrusted devices and hosts to infer the memory access patterns across trust boundaries. We demonstrate that non-cacheable push writes can be exploited by a device to both inject cache lines into the host's last-level cache, and detect their residency through timing, enabling a new Prime+Probe-style attack from the device. This attack, dubbed PUSH+PUSH, does not require access to a host timer, and can bypass partitioning-based defenses that rely on Intel CAT. Furthermore, we analyze the security implications of cross-host memory sharing introduced in CXL 3.0. In particular, we study timing leakage arising from back-invalidation snoops and snoop-filter-based coherence tracking and simulate the vulnerability in CXLMemSim. We then present SnoopDog, a forward-looking cross-host timing attack that distinguishes whether a cache line is privately owned by another host based on access latency. We demonstrate end-to-end cross-host side-channel attacks, including recovery of tokens from large language model inference when model weights are shared across hosts on a CXL memory device.**

***Index Terms*—CXL, Security, Microarchitecture Security, Side-Channel**

I. Introduction

CXL is an open cache-coherent interconnect standard developed collaboratively by major hardware vendors and hyperscalers [1], [2]. It is built upon the PCIe physical layer and defines protocols that enable coherent communication between participating processors and heterogeneous devices such as machine learning accelerators, memory expanders, persistent memory, and storage. By extending devices into the CPU's coherence domain, CXL enables coherent memory operations through native load/store semantics and achieves significantly lower latency than PCIe transactions by eliminating software-managed synchronization mechanisms such as cache flushes and memory fences [1], [3], [4]. CXL is now being widely adopted by all major CPU and device vendors [5]–[13], as well as hyperscalers [14]–[16].

As CXL transitions from specification to deployment, there is a critical time window to evaluate the security implications of the CXL standards before its design choices become difficult to change once hardware and the surrounding ecosystem matures. Additionally, since CXL multiplexes cache-coherent resources across potentially untrusted components, it introduces shared resources analogous to shared CPU caches, which are well known to be highly susceptible to timing channels.

This paper presents the first systematic evaluation of timing channels introduced by CXL protocols and implementations. Across multiple threat models, we demonstrate how an untrusted attacker can exploit CXL capabilities to leak information across trust boundaries. In particular, we examine two major attack vectors: (1) how an untrusted CXL Type-2 device can infer a host's memory access patterns, and (2) how an untrusted host can infer the memory access patterns of other hosts sharing the same CXL memory (CXL 3.0).

CXL is a broad and complex standard comprising multiple protocols and device types, supporting a wide range of use cases and evolving across multiple backward-compatible versions, each introducing additional protocols and use cases. The first version of the standard (CXL 1.0) defines protocols that allow a device to expose cacheable memory to the host via `CXL.mem` and to cache host memory on the device via `CXL.cache`. These capabilities are realized through a set of new cache-coherence messages exchanged between the device and the host. In this paper, we first present a comprehensive study of the extent to which a CXL device can gain unauthorized visibility into host memory and the host memory access patterns through these capabilities.

We find that *non-cacheable push*, a cache-coherent memory operation, can be exploited by an untrusted device to reveal host memory access patterns, despite existing access-control mechanisms. This operation allows a device to place an arbitrary cache line into the host's last-level cache (LLC). Furthermore, the latency of the non-cacheable push itself reveals whether the cache line is still present in the LLC. Based on these observations, we present PUSH+PUSH, a new Prime+Probe–style timing attack launched from the device. Unlike conventional cache attacks, PUSH+PUSH does not rely on high-resolution timers or special instructions on the host, as the device can instead use its own internal timer. Additionally, PUSH+PUSH allows us to bypass defense mechanisms that rely on Intel CAT for partitioning, as we find that CAT does not properly isolate partitions from accesses issued by CXL devices.

Later versions of CXL introduce resource pooling and sharing across multiple hosts. To support cache coherence for shared memory across multiple hosts, CXL 3.0 introduces back-invalidation snoop requests [17], [18]. These back-

invalidation requests are issued from a device to the host to update cache states in the host as part of coherence management. In addition, CXL 3.0 devices can employ a snoop filter [17], [18] to track the coherence state across host caches and avoid broadcasting snoops to all hosts. This paper shows that these new capabilities introduce a new class of security risks by extending the reach of traditional cache attacks beyond a single host, enabling *cross-host cache attacks* for the first time.

Specifically, we present SnoopDog, an attack that exploits this back-invalidation snoop mechanism to infer memory accesses of another host sharing a memory region. SnoopDog exploits the timing difference between those reads that do not trigger a back invalidation snoop (not cached in another host) and those that are delayed by back invalidation snoop (cached in another host) to infer the memory access patterns across hosts.

Using SnoopDog, we demonstrate an end-to-end attack in a setting where multiple hosts share the weights of a large language model (LLM) on a CXL 3.0 memory device. When one host performs inference, we show that an attacker on another host can infer the generated output by observing memory access patterns.

As no CXL 3.0 devices are available at the time of writing, we demonstrate SnoopDog attacks using CXLMemSim [19], a state-of-the-art CXL simulator. On average, SnoopDog achieves 99.8% precision and 99.42% recall when monitoring Mistral7B (FP16) inference running on another host. These vulnerabilities lie in the CXL specification itself, which allows the host's memory access patterns to be leaked through timing. Therefore, we expect future hardware that closely implements this specification to be vulnerable to SnoopDog attacks.

In addition to the security analysis and the attacks, we also propose several defense strategies. In particular, we propose and evaluate a protocol-level defense that mitigates the PUSH+PUSH attack by ensuring that the latency of the non-cacheable push operations is independent of the LLC state of the target address. Our experimental results on a CXL-enabled Intel FPGA show that this defense incurs a worst-case overhead of only 3.13% in data transfer throughput for non-cacheable push operations.

In summary, this paper makes the following contributions:

- We present a comprehensive analysis of timing channels arising from `CXL.cache` and `CXL.mem` operations (§IV, §VI).
- We introduce PUSH+PUSH, a novel cache attack launched from CXL Type-2 devices using `CXL.cache` operations (§V).
- We evaluate PUSH+PUSH through end-to-end side-channel attacks that break existing partitioning-based mitigations (§V-E).
- We introduce SnoopDog, a novel *cross-host* cache attack that exploits delays induced by back invalidation snoops (§VII).
- We demonstrate an end-to-end cross-host attack that infers tokens from LLM inference by observing memory access patterns from another host (§VII-D).
- We propose protocol and hardware level mitigations against PUSH+PUSH and SnoopDog (§VIII).

Responsible Disclosure. We disclosed the discovered vulnerabilities to Intel on February 6, 2026 and to AMD on May 9, 2026.

II. Background and Related Work

A. Compute Express Link (CXL)

Protocols. CXL defines a family of protocols— `CXL.io`, `CXL.cache`, and `CXL.mem`—for seamless cache-coherent memory access between processors and various devices: `CXL.io` handles device discovery, configuration, and is functionally equivalent to PCIe. `CXL.cache` exposes CPU memory as cacheable memory to the CXL-attached devices such as accelerators. `CXL.mem` enables the host CPU to coherently access CXL-attached device memory as part of its unified address space.

Versions. The CXL standard has undergone multiple revisions and each new version is backward compatible with earlier versions. CXL 1.0 defines the `CXL.io` and `CXL.cache` protocols, which allow CXL-attached devices to cache data from CPU memory. Its later revision 1.1 defines the `CXL.mem` protocol enabling CPU to coherently access device memory attached to the CXL bus. CXL 2.0 removes the tight coupling of resources with individual hosts. It defines resource pooling that allows multiple server systems to coherently access a remote pool of memory, which improves the overall device utilization. CXL 3.0 extends the resource pooling to allow cache-coherent sharing of device memory across multiple hosts for message passing interfaces. This revision also introduces direct peer-to-peer memory access where a CXL/PCIe device can directly access memory in another device, bypassing the host CPU. It also supports tree-based fabric topology and enables up to 4096 devices within a CXL system.

Device Types. The CXL standard designates three types of devices capturing various usage models. Type 1 devices (caching devices) are accelerators that do not expose host-addressable memory but require coherent access to the host's memory space, such as SmartNICs. They implement `CXL.io` and `CXL.cache`. Type 2 devices (accelerators with memory) are accelerators that expose their device memory to the host's coherence domain to enable bi-directional cache-coherent memory transactions, such as domain-specific accelerators. They implement `CXL.io`, `CXL.mem`, and `CXL.cache`. Type 3 devices are memory expanders that provide memory capacity or bandwidth expansion to the host, such as hardware memory expanders and pools. They implement `CXL.io` and `CXL.mem`.

Bias Modes. CXL uses bias modes to determine whether the host or device manages coherence for CXL-attached device memory. In host-biased mode, the host manages coherence in hardware and the device must ensure coherence resolution from the host before accessing its local memory. In device-biased mode, the device accesses its local memory directly

TABLE I
SYSTEM CONFIGURATION

Component	Description
Linux Kernel	6.11.5cxleak (v6.11.5 with CXL support added via kconfigs)
Host CPU	Dual-Socket Intel Xeon Silver 4416+ (Sapphire Rapids) @ 2.2 GHz, 20 cores, 37.5 MiB LLC per socket, with SMT/prefetchers disabled.
Host RAM	256 GB DDR5 4K MT/s 1-Bit ECC DRAM (4x32 GB per socket).
CXL Device	Intel Agilex 7 I-Series FPGA (2×R-Tile, 1×F-Tile, CXL Type-2), and 16 GB onboard DDR DRAM [8]

without coherence resolution from the host. So, the host operating system must ensure that any host-cached copies are written back or invalidated before the device accesses them.

Device-Coherent Memory with Back-Invalidate Snoop. To enable cache coherence without routing all coherence traffic through the CPU, CXL 3.0 introduces memory that is device-coherent using the back-invalidate snoops (HDM-DB). This memory relies on an inclusive snoop filter to track host cache state of device memory and a back-invalidate mechanism that allows the device to directly trigger coherence actions.

Existing CXL Security Mechanisms. To provide isolation and security of the physical bus, the specifications provide protocol support for integrity and data encryption (IDE). IDE mitigates impersonation and man-in-the-middle attacks, but it does not prevent timing attacks.

CXL Trust Modes. To provide coarse-grained access control, a CXL-enabled processor supports a configuration known as *trust mode* to prevent an untrusted device from accessing the host memory. In our system, this configuration is applied globally to all attached CXL devices with three possible modes:

- **Fully Trusted Mode.** The device can access both host and device memory.
- **Partially Trusted Mode.** The device can only access its own memory.
- **Untrusted Mode.** The device can only access device memory without host-managed cache coherence.

B. Cache Attacks

The PUSH+PUSH and SnoopDog attacks we describe in this paper are novel types of cache attacks that are enabled by CXL technology. Cache attacks are a well-studied class of side-channel attacks that can infer the memory access patterns by monitoring cache state from access timing [20]–[29] and microarchitectural events [30] to leak information about sensitive applications. Such attacks can be performed on peripheral devices [31] and the network boundaries [32]–[34]. Our PUSH+PUSH and SnoopDog provide new capabilities that are not possible in previous attacks. For instance, PUSH+PUSH allows us to bypass mitigations such as CAT-based partitioning while SnoopDog leaks information across servers. Table II provides a comparison with related work.

TABLE II
SYSTEMATIC COMPARISON OF ATTACK VECTORS AND REQUIREMENTS OF OUR ATTACKS VS RELATED WORK.

Attack Name	Scope	Assumptions/Requirements			
		Special Hardware	No Shared Memory	No Timer	No SMT
Device Driven and Off-chip Memory Attacks					
NVLeak [31]	Cross-Core	NVRAM	✓	×	✓
NetCAT [33]	Cross-Host	RDMA	✓	✓	✓
Packet Chasing [32]	Cross-Host	NIC	✓	✓	✓
DevIOus [34]	Cross-VM/Host	GPU/RDMA	✓	✓	✓
Directory Attacks					
Directory Prime+Probe [23]	Cross-Core	–	✓	×	✓
Coherence and Invalidation Attacks					
S2C [35]	Cross-Core	–	✓	✓	✓
PRIME+ABORT-L3 [30]	Cross-Core	–	✓	✓	✓
Coherence Channels [36]	Cross-Socket	–	×	×	✓
I2SC [37]	Cross-Socket	–	×	×	✓
Invalidate+Transfer [38]	Cross-Socket	–	×	×	✓
PUSH+PUSH (this work)					
Directory Eviction	Cross-Core	CXL T2	✓	✓	✓
Mirroring	Cross-Core	CXL T2	×	✓	✓
Line Demotion	Same-Core	CXL T2	×	✓	×
SnoopDog (this work)					
Downgrade	Cross-Host	CXL T3	×	×	✓
Invalidate	Cross-Host	CXL T3	×	×	✓

Cache Line Mirroring. Our PUSH+PUSH attack can leak the memory access pattern of a target cache line in the last-level cache only. To extend the scope of the attack to cache lines in private caches, we need a method to demote cache lines from private caches to the LLC. Prior work highlights that cache lines in the Exclusive (E) or Modified state (M) in private caches are tracked by directory and cache lines in the Shared state (S) in private caches are inclusive in the LLC [29]. This coherence behavior allows us to deliberately move cache lines from private caches to the LLC. We refer to this process as *cache line mirroring*, where a core accesses a line held privately in another core in the Exclusive state mirrors it to the LLC.

Intel Cache Allocation Technology (CAT). Intel Cache Allocation Technology (CAT) enables software-programmable control over LLC occupancy to provide performance isolation for critical applications (e.g. latency-sensitive interactive applications). It achieves this with classes of service (COS), which associate CPU cores and processes with a subset of LLC ways [39]. While it is designed for performance isolation, several works use this for mitigating cache contention side-channels [22], [23], [26], [40], [41].

III. THREAT MODEL

Based on our analysis of the CXL specification, we identify several vulnerabilities under two threat models. The first threat model targets attacks from an untrusted CXL Type-2 device that can leak memory access patterns from the LLC of the host CPU. The second threat model targets cross-host attacks, where an untrusted host sharing a CXL memory device with other hosts aims to infer the cache states of those hosts. We discuss each threat model in detail below.

A. Attacks from Devices (PUSH+PUSH)

First, we consider vulnerabilities that allow an attacker with access to a CXL Type-2 device to infer the cache state or memory access patterns of the processes running on the attached host. The attacker may either directly configure and program the device to issue attacker-controlled coherence requests, or do so indirectly by invoking a functionality from the device. For example, an attacker can submit rendering requests to a CXL-enabled GPU that trigger specific coherence operations, or cause a CXL-attached network device to inject packets into the host's LLC via non-cacheable push operations [42], [43]

Such capabilities may be available without superuser privileges (e.g., issuing GPU requests from a browser [44]).

We assume the device operates in a partially trusted mode, where it can issue reads and writes to device memory but cannot directly access host memory. The attacker requires access to a timer, but unlike traditional cache attacks, our timer need not be a high-precision CPU timer (e.g., `rdtsc`); any on-device or device-accessible timer is adequate. We assume the attacker can construct eviction sets using well-established techniques from prior work [21], [45]–[48] to identify addresses that map to the same LLC sets as the victim.

Finally, to extend leakage beyond the LLC to private caches, we consider three variants with slightly different attacker capabilities.

- **Cache Line Demotion** uses `CLDEMOTE` and requires the attacker to share a core with the victim.
- **Directory Eviction** targets the cache directory which is local to a socket, and thus requires both attacker and victim to reside on the same socket.
- **Cache Line Mirroring** leverages mirroring [29] and does not require co-location on the same socket but requires memory sharing with the victim.

While memory sharing and co-residence are strong constraints, existing attack models rely on them [20], [24]–[26], [49], [50]. For memory sharing, read-only shared libraries and kernel deduplication across virtual machines may be leveraged, whereas co-residence can be enforced through scheduling on processors that support simultaneous multithreading.

B. Cross-Host Attacks (SnoopDog)

Next, we consider vulnerabilities that allow an untrusted host to infer the cache state of other hosts connected to the same CXL memory. We assume that shared memory is enabled via the back-invalidation mechanism introduced in CXL 3.0, where the device maintains coherence among host caches using back invalidation and employs a snoop filter to avoid broadcasting all accesses. The attacker shares CXL-backed memory regions with the victim. The attacker's goal is to infer the cache state of other hosts for these shared regions. We assume that the attacker can perform load operations to the shared memory region from one of the CPUs of the host. The attacker does not need any super-user privileges, but it requires access to timers (e.g., `rdtsc`). Additionally, the attacker must know the set and slice mapping of the device's snoop filter (or be able to reverse-engineer them). Existing reverse engineering and eviction-set methodologies [23], [45], [46] can be applied for this purpose. Finally, the attacker must be able to cache enough entries to fully occupy a snoop filter set.

IV. Timing Analysis of CXL.cache Operations

To determine whether CXL devices introduce microarchitectural side channels, we examine how device-initiated memory accesses interact with the host cache hierarchy. We systematically characterize the timing behavior of cache-coherent memory operations and use these observations to identify leakage channels that expose cache states.

In our study, we uncover previously unknown timing behavior of different operations and discuss their security implications. We analyze operations that vary along five dimensions:

- **Bias mode**: host-biased vs. device-biased coherence, which determines where coherence is managed.
- **Memory type**: host memory vs. device-attached memory.
- **Trust configuration**: trusted vs. partially trusted vs. untrusted device execution, which constrains accessible memory.
- **Operation**: read vs. write accesses and their different coherence interactions.
- **Cacheability**: cacheable (owned/shared) vs. non-cacheable accesses, which determines cache allocation and coherence participation.

A. A Framework for CXL Timing Analysis

To study the timing behavior and cache effects of cache-coherent memory accesses from a device, we build an experimental framework that allows us to accurately measure the timing of different operations.

We use a CXL-enabled Intel Agilex 7 I-Series FPGA and Intel Xeon Sapphire Rapids CPUs with CXL support. The FPGA features a CXL Type-2 hard IP provided by Intel, as well as memory controllers for interfacing with on-board DDR memory. Table I presents the details of our testbed. Additionally, we develop a telemetry subsystem in Verilog RTL that allows us to analyze the latency of different operations. Figure 1 illustrates the overall experimental framework along with the microarchitecture of the telemetry subsystem. The subsystem features two modules: (1) Custom Control and Status Registers (CSRs), which provide an interface for the host to configure each experiment, and (2) the Memory Access and Timing Module, which issues memory operations and records their timings.

We use this telemetry subsystem to perform a series of security-focused timing analyses described in §IV-B. For the results reported in our analysis, we repeat the experiments at least 100 times and report the median.

This FPGA-based design enables configurable and precise measurement of timings of operations. However, the attacker, as discussed in the threat model (§III), does not rely on an FPGA device; it only needs a device capable of initiating and timing cache-coherent memory accesses, which can be achieved using even a simple microcontroller..

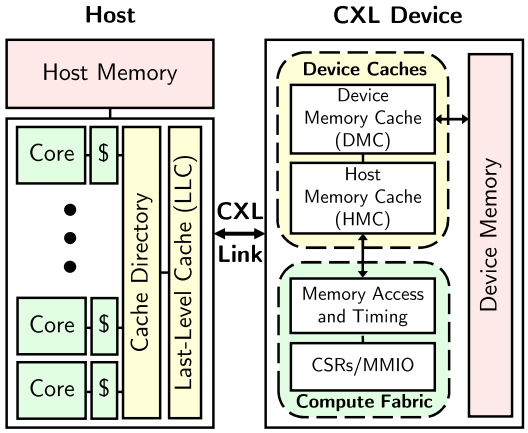


Fig. 1. System-level architecture of our framework for CXL timing analysis.

Cache-Coherent Operations Supported in Intel Agilex 7. We present a list of cache coherent memory operations supported in the framework. We classify these operations based on their cache allocation and coherence behavior. Non-cacheable (NC) reads and writes bypass the caches on the device. They do not allocate cache lines in the device caches and serve requests directly from memory. Cacheable owned (CO) reads and writes allocate the line in the device caches with ownership and writes modify the line in the device cache. Cacheable shared (CS) reads allocate the line in the Shared state and may be used for regular loads. Non-cacheable push writes (NCP) do not allocate the line in device caches. Instead, they push data directly into the host cache hierarchy. This operation may allocate the line in the host's last-level cache (LLC).

B. Security-Focused Timing Analysis

A modern CPU (as shown in Figure 1) features multiple levels of caches and a cache directory. Our goal is to analyze how device operations interact with host caches and directory and leverage these interactions to construct cache attacks. Such cache attacks typically require two capabilities: placing the cache into a known state and observing subsequent changes to that state. Therefore, we focus on the following properties:

- **Host cache state observability:** whether the device can infer the coherence state of cache lines in the LLC.
- **Host cache state modification:** whether the device can alter cache line states or invalidate them in the LLC.
- **Host directory state observability:** whether the device can infer directory metadata maintained by the host.
- **Host directory state modification:** whether the device can cause an eviction to entries in the host directory.

Next, we describe experiments we design to reveal these properties:

Property 1: Host Cache State Observability. To reveal the behavior of each memory operation and configuration, we create an experiment that identifies measurable timing differences in memory accesses from the device, if a target cache line is resident in the LLC. We first place the target cache line in the LLC by performing a `load` followed by a `CLDEMOTE` instruction from one of the CPU cores. Next, the device accesses the same memory location with each configuration in the exploration space, and measures the latency in device cycles. We compare this with a target cache line that is not in the LLC.

Our experiments reveal that observability depends on both memory type and access semantics (operation, bias mode, and cacheability). When the target location resides in a host-biased device memory, only non-cacheable push operations exhibit a measurable timing difference. In contrast, when the target location resides in the host memory, all device-initiated memory accesses exhibit observable timing differences. Figure 2 summarizes these results across all configurations.

Observation 1: The device can observe the LLC state of (1) host memory using any type of operation and (2) device memory in host-biased mode using non-cacheable push (NCP) operations.

Note that operations to host-memory regions are only permitted in fully-trusted mode, which is less security-critical, as the device can directly read memory content in this mode. In contrast, operations to device-memory regions are permitted in partially trusted mode. This makes the NCP operations particularly interesting, as they allow the device to observe the LLC state through accesses to device memory, even in more restricted partially trusted mode. Below, we discuss the use cases of NCP and the protocol-level transactions underlying this behavior.

Non-Cacheable Push (NCP) Transactions. NCP requests allow the device to write cache lines directly to the host LLC. This is a useful primitive for a variety of applications. For example, a network adapter can optimize memory accesses by placing incoming packets directly into the LLC for processing, rather than writing them to memory [43], an approach also employed in Intel Data Directed IO (DDIO) [51].

An important question is why the timing of an NCP is dependent on whether the target cache line is present in the LLC. To answer this, we study protocol-level transactions needed for an NCP.

As illustrated in Figure 3, NCP uses the `ItoMWr` (Invalid To Modified Write) message in CXL.cache protocol to atomically request Exclusive ownership of the cache line and write back to the LLC. When the host receives this message from the device, it sends a `GO+WritePull` message. The global observation (`GO`) message signals to the device that the state is coherent across the system and the device has Exclusive ownership. The device sends data to the host after receiving `WritePull`. The request is considered complete by the device after it has received `GO`. The host ensures coherence by invalidating the line from the cache hierarchy before sending `GO`. We hypothesize that this invalidation request resolves faster when the host finds the cache line in the Exclusive state in the LLC, as it can be resolved locally without consulting remote caches.

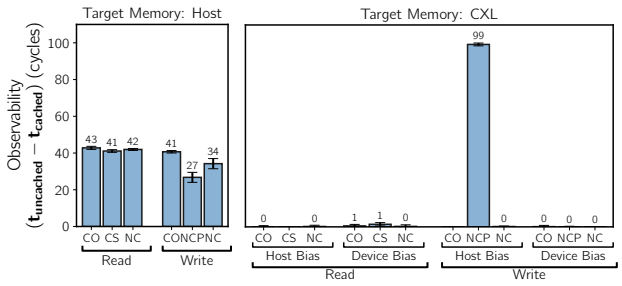


Fig. 2. Latency difference of cache-coherent memory accesses initiated by the device to host memory (left) and device memory (right), comparing cases where the target cache line is present vs. absent. For device memory, only non-cacheable push (NCP) accesses expose LLC residency.

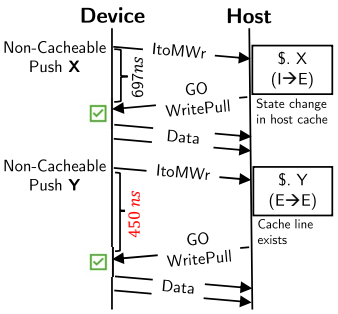


Fig. 3. CXL.cache transaction for NCP. NCPs targeting cached lines exhibit lower completion latency compared to uncached locations.

Property 2: Host Cache State Modification. Next, we study the device's ability to invalidate a cache line in the host caches, a primitive needed for many classical cache attacks such as Flush+Reload [20]. We first construct an eviction set for a target LLC set, i.e., a set of cache lines that all map to the same LLC set and slice. We then choose a target cache line from the eviction set and bring it to the LLC. The device iterates over the eviction set and performs a particular type of memory operation on all cache lines of the set. After completion of device accesses, the host reloads the target cache line and measures the access latency. High latency shows that the target line was evicted from the LLC as a result of the device operations. Our results, shown in Figure 4, reveal that only cacheable owned reads and writes to the host memory addresses can invalidate lines in the LLC. Other combinations such as accesses to device memory or non-cacheable operations do not cause invalidation.

Observation 2: Cacheable owned operations to the host memory can evict CPU lines from the LLC.

Property 3: Directory State Observability. The cache directory is a shared coherence structure that tracks the state of cache lines in the private caches. We hypothesize that lines in the device caches (i.e., DMC and HMC in Figure 1) are also tracked by the host cache directory and thus evictions of their corresponding lines in the host directory cause evictions

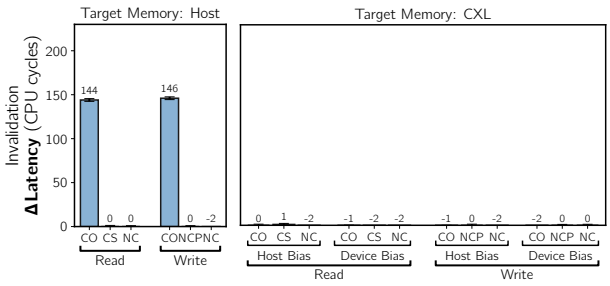


Fig. 4. Change in CPU-measured access latency of a cache line placed in the LLC when a CXL device performs memory accesses to host DRAM (left) and to device memory (right). Latency is reported relative to an idle-device baseline.

from the device caches. Therefore, we investigate whether a device can exploit this contention and infer CPU accesses to the directory by launching a Prime+Probe style attack.

To study the feasibility of this type of leakage, we create two eviction sets that map to the same set in the directory. The device accesses cache lines in the first eviction set to initialize the directory into a known state (Prime). For this experiment we use cacheable owned reads and writes to host memory. Next, the CPU cores access the second eviction set. To ensure we can exhaust the entire directory set we load the second eviction set from multiple CPU cores. Finally, the device reloads the cache lines from the first eviction set (in reverse order) and measures their timings to see if cache lines previously in the device caches are evicted (Probe).

The results of this experiment show that cache lines in the HMC are tracked by the cache directory. Invalidation of a directory entry corresponding to a cache line in the HMC results in eviction from device caches. This allows a device to spy on memory accesses by a private CPU core [21].

Observation 3: The directory tracks both device cache lines (HMC) and private CPU lines, enabling a device to infer directory state by creating contention.

Property 4: Directory State Modification. The previous property established that the directory tracks both host and device caches. Next, we investigate if the device can cause eviction to the directory lines belonging to the host. To that end, we investigate whether cacheable owned reads and writes from the device can invalidate entries in the directory. We design an experiment that proceeds in three steps: First, the CPU brings the directory to a known state by loading an eviction set from multiple cores. Next, the device accesses the second eviction set. We hypothesize that the device access attempts to replace directory entries created in the first step. Finally, the CPU reloads the first eviction set to measure access latency. A higher access latency for any line indicates that the corresponding directory entry was invalidated and in turn it caused an eviction from the private cache.

The results of this experiment show that cache lines in the HMC can create contention in the directory and evict cache

TABLE III
MEMORY OPERATIONS AND VULNERABILITY ANALYSIS FOR INTEL AGILEX 7 I-SERIES CXL FPGA (TYPE-2 IP) IN VULNERABLE TRUST MODES

Access Type	Cache Hint	LLC Obs.	LLC Mod.	Dir. Obs.	Dir. Mod.	Vulnerable Trust Mode
Reads	Cacheable Owned (CO)	✓	✓	✓	–	Fully Trusted
	Cacheable Shared (CS)	✓	–	–	–	Fully Trusted
	Non-Cacheable (NC)	✓	–	–	–	Fully Trusted
Writes	Cacheable Owned (CO)	✓	✓	✓	✓	Fully Trusted
	Non-Cacheable (NC)	✓	–	–	–	Fully Trusted
	Non-Cacheable-Push (NCP)	✓	–	–	–	**Partially Trusted**

lines from private caches. This allows spying on the activity in the private caches using Evict+Time-style attacks.

Observation 4: The device can create contention in the directory, which allows the device to effectively set the directory into a known state.

C. Security Vulnerabilities

Table III presents a summary of memory accesses from the device and the observed properties. The last column shows that only non-cacheable push write is capable of modifying and observing the states in partially trusted mode, while all other access types are only vulnerable in fully trusted mode.

V. PUSH+PUSH: ATTACK FROM CXL TYPE-2 DEVICE

In this section, we present PUSH+PUSH, a novel cache attack that leverages cache-coherent memory requests issued by a CXL Type-2 device.

A. Attack Primitives

A Prime+Probe-style attack from a CXL Type-2 device requires two key primitives: (1) the ability to fill a cache set with attacker-controlled lines, and (2) the ability to detect whether those lines have been evicted by victim accesses.

Probing the LLC. Through the analysis presented in the previous section, we identify non-cacheable push (NCP) operations as a leakage primitive that allows us to infer whether a specific attacker cache line is resident in the LLC. This establishes the probing capability required for the attack.

Priming the LLC. To fill a cache set with attacker-controlled lines, we attempt to use NCP operations, since by definition (per the CXL specification [17]), NCP operations place a device (attacker) cache line into the host's LLC. While this provides the ability to insert lines into the LLC, it does not guarantee that an entire cache set can be filled, as this depends on the LLC's replacement and management policies. To evaluate this, we conduct an experiment to determine whether NCP operations can effectively prime the LLC. We construct eviction sets of size N for an N-way associative cache, with all addresses mapping to the same LLC set. We then issue NCP operations to all N cache lines in the eviction set. If priming is successful, subsequent NCP operations to these lines should result in hits and exhibit lower latency.

However, we do not observe this behavior. We hypothesize that the LLC employs policies that limit the number of cache lines inserted via device pushes, preventing them from fully occupying a set. This is consistent with prior observations on Intel DDIO [52], where the number of I/O cache lines is restricted to avoid evicting useful core data. Alternatively, this behavior may stem from undocumented LLC replacement policies that prevent sequential NCP operations from fully populating a set [53].

To overcome this limitation, we introduce an additional step called *depollution*. The goal of depollution is to reset the target LLC set to a clean state by ensuring that previously cached lines are invalidated before priming. We achieve this by accessing an oversized eviction set that maps to the target LLC set, followed by explicitly flushing those lines. This process enables subsequent NCP operations to effectively occupy all cache ways without self-eviction. In practice, we find that repeating these accesses multiple times is sometimes necessary to reliably reset the cache state. We also perform depollution before each measurement to maintain a stable replacement state for effective priming. We evaluate the impact of depollution on priming effectiveness by comparing the number of cache lines successfully inserted into the LLC with and without depollution. Figure 5 shows that depollution significantly improves the effectiveness of priming. Similar techniques have been explored in prior work to mitigate the effects of unknown replacement policies [21], [23], [29].

B. Observability into Private Caches

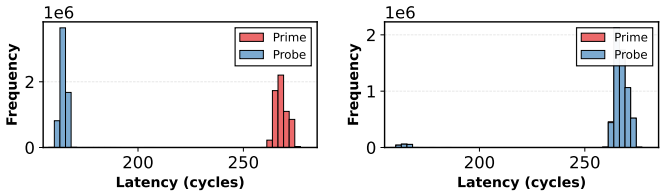
Non-cacheable push writes can only observe and modify the state of the last-level cache (LLC). As a result, under a non-inclusive LLC, they capture only a subset of the memory accesses performed by CPU cores. While this still leaks useful (albeit coarser-grained) information, it does not expose all accesses of a program.

To overcome this limitation, we identify three approaches:

Cache Line Mirroring. Cache line mirroring (described in §II-B) addresses this limitation by ensuring that private-cache accesses are reflected in the LLC. We show that this enables a high-bandwidth covert channel with low bit error rate and can bypass cache partitioning defenses. However, it requires the attacker and victim to share the physical address space (e.g., via shared kernel code).

Directory Eviction. Directory eviction leverages capacity evictions in the coherence directory to indirectly bring cache lines into the LLC, creating a cascading effect observable by the device. The attacker can prime the directory using multiple cores, which reduces self-evictions. While this approach removes the need for shared physical address space, it is slower, less effective, and more prone to noise than cache line mirroring.

Cache Line Demotion. Cache line demotion uses the CLDEMOTE instruction (available in Sapphire Rapids) to explicitly move cache lines from private caches to the LLC. However, this approach requires the attacker to share a core with the victim in a multithreaded processor, limiting its applicability.



(a) (b)

Fig. 5. Impact of cache depollution on the channel. (a) With depollution, the access latency exhibits a distinct bimodal distribution, indicating successful cache state initialization (b) Without depollution, the distribution lacks a bimodal signature, demonstrating ineffective priming of the eviction set.

C. Attack Overview

With the attack primitives established and the challenges of observing private caches addressed, we now present an overview of the PUSH+PUSH attack. The attack consists of three key steps, illustrated in Figure 6.

- **DEPOLLUTE**. The attacker resets the replacement state with this step as described in §V-A.
- **PUSH**. The attacker issues non-cacheable push operations from the device to fill the target set in the LLC with memory addresses belonging to device.
- **PUSH**. The attacker waits and repeats the previous step, timing each access to monitor evictions

D. Establishing a Covert Channel

System Setup. We evaluate the covert channel on a physical testbed summarized in Table I.

Synchronization and Encoding. We use PUSH+PUSH to construct the covert channel between a sender and a receiver, implemented as two processes on the same host. The key idea is that PUSH+PUSH allows the receiver to probe the cache state of specific cache sets and detect whether they have been accessed by the sender. The receiver begins by initializing a set of predetermined cache sets into a known state using the `PUSH` primitive. During transmission, the sender encodes information by selectively accessing memory addresses that map to these cache sets, thereby modifying their cache residency. To enable synchronization, we designate one cache set as a clock and another as a data channel. In each transmission interval, the sender first accesses the clock set to signal the start of a bit. It then conditionally accesses the data set to encode the transmitted value: accessing the data set encodes a logical one, while leaving it untouched encodes a logical zero. After each interval, the receiver probes both cache sets using `PUSH` and measures the resulting cache misses. An increase in misses indicates that the corresponding cache set was accessed by the sender. The receiver uses activity in the clock set to identify valid transmission windows and samples the data set only when a clock event is detected.

Transmission and Error Rate Computation. To evaluate the reliability of this channel, we transmit a 4096-bit sequence from the sender and reconstruct the sequence at the receiver. We quantify the accuracy of this transmission by computing the edit distance between the transmitted and the received sequence, as this robustly captures both bit flips (substitutions) and bit slips (insertions/deletions).

Repetition to Improve Bit Error Rate. To improve reliability of noisy channels (e.g., variant with directory evictions), the sender uses repetition code and transmits each bit multiple times. The receiver then applies majority vote to reconstruct the bit sequence. While this approach reduces the bit error rate (BER), it proportionally reduces channel bandwidth.

Evaluating the Covert Channel. Our evaluation of the covert channels shows an error rate of less than 24% for all configurations. Figure 8 shows that we achieve the best bit error rate and bandwidth trade-off with cache line mirroring which gives us a BER of 2.6% at 15.5 kbps. With cache line demotion, we see a BER ($13.2\,\%$) and lower bandwidth (11 kbps). With directory evictions, the bandwidth is much lower as it relies on multiple cores to prime the directory and uses repetition code to improve the channel. It achieves a bandwidth of 1.15 kbps with a BER of 23.2%.

The peak theoretical bandwidth of this covert channel is 25.4 kbps. The key bottleneck in this attack is the depollute step, which may be improved if the host uses eviction sets built from host memory only for the depollute step. It is important to note that the operating frequency of our device limits the rate at which it performs non-cacheable push writes. which is the second major bottleneck after depollution. A production-scale device with higher clock rate may increase the bandwidth of this covert channel.

Sensitivity Analysis. To understand the impact of background noise on the accuracy of the covert channel, we evaluate it under background noise from other processes. We simulate the background noise with cache workload in `stress-ng`, and evaluate the BER of the covert channel under varying number of worker threads. By introducing noise from four worker threads, we see a drop in the covert channel performance for both PUSH+PUSH and conventional CPU-based attack. However, increasing the number of worker threads beyond this point renders both conventional channel and PUSH+PUSH unusable as shown in Figure 7.

Covert Channel on Partitioned Caches using CAT. To protect sensitive workloads from cache-contention based vulnerabilities, several works propose cache partitioning using Intel Cache Allocation Technology (CAT) [22], [23], [26], [40], [41]. While this approach mitigates contentions, it is often ineffective against attacks targeting shared memory [54], as cache hits can be served from across partition boundaries. To address this limitation, several works propose sophisticated cache designs and randomization [55], [56]. Since PUSH+PUSH can also utilize shared memory, we evaluate its effectiveness in the presence of partitioned caches.

To evaluate the covert channel in partitioned caches, we define three classes of service (COS). We allocate one cache way to background cores, and one mutually exclusive cache way to victim core, each in their own COS. The attacker core

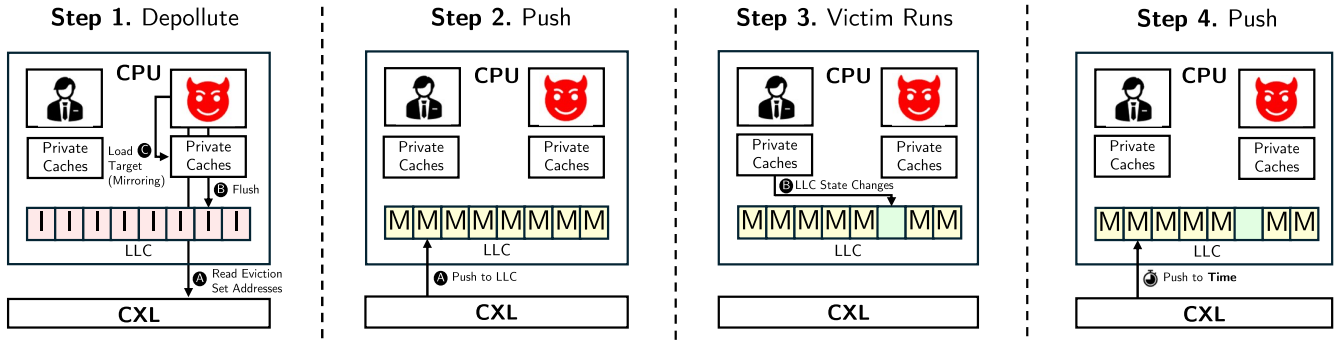


Fig. 6. Overview of PUSH+PUSH.

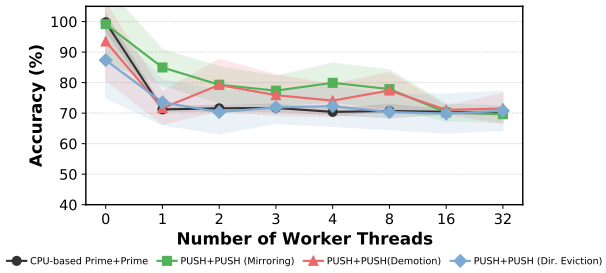


Fig. 7. Characterization of covert channel accuracy under different number of background `stress-ng` threads

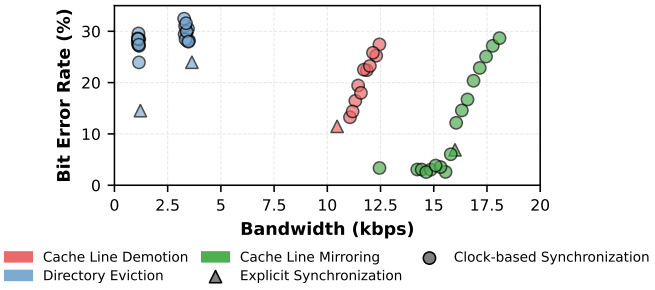


Fig. 8. Bandwidth versus Bit Error Rate trade-offs for different clock rates for three variants of PUSH+PUSH.

is assigned a third COS that occupies remaining cache ways. We observe that the CXL Type-2 device is architecturally restricted to `COS0` so it shares the remaining cache ways with the device.

We see that cache mirroring bypasses this partition, allowing the attacker to observe the effects of victim accesses in an isolated partition. We evaluate the covert channel and we achieve 14.42 kbps bandwidth with 5.87% BER. However for directory evictions and cache line demotion, the covert channel does not exist for partitioned caches. This shows that cache partitioning is not an effective mitigation against PUSH+PUSH if we share memory.

Impact of Different Partitions on Covert-Channel. We reduce the number of cache ways accessible by the attacker to only three cache ways, all of them not overlapping with the victim's cache ways, but the covert channel remains operational.

Evaluating Attacks Across Sockets. To evaluate the covert channel across sockets, the sender is set up on the remote socket and the receiver on the local socket. The local socket connects to the device through its root port, while the remote socket uses Intel Ultrapath Interconnect (UPI) [3] for transiting to the device. NCP only primes the last-level caches of the local socket and not the remote socket, but with cache line mirroring the covert channel can exist to the remote socket as well.

E. Side-Channel Attack

We demonstrate the practical applicability of our attack through a side-channel attack. We use the primitives presented in this paper to leak an AES round key. Our goal is not to demonstrate a newly discovered side channel, but to show the performance of our attack on a standard side-channel attack.

OpenSSL AES Implementation. OpenSSL's AES implementation is a standard library for cryptography and has been extensively used to demonstrate the practicality of side channels [21], [40]. We adopt a one-round synchronous attack to our primitives to recover the four most significant bits for each byte of round key. In this attack, the attacker controls the plaintext and triggers encryption [24]. By repeating the encryption for multiple plaintexts, we collect multiple measurements per plaintext and compute a score for each cache set based on cache activity that is used to recover the round key. With the four most significant bits per byte, the attacker can recover the remaining bits by brute force [24], [57] to reconstruct the complete round key.

Evaluation of Attack. We show this side-channel attack with cache line mirroring and directory eviction that can recover the maximum possible four most significant bits of each byte of the round key. For cache line mirroring, we only need 1024 unique plaintexts, compared to 8192 plaintexts for directory evictions to fully recover the round key.

Key Extraction with Cache Partitioning. We also show the side-channel attack with partitioned LLC described in Section V-D. We can again recover the maximum number of bits

exposed by this attack with only 1024 plaintexts and 640 measurements per plaintext.

VI. Timing Analysis of Back-Invalidation

In this section, we analyze the timing behavior of the back-invalidate snoop protocol introduced by CXL 3.0. Our goal is to identify capabilities that may be leveraged by an adversary as attack primitives.

Simulation Platform. At the time of writing, there is no commodity hardware that supports the back-invalidate snoop protocol. So, we rely on simulation and model the timing behavior of back-invalidate in CXLMemSim [19]. CXLMemSim is a software simulation framework for performance characterization of `CXL.mem`. CXLMemSim integrates seamlessly with QEMU, and intercepts the memory requests from virtual machines to model the performance. It also models the impact of bus traffic to produce timing jitter on the bus.

We create two virtual machines to represent two hosts, and use file-backed shared memory in QEMU. While the file-backed shared memory is coherent, it does not simulate caches and protocol-level details. CXLMemSim forwards memory requests from virtual machines to a CXLMemSim server that runs on the host. This server reads and writes from the shared backing file. The server routes the requests, and simulates coherency management.

We extend CXLMemSim, and add support for back-invalidation protocol. To fully support back-invalidation protocol, we maintain a set-associative cache in QEMU that simulates hosts' CPU cache. The requests cached in the set-associative cache are served directly without retrieving data from the backed file, and a set-associative snoop filter tracks the memory locations cached in each host's cache hierarchy.

Property 1: Cross-Server Cache State Observability. In a setup where two hosts share a memory region, a spy process on one host can infer the cache state of the other host by timing memory accesses to Shared cache lines. The spy observes higher access latency when a request triggers a back-invalidate snoop. This occurs when the target cache line is held in the Exclusive or Modified state in the other host's cache. In such cases, the coherence engine must first issue a back-invalidate request to that host and downgrade the cache-line state before the access can complete. Because the most up-to-date copy of the data may reside in a private cache, the response must wait for that cache to supply or write back the data. This additional coherence step introduces a measurable latency penalty, as shown in Figure 9, which allows the spy to distinguish whether the cache line is privately held by another host. Based on interconnect characteristics and specification latency targets, we expect this penalty to be on the order of tens of nanoseconds [17].

Observation 5: A server sharing device memory with another server can observe cache lines with ownership permissions by loading the target cache line.

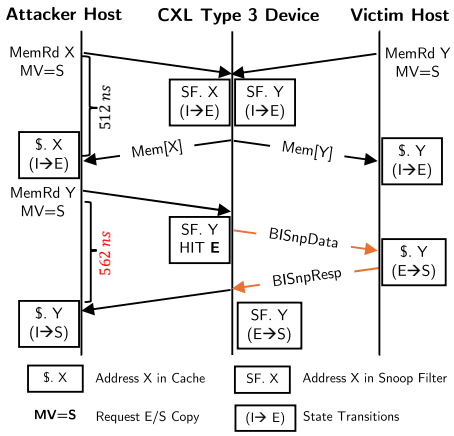


Fig. 9. Illustration of the timing difference exploited by the SnoopDog attacks. Block X (first access) is not cached remotely, therefore is not delayed by a back invalidation while Block Y is in the Exclusive state in victim and thus is delayed by a back invalidation.

Property 2: Cross-Server Cache State Modification. In a setting where two hosts share a memory region, a malicious process on one host may be able to modify the coherence state of cache lines in the other host. By accessing a shared memory location, the malicious process can downgrade a cache line held in the Exclusive state by the other host to the Shared state. Furthermore, with knowledge of the snoop filter's set and slice mapping [45], the malicious process can also evict cache lines belonging to other hosts. This capability is useful for an attacker as it allows the attacker to put a particular cache line into a known state. By strategically loading cache lines from the device memory, one server may be able to evict cache lines from another server by evicting entries from the device snoop filter.

Observation 6: A server sharing device memory with another server can downgrade Exclusive cache lines to Shared state by loading the target cache line.

VII. SnoopDog

This section presents our SnoopDog attack. Our analysis of back-invalidate snoop protocol reveals two potential sources of leakage that we explore below.

A. SnoopDog Downgrade Attack

When a cache line is accessed by only one host, coherence protocols grant that host exclusive caching permissions. In this case, the snoop filter records that the line is cached by a single host, allowing future accesses to avoid broadcast snoops. If another host subsequently accesses the same line, the snoop filter mediates a back-invalidate snoop to revoke exclusive permissions and ensure coherence correctness, thereby revealing the presence of an Exclusive cache line in victim host's cache. The SnoopDog attack exploits the additional delay introduced by the coherence downgrades. The attack works in three steps:

1) **Evict.** The attacker evicts the target cache line from the snoop filter by accessing addresses that map to the same

set or by using a flush instruction to evict the target line directly, ensuring the line is not resident locally.

2) **Victim Access.** The victim executes and conditionally accesses the target cache line, causing it to be cached exclusively in the victim host and recorded as such in the snoop filter.
3) **Reload.** The attacker accesses the target cache line. If the line was loaded by the victim, the access triggers a back-invalidate snoop to revoke the victim's Exclusive state, incurring additional latency that the attacker observes.

B. SnoopDog Invalidate Attack

SnoopDog Invalidate infers cross-host victim cache state by identifying the evictions from the device snoop filter which is expected to invalidate the lines from the hosts' cache. The invalidated cache lines must be reloaded from the memory directly and would incur a latency penalty which we can exploit as a timing signal.

SnoopDog Invalidate may also be designed in a way to allow the attacker to flush and reload one cache line repeatedly but relies heavily on replacement policy of the device snoop filter. If the victim access occupies the snoop filter entry that was flushed from the attacker, the reload from the attacker would incur a latency penalty from blocking back-invalidate snoop.

C. Covert-Channel Characterization

We demonstrate the covert channel for SnoopDog Downgrade in CXLMEMSIM. We construct a covert channel with a data line and a clock line. We estimate the channel bandwidth from the measured access latencies which show that the channel can achieve 420 kbps. For our simulated covert channel, the BER is 7%.

Sensitivity Analysis. We analyze the impact of the number of hosts and the associativity of the device snoop filter on the performance of our covert channel. First, we evaluate the channel without any noise and change the associativity of the device snoop filter. We see that the accuracy drops if a set in the device snoop filter can cache more lines than the host cache hierarchy, and the covert channel collapses. To overcome this challenge the attacker must either control multiple hosts or exploit the replacement policy for the attack [28]. We also observe the impact of background workloads running in other hosts in this system. We sweep over the number of such hosts, where each host runs one thread of random memory access workload. This workload simulates noise due to contentions in the device snoop filter, and congestion in the interconnect bus. We only see a small impact of additional background hosts. We present our findings in Figure 10.

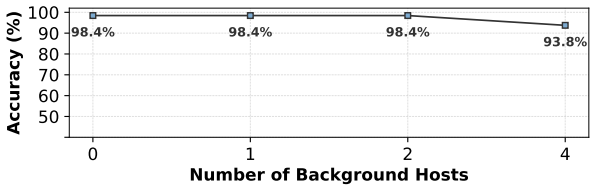


Fig. 10. SnoopDog covert channel accuracy as the number of additional hosts running memory stress workloads.

D. Recovering Tokens from LLM Inference

Next, we show how SnoopDog can recover tokens processed during LLM inference and use them to reconstruct the user prompt and model response. But before going to details of the attack, we first discuss the emulation platform that we use to evaluate the attack. We use CXLMEMSIM to emulate the back-invalidation snoop on two QEMU VMs. The VMs communicate through shared memory to the CXLMEMSIM server to access the emulated CXL device. For this attack to work, the embedding table must be stored in the device-attached shared memory. The victim uses `llama.cpp` to perform inference, and the attacker observes the access pattern to reconstruct the output sequence. For simplicity, explicit synchronization and prompt communication is performed using a shared file, which can be API endpoints in a real-world attack.

We build on top of a state-of-the-art large language model attack [58]. The goal of this attack is to leak a user's prompt and response from a large language model. To transform text to high-dimensional vectors, large language models use the embedding tensor. By monitoring cache activity for embedding tensor, an attacker can leak the tokens. After retrieving the tokens, timing of autoregressive decoding stages can leak the position of the tokens to reconstruct the token sequence. Our evaluation focuses on recovering the prompt and output response, where we recover the prompts given to Mistral 7B (FP16) [59] and output responses in `llama.cpp` inference framework.

The attack requires that embedding lookups execute on the CPU. During probing, an embedding line cached by the victim triggers a back-invalidate request and therefore incurs higher latency. Otherwise, the device can serve the request directly with lower latency.

Because each embedding spans multiple cache lines, the attacker can monitor a token by probing any selected line. We exploit this flexibility to construct a self-evicting probing pattern. The embedding table provides enough candidate lines mapping to each snoop-filter set to fill the set multiple times. We divide these lines into groups large enough to fill a set and use a staggered cache-line offset for each successive group. This shifts the groups to consecutive sets while keeping all probes within the address range of each embedding. After each complete vocabulary sweep, we increment the offset of every probe, shifting the entire probing pattern from set n to set $n+1$. Repeated sweeps therefore form a rolling, self-evicting pattern that eventually wraps around the snoop-filter set space. This optimization eliminates the need for dedicated eviction accesses and avoids aliasing between token probes. For Mistral 7B (FP16), it reduces the number of required eviction sets from one per vocabulary entry to only twice the number of snoop filter sets that would exist without this offset (i.e. 512).

We report the precision of the recovered tokens which shows

```
prompt: "when does man in the high castle come out?"
recover: cl say] nothing/ castle man I.[ written
    Philip Castle,2 thatINSTsay novelwhen series
    science High high9 Prime come by adapted
    November out Video K say published5] nothing?1
    in Man6 seasonsed does on/ first castle into the
     of man television episodes Dick I0. [ has been
    writtening Philip Castle,2 premier season Amazon
     that asking released you subsequent It
    consistedINST annuallysay novelrewhen series ten
     science High high9 show Prime come by' adapted
    November out Video if So K If say5 nothing?1 in
    Man6 seasons fourthed final does on/ first
    castle into when...'
reordered: If you are asking when the science
    fiction series The Man in the High Castle,
    adapted from the novel by Philip K. Dick, was
    released and how many seasons it had... The Man
    in the High Castle is a science fiction
    television series adapted from the novel by
    Philip K. Dick. It premiered on Amazon Prime
    Video in November, with its first season coming
    out and subsequent seasons following. The series
     consisted of four seasons, running until its
    final season concluded the show.
```

Fig. 11. Example Response Recovered from the LLM Side-Channel Attack on llama.cpp and Mistral 7B FP16 Model and Reordering the Token Sequence with Gemini Flash-Lite 3.5

the ratio of correctly leaked tokens, and recall that shows the percentage of tokens leaked. Our CXLMEMSIM emulation achieves 99.8% precision and 99.42% recall across 20 test prompts. We recover only the token identities, which can then be reordered to reconstruct the original text. Since natural language follows learnable structural patterns, we use Gemini Flash-Lite 3.5 to infer the most likely token order for an example reconstruction shown in Figure 11. The wall-clock time to reconstruct the output of each prompt is 7 minutes when the attack is running on the simulator. However, the measurements obtained from this simulation-based experiment will be different from physical hardware as they have different timing characteristics.

We use one thread for probing on the attacker, and introduce a 2-second delay after each generated token. This two-second delay is primarily a limitation of the simulator rather than the attack itself. CXLMemSim intercepts and simulates each access to the CXL device, which disproportionately slows down the SnoopDog attacker because it must continuously probe a large memory region, while the victim accesses CXL memory much less frequently. We therefore add an artificial delay after each generated token to compensate for this simulation-induced imbalance.

VIII. SUMMARY OF ATTACKS AND MITIGATION

We summarize the evaluation results of PUSH+PUSH and SnoopDog in Table IV. Together, these attacks demonstrate that CXL introduces new coherence and device-driven attack surfaces. While PUSH+PUSH highlights practical attack vectors enabled by accelerators, SnoopDog reveals the security risks that may emerge as CXL memory sharing becomes prevalent in future multi-host systems.

TABLE IV
BANDWIDTH AND ERROR RATE PROFILES FOR PUSH+PUSH AND SNOOPDOG ATTACK VARIANTS

Attack Family	Variant	Bandwidth	Error Rate (%)	Demo Platform
PUSH+PUSH	Dir. Eviction	1.15 kbps	18.2 %	Real HW
	Mirroring	15.5 kbps	2.6 %	Real HW
	Line Demotion	11 kbps	13.2 %	Real HW
SnoopDog	Downgrade	420 kbps	7 %	Simulation

A. *Mitigating* PUSH+PUSH

Software Mitigation. PUSH+PUSH has three variants with different attacker assumptions. PUSH+PUSH with cache line mirroring assumes memory sharing between the attacker and victim processes. In the second variation PUSH+PUSH uses evictions from the cache directory which does not need memory sharing for cross-core attacks. PUSH+PUSH with cache line demotion is more constrained and cannot attack across cores and needs memory sharing between the two processes.

PUSH+PUSH breaks isolation from cache partitioning if it uses cache line mirroring. PUSH+PUSH also does not rely on memory sharing if it uses directory evictions, but this variant cannot break cache partitioning isolation. To eliminate the threat from PUSH+PUSH, cache partitioning alone will not be enough to stop attacks that use cache line mirroring. We will also need to stop sharing memory between two processes in addition to cache partitioning.

Protocol Defense. PUSH+PUSH leverages the non-cacheable push write described in §IV-B. The device sends a request to the host to obtain ownership of the cache line it wants to write and receives a global observation (GO) from the host which marks completion at the device. The device pushes the data to the host after completion. We suspect that the global observation (GO) signal from the host is susceptible to leaking timing information that relates to cache state in the host.

We analyze the protocol defense in two threat models. In the first threat model, the attacker is assumed to have a compromised CXL controller on device and controls protocol-level transactions. In the second threat model, the attacker does not have a compromised controller but can send memory requests from device.

If the controller is compromised, we can mitigate PUSH+PUSH by introducing a constant-latency constraint on the host. This requires modifying the host microarchitecture so that it responds to the device request in constant time regardless of the target line's state in the host cache hierarchy. This may introduce a latency penalty on memory accesses but if memory requests are pipelined, the penalty will be amortized. We evaluate the performance impact of this mitigation in Figure 12. We consider three memory access patterns: repeating (REP), sequential (SEQ), and random (RND). To approximate a realistic device workload, we emulate an Ethernet controller issuing DMA transfers for 64 KB packets, with four packets transmitted in bursts. We compare the overhead of constant-latency servicing under both pipelined and serialized execution. Our results show that the repeating

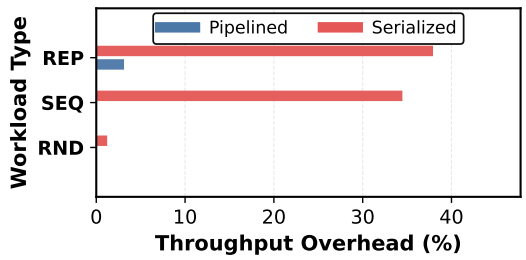


Fig. 12. Throughput overhead of the constant time mitigation.

access pattern incurs the highest overhead, as it frequently results in LLC hits that are artificially delayed under constant-time servicing. In contrast, sequential and random patterns exhibit lower overheads due to their reduced cache locality. Pipelining significantly mitigates these costs by overlapping memory requests and hiding a portion of the added latency.

If the controller is not compromised, we can add support without modifying the host microarchitecture. We propose modifying the protocol for non-cacheable push writes so that the device receives a completion without requesting ownership. However, it introduces overhead as the device must buffer the data for which ownership is pending. Based on the structure of this buffer, the timing information may still be leaked but this may constrain the attack.

B. Mitigating SnoopDog

Protocol Defense. SnoopDog assumes copy-on-write for shared memory. This means any modification of shared memory creates a private copy and does not modify the shared copy. With copy-on-write, no host has Modified cache lines from shared memory. However, they may have cache lines in the Exclusive state which grants ownership to the cache controller to modify the line without notifying other controllers. SnoopDog leaks cache state because back-invalidates need to resolve ownership of these cache lines. So to mitigate SnoopDog, we need to ensure that reads bring cache lines without Exclusive ownership to the host cache hierarchy.

The proposed defense replaces the coherence protocol used for shared memory with MSI instead of MESI. Unlike MESI, MSI does not include an Exclusive (E) state, an optimization that reduces coherence traffic when a core subsequently writes a privately held line.

As SnoopDog relies on transitions involving exclusive ownership, eliminating the Exclusive state removes these ownership transitions, thereby eliminating downgrade-based signaling exploited in our attack. This mitigation comes at the cost of increased coherence traffic due to more frequent upgrade operations.

Mitigation with Cache Write Policy. A cache write policy determines whether the writes update the cache or the main memory and can influence the security. Write-through caches write data directly to main memory without the need for the Modified state, and can prevent the penalty of blocking back-invalidate snoop used for downgrading. However, this is not a practical mitigation as it can create substantial traffic on the memory bus for applications that frequently write data such as databases. We may only apply this policy selectively on memory address ranges that belong to shared memory regions.

Snoop Filter Defense. SnoopDog Invalidate relies on evictions from the device snoop filter and the protocol defenses do not mitigate snoop-filter set contention. This contention occurs because the device snoop filter tracks the lines from the device that are cached in the hosts. SnoopDog Invalidate works by completely filling the device snoop filter by loading the lines in one host, and then observing evictions from the host. This means the host must have enough caching capacity so that it can completely fill the cache set in the device snoop filter, otherwise it will not be able to completely fill the snoop filter and effectively limit the attack.

To defend against the SnoopDog Invalidate, we propose that the device snoop filter should have a higher associativity than the number of lines that can be cached per set in the host cache hierarchy. However, it comes at the cost of increasing the size of the snoop filter on the device which may not be feasible for every device.

IX. Conclusion

CXL brings cache-coherent devices and cross-host memory sharing into mainstream systems, but these same coherence mechanisms introduce new timing side channels that extend beyond traditional co-located process attacks. In this work, we presented the first systematic analysis of timing leakage in CXL protocols and implementations, demonstrating that both device–host and cross-host coherence operations expose exploitable signals. We introduced PUSH+PUSH, a device-based Prime+Probe–style attack that leverages CXL.cache behavior to infer host cache activity and bypass existing detection and partitioning defenses, and SnoopDog, a cross-host timing attack demonstrated in CXLMemSim that exploits back-invalidation snoop delays in CXL 3.0 environments. Our results show that coherent interconnects significantly expand the side-channel attack surface, highlighting the need for device-aware isolation and security-focused coherence design as CXL systems are deployed at scale.

Acknowledgment

We would like to thank Qi Ling for assisting with the recovery of the Intel cache-slice hash function. We would like to thank Ampere Computing and Longview Philanthropy for supporting our research. Co-authors from UC San Diego are supported in part by PRISM and CoCoSys, centers in JUMP 2.0, an SRC program sponsored by DARPA, and NSF grants 2112665, 2211386, 2052809, 2112167.

References

[1] Compute Express Link Consortium, "Compute express link (cxl)," https://computeexpresslink.org/, May 2025, accessed: 2025-10-27.

[2] D. Das Sharma, R. Blankenship, and D. Berger, "An introduction to the compute express link (cxl) interconnect," *ACM Computing Surveys*, vol. 56, no. 11, pp. 1–37, 2024.

[3] Z. Wang, S. Mahar, L. Li, J. Park, J. Kim, T. Michailidis, Y. Pan, M. Shen, T. Rosing, D. Tullsen, S. Swanson, and J. Zhao, "The hitchhiker's guide to programming and optimizing cache coherent heterogeneous systems: Cxl, nvlink-c2c, and amd infinity fabric," 2025. [Online]. Available: https://arxiv.org/abs/2411.02814

[4] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, "Demystifying cxl memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 105–121. [Online]. Available: https://doi.org/10.1145/3613424.3614256

[5] S. Hynix, "Sk hynix develops ddr5 dram cxl memory to expand the cxl memory ecosystem." [Online]. Available: https://news.skhynix.com/sk-hynix-develops-ddr5-dram-cxltm-memory-to-expand-the-cxl-memory-ecosystem/

[6] Samsung, "Expanding the limits of memory bandwidth and density: Samsung's cxl memory expander." [Online]. Available: https://semiconductor.samsung.com/news-events/tech-blog/expanding-the-limits-of-memory-bandwidth-and-density-samsungs-cxl-dram-memory-expander/

[7] V. Petrucci, E. Mirakhur, N. Agarwal, S. W. Lim, V. Tanna, R. Gupta, and M. Wagh, "Cxl memory expansion: A closer look on actual platform," 2023.

[8] Intel Corporation, "Agilex 7 fpga i-series development kit (2x r-tile and 1x f-tile)," https://www.altera.com/products/devkit/a1jui0000049utmmam/agilex-7-fpga-i-series-development-kit-2x-r-tile-and-1x-f-tile, 2025, accessed: 2025-10-27.

[9] AMD, "Amd versal™ premium series gen 2." [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal/gen2/premium-series.html

[10] "4th Gen Xeon Scalable Processors — intel.com," https://www.intel.com/content/www/us/en/products/docs/processors/xeon-accelerated/4th-gen-xeon-scalable-processors.html, [Accessed 04-02-2026].

[11] A. O. Munch, N. Nassif, C. L. Molnar, J. Crop, R. Gammack, C. P. Joshi, G. Zelic, K. Munshi, M. Huang, C. R. Morganti, S. Kandula, and A. Biswas, "2.3 emerald rapids: 5th-generation intel® xeon® scalable processors," in *2024 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 67, 2024, pp. 40–42.

[12] AMD, "4th Generation AMD EPYC™ Processors," https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series.html, 2023, [Accessed 04-02-2026].

[13] ——, "5th Generation AMD EPYC™ Server CPUs," https://www.amd.com/en/products/processors/server/epyc/9005-series.html, 2024, [Accessed 25-03-2026].

[14] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 742–755.

[15] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.

[16] Y. Tang, P. Zhou, W. Zhang, H. Hu, Q. Yang, H. Xiang, T. Liu, J. Shan, R. Huang, C. Zhao, C. Chen, H. Zhang, F. Liu, S. Zhang, X. Ding, and J. Chen, "Exploring performance and cost optimization with asic-based cxl memory," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 818–833. [Online]. Available: https://doi.org/10.1145/3627703.3650061

[17] CXL Consortium, *Compute Express Link (CXL) Specification*, revision 3.0, version 1.0 ed., Compute Express Link Consortium, 8 2022. [Online]. Available: https://www.computeexpresslink.org/

[18] V. Nagarajan, D. J. Sorin, M. D. Hill, D. A. Wood, and N. E. Jerger, *A Primer on Memory Consistency and Cache Coherence*, 2nd ed. Morgan & Claypool Publishers, 2020.

[19] Y. Yang, B. Zhao, Y. Zheng, P. Safayenikoo, T. A. Khan, and A. Quinn, "Cxlmemsim: A pure software simulated cxl.mem for performance characterization," 2025. [Online]. Available: https://arxiv.org/abs/2303.06153

[20] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[21] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+scope: Overcoming the observer effect for high-precision cache contention attacks," 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. 2906–2920. [Online]. Available: https://doi.org/10.1145/3460120.3484816

[22] 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*, 2015, pp. 605–622.

[23] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 888–904.

[24] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.

[25] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," 2016. [Online]. Available: https://arxiv.org/abs/1511.04594

[26] F. Rauscher, C. Fiedler, A. Kogler, and D. Gruss, "A systematic evaluation of novel and existing cache side channels," in *Network and Distributed System Security Symposium (NDSS) 2025*, Feb. 2025, network and Distributed System Security Symposium 2025 : NDSS 2025, NDSS 2025 ; Conference date: 23-02-2025 Through 28-02-2025. [Online]. Available: https://www.ndss-symposium.org/ndss2025/

[27] G. Saileshwar, C. W. Fletcher, and M. Qureshi, "Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1077–1090. [Online]. Available: https://doi.org/10.1145/3445814.3446742

[28] S. Briongos, P. Malagon, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1967–1984. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/briongos

[29] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Last-level cache side-channel attacks are feasible in the modern public cloud," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 582–600. [Online]. Available: https://doi.org/10.1145/3620665.3640403

[30] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A Timer-Free High-Precision l3 cache attack using intel TSX," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 51–67. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen

[31] Z. Wang, M. Taram, D. Moghimi, S. Swanson, D. Tullsen, and J. Zhao, "NVLeak: Off-chip side-channel attacks via non-volatile memory systems," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[32] M. Taram, A. Venkat, and D. Tullsen, "Packet chasing: spying on network packets over a cache side-channel," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 721–734. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00065

[33] 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)*, 2020, pp. 20–38.

[34] 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)*, 2023, pp. 2288–2305.

[35] J. Yu, A. Dutta, T. Jaeger, D. Kohlbrenner, and C. W. Fletcher, "Synchronization storage channels (S2C): Timer-less cache Side-Channel attacks on the apple m1 via hardware synchronization instructions," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1973–1990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jiyong

[36] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 168–179.

[37] F. Thomas and M. Schwarz, "RISCy Cache Coherence: Timer-Free Architectural Cache Attacks via Instruction/Data Cache Incoherence," in *IEEE S&P*, 2026.

[38] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 353–364. [Online]. Available: https://doi.org/10.1145/2897845.2897867

[39] Intel Corporation, "Improving performance by utilizing Cache Allocation Technology," Intel Corporation, White Paper 331843-001US, 2015. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf

[40] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 406–418.

[41] F. Yao, H. Fang, M. Doroslovački, and G. Venkataramani, "Cotsknight: Practical defense against cache timing channel attacks using cache monitoring and partitioning technologies," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 121–130.

[42] Y. Wang, L. Wu, S. Gao, Y. Tang, J. Luo, Z. Wang, Y. Ou, D. Dong, N. Xiao, and M. Lai, "Cohet: A cxl-driven coherent heterogeneous computing framework with hardware-calibrated full-system simulation," 2026. [Online]. Available: https://arxiv.org/abs/2511.23011

[43] H. Ji, Y. Yuan, Y. Zhou, I. Jeong, R. Wang, S. Agarwal, and N. S. Kim, "Re-architecting end-host networking with cxl: Coherence, memory, and offloading," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1809–1823. [Online]. Available: https://doi.org/10.1145/3725843.3756102

[44] L. Giner, R. Czerny, C. Gruber, F. Rauscher, A. Kogler, D. D. A. Braga, and D. Gruss, "Generic and automated drive-by gpu cache attacks from the browser," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 128–140. [Online]. Available: https://doi.org/10.1145/3634737.3656283

[45] J. D. McCalpin, "Mapping addresses to l3/cha slices in intel processors," Texas Advanced Computing Center, Tech. Rep. TR-2021-03, 2021. [Online]. Available: https://hdl.handle.net/2152/87595

[46] Y. Han, P. Qiu, Y. Yang, C. Wang, D. Wang, J. Dong, and G. Qu, "Split+prune: Constructing minimal eviction set under the lru replacement policy," in *2025 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2025, pp. 1–6.

[47] Z. Xue, J. Han, and W. Song, "Ctpp: A fast and stealth algorithm for searching eviction sets on intel processors," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 151–163. [Online]. Available: https://doi.org/10.1145/3607199.3607202

[48] Y. Zhang, J. Yang, Y. Yue, Y. Vigfusson, and K. V. Rashmi, "Sieve is simpler than lru: an efficient turn-key eviction algorithm for web caches," in *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'24. USA: USENIX Association, 2024.

[49] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A timing attack on OpenSSL constant time RSA," Cryptology ePrint Archive, Paper 2016/224, 2016. [Online]. Available: https://eprint.iacr.org/2016/224

[50] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "TLBleed: When Protecting Your CPU Caches is not Enough," in *Black Hat USA*, Aug. 2018. [Online]. Available: Slides=https://i.blackhat.com/us-18/Thu-August-9/us-18-Gras-TLBleed-When-Protecting-Your-CPU-Caches-is-Not-Enough.pdfWeb=https://vusec.net/projects/tlbleed

[51] Intel Corporation, "Effective utilization of Intel Data Direct I/O Technology," Intel Corporation, Tech. Rep., 2012, white Paper. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-paper.pdf

[52] H. Park, J. Lou, S. Lee, Y. Yuan, K. Park, Y. Son, I. Jeong, and N. S. Kim, "A4: Microarchitecture-aware llc management for datacenter servers with emerging i/o devices," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 679–693. [Online]. Available: https://doi.org/10.1145/3695053.3731114

[53] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim, " Don't Forget the I/O When Allocating Your LLC ," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2021, pp. 112–125. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISCA52012.2021.00018

[54] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 974–987.

[55] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021, pp. 37–49.

[56] A. Bhatla, H. R. Bhavsar, S. Saha, and B. Panda, "Sok: so, you think you know all about secure randomized caches?" in *Proceedings of the 34th USENIX Conference on Security Symposium*, ser. SEC '25. USA: USENIX Association, 2025.

[57] T. Hornetz, H. Yavarzadeh, A. Cheu, A. Gascon, L. Gerlach, D. Moghimi, P. Schoppmann, M. Schwarz, and R. Zhang, "TDXRay: Microarchitectural side-channel analysis of intel TDX for real-world workloads," in *IEEE Symposium on Security and Privacy (S&P)*, 2026.

[58] Z. Gao, J. Hu, F. Guo, Y. Zhang, Y. Han, S. Liu, H. Li, and Z. Lv, "I know what you said: Unveiling hardware cache side-channels in local large language model inference," in *Proceedings of the 34th USENIX Conference on Security Symposium*, ser. SEC '25. USA: USENIX Association, 2025.

[59] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

APPENDIX
ARTIFACT EVALUATION

A. Abstract

This artifact provides the source code, experimental and plotting scripts for timing analysis, covert channels, and side-channel attacks for PUSH+PUSH primitives. We outline the hardware requirements, provide access instructions, and describe how to reproduce the primary experimental results presented in this paper.

B. Artifact check-list (meta-information)

- **Hardware:** CXL-enabled system (detailed in Table I) will be provided directly to AE reviewers.
- **Output:** Execution logs, timing traces, and generated figures.
- **Publicly Available:** Yes (excludes proprietary/licensed bitstream components from Intel).
- **Experiments:** PUSH+PUSH analysis and evaluations (Figures 2, 4, 5, 7, and 8), SnoopDog Sensitivity Analysis (Figure 10), and Mitigation Overhead Analysis (Figure 12).
- **Execution Time:** PUSH+PUSH reproduction takes approximately 16h (Human effort approx. 30m) and SnoopDog reproduction takes approximately 8h (Human effort approx. 1h)
- **Archived:** https://doi.org/10.5281/zenodo.21549983

C. Description

1) How to access: We provide the archived source code at https://doi.org/10.5281/zenodo.21549983

2) Hardware dependencies: Hardware dependencies are listed in Table I.

3) Software dependencies: To reproduce the AES side-channel evaluation, download and build OpenSSL 1.0.1f. Detailed build instructions follow standard procedures from the prior Flush+Reload repository, however it is handled by configuration setup by default.

Python dependencies required for data processing and plotting are handled automatically by the setup script. Ensure the Python virtual environment containing these packages is activated prior to executing `make build-exps`. Before building QEMU ensure that no Python virtual environment is running.

D. Installation

Follow the detailed instructions in `README.md` to configure the environment, compile the attack primitives, and prepare the experiment workspace. Detailed guidelines for PUSH+PUSH and SnoopDog are provided in `README.md` in respective directories.

E. Evaluation and expected results

To reproduce the key empirical findings for PUSH+PUSH (Figures 2, 4, 5, 7, and 8), follow the instructions in `push-push/README.md`. The experimental plots and reference plots will be produced in `push-push/plots/<fig-num>`.

Additionally, the experimental suite generates:

- Directory timing analysis plots (Section IV-B) in `push-push/plots/additional-plots/`.
- Empirical metrics for the AES side-channel attack in `push-push/results/`.
- Mitigation overhead analysis for PUSH+PUSH (Figure 12)

To reproduce the results for SnoopDog (Figure 10), follow the instructions in `snoopdog/README.md`. The experimental plots and reference plots will be produced in `snoopdog/plots/<fig-num>`.

Note: *Due to AE timeframe constraints, sample sizes in automated run scripts have been reduced relative to the paper runs. Minor variance in exact figures may occur, but all qualitative conclusions and attack accuracies remain fully verifiable.*

F. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae