MeshUp: Stateless Cache Side-channel Attack on CPU Mesh

Junpeng Wan, Yanxiang Bi, Zhe Zhou†
Fudan University
{19210240003,19210240167,zhouzhe}@fudan.edu.cn

Zhou Li
University of California, Irvine
zhou.li@uci.edu

***Abstract*—Cache side-channel attacks lead to severe security threats to the settings where a CPU is shared across users, e.g., in the cloud. The majority of attacks rely on sensing the micro-architectural state changes made by victims, but this assumption can be invalidated by combining spatial (*e.g.*, Intel CAT) and temporal isolation. In this work, we advance the state of cache side-channel attacks by showing stateless cache side-channel attacks on server-grade CPUs, that can bypass both spatial and temporal isolation.**

Unlike stateful cache side-channel attacks that rely on the timing difference between a cache hit or miss, our attack exploits the timing difference caused by the interconnect congestion. Specifically, to complete cache transactions, for Intel server CPUs, which use non-inclusive and mesh interconnect, cache lines would travel across cores via the CPU mesh and UPI interconnects. Nonetheless, the interconnects are shared by all cores, and cache isolation does not segregate the traffic. An attacker can generate traffic to contend with a victim on a link, measure the extra delay, deduce the memory access pattern of the victim's program, and infer its sensitive data. Based on this idea, we implement MESHUP, a stateless cache side-channel against mesh interconnect, and test it against the existing RSA implementations of JDK for the cross-core attack and application fingerprinting for the the cross-CPU attack. We found the RSA private key used by a victim process can be partially recovered and the co-running application can be inferred at high accuracy.

I. INTRODUCTION

Memory isolation is one of the fundamental security principles to protect sensitive information. Yet, memory isolation does not protect the computing resources, like cache, resulting in side-channel attacks under machine sharing scenarios, *e.g.*, in the cloud. By timing the interval of accessing a cache line, existing cache attacks learn whether the associated memory addresses have been loaded by a victim program, and further deduce sensitive information. Many techniques have been proposed, like FLUSH+RELOAD [1], PRIME+PROBE [2], and the recent Xlate [3], based on different assumptions, *e.g.*, using shared libraries [1], sharing LLC (Last-level Cache) across cores [4], or MMU (Memory Management Units), *etc*.

These attacks can be categorized into *stateful cache attacks*, as the victim program introduces micro-architectural state changes that can be sensed by an attacker [5], *e.g.*, through creating eviction sets. However, this attack condition may not be fulfilled nowadays due to the rise of spatial and temporal isolation. For example, Intel Xeon CPUs introduce *Cache Allocation Technique (CAT)* [6], which was designed to maintain QoS of cache usage, but later found to be a panacea for cache attacks [7]. CAT assigns LLC cache ways to cores exclusively, which spatially isolate LLC and break attacks based on eviction set conflict on LLC (*e.g.*, [4]). Temporal isolation [5] provides more principled protection, which is effective against nearly every cache attack with the existing hardware support.

Stateless Cache Side-channel Attack. Although spatial and temporal isolation can eliminate the root cause of cache side-channel attacks, or micro-architectural state change, stateless cache side-channel attacks could bypass both of them. One such stateless channel is the CPU interconnect, which links different CPU units like cache and cores. Due to the complex interplay between these CPU units, it is possible that cache access can be observed on the interconnect by an adversary. In fact, a recent work [8] showed that the CPU *ring* interconnect can be exploited to launch stateless attacks. Ring interconnect is prevalent in consumer-grade CPUs, but not in server-grade CPUs. Whether these server-grade CPUs, which are more relevant to the cloud settings, are vulnerable is unclear.

In this work, we explore the attack surface on the server-grade CPUs. By investigating the latest architecture of Intel server CPU, *e.g.*, Xeon Scalable Processor (SP) [9], we found that cores and uncore units inside a CPU are connected with *mesh*, constituting an NoC (Network on Chip). Besides, different CPUs are also linked with an interconnect, *i.e.*, *UPI (Ultra Path Interconnect)*[1]. Although these interconnects showed a great advantage in latency and bandwidth on multi-core CPUs [10], [11], they could leak information about the memory access pattern of a program, because of the timing difference resulting from *congestion* on those interconnects.

Based on this insight, we propose MESHUP, a new stateless cache side-channel attack against CPU interconnects. Our key idea is to let a core occupied by an attacker program keep probing the path that the cache transactions of a victim program might pass by, and measure the delay. When the core occupied by the victim program accesses a remote cache agent, the accumulated mesh traffic volume will rise, hence increasing the delays observed by the attacker. By probing the interconnects at high frequency, the attacker could deduce the

† Zhe Zhou is the corresponding author.

[1]UPI is also used as interconnect inside CPU by extreme high-end CPUs, *e.g.*, Xeon 9200 series.

victim's secret with the delay traces.

Challenges. Still implementing the idea of MESHUP is challenging. 1) We assume the attack can be executed on a cloud VM, which prohibits an attacker to choose a core or mesh path at his/her will. 2) There is no API to let a program direct mesh packets to a given target and the information retained from mesh congestion is expected to be coarse-grained (*i.e.*, it does not tell which cache lines are conflicted). 3) Cross-CPU attack is even more difficult as there are more units involved.

Attack Techniques. We have investigated the side-channel leakages in both cross-core and cross-CPU settings. For the first case, we develop a new eviction-based probe, which allows an attacker to probe a mesh route and measure the delays. In particular, our approach constructs an L2 eviction set, which can be mapped to the desired LLC slice, and contained in an L2 set. *To notice is that the eviction set does not conflict with the victim.* It is only used to cause cache eviction and generate mesh traffic. Therefore, defenses trying to prevent adversarial cache eviction can be evaded. For the second case, we found that, though the eviction-based probe cannot reliably generate the cross-CPU traffic, *cache synchronization* by two CPUs could introduce a high volume of traffic, and congest the link. Hence, we develop a new coherence-based probe for cross-CPU attacks. Our analysis of the two probes shows they can achieve good temporal resolution, high Signal to Noise Ratio (SNR), and spatial resolution that allows the attacker to contend to the victim's traffic even starting off from a random core.

As a showcase for the attack effectiveness, we analyzed the Sliding Window algorithm of RSA with the eviction-based probe to recover 2048-bit private RSA keys. On the off-the-shelf implementation of JDK, the attacker can recover over 31% of the 2048 bits, with the help of a cryptographic method [12]. For the cross-CPU attack, we assume a victim runs an application (app) in server, and let the attacker infer which app the victim is running, with the coherence-based probe. The attacker has over 82% accuracy in recognizing the apps co-located on the machine.

Contributions. We summarize the main contribution of our work as follows.

- We identify a new security implication of server-grade CPU interconnects, and show it can be exploited to construct a powerful stateless side-channel.
- We develop MESHUP, using cache eviction and cache synchronization as probe techniques to conduct cross-core and cross-CPU cache side-channel attacks.
- We systematically analyze the properties of MESHUP channels.
- To show the consequences of MESHUP, we evaluate it with RSA key recovery and app fingerprinting.

II. BACKGROUND AND RELATED WORKS

In this work, we investigate the security of the cache architecture of Intel Xeon Scalable Processors (SP) [9], which have gained a prominent market share in cloud computing [13]. We first overview their cache design. Then, we introduce the prior cache side-channel attacks. Finally, we overview the research of stateless channels that serve as our attack primitives.

A. Architecture of Intel Xeon SP

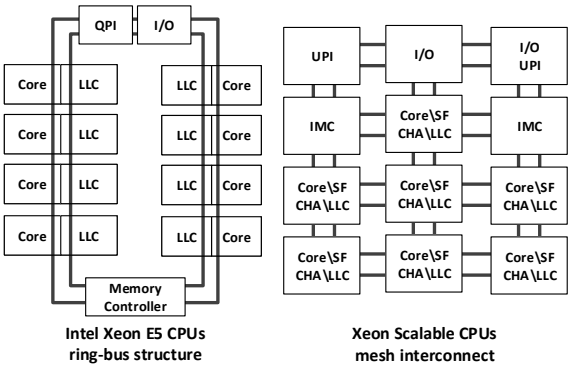

Fig. 1: Comparison between ring-bus and mesh structures. Not all cores/components are drawn to save space.

Mesh Interconnect. When a CPU chip contains multiple cores, the connection topology among them, or *on-chip interconnect*, is a key factor determining the CPU performance. The old design of interconnect (*e.g.*, in Intel Xeon E5 and Intel Core) mimics the multiprocessor architecture, in that a shared *ring-bus* connects all cores together [14], as illustrated in Figure 1 left. However, the core-to-core latency could increase linearly along with the growth of cores within one CPU die, because the communication between two cores could be routed through all other cores.

Since Xeon Skylake-SP server CPU family [15] (released in 2017), Intel revamped the interconnect design with *mesh*, which is also adopted by the latest generation of Intel server CPUs, *e.g.*, Xeon Icelake-SP, and expected to be the default design in the near future [16]. Besides Intel CPUs, mesh interconnect has also been adopted by other processors, like Tile Processors [10], [17], [11], and ARM server CPUs [18]. In essence, the chip is structured as a 2D matrix of *tiles* [19] and each tile either consists of a core (together with cache), or an uncore component like *IMC (Integrated Memory Controller)*, *UPI (Ultra Path Interconnect)* controller, and I/O unit. Each tile is connected to its vertical and horizontal neighbors, and the traffic of each direction (in total 4) is managed through a *mesh stop* inside the tile. Figure 1 right illustrates the mesh structure. Mesh interconnect caps the core-to-core latency at a much lower rate, because the number of hops between any pair of tiles is only proportional to the square root of the number of tiles, which satisfies the growth of core density.

UPI. To further increase core density, Intel allows linking CPUs on different sockets with UPI connection [20], which wires UPI mesh stops from different CPUs. One CPU can have up to 3 UPI stops. Cross-socket mesh traffic firstly reaches the UPI stop of the originating CPU via mesh, and then will be

forwarded to the UPI stop of the destination CPU. At last, the UPI stop forwards it to the destination mesh tile.

Cache Hierarchy. Modern processors all feature a hierarchy of cache to localize the frequently accessed data and code, in order to reduce the access latency. Each core has its own private cache, *e.g.*, L1 and L2 cache. And there is also L3 cache, or last-level cache (LLC), which is shared across cores.

Xeon SP breaks LLC into slices and evenly distribute them among cores. As shown in Figure 1 right, each tile consists of a core (with L1 and L2 embedded), a LLC slice, a *Snoop Filter (SF)* and a *Cache Home Agent (CHA)*. Appendix A describe the cache configurations of Intel Skylake-SP.

How a memory address maps to a cache line depends on the set index of the address. Intel has a proprietary hash algorithm to do the mapping. We illustrate the structure under Xeon SP in Figure 9 in Appendix A, summarized from [21], [22].

Cache Coherence. As the same address can be read/write by different cores, the cache should be kept coherent to avoid the access of outdated data. To this end, Intel uses the *MESIF (Modified, Exclusive, Shared, Invalid, Forward)* protocol [23] for cache coherence, managing five cache states. "Modified" means the line is in the private cache of the owing core and is dirty as it has been written. "Exclusive" indicates the line is stored in a single core. "Shared" means the line is potentially shared by multiple cores. "Invalid" means the line is not in cache. "Forward" suggests the core holding the line is responsible to forward the line to cores reading the line.

The CHA is responsible for coordinating all the cores and maintaining the coherence. For example, when a requesting core requests to write a cache line that is cached by another core (or caching core) in the Exclusive state, the CHA will ask the caching core to send the line to the requesting core and turn the state to Invalid, as a line can only be writable to one core. After this, the requesting core can turn the state of the line to Exclusive. In Section V-B, we exploit the messages delivered under the MESIF protocol as a side-channel for cross-CPU attacks.

B. Cache Side-channel Attack

Cache side-channel attack bypasses memory isolation and it is particularly concerning in the cloud setting, where multiple users share the same physical machine [4], [24], [25]. Below we overview the existing attack methods, and classify them by whether they assume memory sharing between victim and attacker. The overview is not meant to be exhaustive and we refer interested readers to surveys like [26].

Sharing Memory. Running processes often share identical memory pages to save memory. Shared memory leads to shared cache, and FLUSH+RELOAD exploits such a condition for cache side-channel attack. It takes three steps. First, the attacker sets all the cache lines mapped to the shared memory as invalid, by using cache clearance instruction `clflush`. Then, the attacker waits a period of time for the victim to access the shared memory. Finally, the attacker accesses the shared memory and counts the cycles (*e.g.*, through `rdtsc`) to measure the latency, and infer the code/data access pattern of the victim.

FLUSH+RELOAD has been demonstrated effective on LLC [1] of a PC and cloud instances [24], resulting in leakage of encryption keys [25] and keystroke events [27]. It has been evolved to variations [26] like FLUSH+FLUSH [28], which is stealthier by avoiding the extra memory access. On the other hand, this attack can be mitigated when `clflush` is banned [29]. To address this limitation, EVICT+RELOAD [30] was proposed, which uses cache conflicts as a replacement for `clflush`.

Not Sharing Memory. When memory is not shared, an attacker can still force cache contention because memory addresses of different programs can share a cache set. PRIME+PROBE exploits such feature, and it also takes three steps. First, the attacker collects a set of cache lines that can fill a cache set and access the related memory addresses. Next, the attacker waits for the victim to evict the cache lines. Finally, the attacker measures the access latency.

Though PRIME+PROBE initially targets L1 cache [31], LLC that is inclusive has also been attacked [25], [4]. A number of variations have been developed [26]. For instance, PRIME+ABORT [2] measures the Intel TSX (Transactional Synchronization Extensions) abort rather than access latency. Instead of letting the victim evict the cache lines, EVICT+TIME lets the attacker evict a cache set, and then invokes the victim operation [32], [33].

Indirect Attacks. Recently, researchers started to investigate the interplay between other CPU units and cache, to make the attack more evasive. For instance, XLATE [3] and TLBLEED [34] exploited MMU (Memory Management Units) and TLB (Translation Lookaside Buffers) to leak victim's cache activity. The recent Intel Xeon SP started to use non-inclusive LLC, which raised the bar for LLC cache attacks. Yet, Yan *et al.* [22] showed that by targeting cache *directories* (or Snoop Filter), the units tracking which core contains a copy of a cache line, attacking non-inclusive LLC is feasible.

One major assumption of the prior attacks is that the attacker's code is on the same machine as the victim's. Recently, attacks over network connections were studied. By exploiting RDMA (Remote Direct Memory Access) and DDIO (Data Direct I/O), a remote attacker can access LLC [35] of CPU and cache inside NIC [36], launching side-channel attacks.

On an orthogonal direction, transient execution attacks [37] like SPECTRE [38], MELTDOWN [39] and FORESHADOW [40] modulate the state of the cache to construct *covert channels*, and exfiltrate information from speculatively executed instructions. MESHUP focuses on side channels and we will investigate whether MESHUP can be leveraged by transient execution attacks in the future.

C. Stateless Channels

According to Ge *et al.* [5], microarchitectural side-channels exploit the competition of hardware resources, which can be classified into two categories: *microarchitectural state* and *stateless interconnects*. The first category includes

caches, TLBs [34], branch predictors [41], and DRAM row buffers [42], on which resource contention leads to the state changes observable to the adversary. The second category includes I/O buses [43], [44], [45], [46], execution ports [47], [48], [49], cache-bank [50], memory bus lock [51], CPU interconnect [52], *etc*. Yet, we found the security implications of CPU interconnect contention have not been thoroughly investigated, even though it has some very attractive properties, like residing in all contemporary CPUs and requiring no special features to be used by victim applications. These properties do not always hold in other stateless channels: e.g., VAX security kernel has led to I/O bus contention but it is outdated [43], [44], and ports contention rely on Intel SMT [47], [48], [49]. Though Wang *et al.* used a simulator to study the timing side-channel of on-chip network [52], the attack has not been demonstrated on real CPUs. In this work, we investigate this under-studied channel.

Comparison to LoR ("Lord of Rings") [8]. LoR was proposed to build side-channel over CPU *ring interconnect*, which is equipped by *consumer CPUs and old server CPUs*. Different from LoR, our attack MESHUP focuses on *mesh interconnects* that is equipped by current *server-grade CPUs*. There are also conceptual differences between LoR and MESHUP. First, consumer-grade CPUs come with inclusive cache, but server-grade CPUs use distributed and non-inclusive cache, in which case the path is unknown to attackers. Second, in addition to the eviction-based probe, MESHUP has a new coherence-based probe that enables cross-CPU attack.

III. ATTACK OVERVIEW

In this section, we first introduce the threat model and compare it with prior works. Then, we overview our attack MESHUP.

A. Threat Model

We assume the attacker who intends to extract secret (*e.g.*, encryption keys) shares the same CPU with the victim but resides in different cores. We envision MESHUP is effective in the cloud environment, when co-residency on CPU or machine can be achieved [54]. We also assume the existing hardware and software defenses against the cache side-channel attacks are deployed and turned on, like page coloring [55], [56] and CATalyst [7]. We target Intel Xeon SP, where core-to-core communication goes through mesh interconnects or UPI connections. We assume that the victim application is non-trivial, which generates observable mesh traffic, either through a large secret working set to cause private cache eviction, or frequent memory access. Our evaluation in Section VII shows this feature is common in off-the-shelf applications.

Here we compare our setting to the existing cache attacks. The strongest assumption made by the prior works is the sharing of memory addresses (shown in Figure 2 left), like FLUSH+RELOAD. However, memory sharing can be turned off for the critical data/code. A weaker assumption is that cache sets are shared (shown in Figure 2 right), so the attacker can evict cache lines of the victim (or vice versa), like

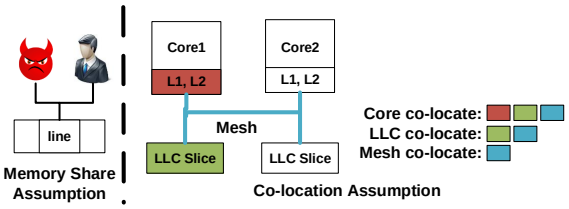

Fig. 2: Comparison of the settings between different cache side-channel attacks.

PRIME+PROBE. Under this assumption, the attacker either shares in-core private L1/L2 cache with the victim [31], out-of-core LLC (either inclusive LLC [25], or non-inclusive LLC [22]). However, the assumption does not always hold when software defenses like page coloring or hardware defenses like CATalyst are deployed to enforce spatial isolation. Recently, temporal isolation has been proposed to mitigate all existing cache side-channels. Appendix B describes these defenses in detail and Table I summarizes the representative attacks and how they are impacted under the existing defenses.

Stateless Cache Side-channel Attack on Mesh. Since the data movement on the mesh keeps occurring for all sorts of program activities, if the attacker's and victim's programs happen to share a mesh route, the victim's activities might be inferred, which potentially include cache accesses. In this process, the attacker could just access his/her own resources. Based on this insight, we develop MESHUP to exploit the contention on the mesh.

MESHUP is expected to *bypass page coloring, hardware isolation and temporal isolation*: it does not cause cache conflicts between victim and attacker, so page coloring and hardware isolation like CAT and TSX can be evaded. Though the time protection of [5] is very effective against the existing stateful cache side-channel attacks, the authors admit that they are *"powerless"* against stateless channels, since there is no "appropriate hardware support" to partition interconnects.

B. Attack Steps

MESHUP consists of two phases to tackle the challenges (summarized in Section I) of creating a reliable side-channel.

Phase 1: Probe & Measurement (Section V). When launching attacks, the attacker runs an application sharing CPU with the victim. The attacker randomly selects a mesh path and then tries to trigger traffic along the path. To direct mesh packets over the selected path, the attacker either 1) constructs an eviction set and probes the memory addresses related to it, or 2) causes cache line synchronization across CPUs. The probe is issued repeatedly and the delays are logged.

Phase 2: Secret Inference (Section VII). After the prior step, the attacker obtains the delay trace, and the secret underlying the trace is to be decoded. This step is application-specific, as different victim programs produce different patterns. This step can be done at the attacker's own machine. We use RSA

	Co-location Assumption	Attack Channel	Page Coloring	Hardware Isolation	Temporal Isolation	Key Feature
Flush+Reload [1]	Memory	LLC	✓	CAT	✓	Exploit shared memory
Prime+Probe [25], [4]	Cache	LLC	✓	CAT	✓	No need to share memory
TLBleed[34]	Core	TLB	-	Disable HT	✓	Attack TLB not cache
Attack Directories, not caches [22]	Cache	Directory	✓	SecDir [53]	✓	Work on non-inclusive cache
Prime+Abort[2]	Cache	TSX Status	✓	CAT	✓	Does not rely on timing instruction
Xlate[3]	Cache	MMU	-	-	✓	Lure MMU to access cache
LoR[8]	Ring	Ring	-	-	-	Stateless, consumer-grade CPUs
MESHUP	Mesh	Mesh	-	-	-	Stateless, server-grade CPUs, cross cores and CPUs

TABLE I: Cache side-channels under defenses. "✓" means the attack can be defended. "-" means the defense is ineffective.

encryption and app fingerprinting to showcase how to decode a delay trace.

IV. CHARACTERIZATION OF MESH TRAFFIC

Before introducing the mechanisms of MESHUP, we categorize mesh traffic generated during the lifetime of a cache line, and show the characteristics of each type, which motivates the design of MESHUP.

A. Cache Access

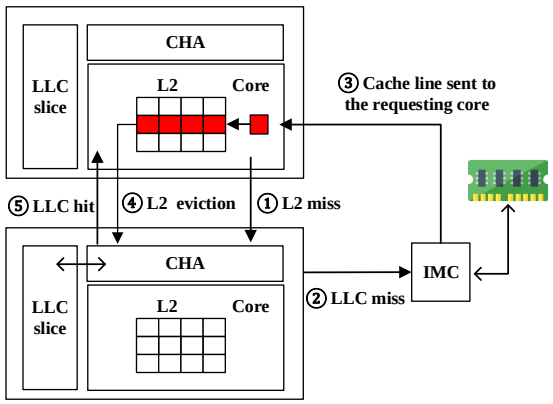

Fig. 3: The process of cache access.

Here we describe the process of cache access, focusing on the non-inclusive LLC adopted by Xeon SP (also illustrated in Figure 3). When a core accesses a fresh memory address (the core is called *requesting core*), both L1 and L2 cache would miss. The memory sub-system of the requesting core computes a CHA ID (*target CHA*) from the memory address with a proprietary hash algorithm, to know which CHA (*target CHA*) is responsible for the address. Then a read transaction is sent to the target CHA (Step ①), which replies to the requesting core and updates the *directory* accordingly. Since the line is not cached in the LLC nor another core, the CHA cannot complete the request by itself, and it will ask the IMC to fetch the line from memory (Step ②), and IMC will send the line directly to the requesting core (Step ③). The cache line will be inserted into the L2 of the requesting core, but the LLC will be kept untouched, because it is non-inclusive. When the requesting core runs out of the L2 cache, it will follow a pseudo-LRU policy [22] to evict a line from L2 to LLC (or drop directly, depending on eviction policy) (Step ④). Evicted lines will be accessed from LLC the next time (Step ⑤).

To be noticed is that all the five steps leverage mesh (plus UPI when crossing CPUs) to deliver transaction messages as well as the cache lines. This is critical to our attack as it exploits the statistics of the transactions.

B. Mesh Traffic Categorization

Below we introduce the types of mesh traffic (termed **T1-T7**) related to cache access, summarized from the existing documents [57]. We use r and t to refer to the entity issuing the request and the target. Hence, the requesting core is termed $Core_r$, and the co-located L2, LLC slice and CHA are termed $L2_r$, LLC_r and CHA_r. For the target, the terms are $Core_t$, $L2_t$, LLC_t and CHA_t.

- **T1: $Core_r$ to LLC_t.** When $Core_r$ encounters L2 cache miss, it will send messages to CHA_t co-located with LLC_t, asking if the cache line is presented. Also, $L2_r$ may evict a line to LLC'_t when the L2 cache set to be inserted is full. The memory sub-system of the core will pass the line from $L2_r$ to LLC'_t .
- **T2: LLC_t to $Core_r$.** Following T1, if the cache line is in LLC_t, LLC_t will send the cache line to $Core_r$.
- **T3: IMC to $Core_r$.** Alternatively, if the cache line is not present in LLC_t, CHA_t will ask the IMC to fetch the line from memory and send it to $Core_r$. To be noticed is that the line is directly sent to $Core_r$, when LLC is non-inclusive.
- **T4: LLC_t to IMC.** When LLC_t is full, to accept new cache line insertion, it will evict the least recently used cache line to the IMC.
- **T5: between $Core_r$ and $Core_t$.** $Core_r$ can access a cache line in the private cache of $Core_t$ when they share memory. $L2_t$ will pass the cache line to $Core_r$ through mesh, once $Core_r$ is going to write the line.
- **T6: LLC_t to I/O Unit.** Intel CPUs allow I/O devices to directly access LLC and bypass memory for better performance, under DDIO [14]. In this case, cache lines will be passed between a PCIe stop (stop inside a PCIe tile) and LLC_t.
- **T7: Other traffic.** It characterizes the mesh traffic undocumented by Intel, which is expected to have a smaller volume than T1-T6.

In Section VI-F, we summarize our insights about different traffic types.

V. The Probe Design

Below we describe two probes under MeshUp, which selects a path on the interconnect, hoping to contend with the victim's mesh traffic and triggers mesh transactions.

A. Probe based on Cache Eviction

Different from routing a packet on the Internet, in mesh interconnect, a program *cannot* explicitly sends traffic to the destination, because cache transactions are triggered implicitly. To address this challenge, we adapt the existing methods for *constructing evictions sets* [4], [22]. An eviction set is a large set of memory addresses that are mapped to the same cache set, so accessing the whole set will result in cache-set overflow and cache eviction. Previous attacks, *e.g.*, Prime+Probe, use an eviction set to evict lines of the private caches in the victim core. Though MeshUp uses eviction set, our goal is *not* to evict victim addresses. To the contrary, **MeshUp evicts lines of its *own* private L2 cache**, in order to generate mesh traffic flowing to a designated LLC slice, which is distinguished by the mesh tile. As such, MeshUp stays out of the protection realm of any existing defense. In Figure 4, we illustrate the concept of our probe.

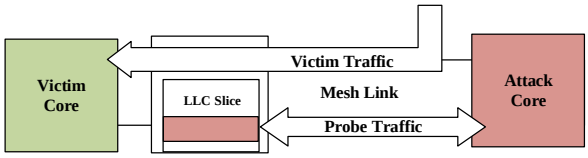

Fig. 4: The probe designed by the attacker.

Noticeably, we assume the attack can be executed on a cloud VM, where *the attacker cannot pin the process to his/her desired core or learn which core is occupied by the victim application*. In other words, the attacker cannot select the optimal path for contention. Still, our evaluation shows the chances of contention are high on a random path.

Constructing Eviction Set. First, the attacker prepares a set of memory addresses (denoted as EV) that are mapped to one L2 cache set. The number of addresses (denoted as n) in EV is set to be larger than the number of ways (denoted as w) an L2 set has, therefore when requesting addresses of EV, L2 cache misses always happen after w requests. From $w+1$ to n requests, each time a line is evicted from L2 to LLC, a new line from memory will be inserted to L2. After that, when requesting EV again, n lines will be evicted from L2 to LLC, and n lines will be passed from LLC to L2 in return, resulting in stable bi-directional mesh traffic on the attack path.

To get traffic on the fixed mesh route, MeshUp needs to force all L2 misses to be served by one LLC slice. Thus, EV is not only mapped to a set of L2, but also a set of LLC slices. To find addresses for such EV, we use the two routines proposed by [22], `check_conflict` and `find_EV`, which are designed for non-inclusive LLC. In essence, `check_conflict` tries to test if removing an address of a set makes cache conflict disappear. `find_EV` tries to utilize `check_conflict` to filter out a set of addresses that are all mapped to the same LLC slice. In Appendix C, we describe them in detail. We split the EV into EV_0 and EV_1 of equal size, and the addresses on the two sets have different 16th bit (0 and 1). As shown in Figure 9 of Appendix A, bits 15:6 of a memory address point to an L2 set, while bits 16:6 point to an LLC slice set. As such, EV is associated with a single L2 cache set, and EV_0, EV_1 are mapped to two different sets of the same LLC slice.

Even when all cache lines fall into one LLC set, the attacker cannot decide which LLC slice serves the $Core_r$ yet. To identify the LLC slice, Yan *et al.* suggest testing an EV (multiple EV have been constructed as candidates) to see if it co-locates with $Core_r$ [22]. We adopt the same approach and let the attacker enumerate every core on his/her own CPU, and measure its access time to an EV. When the serving LLC slice is local to the core, the access latency is the lowest, and we link EV to the LLC slice ID.

Finally, the size of EV (n) has to be tuned carefully by the attacker. Not only n should be larger than w to force L2 misses, but it should also avoid being too large to overflow LLC and L2 together. Otherwise, a line will be evicted from LLC to memory (T4 traffic), a request that takes a long time to respond, reducing the probing frequency. As the Xeon SP uses 11-way LLC set and 16-way L2 set [9], we varied n of EV from 18 to 38 (2*11+16), and found when n equals to 24 (EV_0 and EV_1 each has 12 addresses), the mesh traffic generated within an interval is the highest.

For the attack prototype, we exploit a CPU feature called *huge page* [58], which allows pages of more than 4KB to be accessed, to make the attacker's measurement more stable. However, this assumption is not necessary during the real-world attack as Vila *et al.* has a solution for EV construction with normal pages [59].

Delay Measurement. When the victim application is running, the attacker sequentially visits every address within EV immediately after receiving the response to the prior request, and records the timestamp of each request (*e.g.*, using the instruction `RDTSCP` to read the CPU counter). The interval between consecutive requests reflects the cache transaction latency. When the interval is increased, the victim is supposed to have cache transactions concurrently. To cope with random noises, the attacker repeatedly visits the EV for x times (we set x to 20 during the experiment) to obtain a sample for an interval, and analyzes the interval trace to infer the access patterns of the victim applications.

The Pseudo-code. Here we summarize the probe in Algorithm 1. The process of EV generation is adapted from [22], by setting the size of the set of candidate EV (EVs) three times as the number of LLC slices, in order to increase the chance of getting an EV for a designated LLC slice. This size can be adjusted based on the running environment.

Algorithm 1: The pseudo-code of eviction-based probe

```
Result: IntervalSeq
path = rand_path();
L2_set_index = rand(0x3ff);
for i in 3* (num of LLC slices) do
    EVs.append(get_EV( L2_set_index));
end
// select an EV
set_affinity(path.dst);
for each EV in EVs do
    access(first addr of EV);
    if access time ≤ TH then
        break;
    end
end
// start attack
set_affinity(path.src);
while True do
    for i in range(20) do
        access(EV);
    end
    IntervalSeq.append(access_time);
end
```

The `get_EV()` function employs `check_conflict` and `find_EV` to find an EV for an LLC slice. The attacker uses `set_affinity()` to place its program to a random core (source) and repeatedly test if an EV is in the desired LLC (destination). Noticeably, calling `set_affinity()` from VM will not pin the program to a desired core. We use this routine to make the core ID constant. Sometimes, core remapping could move the applications to different cores during execution, and we discuss its impact in Section VII-B ("Core Remapping").

B. Probe based on Cache Coherence

The eviction-based probe can generate sufficient and steady mesh traffic on a single CPU, but cannot produce enough cross-socket traffic that is needed for the cross-CPU attack. When we test the eviction-based probe on a server with two CPUs and let a core evict lines to an LLC slice in the remote socket, we found the CPU core does not always evict the desired cache lines. Instead, the core may just drop the lines if they are clean. In this case, accessing the eviction set only triggers T2 traffic, so some accesses related to victim memory may not be captured by the probe.

By inspecting the different types of traffic, we found if an attacker intentionally triggers communications through UPI, cross-core T5 traffic can be captured, so we design the probe accordingly. When two different cores keep writing to the same cache line, the CHA will be busy maintaining the cache coherence (under MESIF described in Section II-A), and the interconnects would be filled with T5 messages *synchronizing* the line. The completion time of such cache accesses can be influenced by the victim's traffic.

We design a cross-CPU probe based on this insight. The attacker first occupies a core and constructs a set of cache lines that are all mapped to an LLC slice of the core. Then he/she starts a thread in the core (core A) to keep sequentially writing the lines with random values. At the same time, the attacker starts a thread in the remote socket (core B) to also sequentially write the lines within the set, and records the time spent on accessing the set. In this way, every cache line to be accessed by core B is expected as Invalid, because core A should have written the line, during which core B turned the line to the invalid state and core A reaches the exclusive state. As a result, writing the invalid line causes messages sent to the CHA in core A, and core A will send the line to core B, resulting in bi-directional UPI traffic. Both cores of the attacker can record the timestamps to infer the victim access pattern. Algorithm 2 shows the pseudo-code of this probe. A prominent advantage of this probe is that the contention to the victim traffic through the UPI path is *deterministic*, so all the measured delay traces can be used for inference, without worrying about path co-location.

Algorithm 2: The pseudo-code of coherence-based probe

```
Result: IntervalSeq
path = rand_cross_CPU_path();
set_affinity(path.dst);
for i in 3* (num of LLC slices) do
    AS = rand_slice_addr_set();
    access(first addr of AS);
    if access time ≤ TH then
        break;
    end
end
if clone() == parent then
    while True do
        access(AS);
    end
else
    set_affinity(path.src);
    while True do
        access(AS);
        IntervalSeq.append(access_time);
    end
end
```

According to our exploratory experiment, 4 pairs of probes can *saturate the two UPI links by 93%*. In this case, extra cross-socket traffic made by the victim can cause congestion and enlarge the attacker's probe delays.

Our probe differs from the previous cross-CPU channels. DRAMA exploits the contention on memory shared across CPUs rather than cache [42]. Irazoqui *et al.* exploits the directory protocol of CPU interconnects [60], and attacks

QuickPath Interconnect (QPI) [61] on Intel CPU. Our probe attacks UPI, which succeeds QPI in Intel SP. We also believe our probe has the potential to be generalized to other CPU interconnect, like ring on old CPUs which use QPI, but the channel quality might be downgraded.

VI. Analysis of the MeshUp Side-channel

In this section, we provide quantitative analysis on the probes described in the prior section. We first describe an approach to reverse engineer CPU layout and leverage it to examine the variance of delays caused by the contention between the victim and attacker applications. Then, we inspect the spatial and temporal distribution of the mesh traffic. We also identify the reasons for the delay increase resulting from mesh congestion, according to CPUs' performance counters. A simple mitigation based on LLC slice isolation is tested against MeshUp probe. Finally, we summarize our insights into the MeshUp side-channel.

A. CPU Layout Reverse Engineering

For the quantitative analysis, we aim to enumerate various combinations of attacker's and victim's mesh traffic. The CPU layout needs to be reverse engineered, so that we can pin a program to the desired core and direct it to talk to another. To notice, this step requires root privilege to execute some profiling instructions, but during the actual attack described, this step is not taken.

We have reverse engineered Intel Xeon Scalable 8260 and 8175. Due to the space limit, we describe the layout of 8175 in Appendix D.

Identifying the Enabled Tiles. There are three types of CPU dies, named LCC, HCC, XCC (for low, high, and extreme core counts), for Intel Xeon family, with 10, 18, or 28 cores in a die respectively [62]. However, when a CPU is shipped to the customers, Intel might intentionally disable some cores. For example, the Xeon Scalable 8260 has 24 active cores, because Intel disabled 4 cores.

Here, we exploit the hardware features of Intel CPUs to reveal such information. According to Intel's document [63], a user can query the `CAPID6` register to learn the ID of the tile whose CHA is disabled. When CHA is disabled, the whole tile including the core and LLC inside are also disabled. Take Xeon Scalable 8260 as an example. Its `CAPID6` contains 28 bits to indicate the status of all tiles, and a CHA is disabled if its associated bit is 0. By reading all bits of `CAPID6`, we found bit 2, 3, 21 and 27 are set to 0. A previous research [64] suggests tiles are numbered from north to south on each column and the west column is the smallest, so we number all the tiles and mark the disabled tiles based on the `CAPID6` bits. Table II shows the tile IDs and the disabled ones (in gray).

Mapping CHAs and Cores to Tiles. Next, we try to infer the relation between the core/CHA IDs and the tile IDs. According to [64], CHAs are sequentially numbered along with the tiles, but when a tile is disabled, the CHA ID is skipped. As such, CHA is numbered from 0-23 for Xeon Scalable 8260, and tile #4 has CHA #2 because tile #2 and #3 are disabled.

UPI	PCIE	PCIE	RLINK	UPI2	PCIE
0	4	9	14	19	24
IMC0	5	10	15	20	IMC1
1	6	11	16	21	25
2	7	12	17	22	26
3	8	13	18	23	27

TABLE II: Layout of Xeon Scalable 8260 CPU. Gray cell indicates the tile is disabled.

UPI	PCIE	PCIE	RLINK	UPI2	PCIE
0, 0	2, 16	7, 19	12, 3	17, 16	21, 17
IMC0	3, 18	8, 2	13, 15	18, 10	IMC1
1,12	4, 1	9, 14	14, 9		22, 11
	5, 13	10, 8	15, 21	19, 22	23, 23
	6, 7	11, 20	16, 4	20, 5	

TABLE III: The IDs of CHAs and cores of Xeon Scalable 8260 CPU.

Regarding cores, the task becomes non-trivial, as they are *not sequentially numbered*. McCalpin proposed a method to infer how the cores are aligned by reading "mesh traffic counters" [64]. However, the author also admitted the result needs to be disambiguated [65]. To improve the accuracy of the inferred layout, we propose a new method. Specifically, we bind a thread to a core (by setting its affinity [66] to the core ID), and use it to access 2GB memory. We monitor the performance counter `LCORE_PMA GV` (Core Power Management Agent Global system state Value) of Intel PMU [63] and found the CHA yields the highest value when it co-locates with the core in the same tile. We assign the core ID to the tile with CHA with the highest `LCORE_PMA GV` reading. We repeat the process for every core, and the layout can be reconstructed. In Table III, we show the inferred CHA and core IDs for Xeon Scalable 8260.

B. Temporal Resolution

The key assumptions of MeshUp are that 1) the probe delay increases when the mesh links are congested; 2) the granularity of memory access is sufficient to introduce noticeable mesh traffic. Here we try to validate these assumptions. We first inspect the delay traces resulting from on-off style memory access. If the access pattern can indeed be recognized, the delay sequence should resemble a square wave, where a rise is related to contention.

Leveraging the result of Section VI-A, we fix a sender process at CHA 0 and force the process to access the memory mapped to LLC slice at CHA 21, which generates mesh traffic along the top horizontal row. The sender process accesses memory for 15us and then rests for 15us periodically (*i.e.*, Duration T = 30us), simulating on-off access. We place 3 eviction-based probes as a receiver at the top horizontal path of the mesh structure, *i.e.*, from CHA 2 to 17, from 17 to 2, and from 7 to 12.

Results. We gradually decrease T (from 30 to 5 and 2) and Figure 5 shows the observed interval traces. As we can see, the traffic made by the sender indeed causes delay increase by up to 50%, and it disappears every time the sender stops

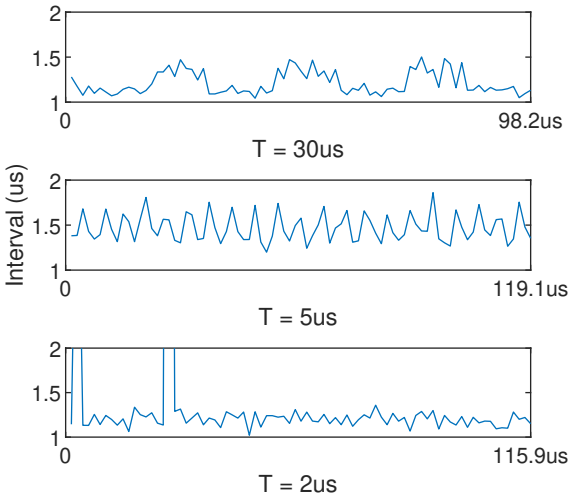

Fig. 5: Delay traces collected by the receiver of different T. The x-axis is the running period of the sender.

accessing the memory. When T is larger than 5us, clear peaks can be observed on the delay trace. However, when T drops to 2us, we are unable to identify those peaks. As such, the access pattern is leaked to mesh links, and memory access lasting longer than 2.5us($T/2$) can be recognized from the delay trace.

On the Intel SP platform, the probe can infer victim activity at the granularity of around 17 memory accesses. One DRAM access takes around 150ns (370 cycles) as reported in [22]. When T is 5us, the access is sustained for only 2.5us as the victim accesses memory in on-off style. Therefore, the granularity is 17 (*i.e.*, 2500/150). Though the resolution is lower than fine-grained probes, like 25ns for PRIME+SCOPE [67], we found it is sufficient to leak secret like encryption key.

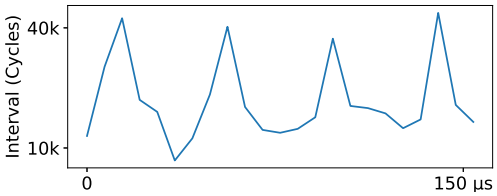

Fig. 6: Delay trace collected by coherence-based probe.

Cross-CPU delays. We also tried to use the coherence-based probe to evaluate the cross-CPU traffic. The sender process accesses the memory of another socket in the on-off style, and a pair of coherence-based probes are placed at each CPU to collect delay traces. When T is 35us, the congestion can be clearly sensed by the probe. Figure 6 shows the delay trace. Though the temporal resolution is worse than the eviction-based probe, the delay variance resulting from UPI congestion can be over 4 times (from 10K cycles to 40K cycles). The variance of the eviction-based probe is around 50%.

Signal to Noise Ratio (SNR). To quantify the quality of the collected traces, we calculate the SNR of each trace. To this end, we first convert a trace from the time domain into the frequency domain with Fast Fourier Transform (FFT). We denote the magnitude value at the square wave frequency (*i.e.*, $\frac{1}{T}$) as signal strength and the average magnitude at other frequencies as noise strength. SNR is their ratio. Figure 10 in Appendix E shows traces of different SNR values under the eviction-based probe. As we can see, for the trace of SNR larger than 10, each bit can be clearly recognized. For the trace with SNR 10, some bits cannot be decoded. When SNR is less than 10, the bits can hardly be decoded.

C. Distribution of Mesh Traffic

The prior evaluation proves the variance of probe delay is related to the program's activities. In this subsection, we try to quantify the distribution of mesh traffic and assess the spatial resolution of the channel.

Distribution by Traffic Types. We place a victim application described in Section VI-B at each tile and access every LLC slice to generate mesh traffic. Then, we select a route that is related to every traffic type (*i.e.*, T1-T7) and place eviction-based probes to collect delays. Table IV left ("w/o LLC slice isolation") shows the result.

It turns out, on the paths from the victim core to the victim LLC slice (*i.e.*, T1 and T2), the delay traces are highly related to the memory access (the average SNRs are all larger than 10). Besides, the probes closer to LLC (*i.e.*, X route for T1 and Y for T2) yield better SNR (the average SNRs are larger than 30 and the ratio of SNR $>$10 traces is more than 0.6). This result shows which LLC is accessed by an application can be discerned, suggesting the MESHUP side-channel has LLC-slice level spatial resolution.

On routes not directly related to core-to-LLC communication, like the route from IMC (T3), the collected delay traces still have non-negligible SNR. Moreover, we found there are delays related to T7, *i.e.*, unknown mesh traffic. One explanation for the existence of these seemingly irrelevant mesh traffic is that CPU might *broadcast* messages under certain events, *e.g.*, credit exhaust messages to notify that congestion is happening.

Distribution by Geometry. Though we assume the attacker cannot select a path at his/her will, contending to victim's mesh traffic by running the application on any tile is feasible, as the mesh traffic should be distributed across the mesh network, hitting different tiles. We validate this assumption through an emulation. We run a victim process to access a chunk of consecutive memory in on-off style, and we pin the process to CHA 9. Then we run the attacker process to probe *all* mesh paths: we enumerate the tiles other than CHA 9, pin the attacker process to the core, and then probe from the core towards all LLC slices.

We compute the max and median SNR over all paths for each assigned attacker core. Table V shows our results. CHA 14 yields the best SNR. For most of the other tiles, the SNR values are sufficient for recovering the access pattern (*i.e.*, SNR over 10), suggesting MESHUP is effective even when a random core is occupied.

Probe Route	w/o LLC slice isolation										w/ LLC slice isolation					
	T1		T2		T3		T4		Others		Others		T3		T4	
	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X	Y	X
Avg. SNR	12.74	38.97	32.74	14.51	4.73	9.34	5.46	8.12	5.36	7.40	3.46	3.36	6.19	4.60	6.53	6.95
Ratio (SNR >10)	0.20	0.72	0.64	0.36	0.05	0.28	0.07	0.21	0.07	0.12	0.00	0.00	0.08	0.04	0.08	0.13

TABLE IV: The SNR of delay traces probed at different links grouped by traffic types. Y and X are the traffic encountered at the Y-route and X-route. The last row shows the ratio of the delay sequences whose SNR are larger than 10.

UPI	PCIE	PCIE	RLINK	UPI2	PCIE
0 27.79 9.74	**2** 21.03 10.30	**7** 30.34 11.74	**12** 23.42 11.92	**17** 15.74 10.24	**21** 18.61 10.67
IMC0	**3** 21.82 11.63	**8** 19.28 11.30	**13** 17.17 9.60	**18** 10.89 7.31	IMC1
1 15.60 9.52	**4** 16.92 11.77	**9** Victim	**14** 49.75 26.25		**22** 23.18 10.37
	5 17.32 12.17	**10** 15.42 7.32	**15** 23.28 13.21	**19** 19.89 12.43	**23** 16.40 8.39
	6 16.47 11.46	**11** 24.25 11.77	**16** 21.57 9.65	**20** 19.35 10.58	

TABLE V: The measured SNR when the attack is pinned to each core. Numbers in every tile represents the CHA ID, maximum and median SNR values respectively.

Events	w/o victim	w/ victim
HORZ_RING_AD_IN_USE	1004151	822149631
HORZ_RING_AK_IN_USE	1952646432	2530753880
HORZ_RING_BL_IN_USE	3914059524	4665246823
HORZ_RING_IV_IN_USE	35639	102218
TxR_HORZ_OCCUPANCY	4414430426	5346595687
AG1_BL_CRD_OCCUPANCY	60996	482979
RxR_OCCUPANCY	4690	141706754
STALL_NO_TxR_HORZ_ CRD_BL_AG1	350	5005
RxR_BUSY_STARVED	9227	41261801
RxR_CRD_STARVED	2041	65177026
TxR_HORZ_STARVED	0	6942
TxR_HORZ_NACK	38951	188482
TxR_VERT_NACK	2	19405478
VNA_CREDIT_RETURN _OCCUPANCY	1301612583	2668519186
M3_CRD_RETURN_BLOCKED	1363707	60557495

TABLE VI: PMU events that are changed most rapidly.

D. Causes of Delay Increase

To obtain an in-depth understanding of the mesh delays, we leverage Intel PMU [63] to record the events related to mesh stops, and use the event counters to identify the root cause. The experiment settings follow Section VI-B.

PMU Event Comparison. We launch and stop the victim application and record the PMU events separately. Table VI lists the PMU events that are changed most among all the events. These events mainly fall into three categories: resources in use, resource starvation, and NACKs. The increase of the first category counters indicates that the related components inside the mesh stop become busier, as the growth of these counters means more time or buffer resources are spent. For example, the increase of `HORZ_RING_XX_IN_USE` indicates that the mesh stops spent more cycles in forwarding packets to horizontal direction; the increase of events that ends with `_OCCUPANCY` suggests more buffers or credits have been occupied.

The second category indicates that the forwarding components get stalled more frequently, because the related resources are used up, which back-pressures the prior sending components. For example, the increase of `STALL_NO_TxR_HORZ_CRD_BL_AG1` means that more Egress buffer of BL ring [63] of agent 1 is stalled because of waiting for credits. In this case, it stops forwarding packets, which will increase the delay of the transactions held in the components.

The third type indicates that the number of packets gets lost due to congestion. For example, `TxR_HORZ_NACK` counts how many Egress packets have not got responded to the horizontal ring, and NACKs were received. Packet loss would sharply increase the network delay and cost considerable time to recover under re-transmission.

Based on the observation, we conjecture that when the two mesh flows (*i.e.*, attack and victim) go through a mesh stop, components inside mesh stops become busier, or even stalled because of the shortage of credits and buffer, resulting in packet losses and delay increase.

We found the similar patterns also exist for the coherence-based probe. In particular, the VNA credits and M3UPI (the interface between mesh and UPI) credits have seen a significant increase (see the last two rows of Table VI).

E. LLC Slice Isolation

MESHUP mainly targets the cross-tile traffic related to LLC. Intel CAT allocates different cache ways to different applications to implement cache isolation, but it does not partition the cross-tile LLC accesses. If cache partition can be done at the level of LLC slice, *e.g.*, placing the data frequently accessed by an application to the LLC slice local to its occupied cores, MESHUP might be deterred. Farshin *et al.* [21] designed a slice-aware memory management mechanism and showed it can realize cache partition. Here, we evaluate this idea under the attack of MESHUP.

In particular, we create a victim application that only accesses its *local LLC* in on-off style, and let the eviction-based probe collect delay sequences on other routes. The code of real-world applications can be changed to implement this idea: *e.g.*, reading `LCORE_PMA GV` when accessing the

memory and conducting sensitive operations only when the memory is mapped to the application's local LLC slices. In this case, the mesh traffic from the victim is minimized, so the contention with the attacker's probes is unlikely to happen.

Table IV right shows the SNR by traffic types when this defense is deployed. Though no more T1-T2 traffic, T3-T4 traffic still carries meaningful information (*e.g.*, the SNR of 13% X-route T4 traffic is larger than 10). The reason is that the victim core cannot terminate the communication with IMC, which exposes the access patterns to memory. As a result, we conclude LLC slice isolation does not mitigate MESHUP.

F. Insights into Mesh Traffic

Limited forwarding capability causes congestion on mesh stops. A CPU core can execute more than one load/store instruction per cycle, as there are multiple ports coming with load store unit, indicating that the memory sub-system potentially can issue more than one cache line (64 bytes) per cycle. However, according to [68], each mesh link has only 32-byte wide bi-directional physical data bus (*i.e.*, BL ring), which can be easily saturated by one core. What's worse, a mesh stop serves requests from cores coming from all four directions, so a mesh link can be severely congested when there is more incoming traffic than outgoing traffic.

Congestion causes extra delays. Like routers in computer networks, mesh stops queue the packets to be processed in the buffer. But unlike routers that drop packets as they will, Intel adopts credit-based flow control for mesh [63], which is more reliable. When a mesh stop is unable to process more incoming packets, it back-pressures the senders by cutting down their credits. In this case, the packets will be queued in the sender, and suffer from extra delays. In Section VI-D, we show that MESHUP can indeed cause extra cycles, credit exhaust, and even packet NACK in mesh stops, which results in up to 50% delay increase for mesh.

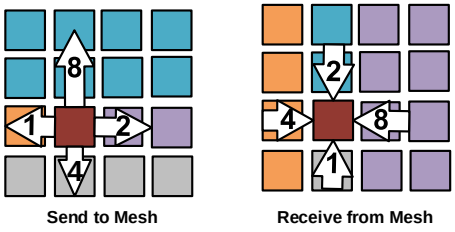

Fig. 7: Mesh traffic to and from a core. The number inside the arrow shows the number of tiles that may receive/send traffic, which is derived according to the policy of YX routing and the hash function that decides the home of an address.

Uneven traffic distribution. Mesh traffic is unevenly distributed across the mesh network. To maximize the overall throughput of LLC and avoid the bottleneck in LLC slices, Intel uses the hash algorithm such that each LLC slice serves an address with equal chances. As a consequence, the traffic may be more congested at some links *near* the victim core, as they serve more LLC slices. For example, for a victim at the southwest core, more than half of the incoming traffic may be carried by the right-hand tile of the victim, and more than half of the outgoing traffic will go through the north tile, which is illustrated in Figure 7. This aligns with our observation in Table V, as the right side of the victim tile has the most tiles and CHA 14 carries most of the traffic (T2). For T3 and T4, tiles near IMC may carry more transit traffics. For T6, tiles near the PCIe stop may carry more transit traffics. Cross-socket T5 will influence tiles near UPI stops.

VII. ATTACKING REAL-WORLD APPLICATIONS

In this section, we evaluate Phase 2 (Secret Inference) described in Section III-B. We choose JDK RSA implementations as the primary target to evaluate MESHUP within a CPU. Besides, we choose app fingerprinting to evaluate MESHUP across CPUs.

A. Experiment Settings

CPU	Intel Xeon Scalable 8260 / 8175*2
Main board	Intel C621
Memory	64GB
OS	Ubuntu 18.04
JDK version	11
Chrome version	89.0.4389.82
CAT Version	1.2.0-1

TABLE VII: Hardware and software for the attack evaluation.

Table VII shows the hardware and software specification of our experiment platform. We use 8260 to evaluate the eviction-based probe, and two-way 8175 for the coherence-based probe.

To validate our claim that MESHUP bypasses cache partition, we turn on Intel CAT for the entire experiment duration. This is the common setting for other side-channel attack works like Xlate [3], but Intel CAT does not represent the strongest defense, and we acknowledge this limitation of setting in Section VIII. For Intel CAT, we create two COS. The core running the victim program is bond with COS 1, while other cores are bond with COS 2, by using the command `pqos` of `intel-cmt-cat` package [69] (with parameter `-e`). COS 1 is exclusively allocated with one way of LLC while the rest 10 ways are allocated to COS 2 exclusively. Therefore, the attacker program will not share any L1/L2/LLC cache with the victim program.

For the attack within a CPU, we evaluate MESHUP against RSA encryption and choose its Java implementation because Java yields more distinguishable patterns of mesh traffic, compared to other languages without automated memory management, like C++. The Java Virtual Machine (JVM) creates a new *object* for the same variable for each iteration within the loop, and uses Garbage Collection (GC) to manage the old object, which triggers frequent memory access. In the meantime, huge integer *arrays* are reused across loops, e.g., `mult`, and their addresses are constant, which produces stable contention patterns. We assume a 2048-bit private key is used. The attacker aims to infer the bit sequence of the

private key. We tested the official JDK implementation of RSA (`javax.crypto.Cipher`) as the victim. We assume the victim is assigned to a randomly selected core for RSA encryption, and the attacker is also assigned to a random core for probing. This setting aligns with the cloud environment, as both victims and attackers have no control over core selection. The experiment was repeated 1,000 times and each run uses a random RSA key, so each delay trace collected by the attacker is unique.

For the cross-CPU attack, we choose application (or app) fingerprinting to evaluate the coherence-based probes. We assume the attacker is interested in learning what app is running on a machine, so he/she launches MESHUP to probe the UPI bus. The attacker employs a DNN classifier for secret inference. Gulmezoglu *et al.* also launched a cross-CPU attack for app fingerprinting (the exploited channel is directory protocol, different from MESHUP) [70], and used 40 apps as the test suite (see Appendix F). We use the same test suite. The attacker runs each app 50 times and each run lasts for 5 minutes. Under each app, 38 traces are used for training and the remaining 12 are for testing.

B. Attacking Sliding Window RSA

Sliding Window is a popular implementation of RSA (e.g., GnuPG has adopted Sliding Window after version 1.4.13). Compared to the older Fast Modular Exponentiation, it has better efficiency and partially fixes the side-channel vulnerabilities. In particular, Sliding Window decouples key bit stream from `mul`/`sqr` execution sequence, so learning the occurrences of `mul`/`sqr` with timing side-channel does not let the attacker directly learn the key bits. However, a recent work [12] showed that `mul`/`sqr` execution sequence can still be utilized to crack Sliding Window RSA. Given a `mul`/`sqr` sequence, their algorithm (*Sliding right into disaster*, or *SRID* for short) is able to either output the 100% correct inference for a key bit (*i.e.*, either 0 or 1), or output X, meaning the algorithm is unable to get a correct inference. When using SRID to crack 2048-bit RSA key, 5-bit sliding window RSA implementation leaks over 33% of the key bits. JDK uses 7-bit sliding window, in which case around 30% bits are expected to be recovered[2]. MESHUP can make full use of SRID to recover key bits. The attacker needs only recover `mul`/`sqr` sequence, then SRID is applied to output 0, 1 and x.

LoR has tested the eviction-based probe on CPU ring against RSA Fast Modular Exponentiation, and achieved 90% accuracy (with prefetchers on) for key bits recovery [8]. As a comparison, we also tested MESHUP against RSA Fast Modular Exponentiation, and the details are elaborated in Appendix G.

Decoding the delay traces. After analyzing the collected delay traces, we found they have discernable patterns and fall into three categories, Pattern A/B/C, as shown in Figure 8.

[2]For the left-to-right algorithm, the one used by JDK, 28% bits can be recovered for each iteration when windows size is 7 [12]. Multiple iterations increase the number of recovered bits, which makes it possible to recover around 30% bits, as we show later.

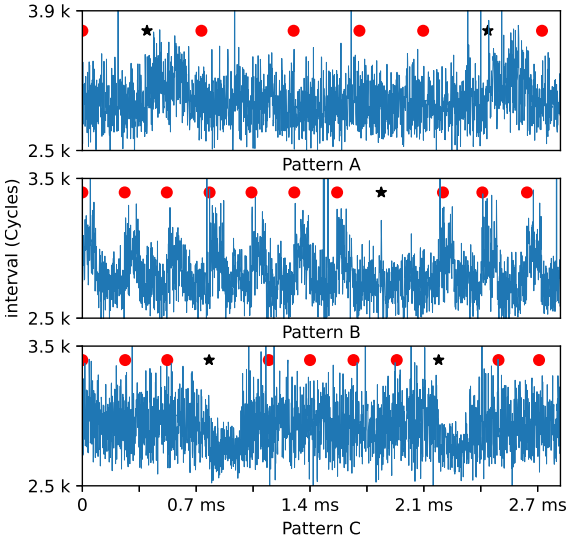

Fig. 8: A delay sequence collected for Sliding Window RSA. Black stars represent ground truth positions of `mul`, and red dots represent the positions of `sqr`.

For Pattern A, each `mul` incurs an obvious rise, while for Pattern C, each `mul` incurs a discernible valley. For Pattern B, each `sqr` incurs a rise. Because the interval of the rises are different, their peaks in the frequency spectrum are different. If a sequence has peaks at around 10.5 kHz, we consider it Pattern B, and 8 kHz for Pattern A and C. Pattern A and C can be distinguished as A has many rises while C has valleys.

As such, our decoder is implemented as a script to find the traces that match Pattern A/B/C and then infer the key bits from them. It 1) clips each samples to a range; 2) smooths the trace; 3) finds peaks (valleys) and calculate how many `mul`s or `sqr`s are between the found peaks (valleys), 4) selects the traces matching Pattern A/B/C, 5) launches SRID on the `mul`/`sqr` traces to infer key bits.

	`mul`/`sqr` Sequence Recovery			SRID	
Pattern	#	Acc_1	Perfect	Acc_2	Acc_3
A	5	99.0%	1	31.1%	23.5%
B	17	98.8%	10	30.7%	25.2%
C	34	96.1%	3	30.8%	20.4%
All	56	97.2%	14	30.8%	22.1%

TABLE VIII: Sliding Window RSA key bits recovery results. "#" is for the number of pattern traces. The slight differences on ACC_2 are attributed to the use of random keys.

Results. We define three evaluation metrics for this experiment. For a sequences exhibit Pattern A/B/C (called pattern sequence), a `mul`/`sqr` sequence (called `ms` sequence) is recovered, and its Largest Common String (LCS) is derived by matching the ground-truth. ACC_1 is computed as $\sum_{i=1}^{N_{MS}} \frac{LCS_i}{MS_i}$, where LCS_i and MS_i are the lengths of ith LCS and `ms` sequence, and N_{MS} is the number of all `ms`

sequence. On a ms sequence, SRID is executed. Assume the number of all key bits is K, and correctly recovered key bits for MS_i is C_i. For the perfectly recovered ms sequences (counted N_P), we define ACC_2 as $\sum_{i=1}^{N_P} \frac{C_i}{K}$. For all sequences, we define ACC_3 as $\sum_{i=1}^{N_{MS}} \frac{C_i}{K}$.

Table VIII summarizes the overall results and the results by Pattern A/B/C. Among all these 1,000 traces, we found 5, 17, and 34 traces exhibit Pattern A, B, and C. So in total 56 sequences can be analyzed with SRID. The remaining traces (944) do not exhibit discernible patterns because their SNR is not sufficient. We tried to recover the ms sequences from the 56 useful traces, and found 14 can be recovered perfectly. Assuming the victim repeatedly runs RSA and the attacker keeps profiling mesh traffic, it will take on average 18 rounds to get a ms, or 71 (1,000/14) rounds to get a perfect ms sequence. The overall ACC_1 is 97.2%, suggesting mul and sqr can be derived at high accuracy from the pattern traces. Regarding the result on each pattern, we found the accuracy differs. For Pattern B, the attacker has 59% chances (10 out of 17) to obtain a perfect ms sequence. Therefore, the attacker can focus on Pattern B when enough traces are observed.

Then, we tested the SRID algorithm on the perfect and ms sequences. In average 30.8% key bits (ACC_2) can be inferred from the 14 perfect ms sequences. The accuracy drops to 22.1% (ACC_3) when all 56 ms sequences are considered. A recent work [71] pointed out that SRID algorithm can be improved for better recovery ratio, and we believe more key bits can be recovered upon it.

Background Workload. The previous experiment is conducted in an environment with no prominent background workload. Here we evaluate how normal background workload (or noises) would impact the delay traces and inference accuracy. To produce the background noise, we run snapd, sshd, tmux, and a docker with Ubuntu image concurrently with RSA encryption on the server. Besides, the server runs the default website of Apache HTTP server, and another machine repeatedly visits the website. The attacker launches the attack against RSA as described in Section VII-B.

Table IX shows the results (columns with "N"), and fewer pattern traces are observed (dropped from 56 to 40), mainly because the noises interfere with the probe and make the delay traces be filtered out by our script. Yet, ACC_1 is similar (97.1% vs. 97.2% in Table VIII). Hence, we expect the attack can be successful in the noisy environment when more delay traces are collected.

Pattern	# N, CR	Acc_1 N, CR	Perfect N, CR	Acc_3 N, CR
A	2, 3	96.2%, 99.5%	0, 2	21.0%, 30.1%
B	14, 11	98.5%, 99.0%	4, 5	22.8%, 26.7%
C	24, 18	96.3%, 95.1%	1, 1	20.8%, 19.7%
All	40, 32	97.1%, 96.9%	5, 8	21.5%, 23.1%

TABLE IX: Results with background workload (columns "N") and core-remapping (columns "CR").

Core Remapping. In the previous evaluation, we fixed the victim at a core when RSA encryption is executed. On cloud platforms, neither the attacker nor the victim has control over core allocation, as the OS or hypervisor may remap tasks to other cores silently [72]. The core remapping process may introduce noises, and we evaluate the impact here.

First, we try to estimate the frequency of core remapping. We ran the attack of Section VII-B in a VM hosted by QEMU, and observed that core remapping has happened 23 times in 10,395 seconds, meaning that core remapping happens every 7.5 minutes on average, which is far longer than a round of RSA encryption. Therefore, most delay traces would not be impacted. Then, we estimate how many bits can be recovered from the traces when core remapping is happening. Initially, we tried to force the OS to remap the victim task to another core with taskset, but found Linux does not remap a task under this instruction. Then, we tried to launch a large number (e.g., 72) of dummy threads, which forces OS to schedule tasks among cores. We found core remapping happens 52 times during the 1,000 runs of RSA, which means 52 traces are directly polluted by core remapping while the rest are indirectly interfered. In Table IX (columns with "CR"), we can see that the results are similar as the case with background workload. The number of pattern traces is 32, while ACC_1 is 96.9%.

Another OS factor that can impact MESHUP is memory swapping. During evaluation, we did not prevent memory swapping, so noises introduced by it have been taken into consideration.

C. App Fingerprinting

We tested the coherence-based probe for app fingerprinting, with the 40 apps in Phoronix Test Suite used by [70]. Our testbed has two CPUs, so we let the attacker's code runs on two different CPUs and a random core is occupied in each CPU. The victim's app runs on a random core on one of the two CPUs. When collecting delay traces, the attacker probes the UPI bus in a 10ms interval, so 30,000 samples can be collected for each app run (i.e., 5 min). In the pre-processing phase, we remove the abnormal samples (or noises) that are out of the normal range (30000, 600000) and smooth the trace with a window size of 80 samples. We train an RNN classifier with the processed training traces and classify the traces in the testing data. To further reduce the length of the traces sent to the RNN model, we split a trace by window of 120 samples, and each window is convoluted to a 16-dimension vector. Therefore a trace will be compressed to 250 (30000/120) 16-dimension vectors. Compressing the time-series has also been adopted by other works in intrusion detection [73].

In our evaluation, we employ *Bi-directional LSTM with Attention (AttBLSTM)* [74], which has been broadly used to classify time-series, to classify a test trace to an app. Table X shows the model hyper-parameters.

Results. MESHUP has **82.27%** classification accuracy. In comparison, the PRIME-PROBE method leveraged by [70] yields 78% accuracy. We conclude coherence-based probe can achieve satisfactory attack accuracy under the cross-CPU setting. Admittedly, the accuracy is not high enough, mainly

batch size	hidden layer size	drop out	attention layer size
64	384	0.1	512

TABLE X: Hyper-parameters of the AttBLSTM model used for app fingerprinting.

because the execution of those apps comes with a lot of random factors, like random core location and random input. Yet, it is surprising the coarse-grained coherence-based probe has better accuracy than the fine-grained probe [70]. We speculate the rise is resulting from better coverage of cache transactions: our attack can probe transactions from all cache sets of a CPU, while a stateful cache attack can only probe one cache set at a time. The overall cache accessing frequency derived under our probe characterizes an app better.

Website Fingerprinting. We have also tested the coherence-based probe on website fingerprinting. The details are elaborated in Appendix H.

VIII. Discussion

Limitations. 1) To evaluate the eviction-based probe, we use RSA as the victim application, whose mesh traffic mainly belongs to T1-T2. Other applications may show different patterns: *e.g.*, I/O intensive applications could introduce prominent mesh traffic between PCIe stop and LLC slices. We plan to test such applications in the future. 2) MeshUp bypasses the existing defenses at the cost of obtaining coarse-grained information about cache activities. In stateful cache attacks like Prime+Probe, the attacker can precisely evict a cache set shared with a victim and learn the *which* memory address/range accessed by the victim. However, MeshUp at most tells *which LLC slice* the victim accesses. As such, we choose the Java-based RSA implementations which leak more information. 3) We chose Intel CAT as the defense following recent cache side-channel attacks like Xlate [3]. The other stronger defenses like DAWG [75] and temporal isolation [5] are not tested because we cannot find their implementation on Intel CPU (e.g., DAWG requires hardware modification) or the mainstream OSes (e.g., temporal isolation builds on seL4). However, as MeshUp is based on the stateless channel, which admittedly is out of the mitigation scope of stateful protections [5], we expect MeshUp to be effective under these defenses. 4) For the attack against RSA encryption, we used a script to filter the useful traces with hard-coded parameters, but we expect these parameters can be replaced with a learned classifier, e.g., an LSTM model. 5) Our evaluation is done on Intel CPUs. In the future, we plan to test other CPUs with mesh interconnects like ARM Neoverse [18]. 6) For the core remapping experiment, the measurements are not isolating the cases where core mappings change, which might underestimate its impact.

Implications and Defenses. The key takeaway message from our study is that the mesh interconnect on server-grade CPUs introduces stateless cache side-channels. This is counter-intuitive at first sight, but the new interconnect design intertwines the cache lines *on the move* from different applications, introducing new types of resource contention. Together with LoR [8], MeshUp draws a comprehensive picture about the contention side-channel of CPU interconnect, and we hope more attention can be spared on this class of issues.

For defenses, in Section VI-E, we proposed a simple solution that isolates LLC to reduce the SNR for T1-T2 traffic exploited by MeshUp. Yet, the other types of traffic still leak information. On the other hand, if spatial isolation can go a step further to partition interconnect bandwidth, MeshUp might be thwarted. However, as mentioned in [5], "no support for bandwidth partition exists on contemporary mainstream hardware". Though Intel recently proposed a technique named Memory Bandwidth Allocation (MBA) [76], which limits the bandwidth a core can issue to memory, the limit is an approximation and insufficient for threat mitigation [5].

Instead of strong mitigation based on isolation, we believe mechanisms that increase the aggression difficulty is more likely to be adopted. One example is cache randomization. By forcing the mapping between physical addresses and cache set index dynamic and unpredictable, finding the right eviction sets is expected to be more difficult, which could make the probes of MeshUp unstable. However, recent studies [77], [78], [79] have shown the previous approaches like CEASER-S [80] and ScatterCache [81] are broken under new attack methods. Following these discoveries, Song *et al.* proposed to fix the flaws of the existing mechanisms [77] and Saileshwar *et al.* proposed fully associative cache [82]. We plan to evaluate whether the new methods are effective against MeshUp.

IX. Conclusion

In this work, we show stateless cache side-channel on the mesh interconnect, or MeshUp, that can leak memory access patterns of a victim program on server-grade CPUs. The side-channel is different from previous cache side-channels, in that it does not rely on stateful micro-architectural changes made by the victim. Therefore it can bypass the existing defenses based on spatial and temporal isolation. To reveal the consequences of MeshUp, we analyzed RSA encryption and application fingerprinting. The results show that MeshUp is very effective. We believe mesh interconnect opens up new opportunities for security research, and its implications should be further examined.

Acknowledgement

We thank the valuable feedback from the anonymous reviewers, which help us significantly improve this paper. The Fudan authors are supported by NSFC 61802068. The UCI author is partially supported by gift from Microsoft and Cisco.

References

[1] 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)*, 2014, pp. 719–732.

[2] 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)*, 2017, pp. 51–67.

[3] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious management unit: Why stopping cache attacks in software is harder than you think," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 937–954.

[4] 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*. IEEE, 2015, pp. 605–622.

[5] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: the missing os abstraction," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.

[6] K. T. Nguyen, "Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family," https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html, 2016, [Online; accessed 10-August-2020].

[7] 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)*. IEEE, 2016, pp. 406–418.

[8] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring (s): Side channel attacks on the {CPU} on-chip ring interconnect are practical," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[9] Intel, "Intel Xeon Scalable Processors," https://www.intel.com/content/www/us/en/products/processors/xeon/scalable.html, 2020, [Online; accessed 10-August-2020].

[10] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown *et al.*, "Tile64-processor: A 64-core soc with mesh interconnect," in *2008 IEEE International Solid-State Circuits Conference-Digest of Technical Papers*. IEEE, 2008, pp. 88–598.

[11] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob *et al.*, "An 80-tile 1.28 tflops network-on-chip in 65nm cmos," in *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. IEEE, 2007, pp. 98–589.

[12] D. J. Bernstein, J. Breitner, D. Genkin, L. G. Bruinderink, N. Heninger, T. Lange, C. van Vredendaal, and Y. Yarom, "Sliding right into disaster: Left-to-right sliding windows leak," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 555–576.

[13] T. T. Kevin Fogarty, "Chasing profits and Intel, AMD sets sights on data centers," https://www.spglobal.com/marketintelligence/en/news-insights/latest-news-headlines/chasing-profits-and-intel-amd-sets-sights-on-data-centers-57996772, 2020.

[14] J. Gilbert and M. Rowland, "The intel® xeon® processor e5 family architecture, power efficiency, and performance," in *2012 IEEE Hot Chips 24 Symposium (HCS)*. IEEE, 2012, pp. 1–25.

[15] Unknown, "Mesh Interconnect Architecture - Intel," https://en.wikichip.org/wiki/intel/mesh_interconnect_architecture, 2020, [Online; accessed 5-August-2020].

[16] ——, "Intel Unveils 3rd Gen Ice Lake-SP Xeon CPU Family," https://wccftech.com/intel-unveils-ice-lake-sp-xeon-cpu-family-10nm-sunny-cove-cores-28-core-die/, 2020, [Online; accessed 18-August-2020].

[17] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, W. Lee, A. Saraf *et al.*, "A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network," in *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.* IEEE, 2003, pp. 170–171.

[18] J. D. Gelas, "New ARM IP Launched: CMN-600 Interconnect for 128 Cores and DMC-620, an 8Ch DDR4 IMC," https://www.anandtech.com/show/10711/arm-cmn-600-dmc-620-128-cores-8-channel-ddr4, 2020, [Online; accessed 18-August-2020].

[19] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE micro*, vol. 27, no. 5, pp. 15–31, 2007.

[20] Unknown, "Intel Ultra Path Interconnect," https://en.wikipedia.org/wiki/Intel_Ultra_Path_Interconnect, 2021, [Online; accessed 13-April-2021].

[21] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić, "Make the most out of last level cache in intel processors," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.

[22] 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)*. IEEE, 2019, pp. 888–904.

[23] R. Schöne, T. Ilsche, M. Bielert, A. Gocht, and D. Hackenberg, "Energy efficiency features of the intel skylake-sp processor and their impact on performance," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2019, pp. 399–406.

[24] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 990–1003.

[25] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S $ a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 591–604.

[26] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[27] D. Wang, A. Neupane, Z. Qian, N. B. Abu-Ghazaleh, S. V. Krishnamurthy, E. J. Colbert, and P. Yu, "Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries." in *NDSS*, 2019.

[28] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.

[29] mseaborn, "Security: Disallow the x86 "clflush" instruction due to DRAM "rowhammer" problem," https://bugs.chromium.org/p/nativeclient/issues/detail?id=3944, 2020, [Online; accessed 18-August-2020].

[30] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 897–912.

[31] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

[32] N. Lawson, "Side-channel attacks on cryptographic software," *IEEE Security & Privacy*, vol. 7, no. 6, pp. 65–68, 2009.

[33] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "Aslr on the line: Practical cache attacks on the mmu." in *NDSS*, vol. 17, 2017, p. 26.

[34] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 955–972.

[35] S.-Y. Tsai, M. Payer, and Y. Zhang, "Pythia: remote oracles for the masses," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 693–710.

[36] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "Netcat: Practical cache attacks from the network," in *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[37] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 249–266.

[38] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[39] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[40] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 991–1008.

[41] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Understanding and mitigating covert channels through branch predictors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1–23, 2016.

[42] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "{DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 565–581.

[43] J. W. Gray, "On introducing noise into the bus-contention channel," in *Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE, 1993, pp. 90–98.

[44] J. W. Gray III, "Countermeasures and tradeoffs for a class of covert timing channels," Tech. Rep., 1994.

[45] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of computer security*, vol. 1, no. 3-4, pp. 233–254, 1992.

[46] ——, "Lattice scheduling and covert channels," in *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society, 1992, pp. 52–52.

[47] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 870–887.

[48] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 785–800.

[49] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures." in *NDSS*, 2020.

[50] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[51] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud," *IEEE/ACM Transactions on Networking*, vol. 23, no. 2, pp. 603–615, 2014.

[52] Y. Wang and G. E. Suh, "Efficient timing channel protection for on-chip networks," in *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*. IEEE, 2012, pp. 142–151.

[53] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: a secure directory to defeat directory side-channel attacks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 332–345.

[54] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift, "A placement vulnerability study in multi-tenant public clouds," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 913–928.

[55] T. Kim, M. Peinado, and G. Mainar-Ruiz, "{STEALTHMEM}: System-level protection against cache-based side channel attacks in the cloud," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 189–204.

[56] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 871–882.

[57] O. M. Michael Papamichael, "Interconnection Networks," http://www.cs.cmu.edu/~418/lectures/15_interconnects.pdf, 2020, [Online; accessed 18-August-2020].

[58] R. Coker, "Hugepages," https://wiki.debian.org/Hugepages, 2017, [Online; accessed 18-August-2020].

[59] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 39–54.

[60] 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*, 2016, pp. 353–364.

[61] Unknown, "Maximizing multicore processor performance," https://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html, 2021, [Online; accessed 13-April-2021].

[62] ——, "Skylake (server) - Microarchitectures - Intel," https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server), 2020, [Online; accessed 10-August-2020].

[63] Intel, "Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual," https://software.intel.com/content/www/us/en/develop/download/intel-xeon-processor-scalable-memory-family-uncore-performance-monitoring-reference-manual.html, 2020, [Online; accessed 5-August-2020].

[64] J. D. McCalpin, "Topology and Cache Coherence in Knights Landing and Skylake Xeon Processors," https://www.ixpug.org/documents/1524216121knl_skx_topology_coherence_2018-03-23.pptx, 2018, [Online; accessed 10-August-2020].

[65] Intel, "Core and L3 numbering vs physical layout on Xeon Platinum 8160 (and KNL)," https://community.intel.com/t5/Software-Tuning-Performance/Core-and-L3-numbering-vs-physical-layout-on-Xeon-Platinum-8160/td-p/1178596, 2018, [Online; accessed 10-August-2020].

[66] Linux manual page, "pthread_setaffinity_np(3)," https://man7.org/linux/man-pages//man3/pthread_setaffinity_np.3.html, 2020, [Online; accessed 10-August-2020].

[67] 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*, 2021, pp. 2906–2920.

[68] Unknown, "What is the data width of the mesh in skx," https://community.intel.com/t5/Software-Tuning-Performance/What-is-the-data-width-of-the-mesh-in-SKX/td-p/1181487, 2020, [Online; accessed 13-April-2021].

[69] debian, "pqos(8) intel-cmt-cat," https://manpages.debian.org/testing/intel-cmt-cat/pqos.8.en.html, 2020, [Online; accessed 18-August-2020].

[70] B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "Cache-based application detection in the cloud using machine learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 288–300.

[71] K. Oonishi, X. Huang, and N. Kunihiro, "Improved crt-rsa secret key recovery method from sliding window leakage," in *Information Security and Cryptology – ICISC 2019*, J. H. Seo, Ed. Cham: Springer International Publishing, 2020, pp. 278–296.

[72] C. S. Bae, L. Xia, P. Dinda, and J. Lange, "Dynamic adaptive virtual core mapping to improve power, energy, and performance in multi-socket multicores," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, 2012, pp. 247–258.

[73] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: an ensemble of autoencoders for online network intrusion detection," in *NDSS*, 2018.

[74] P. Zhou, W. Shi, J. Tian, Z. Qi, B. Li, H. Hao, and B. Xu, "Attention-based bidirectional long short-term memory networks for relation classification," in *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 2: Short papers)*, 2016, pp. 207–212.

[75] 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)*. IEEE, 2018, pp. 974–987.

[76] Intel, "Introduction to Memory Bandwidth Allocation," https://software.intel.com/content/www/us/en/develop/articles/introduction-to-memory-bandwidth-allocation.html, 2019, [Online; accessed 13-January-2021].

[77] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it," 2020.

[78] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "Casa: End-to-end quantitative security analysis of randomly mapped caches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1110–1123.

[79] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *42th IEEE Symposium on Security and Privacy*, vol. 5, 2021.

[80] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 360–371.

[81] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 675–692.

[82] G. Saileshwar and M. Qureshi, "{MIRAGE}: Mitigating conflict-based cache attacks with a practical fully-associative design," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[83] J. D. McCalpin, "Mapping addresses to l3/cha slices in intel processors," Tech. Rep., 2021.

[84] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen, "Avoiding conflict misses dynamically in large direct-mapped caches," in *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, 1994, pp. 158–170.

[85] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 338–359, 1992.
[86] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: a dynamic cache partitioning system using page coloring," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2014, pp. 381–392.
[87] ARM, "ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition," https://developer.arm.com/documentation/ddi0406/latest/, 2008, [Online; accessed 13-January-2021].
[88] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 217–233.
[89] G. Bouffard, "A generic approach for protecting java card™ smart card against software attacks," Ph.D. dissertation, Limoges, 2014.
[90] Y. Lin, J. Yuan, M. Kolmogorov, M. W. Shen, M. Chaisson, and P. A. Pevzner, "Assembly of long error-prone reads using de bruijn graphs," *Proceedings of the National Academy of Sciences*, vol. 113, no. 52, pp. E8396–E8405, 2016.
[91] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 191–206.
[92] Unknown, "Top sites in china," https://www.alexa.com/topsites/countries/CN, 2021, [Online; accessed 13-April-2021].

APPENDIX

A. Intel Skylake-SP cache Spec

We describe the cache specification of Intel Skylake-SP processors, which we used as the evaluation platform.

Address Mapping. The lowest 6 bits reflect the block offset within a cache line. The bits in the middle indicate the index of the cache set containing the line (bits 15:6 for L2 and 16:6 for LLC). The upper bits form a cache *tag*, which indicates whether the data is in the cache.

As LLC requests are all managed by CHA inside a core, for LLC access, the CPU has to decide which CHA to talk to. The decision is based on a proprietary hash function that is not fully reverse-engineered yet [22], [83].

Cache Structure. For Skylake-SP processors, LLC is designed as *non-inclusive* to the private caches. Before Skylake-SP processors, LLC is *inclusive*, meaning that a cache line in L2 cache has a replicate in LLC. For non-inclusive LLC, an L2 cache line may or may not have a replicate in LLC, which is determined by the cache eviction policy. As a result, Skylake-SP has a much larger effective cache size (the sum of L2 and LLC) compared to the previous generations (LLC only).

<table>
<tr><td>63:17</td><td>16</td><td>15:6</td><td>5:0</td></tr>
<tr><td colspan="2">L2 Tag</td><td>L2 set index</td><td rowspan="3">Offset</td></tr>
<tr><td>LLC Tag</td><td colspan="2">LLC set index</td></tr>
<tr><td>Hash to LLC slice ID</td><td colspan="2"></td></tr>
</table>

Fig. 9: Mapping between memory address (physical) and cache.

B. Defenses against Cache Side-channels

Software Defenses. Since OS controls the allocation of memory to programs, the access to the physically-indexed caches can be isolated along with memory. *Page colouring* takes the advantage of the overlapping bits between cache set index and page index (for virtual-to-physical address translation). Pages can be assigned with different "colours" (i.e., the overlapping bits), and the colour decides which cache *sets* they are mapped to. Therefore, cache accesses are isolated along with memory access. Initially being used to improve system performance [84], [85], [86], page colouring has been re-purposed to build defenses, by isolating the cache that the untrusted code has direct access [55], [56].

	Size	Associative	Set
L1-I	32KB	8-way	64
L1-D	32KB	8-way	64
L2	1 MB	16-way	1024
LLC slice	1.375MB	11-way	2048

TABLE XI: Cache configuration for Skylake-SP, Cascade Lake-SP and Cooper Lake-SP CPU families [62]. L1-I and L1-D are for instruction and data separately.

Hardware Defenses. To provide better cache QoS, Intel has implemented a technique named *Cache Allocation Technology (CAT)* [6]. It allocates different cache *ways* to different COS (class of service). Each core is also associated with one or more COS. A core can access a cache way only when they share at least one common COS. Still, directly enforcing cache isolation with CAT is not straightforward, as the provided COS is limited to 4 or 16. CATalyst [7] adjusted CAT to protect LLC by separating it into a secure and a non-secure partition and forcing the secure partition to be loaded with cache-pinned secure pages. Comparing to CAT, Dynamically Allocated Way Guard (DAWG) provides a more principled isolation mechanism with modification to the existing hardware [75]. On ARM, hardware mechanisms like Cache Lockdown can enable similar protection by pinning whole sets of the L1-I and L1-D caches [87].

Hardware-based cache isolation and its extension are supposed to mitigate the attacks that bypass the software defenses. For example, by partitioning the page table and TLB with CATalyst, XLATE [3] and TLBLEED [34] can be mitigated. To defend against directory-based attack [22], Yan *et al.* proposed SecDir to partition and isolate directories [53].

In addition to CAT, Intel TSX has been used for defense [88]. Intel TSX introduces hardware transactions, in which case transactions would abort if they are interfered. By putting sensitive data and code in a transaction and pinning it to a cache set, cache eviction triggered by adversaries will lead to abort.

Temporal Isolation. Both hardware and software defenses aim at isolating resources spatially, which could not cover all resources. For example, small cache like L1 [3] cannot be isolated due to insufficient page coloring granularity. Ge *et al.* proposed to enforce temporal isolation with OS abstraction [5], so the existing cache side-channels [22], [2] can be mitigated when combining with spatial isolation techniques (hardware and software). However, the defense is only applicable to the seL4 microkernel.

C. `check_conflict` and `find_ev`

According to [4], [22], The `check_conflict` function checks if an address x conflicts with a set of addresses U, by checking if x is evicted when traversing x followed by U. If x is evicted by U, it indicates U conflicts with x, otherwise, it does not. [22] adapted this function to CPUs with non-inclusive LLC, by pushing all lines in an L2 set to LLC before measuring accessing time of x, which reduces false positives and negatives.

According to [22], the `find_ev` function tries to find a minimal EV within a given set of addresses CS. It starts by randomly picking out an address x from CS and assigning the rest addresses in CS'. It then repeatedly deletes addresses from CS' except those addresses making CS' no longer conflict with x. Those addresses should be in the EV. EV could be extended by picking out those addresses in CS but conflict with EV.

D. Layout of Xeon Scalable 8175

Table XII and XIII show the reverse-engineered layout of Xeon Scalable 8175.

UPI	PCIE	PCIE	RLINK	UPI2	PCIE
0	4	9	14	19	24
IMC0	5	10	15	20	IMC1
1	6	11	16	21	25
2	7	12	17	22	26
3	8	13	18	23	27

TABLE XII: The disabled tiles of Xeon Scalable 8175 CPU. Gray cell indicates the tile is disabled, including its core, CHA, SF and LLC. The number in each cell is the ID of tile.

UPI	PCIE	PCIE	RLINK	UPI2	PCIE
0, 0		7, 19	12, 3	17, 16	
IMC0	3, 18	8, 2	13, 15	18, 10	IMC1
	4, 1	9, 14	14, 9	19, 22	22, 11
1, 12	5, 13	10, 8	15, 21	20, 5	23, 23
2, 6	6, 7	11, 20	16, 4	21, 17	

TABLE XIII: Layout of Xeon Scalable 8175 CPU. Gray cell indicates the tile is disabled. The two numbers in each cell indicates the ID of CHA and core respectively.

E. Sequences of different SNR

Figure 10 shows the delay sequences of different SNR.

F. Phoronix Test Suite

We use 40 apps in the test suite of [70] for app fingerprinting. The 40 apps are selected across many categories, like image processing, AI, scientific computing, database, *etc*.. The 40 apps are: Tesseract OCR, Bork, Stockfish, Ebizzy, Sunflow, Mafft, Octave-benchmark, Compress-7zip, Parboil, Npb, Hmmer, Ttsiod-renderer, Postmark, Gimp, Git, Vpxenc, Hpcc, Influxdb, Tinymembench, Stress-ng, Numpy, Tensorflow, Mbw, Ramspeed, Cloverleaf, Askap, Mocassion, Oidn, Mlpack, Cassandra, Av1, Clomp, Arrayfire, Build2, Intel-mlc, Hpl, Lzbench, Osbench, Cloudsuite-ma, Memtier-benchmark.

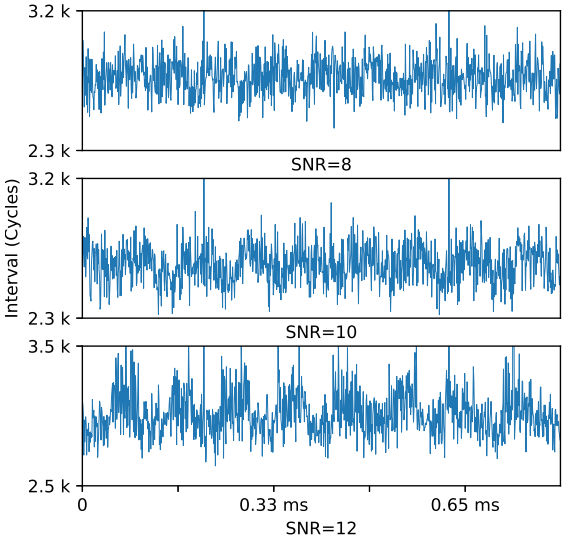

Fig. 10: Delay sequences of different SNR values.

G. Attacking Fast Modular Exponentiation

We evaluate MESHUP with the Fast Modular Exponentiation algorithm implemented in Java following the code found in [89]. This implementation has a timing side-channel. Specifically, different key bit leads to different memory access patterns and different calculation duration. MESHUP is expected to probe the timing difference key bits and recover the entire key sequence.

We assume the victim program runs in a core randomly assigned by the OS. The attacker can then select a core among the rest to construct key paths, according to the characterization in Section IV. We do not randomize the attacker's core for a fair comparison with LoR [8], which fixes the core for attacker. We tested 100 different keys with this RSA program as the victim, and for each key, we let the program run 20 times. Hence, there are in total 2,000 traces collected by the MESHUP probe.

Analyzing the Interval Sequence. Figure 11a shows the interval sequence mapped to the first 8 bits of a d, which is `01001010`. As we can see, the start of each key bit calculation comes with a sharp rise in the collected interval sequence, mainly because of cache misses. Besides, bit 1 takes longer so the interval to the next rise is larger than that of bit 0.

To automatically recognize all the rises and then recover key bits, we use a threshold $(2900, 4200)$ to keep rises first. Figure 11b shows the data points after filtering. Then, the data points are smoothed, *i.e*., taking the average of the points in a window, as shown in Figure 11c. With the smoothed sequence, the attacker starts to find the peaks over 600 cycles, which are expected to be rises. Then, he/she examines the interval between peaks to infer which key bit each rises corresponds to.

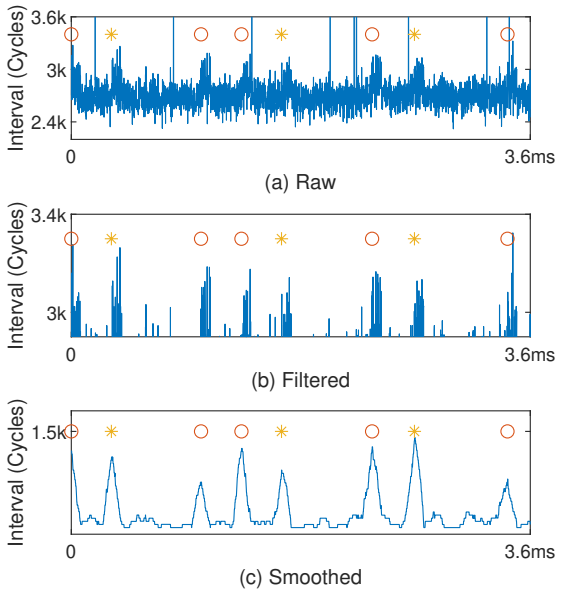

Fig. 11: (a) The raw interval sequence collected by attacker's probe. (b) The sequence after filtering. (c) The sequence after smoothing, with circles and stars representing the starting time of bit 0 and 1. The key stream is `01001010`.

	Edit Distance					Avg. LCS	
	≤10	≤50	≤100	>100	Avg.	Str	Seq
Recovered	17	67	2	14	15.4	760.6	2039.5
Corrected	47	9	2	25	7.8	2040.0	2040.4

TABLE XIV: Results of the RSA key recovery experiment (Fast Modular Exponentiation). "Str" means the longest common sub-string. "Seq" means the longest common subsequence. Average metrics for the recovered cases are computed on the inferred keys whose edit distances are less than 50. For the error-corrected case, we choose 10.

Results. For each key used by the victim program, we compute the edit distances between all the inferred keys with the ground truth, and take the smallest value. Table XIV row "Recovered" shows the distribution of the edit distances. Out of the 100 keys, 17 keys have edit distances less than 10, and 84 (17+67) have at most 50. We look into the 84 keys, and compute the average edit distance and two other metrics: the longest common sub-string (LCSStr) and the longest common subsequence (LCSSeq). The result suggests a large portion of the key has been inferred. For instance, the average LCSStr is 760.6, meaning that a chunk of 760 consecutive bits can be precisely recovered.

Moreover, we found the inference accuracy can be enhanced with error-correction techniques. We chose De Bruijn graph [90], a technique widely used to correct gene sequence errors, for this task. In essence, for a group of long sequences, it breaks each one into sub-sequences and drops the less frequent ones. Then it concatenates the remaining sub-sequences back to a complete sequence. With the De Bruijn graph, for a group of 20 inferred keys, we can correct the errors and generate 1 key. We compare each generated key to the ground truth, and the row "Corrected" of Table XIV shows the result. This time, 47 inferred keys have an edit distance less than 10, and we further compute the average edit distance, LCSStr, and LCSSeq for them. It turns out the average LCSStr can be as high as 2040, meaning that **only 8 bits are incorrectly predicted**.

H. Website Fingerprinting

Since rendering different websites introduces different patterns of network I/Os like webfile downloading [91], which can travel across the UPI (e.g., I/O traffic to NIC), sensitive information about the victim could be leaked. We evaluate such information leakage under MESHUP. In particular, we fix Chrome on the CPU that does not connect to the PCH, and invoke Chrome in headless mode to visit websites without rendering their pages. We select the Alexa top 100 websites [92] as the candidate set. Each website was visited 100 times, during which a pair of coherence-based probes collects the delay traces. For each website, 80 traces are used for training and the rest 20 are for testing.

Like application fingerprinting (Section VII-C), we first remove the abnormal samples and locate the starting point of website rendering (all the parameters like normal range and window size are the same). Since the rendering duration of websites can be different, we pad or clip the traces to 20,000 samples. After that, we also employ AttBLSTM [74] to classify the traces. Table XV shows the hyper-parameters of the model we used.

batch size	hidden layer size	drop out	attention layer size
64	256	0.3	256

TABLE XV: Hyper-parameters of the AttBLSTM model used to website fingerprinting.

Results. The attacker has **82.25%** chances to correctly predict which website a delay trace corresponds to. Besides, the top-3 accuracy can be raised to **92.61%**. Regarding the errors, we believe they are mainly caused by UPI traffic rather than the T6 produced by Chrome (T6 should be the main UPI traffic, as described in Section IV-B), which interferes with the data collected by the probes. In addition, the network connection of our server is not always stable, which varies the downloading duration of web files and introduces noises.