Uncore Encore: Covert Channels Exploiting Uncore Frequency Scaling

Yanan Guo*
University of Pittsburgh
Pittsburgh, PA, USA
yag45@pitt.edu

Dingyuan Cao*
University of Illinois
Urbana-Champaign
Champaign and Urbana, IL, USA
dc29@illinois.edu

Xin Xin
University of Pittsburgh
Pittsburgh, PA, USA
xix59@pitt.edu

Youtao Zhang
University of Pittsburgh
Pittsburgh, PA, USA
zhangyt@cs.pitt.edu

Jun Yang
University of Pittsburgh
Pittsburgh, PA, USA
juy9@pitt.edu

ABSTRACT

Modern processors dynamically adjust clock frequencies and voltages to reduce energy consumption. Recent Intel processors separate the uncore frequency from the core frequency, using Uncore Frequency Scaling (UFS) to adapt the uncore frequency to various workloads. While UFS improves power efficiency, it also introduces security vulnerabilities. In this paper, we study the feasibility of covert channels exploiting UFS. First, we conduct a series of experiments to understand the details of UFS, such as the factors that can cause uncore frequency variations. Then, based on the results, we build the first UFS-based covert channel, UF-variation, which works both across-cores and across-processors. Finally, we analyze the robustness of UF-variation under known defense mechanisms against uncore covert channels, and show that UF-variation remains functional even with those defenses in place.

CCS CONCEPTS

• Security and privacy → Side-channel analysis and countermeasures; **• Computer systems organization → Architectures**; *Multicore architectures.*

KEYWORDS

Security; Side channel; Cache

ACM Reference Format:
Yanan Guo, Dingyuan Cao, Xin Xin, Youtao Zhang, and Jun Yang. 2023. Uncore Encore: Covert Channels Exploiting Uncore Frequency Scaling. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3613424.3614259

*Both authors contributed equally to this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614259

1 INTRODUCTION

Modern processors often contain complex microarchitectural structures that are shared among applications. Although such resource sharing provides efficiency and cost benefits, it also creates opportunities for new attacks that take advantage of these features. One category of such attacks involves software-based covert channels and side channel attacks (e.g., [1, 12, 23, 24, 38, 43, 44, 48, 49, 51, 53, 54, 62, 66]). Specifically, the execution of one application may cause side effects on the states of the shared structures, which can be observed by another application (e.g., through timing variations). Adversaries can utilize this to surreptitiously transfer data (in covert channel scenarios), or infer secrets from a victim process (in side channel scenarios), bypassing sandboxes and traditional privilege boundaries.

Following Intel's terminology, a multi-core processor consists of multiple cores and an uncore. The uncore typically includes the last-level cache (LLC), the on-chip interconnect, the memory controllers (MCs), and other components. Over the last two decades, the microarchitectural resources in both the cores (e.g., branch predictor [2, 17, 18]) and the uncore (e.g., the LLC [35, 42, 63, 64]) have been exploited to mount covert channels (and side channel attacks). However, covert channels based on the uncore components are a more serious threat to the security of modern systems, as the uncore is shared among all the applications running on the processor.

Fortunately, in recent years, there has been a growing focus on uncore covert channels, leading to the development of countermeasures against them (e.g., [3, 5, 27, 31, 47]). Since most uncore covert channels are based on *uncore resource contention/conflict*, partitioning the uncore hardware resources among users is a promising countermeasure approach. Various partitioning strategies with different granularities have been proposed to mitigate different uncore covert channels. For example, inside the uncore of a processor, LLC set partitioning [55] can defend covert channels based on LLC set conflicts (e.g., Prime+Probe [45]); tile partitioning may mitigate covert channels based on interconnect contention (e.g., the mesh contention [11]). In addition, for a multi-processor (multi-socket) system, one can also use a coarse-grained mechanism which assigns each user to a separate processor. In this scenario, each user has its own uncore, and users are not allowed to make cross-socket memory allocations/accesses, resulting in no cross-socket uncore contention. With all these partitioning designs, we ask the following questions:

Can uncore partitioning prevent all uncore covert channels? Can we build a practical uncore covert channel that remains functional even with one or more partitioning mechanisms in place?

There is an ever-growing need for improving power efficiency, as higher power usage directly translates into higher operational costs for data centers. The power consumption of a processor is closely related to its frequency. Thus, adjusting the processor frequency based on the workloads has been widely used on Intel processors to reduce power usage. Earlier Intel processors use a common frequency for the cores and the uncore. On more recent processors, the uncore frequency is controlled independently and can be different than the core frequency. In addition, Intel has introduced a mechanism called uncore frequency scaling (UFS) for their Xeon processors, which adjusts the uncore frequency based on uncore needs [7, 29]. This UFS mechanism, however, may actually introduce practical uncore covert channels that cannot be prevented by uncore partitioning.

In this paper, we conduct a series of experiments to study the detailed behavior of UFS. We have three important observations from the results. First, the power monitoring unit (PMU) continuously monitors the system status, and adjusts the uncore frequency by increasing, decreasing, or maintaining it approximately every 10 ms. Second, when there is low demand for uncore resources, the uncore frequency remains relatively low. There are (at least) two situations that can cause the uncore to operate at a higher frequency: 1) high uncore utilization, such as frequent LLC accesses and dense interconnect traffic, and 2) a significant proportion of active cores being stalled. Third, we found that for a multi-processor system, all the uncores in different processors always maintain similar frequencies.

Next, based on these observations, we propose a new uncore covert channel that can operate as both a cross-core and cross-processor channel on systems with UFS enabled. We name this covert channel UF-variation. Specifically, the sender manipulates the uncore frequency (e.g., by controlling the density of LLC accesses) and encodes the data into the uncore frequency variation within each transmission interval. For example, the sender increases the uncore frequency in the interval to send a bit “1”, and decreases it to send a bit “0”. The receiver then obtains the data by observing the uncore frequency variation within a transmission interval. We found that the LLC access latencies differ significantly when the uncore operates at different frequency levels. Thus, the receiver can indirectly determine the uncore frequency by timing the LLC accesses. We test the channel capacities of UF-variation and show that UF-variation can achieve a capacity of 46 bit/s in the cross-core case, and 31 bit/s in the cross-processor case. Compared to other covert channels, the capacities of UF-variation are limited. However, we show that UF-variation remains functional even with one or more uncore partitioning mechanisms enabled, while most prior uncore covert channels can be prevented by those mechanisms.

Finally, we demonstrate how UFS can be used for side channel attacks to profile the activities of co-located users. For example, when used for website fingerprinting, the UFS-based attack can achieve a top-1 accuracy of 82.18%.

2 BACKGROUND

2.1 Architecture of Intel CPUs

On-chip interconnect. On multi-core processors, an on-chip interconnect is used to connect the processor cores, LLC slices, MCs, and other components (e.g., the PCIe controller). This interconnect facilitates efficient data transmission and coordination between these essential components. Early generations of Intel server-grade processors (Intel Xeon processors) use a ring interconnect (often referred to as a ring bus), allowing data to circulate in a loop-like manner. Recent Intel Intel Xeon processors use a mesh interconnect which has a grid-like layout with multiple horizontal and vertical channels, enabling more direct on-chip communication.

As shown in Figure 2, with a mesh interconnect, the processor chip is structured as a 2D matrix of tiles; each tile can be either a *core tile* which consists of a core (and an LLC+directory slice), or a *controller tile* which consists of an integrated MC. Note that there are three types of CPU dies for Intel Xeon processors based on Skylake: LCC, HCC, and XCC, which represent low, high, and extreme core counts, respectively. The XCC die features 30 tiles (28 core tiles + 2 controller tiles), arranged in a 5×6 grid. However, some tiles might be intentionally disabled[1] by Intel. For example, our Xeon Gold 6142 processor which uses the XCC die, has 16 cores and 16 LLC slices, meaning 12 out of 28 core tiles are disabled (Figure 2).

On an Intel Xeon processor, physical addresses are uniformly distributed to LLC slices using a slice hash function. The specific hash function used in a processor is determined by the number of tiles in the processor. For example, all processors with 28 active core tiles use the same hash function, which has been reverse engineered [46]. Note that an unprivileged user, who cannot access the physical address of a given virtual address, may not directly know the LLC slice a virtual address is mapped to. However, the user can infer this mapping indirectly using timing information, as access latencies (from a specific core) may vary across different LLC slices.
CPU uncore. According to Intel's terminology, a multi-core processor is composed of several cores and an uncore. A core is a logically independent computing unit with ALUs, FPUs, registers, and private caches. The uncore, on the other hand, comprises the components that are not part of the individual cores but are essential for the overall functionality and performance of the processor. Typically, the uncore includes the LLC, on-chip interconnect, MCs, and other components (such as the PMU). The uncore is shared by all the cores on the processor.

2.2 CPU Power Management

Reducing the power consumption of processors (especially server processors) is becoming increasingly vital these days. As a result, Intel has integrated many power efficiency features into its processors. In this section, we focus only on the features that are related to our design.

2.2.1 CPU Frequency Scaling. Intel processors use the common power saving approach, Dynamic Voltage and Frequency Scaling (DVFS) [33], to adjust the frequencies of the cores, based on the workloads. This frequency adjustment works at the granularity of

[1]The routers in the disabled tiles are still functional.

P-states. Each P-state corresponds to a different operating point of the core, in 100 MHz frequency increments. Recent Intel processors offer two mechanisms for P-state selection, namely SpeedStep and SpeedShift. With SpeedStep, the operating system (OS) is responsible for controlling and selecting P-states. In contrast, when SpeedShift is enabled, the P-state selection is controlled by the processor hardware rather than the OS. However, the OS can still give hints to the hardware, such as restricting the range of allowed P-states.

Uncore frequency scaling. Early Intel processors use either a fixed uncore frequency (e.g., for Nehalem and Westmere) or a common frequency for both cores and the uncore (e.g., for Sandy Bridge and Ivy Bridge). Since Haswell, the uncore frequency can be set independently of the core frequencies, and UFS was introduced on Intel Xeon processors to dynamically control the frequency of the uncore, based on the needs for the uncore [29]: UFS increments, decrements, or leaves unchanged uncore frequency based on whether the uncore is under stress, under-utilized, or stable, respectively. This ensures that the uncore components can deliver optimal performance when required, while conserving energy during periods of reduced activity or demand. Unlike DVFS (for cores), the selection of uncore frequency is always managed by hardware using built-in power management algorithms. However, the OS can restrain the uncore frequency selection, through model specific registers (MSRs). Specifically, the OS can specify the maximum and minimum uncore frequencies by writing to `UNCORE_RATIO_LIMIT`, as shown in Figure 1, and the hardware will only adjust the uncore frequency within this range. On our processors specified in Table 1, the default minimum frequency is 1.2 GHz, and the maximum frequency is 2.4 GHz.

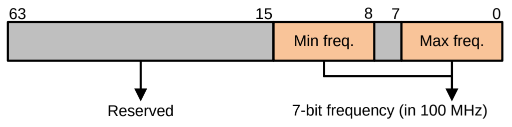


Figure 1: The layout of the uncore freq. limitation register.

The specifics of UFS on Intel processors are undocumented. According to our experiments (and prior studies such as [29]), in general the uncore frequency is dynamically adjusted only when all the active cores are running at a frequency lower than (or equal to) the base frequency[2] (i.e., UFS is enabled). When at least one core is running at a higher frequency, the uncore consistently stays at the maximum frequency specified in `UNCORE_RATIO_LIMIT` (i.e., UFS is disabled). Note that UFS is also disabled if the OS sets the minimum and maximum uncore frequencies to be the same.

2.2.2 *CPU Idle Power Management.*

Computing tasks often involve idle periods, during which the processor core enters a low-power state to save energy. Modern processors support multiple core power states, known as C-states [32]. A specific C-state is denoted as *Cn*, where *n* is the index. *C*0 is the normal operating state where the core is 100% active, while other states (C1 to Cn) represent idle states (also called sleep states) where the core is inactive and some components of the core are powered down. A deeper C-state indicates more powered-down components and better power efficiency; however, it also means that it takes more time for the core to become fully active (i.e., longer *exit latency*). The OS primarily manages the C-state selection, striking a balance between power efficiency and performance. Typically, the OS chooses a C-state based on the intensity of the workloads running on the core. If the workloads are intense, the core is more likely to stay in a shallow C-state during idle periods; otherwise, it stays in a deeper C-state.

When all the cores of a processor are idle, the uncore is also (partially) turned off to further reduce the idle power consumption. Similar to C-states, modern processors support several package C-states (i.e., PC-states), which indicate the power state of the uncore [9]. Again, the uncore is fully active when it is in PC0, with deeper PC-state meaning more uncore components are turned off and there is a longer exit latency for the uncore. The selection of PC-state is usually driven by the selection of C-state. On Intel processors, the PC-state index is *no larger than* the smallest C-state index (among all the cores on the processor). For example, if one or more cores are in C0, the uncore is in PC0.

[2]The specific conditions vary depending on the processor model. For example, the base frequency is included for Intel Xeon E5 v3 processors, but not for Intel Xeon SP processors.

2.3 CPU Uncore Covert Channels

Over the last few years, researchers have proposed many covert channels that exploit the core components. We discuss these covert channels in this section.

Covert channels based on the LLC. In this type of covert channels, the sender intentionally modifies the LLC state to send the data; the receiver then checks the LLC state (e.g., through timing information) to receive the data [8, 25, 26, 28, 40, 42, 53, 65]. For example, in Prime+Probe [42], the data is transmitted through set conflicts. To send a bit "1", the sender loads its own data into an LLC set, evicting the receiver's data in this set; to send a bit "0", the sender does not load the data and the receiver's data remains in this set. The receiver then determines the bit by checking whether its data is evicted from the LLC, based on the access latency of this data.

Covert channels based on the interconnect. As explained in Section 2.1, modern processors use interconnects for on-chip data transmission. Recent work [11, 16, 50, 57] discovered that LLC accesses from different cores may contend for interconnect bandwidth (for both ring and mesh interconnects), resulting in longer access latencies. Thus, the interconnect can be utilized for a covert channel: the sender sends a bit by generating LLC accesses that are transmitted through the interconnect (for "1") or not (for "0"). At the same time, the receiver generates LLC accesses that contend with the sender's accesses on the interconnect, and measures the LLC access latencies to receive the data.

Covert channels based on other uncore components. Some covert channels exploit contention in other uncore components, such as MCs, PMUs, and PCIe controllers [20, 58]. Although the specifics may vary across channels, the overall concept remains the same: contention for limited hardware resources can affect the access time to those resources, enabling covert communication.

Covert channels based on the idle power states. Since multiple PC-states exist, it is possible to encode information into these PC-states and form a covert channel. Specifically, as PC-states are driven

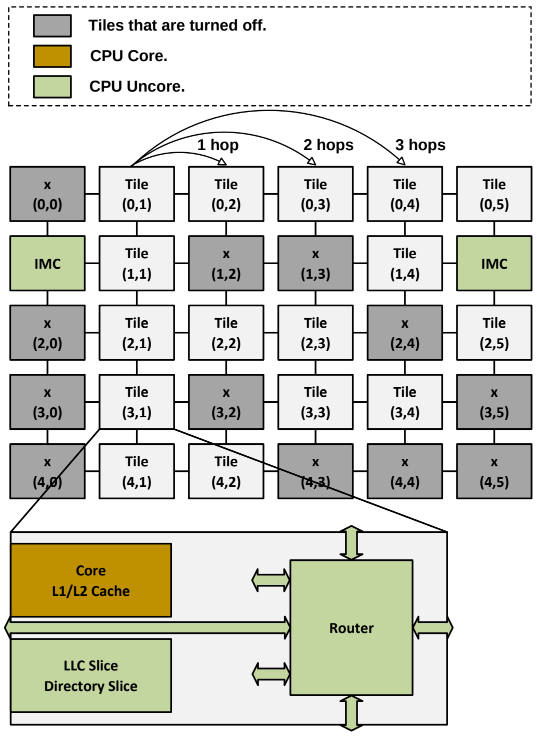


Figure 2: The architecture of our Intel Xeon Gold 6142 processor; the I/O controllers are omitted.

by C-states, the sender can force the uncore to enter a certain PC-state by controlling the workload on a core (assuming there is no other active cores). The receiver can then infer the PC-state by examining the exit latency of the uncore (cf. Section 2.2.2). For example, this latency can be measured through a network interface card: the receiver records the timestamp when a packet arrives (T_1), and the timestamp when the interrupt service routine starts (T_2). Since serving this package requires waking up the uncore and a core, $T_2 - T_1$ is the sum of this core's exit latency and the uncore's exit latency. If the receiver is aware of the core's C-state and its exit latency, the receiver can infer the uncore's exit latency and thus the PC-state from $T_2 - T_1$. Compared to other uncore-based covert channels, this Uncore-idle channel is much less reliable, as it can only work in an "idle" environment. If there are other workloads running on the same processor, keeping at least one core fully active, the uncore stays in PC0, and this channel can no longer function.

3 UFS CHARACTERIZATION

Although UFS can improve the processor's power efficiency, it may also lead to security problems. In this work, we demonstrate that UFS can be exploited to build new uncore covert channels. In order to do this, we must first answer a fundamental question: "how do Intel processors dynamically adjust the uncore frequency using UFS?". For example, we must understand which factors lead to uncore frequency changes. Thus, in this section, we study the details of UFS.

Experiment platform. Unless otherwise specified, all the experiments in this paper are performed on a dual-socket system consisting of two Intel Xeon processors. An overview of this system is given in Table 1. Figure 2 shows the architectural details of one of these processors. Note that the basic architectures of the two processors are the same; however, the tiles that are turned off are different (cf. Section 2.1). Figure 2 corresponds to Processor 0 on our platform; the details of Processor 1 are omitted due to the limited space. UFS is enabled on both processors with the powersave frequency governor chosen. In this section, the uncore frequency is obtained by reading the MSR. Specifically, Intel provides the MSR, `U_PMON_UCLK_FIXED_CTR`, which is incremented by one at each tick of the uncore clock. Thus, by repeatedly reading this MSR, we can indirectly obtain the current uncore frequency.

Table 1: Platform details.

Processor	2× Intel Xeon Gold 6142
Microarchitecture	Skylake-SP
Num of cores	2×16
Core base frequency	2.6 GHz
UFS	1.2-2.4 GHz
L1 cache	8-way associative, private, 32KB+32KB
L2 cache	16-way associative, private, inclusive, 1024KB
LLC	11-way associative, shared, non-inclusive, 22528KB
Operating system	Ubuntu 22.04.1
Frequency driver	Intel_cpufreq
Frequency governor	Powersave

3.1 UFS with LLC/Interconnect Utilization

The idea behind UFS is to adjust the uncore frequency based on the needs for uncore. Naturally, we then expect that the uncore frequency is higher when there is higher uncore utilization. In this section, we conduct experiments to verify this hypothesis.

Experiments. We study the uncore frequency under various uncore utilization levels, focusing on the utilization of the LLC and the interconnect. To control the uncore utilization level, we need to regulate the amount of LLC accesses and interconnect traffic. To achieve this, we use a group of threads to generate *LLC accesses* and pin each thread to a different core on the same processor. All the accesses from the same thread target the same LLC slice, while accesses from different threads target different LLC slices. Then, we can manipulate the uncore utilization by varying two parameters: 1) the total number of threads and 2) the on-chip distance (hops, cf. Figure 2) between the CPU core and the target LLC slice for each thread. The first parameter mainly affects the LLC utilization, with more threads indicating a higher LLC access density. The second parameter mainly affects the interconnect utilization, with a longer core-to-LLC distance indicating more interconnect traffic.

Generate LLC accesses. To create LLC accesses, we must bypass the L2 cache (and L1). We achieve this using eviction lists: we define an eviction list, $EV_j(i)$, as a group of cache lines (addresses) that are mapped into the i^{th} L2 **set**, as well as the j^{th} LLC **slice**. Then, a thread that targets the s^{th} LLC slice operates as follows:

Step 1: Create n eviction lists, $EV_s(0)$ to $EV_s(n-1)$; each list contains m addresses.

Step 2: Repeatedly access the addresses in $EV_s(0)$ to $EV_s(n-1)$ and in each round, alternate the accesses to addresses in different eviction lists, as shown in Listing 1. With proper m and n, all these accesses are likely to miss in the L2 cache and hit in the s^{th} LLC slice (explained below).

```
/* n is the number of eviction lists. */
/* m is the number of addresses in each list. */
/* EV_lists[n][m] is used to store all the eviction
   lists, EVs(0) to EVs(n-1). */
while ((i++) < Total_Rounds) {
    for(j = 0; j < m; j++)
        for(k = 0; k < n; k++)
            memaccess(EV_lists[k][j]);}
```

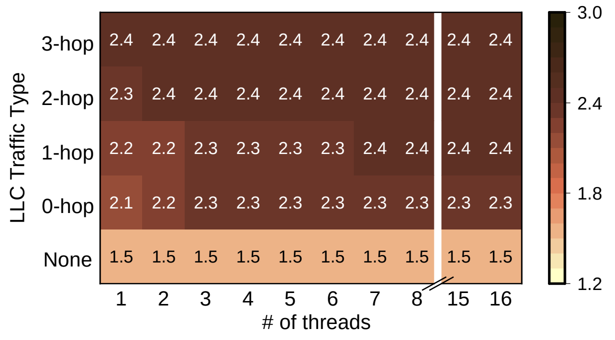
Listing 1: The loop used in each thread to creat LLC accesses (and traffic on the interconnect), referred to as the traffic loop.

Let W_{L2} and W_{LLC} be the associativities of the L2 cache and the LLC, respectively. Then, to ensure that all the accesses in Step 2 are likely to be served by the LLC, m should be large enough (e.g., larger than W_{L2}) to avoid L2 hits, and small enough (e.g., smaller than $W_{L2} + W_{LLC}$) to avoid LLC misses. For our processor where $W_{L2} = 16$ and $W_{LLC} = 11$, we use 20 cache lines in each eviction list, i.e., $m = 20$. In addition, to guarantee L2 misses, the m addresses in the same L2 set (same eviction list) must always be accessed in a fixed order (assuming the LRU policy). To guarantee this order, we use multiple L2 sets (multiple eviction lists) and alternate the accesses to different sets: in this case the accesses to the same L2 set are separated (by accesses to other L2 sets) in the program order, and are thus unlikely to be close enough to get reordered by hardware. Here we use 64 L2 sets (64 eviction lists), i.e., $n = 64$.

Results. We measure the uncore frequency with varying thread counts and core-to-LLC distances. Our results show that when we first launch the thread(s), the uncore frequency adjusts accordingly and eventually stabilizes at a certain level (since each thread executes a loop). The stabilized frequencies are given in Figure 3[3]. First of all, for a given core-to-LLC distance (for all the threads), executing more threads results in higher uncore frequencies. For example, when all the threads are accessing their local LLC slices (i.e., 0-hop traffic), the uncore frequency increases from 2.1 GHz to 2.3 GHz if the thread count increases from 1 to 16. Likewise, given a specific thread count, the uncore frequency is higher if the threads are accessing further LLC slices. In addition, when accessing LLC slices that are 3 hops away from the core, the uncore frequency reaches 2.4 GHz (i.e., the maximum uncore frequency, cf. Table 1), even with just one thread running.

For reference, in Figure 3 we also show the uncore frequency with only L2 accesses and no LLC access. In this scenario, the uncore does not stay at a certain frequency; instead, it alternates between 1.4 GHz and 1.5 GHz. *For simplicity, we refer to this situation as "staying at 1.5 GHz" in the rest of this paper.*

[3]With a given thread count and LLC access type, the uncore frequency still varies slightly depending on the traffic direction.



LLC Traffic Type	1	2	3	4	5	6	7	8	15	16
3-hop	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4
2-hop	2.3	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4
1-hop	2.2	2.2	2.3	2.3	2.3	2.3	2.4	2.4	2.4	2.4
0-hop	2.1	2.2	2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3
None	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5

Figure 3: The median uncore frequencies (in GHz) with different thread counts and LLC access types.

These results confirm that the uncore frequency changes based on the uncore utilization, including both the LLC and the interconnect utilization. Higher utilization results in higher uncore frequency. Without any traffic on the interconnect, the frequency can only go up to 2.3 GHz; in contrast, it can go up to the maximum uncore frequency (2.4 GHz) with interconnect traffic.

```
/* EV_list[m] contains all the addrs in the eviction
   list. */
/* *EV_list[i] = EV_list[i+1], with i in [0, m-2]. */
/* *EV_list[m-1] = EV_list[0]. */
current_addr = EV_list[0];
while ((i++) < Total_Rounds) {
    current_addr = *(current_addr);}
```

Listing 2: The loop used in each thread to stall the core, referred to as the stalling loop.

3.2 UFS with Core Stalling

We conducted several supplementary experiments and studies, in addition to those presented in Section 3.1, to determine if other factors influence uncore frequency changes. It is shown that the uncore frequency is also related to the number of cores that are stalled due to waiting for load or store operations[4].

Experiments. We use a similar approach to the one in Section 3.1. We again launch a group of threads accessing the LLC slices; however, instead of accessing each address independently, we access them through pointer chasing. That is, the data at a pointer address dictates the subsequent pointer address. This ensures that the subsequent load cannot be executed until the current load is completed, i.e., the CPU core is stalled due to waiting for a load to finish. The example code is shown in Listing 2; here we only use one eviction list since the access order is already guaranteed by pointer-chasing.

Results. Our experiments show that with pointer chasing, the uncore frequency always stabilizes at 2.4 GHz, regardless of the thread count and the core-to-LLC distance. This means, the uncore frequency reaches 2.4 GHz even when running just one thread accessing the local LLC slice. Recall that without pointer chasing (as in Section 3.1), with this setup the uncore frequency is only 2.1 GHz. To better understand this difference, we use Linux perf tools

[4]This aligns with the design in Intel's patent [7].

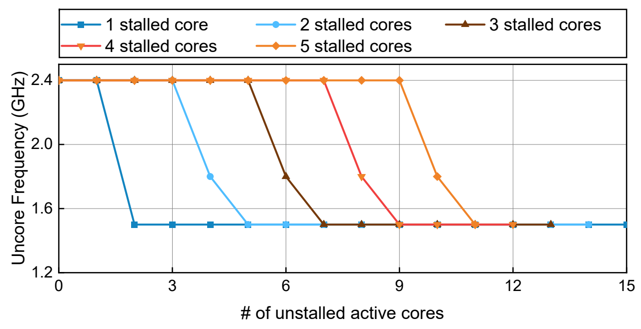


Figure 4: The uncore frequencies based on the number of stalled cores and active but not stalled cores.

to profile the pointer-chasing threads and gather data from two counters: 1) `cycle_activity.stalls_mem_any`, which represents the total time that the execution is stalled due to an outstanding memory operation, and 2) `cycles`, which is the total execution time. The results show that, the ratio of these two data is approximately **0.77** for each pointer-chasing thread. For comparison, this ratio is only about **0.3** for the traffic threads used in Section 3.1. It is notable that if the pointer chasing happens within L2 (no uncore activity), the stalling ratio is **0.14**, and uncore will not boost its frequency. Thus, we hypothesize that the uncore frequency increases (to the maximum uncore frequency) when the stalling time within a given time period for one or more cores surpasses a certain threshold. In the rest of this paper, we use the term "a core is stalled" to indicate that "the stalling time of a core in almost every time period is above this threshold".

In the above experiments, all the threads running on the processor are the pointer-chasing threads. As a result, all the cores that are active are stalled. We found that, the uncore frequency may not reach 2.4 GHz when only some of the active cores are stalled, and others are not. We test this by launching some threads that do not stall the CPU cores alongside the pointer-chasing threads. As shown in Figure 4, when two active cores are stalled, if there are four (or more) other active cores which are not stalled, the uncore frequency stabilizes at 1.8 or 1.5 GHz, rather than 2.4 GHz. Similarly, when three cores are stalled and six (or more) cores are active but not stalled, the uncore frequency is 1.8 or 1.5 GHz. These observations indicate that the uncore frequency is indeed influenced by the proportion of the active cores that are stalled; the uncore frequency only rises to 2.4 GHz if more than **1/3** active cores are stalled.

3.3 UFS Granularity

In the previous experiments, we focused on the stabilized uncore frequencies under specific workloads. However, for building a covert channel with UFS, it is also important to understand the details of the frequency adjustment period (before stabilization). For instance, we need to know the total time it takes for the frequency to rise from 1.5 GHz to 2.4 GHz after a core becomes stalled. In this section, we investigate this aspect.

Frequency increase. We launch a thread that first runs a nop loop, and then switches to a stalling loop (cf. Listing 2). We record the uncore frequency trace while the thread is running, collecting the

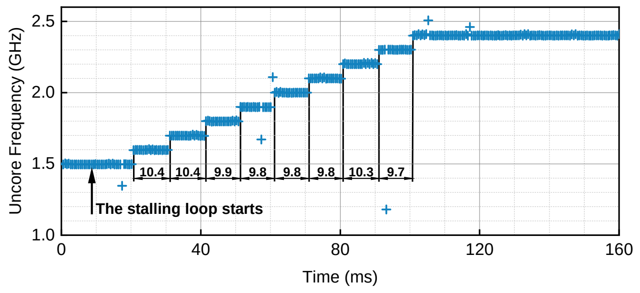


Figure 5: The uncore frequency trace upon initiating the stalling loop.

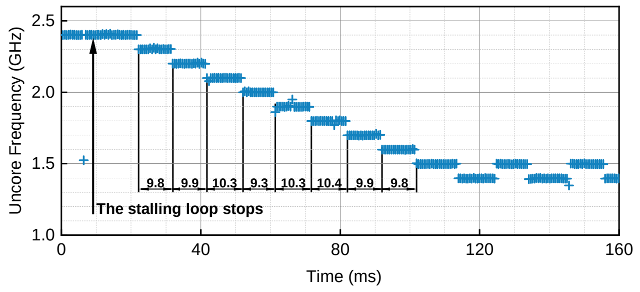


Figure 6: The uncore frequency trace upon stopping the stalling loop.

uncore frequency every 200 μs. The result is shown in Figure 5. Before the stalling loop starts, the uncore frequency is 1.5 GHz. Once the stalling loop starts, the uncore frequency increases by 100 MHz approximately every 10 ms; after the frequency reaches 2.4 GHz, it stabilizes. In addition, we tried launching multiple such threads together and letting each thread access a further LLC slice, but neither of these options can make the uncore frequency increase faster, i.e., it still only changes every 10 ms. Similar results apply when using a traffic loop (cf. Listing 1) instead of the stalling loop. Thus, we believe that the frequency control hardware checks the system status approximately every 10 ms and decides whether and how to update the uncore frequency. Additionally, similar to the P-states for cores, the uncore also has different operating points in 100 MHz frequency increments. Moreover, it takes slightly longer than 10 ms to change from 1.5 GHz to 1.6 GHz. We believe this is because the starting time of the stalling loop is not aligned with the frequency update periods.

Note that in Figure 5, the frequency is 1.5 GHz when the stalling loop starts. Since the uncore frequency alternates between 1.4 GHz and 1.5 GHz with no uncore utilization (cf. Section 3.1), it is also possible that the frequency is 1.4 GHz when the loop starts.

Frequency decrease. We use a similar method to measure how the uncore frequency decreases. Specifically, we launch a thread which first runs a stalling loop and then switches to a nop loop. The recorded uncore frequency trace is shown in Figure 6: once the stalling loop stops, the frequency decreases by 100 MHz every 10 ms, until it reaches 1.5 GHz (and starts to fluctuate around 1.5 GHz). Again, similar results apply when using a traffic loop.

3.4 UFS across Processors

After analyzing UFS within a processor, now we study how UFS works across processors (sockets). Figure 7 shows the uncore frequency traces for both processors when starting a stalling loop on a core of Processor 0. As discussed earlier, the uncore frequency of Processor 0 increases after the loop starts. Interestingly, the uncore frequency of Processor 1 also increases, even though there is nothing running on Processor 1 that can trigger this increment. In addition, the frequency increment on Processor 1 starts about 10 ms later than the increment on Processor 0. Thus, during the frequency adjustment period, the uncore frequency of Processor 1 is always 100 MHz less than the uncore frequency of Processor 0. Eventually, the uncore frequency of Processor 1 stabilizes at 2.3 GHz instead of 2.4 GHz.

We perform further tests where we run different workloads on Processor 0 to make its uncore frequency stay at different levels (e.g., 2.1 GHz). Then we examine the uncore frequency of Processor 1. It turns out that the uncore frequency adjustment on Processor 1 always starts later than the one on Processor 0. In addition, its stabilized frequency is always the same or slightly lower than the one of Processor 0.

3.5 Summary of UFS Behavior

The UFS behavior discussed in this section can be summarized as follows:

- The uncore has different operating points in 100 MHz frequency increments. The system status is checked about every 10 ms to decide whether to increase, decrease, or maintain the uncore frequency.
- The uncore frequency is influenced by the uncore utilization; higher utilization leads to higher frequency (within the allowed frequency range).
- The uncore frequency is also affected by the proportion of active cores stalled due to cache/memory accesses; the uncore remains at the maximum frequency when more than 1/3 of the active cores on the processor are stalled.
- The uncore frequencies of different processors (in the same system) are correlated: when the uncore frequency of one processor increases, the ones of other processors increase as well.

Note that this summary does not represent the complete design of UFS. There might be other factors that can affect the uncore frequency. Our goal is to utilize UFS to build a covert channel, rather than uncovering every detail about UFS.

4 UFS-BASED COVERT CHANNEL

We use the findings in Section 3 to build the first covert channel based on UFS. The basic idea of the sender is to transmit information by manipulating the uncore frequency (through controlling the workload it executes). Simultaneously, the receiver obtains the information by monitoring the uncore frequency. In this section, we first introduce the threat model, and then provide an in-depth discussion of this channel's details.

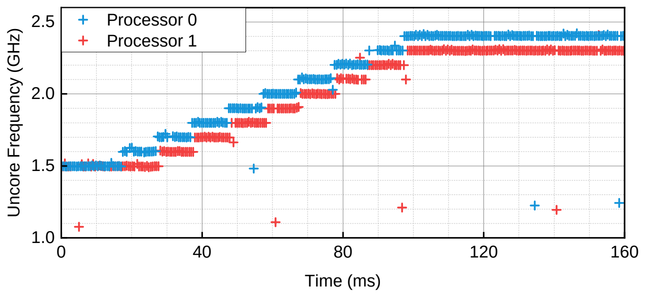


Figure 7: The uncore frequency trace upon initiating the stalling loop, for both processors.

4.1 Threat Model

Like all other covert channels, the UFS-based covert channel involves two parties: the sender and the receiver. We assume that the sender and the receiver are two unprivileged processes or virtual machines that are either 1) running on the same processor (but different cores) or 2) running on the same computing system (but different processors). We also assume the processors in the system are Intel processors that dynamically adjust their uncore frequency using UFS (i.e., UFS enabled). In addition, the sender and the receiver agree on pre-defined channel protocols, such as the synchronization protocol.

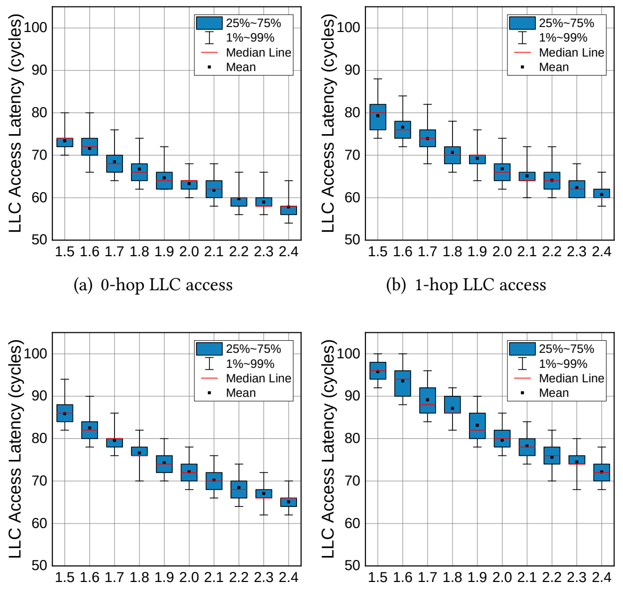
Apart from the above, no additional assumptions are made on the sender or the receiver. For example, we do not assume memory sharing techniques (e.g., page deduplication [4]) or HugePages which are required in many uncore covert channels (e.g., [36, 42, 63, 65]). We also do not require memory allocations/accesses across non-uniform memory access (NUMA) domains, unlike some prior cross-processor covert channels [57].

4.2 Measuring Uncore Frequency

The receiver in the UFS-based covert channel needs to monitor the uncore frequency, in order to receive information. In Section 3, we obtain the uncore frequency by reading the MSR. However, accessing MSRs is generally only allowed for privileged users. Since we do not require the receiver to have privileged permission in the threat model, we need to find a different and more accessible method to probe the uncore frequency.

Our insight is that, the uncore frequency can be obtained indirectly, by measuring the access latencies to uncore components (e.g., the LLC). Intuitively, a lower uncore frequency means that the uncore components are working at a lower speed, resulting in slower accesses to those components (and vice versa). Figure 8 shows the LLC access latencies at various uncore frequencies. We force the uncore to operate at a certain frequency by setting the minimum and maximum uncore frequencies to be the same (cf. Figure 1). It is shown that, for a given LLC slice, the average access latency decreases as the uncore frequency increases. Consequently, the receiver can use the LLC access latency to accurately determine the uncore frequency.

Measurement noise. Listing 3 shows the code snippet of the measurement loop in the receiver: it sequentially accesses every



(a) 0-hop LLC access (b) 1-hop LLC access

(c) 2-cop LLC access (d) 3-hop LLC access

Figure 8: The LLC access latencies under different uncore frequencies; the latencies are measured all on core (3,3). 0-hop latencies, 1-hop latencies, 2-hop latencies, and 3-hop latencies are collected when accessing LLC slice (3,3), LLC slice (2,3), LLC slice (2,2), and LLC slice (2,1), respectively. The latencies are collected in a 10 ms window.

cache line in the eviction list and times the access. All of these accesses should hit in the LLC. Since this measurement loop creates a lot of LLC accesses, it is essential to know how this loop affects the uncore frequency. If running this loop makes the uncore constantly stay at a very high frequency, it will be difficult or even impossible for the sender to manipulate the uncore frequency (to send data). In fact, we found that when only running this loop, the uncore frequency stays low (at 1.5 GHz). This is because the memory fences used in the loop keep the LLC access density relatively low.

```
/* EV_list[m] contains the addrs the evic. list. */
while ((i++) < Total_Measure) {
    for(j = 0; j < m; j++) {
        mfence();
        lfence();
        t1 = rdtscp();
        memaccess(EV_list[j]);
        t2 = rdtscp();
        access_latency[i*m+j] = t2-t1;}}
```

Listing 3: The measurement loop in the receiver.

4.3 UF-variation

In this section, we explain the covert channel in detail. For generality, in the rest of this paper we use `freq_max` to represent the maximum uncore frequency (2.4 GHz on our processor), and use

Algorithm 1: The UF-variation Covert Channel

```
Input: $T_{freq\_max}$: the LLC latency at freq_max.
Input: $T_{freq\_min}$: the LLC latency at freq_min.
Input: message[n]: the n-bit message to be transmitted.
Sender:
    // Algorithm steps for the sender
    for $i = 0; i < n; i++$ do
        sync_channel();
        if message[i] == 1 then
            stalling_loop(); // Or a heavy LLC traffic loop
        end
    end
Receiver :
    // Algorithm steps for the receiver
    for $i = 0; i < n; i++$ do
        sync_channel();
        $T_1$ = measure_avg_LLC_latency();
        wait();
        $T_2$ = measure_avg_LLC_latency();
        if $T_2 < T_1$ or $T_1 = T_2 = T_{freq\_max}$ then
            Received a bit "1";
        end
        if $T_2 > T_1$ or $T_1 = T_2 = T_{freq\_min}$ then
            Received a bit "0";
        end
    end
```

`freq_min` to represent the minimum active uncore frequency (1.5 GHz on our processor).

4.3.1 Channel Protocol. Intuitively, with UFS, the sender can encode the data into the uncore frequency values: to send different data, the sender creates different amounts of LLC traffic (cf. Figure 3) or different levels of core stalling to make the uncore frequency stay at different levels. However, we do not use this approach because it results in a very long transmission interval and thus a limited transmission rate. In Figure 5, the uncore frequency increases by 100 MHz every time the hardware checks the system status (every 10 ms), during the frequency adjustment period. We found that this only happens when we apply heavy LLC traffic or have severely stalled cores (where the stabilized frequency is `freq_max`). In contrast, with lighter LLC traffic or less severely stalled cores where the stabilized frequency is lower than `freq_max`, the uncore frequency is not increased in every 10 ms during the adjustment period. As a result, it takes much longer for the uncore frequency to adjust. For example, when launching one thread accessing the local LLC slice (where the stabilized frequency is 2.1 GHz, cf. Figure 3), it takes over 50 ms for the uncore frequency to even change from 1.5 GHz to 1.6 GHz.

Thus, in our covert channel, we only use heavy LLC traffic or severely stalled cores (to control the uncore frequency). This ensures that the uncore frequency changes frequently (every 10 ms), which allows a shorter transmission interval and thus a higher transmission rate.

The covert channel we propose, named UF-variation, encodes data into the uncore frequency variation. The detailed channel protocol is shown in Algorithm 1. The sender uses the stalling loop (Listing 2) which can severely stall the core to control the uncore frequency[5]. 1-bit of data is transmitted in each transmission interval. To send a bit "1", the sender executes this loop and the uncore frequency increases every 10 ms (unless it's already at `freq_max`). To send "0", the sender does not execute the loop and the frequency decreases every 10 ms (unless it's already at `freq_min`). On the other hand, the receiver monitors the LLC access latencies, and compares the average latency near the beginning of the interval (T1) and the average latency near the end of the interval (T2). If 1) T2 < T1 or 2) both T1 and T2 match the latency at `freq_max`, it means the uncore frequency is increasing or staying at `freq_max` in this interval. Thus, the receiver receives a bit "1". Otherwise, if 1) T2 > T1 or 2) both T1 and T2 match the latency at `freq_min`, it means the uncore frequency is decreasing or staying at `freq_min`, and the receiver gets a bit "0".

In this channel, the transmission interval should be long enough for the frequency to change (i.e., at least 10 ms). Figure 9 provides an example of sending "1101001011" through this channel. In the first interval, the sender sends "1" by executing the stalling loop, the frequency increases from 1.5 to 1.8 GHz and the LLC latency decreases from 79 to 71 cycles. Then in the second interval, the sender continues the stalling loop to send "1", the frequency continues to increase from 1.8 to 2.2 GHz and the LLC latency further decreases from 71 to 63 cycles. In the third interval, the sender sends "0" and stops the stalling loop, the frequency thus decreases from 2.2 to 1.9 GHz and the LLC latency increases from 63 to 68 cycles.

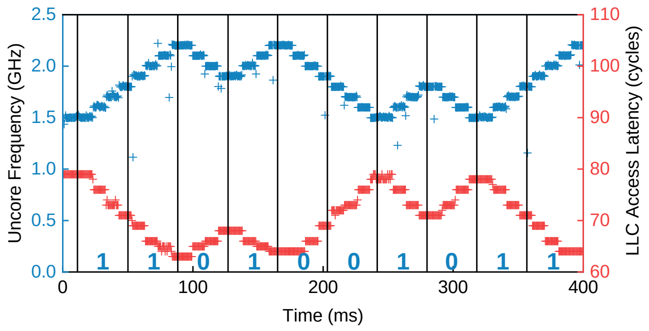


Figure 9: The LLC access latency trace and the corresponding uncore frequency trace when sending "1101001011" through the channel. The transmission interval is 38 ms. The LLC access latencies are 1-hop latencies.

4.3.2 *Channel Capacity.*

In this section, we evaluate the throughput of UF-variation as a cross-core covert channel and a cross-processor covert channel, respectively.

Configuration. We create a proof-of-concept implementation of UF-variation, where the sender and the receiver are single-threaded processes that synchronize using time stamp counters. The receiver calculates the average LLC latencies for the first and last 5 ms in an interval and compares them. We use the metric *channel capacity* (as in [9, 52]) to quantify the throughput performance. It can be calculated by multiplying the raw transmission rate with $(1-H(e))$, where e represents the bit error rate and H denotes the binary entropy function.

Figure 10 shows the channel capacities and bit error rates of UF-variation under different raw transmission rates (i.e., different transmission intervals). When the transmission rate is low (e.g., below 47 bit/s for the cross-core channel), the error rate is very low and remains almost constant. Thus, the channel capacity increases proportionally to the transmission rate. When the transmission rate is higher (i.e., intervals are smaller), the error rate starts to increase, which causes a decrease in the channel capacity. In the cross-core scenario, the channel capacity peaks at 46 bit/s given a transmission rate of 47.6 bit/s (interval of 21 ms). In the cross-processor scenario, the capacity peaks at 31 bit/s given a transmission rate of 33 bit/s (interval of 33 ms). Although the channel capacity of UF-variation is much lower than many prior uncore covert channels, it is effective under a wider range of situations than prior channels. We discuss the details of this later in Section 4.4.

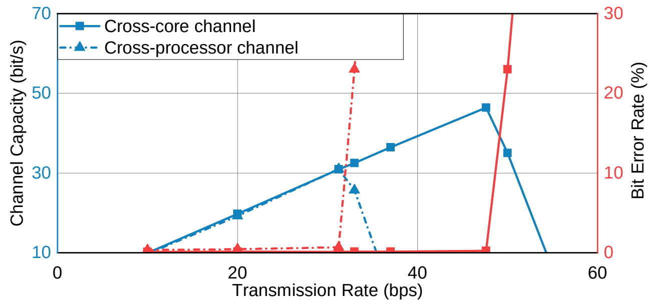


Figure 10: The channel capacities and error rates of UF-variation, in the cross-core and cross-processor scenarios, respectively.

4.3.3 *Channel Reliability.*

Like other uncore covert channels, UF-variation can be affected by noise from other processes running on this system. There are mainly two categories of noise that can influence UF-variation. First, the execution of other processes may affect the proportion of the active cores that are stalled, and thus affect the uncore frequency. For example, in Algorithm 1, the sender only launches one thread and uses the stalling loop to control the uncore frequency. This works well when only the sender and receiver are using the processor: when the sender sends a "1", 1/2 of the active cores are stalled, causing the uncore frequency to rise. However, if there are two threads from other processes running on this processor (which do not stall the cores), only 1/4 active cores are stalled when the sender sends a "1". Consequently, the uncore frequency does not increase, and the receiver cannot differentiate between "1" and "0". Nevertheless, this type of noise can always be avoided by using the traffic loop to control the uncore frequency instead. Moreover, if the sender can access multiple cores, this issue can be resolved by stalling multiple cores simultaneously. For example, on a 16-core processor, if the sender stalls 6 cores, then it is guaranteed that over 1/3 active cores are stalled when the sender sends a "1".

Table 2: The maximum channel capacities of UF-variation (as a cross-core channel) with the stress-ng tool.

stress-ng -N	N=1	N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=9
Capacity (bit/s)	8.6	7.2	6.8	5.1	4.4	3.0	2.4	0.2	~0

Second, other processes with heavy LLC utilization or stalling loops may keep the uncore frequency high even when the sender

[5]The sender can also use a heavy traffic loop (cf. Listing 1) instead of a stalling loop.

Table 3: The comparison of uncore covert channels; ✓ means the channel is functional while ✗ means it is not.

Attack technique	Leakage source	Prerequisites			Defenses			Reliability
		No shared mem.	No `clflush`	No TSX	Random. LLC	Fine partition	Coarse partition	`stress-ng –cache 4`
Flush+Reload [65]		✗	✗	✓	✓	✗	✗	✓
Flush+Flush [25]	Data reuse	✗	✗	✓	✓	✗	✗	✓
Reload+Refresh [8]		✗	✗	✓	✗	✗	✗	✓
Prime+Probe [42]		✓	✓	✓	✗	✗	✗	✓
Prime+Abort [14]	LLC set conflict	✓	✓	✗	✗	✗	✗	✓
SPP [56]		✓	✓	✓	✓	✗	✗	✓
Mesh-contention [11]	Interconnect contention	✓	✓	✓	✓	✗	✗	✓
Ring-contention [50]		✓	✓	✓	✓	✗	✗	✓
IccCoresCovert [30]	PMU contention[1]	✓	✓	✓	✓	✓	✗	✓
Uncore-idle [9]	Idle power control	✓	✓	✓	✓	✓	✓	✗
UF-variation	**UFS**	✓	✓	✓	✓	✓	✓	✓

[1]It cannot be defended by partitioning, but can be defended using a per-core voltage regulator in the PMU.

sends a "0". Here we test the channel capacity of UF-variation while running `stress-ng –cache N` to stress the CPU cache in the background (using `N` threads), similar to prior work [11, 28, 50]. The results are shown in Table 2. The channel is affected by the phases where `stress-ng` keeps the uncore frequency at `freq_max`. The channel capacity is lower when those phases appear more often or last longer. As shown in the table, UF-variation can tolerate the cache stressing when `N < 9`. When `N` is higher, the error rate becomes excessive and the covert channel is no longer functional. We compare the reliability of UF-variation to the reliabilities of other covert channels in Section 4.4.

4.4 Comparison of Uncore Covert Channels

Table 3 compares UF-variation to the existing uncore covert channel techniques based on prerequisites, robustness against defenses, and reliability.

Prerequisites. Covert channels typically have some requirements for the system setup. The most fundamental requirement for an uncore covert channel is co-location, i.e., the sender and receiver are able to run simultaneously on the same system. In addition to this basic prerequisite, some uncore covert channels have further requirements. Common additional requirements include memory sharing, the presence of the `clflush` instruction (or similar instructions), and transactional memory techniques (e.g., Intel TSX). Although these requirements facilitate powerful covert channels in terms of speed and reliability, they also significantly restrict the channel's applicability. For example, memory sharing is discouraged in cloud environments, and special instructions like `clflush` may not be accessible to users in non-native environments (e.g., within a browser).

Effectiveness under defenses. In recent years, numerous defense approaches against uncore covert channels have been proposed. It is expected that real processors will soon implement one or more of these defenses. Thus, it is important to know whether a covert channel remains functional under a specific defense mechanism. The main lines of defense designs are based on randomization and isolation. First, by randomizing the address-to-set mapping in the LLC (e.g., [22]), it becomes challenging to force or observe LLC set conflicts, which are essential for many LLC covert channels. Arguably, a more promising method is partitioning uncore components among users (e.g., [6, 15, 55, 59]), since most uncore covert channels are based on uncore resource contention/conflict. We discuss two types of partitioning mechanisms here.

The first one is a coarse-grained partitioning approach where the sender and receiver are on different processors in the system and the NUMA-strict policy is enforced, i.e., memory allocations/accesses across NUMA domains are not allowed. The second one is a more fine-grained partitioning mechanism, where the sender and receiver can run on the same processor but in different security domains. In this case, all the uncore buffering structures such as LLC slices and queues in the MCs are partitioned among domains: for example, with two domains, each domain is assigned with half of the LLC slices (8 on our processor). Additionally, all the communication paths such as the interconnect work with a time-multiplexed scheduling policy so that traffic from different domains is partitioned and served in different time periods [61], avoiding contention[6].

Reliability. We evaluate the reliabilities of the channels by examining their functionalities while running `stress-ng –cache 4` in the background, i.e., whether the receiver can still distinguish between "1" and "0". Note that we use four stressing threads here because this results in a processor load of 37.5%, which is close to the processor load observed in modern data centers [21].

Comparison. As shown in Table 3, covert channels based on data reuse usually require memory sharing and special-purpose instructions, meaning they can be defended by simply disabling the required features. Covert channels based on LLC set conflicts typically do not have additional requirements; however, most of them can be mitigated by randomized LLC designs (other than SPP), and all of them can be prevented by uncore partitioning. Covert channels based on interconnect contention do not have additional requirements as well, but they also cannot work under either of the two partitioning designs. IccCoresCovert relies on the contention for the voltage regulator in the PMU. Thus, it cannot work under the coarse-grained partitioning where the PMU is no longer shared. In addition, all these above-mentioned covert channels remain functional with four cache stressing threads.

Uncore-idle (cf. Section 2.3) and UF-variation are the only two channels that cannot be stopped by any of the listed defenses. Unlike

[6]We do not assume a spatial partitioning design like Intel Sub-NUMA Clustering [34] for preventing interconnect contention, since the traffic to peripheral devices from different domains may still contend in this scenario.

other uncore channels, these two channels are not based on hardware contention or conflict. Thus, randomizing the LLC mapping or partitioning uncore resources cannot prevent them. However, Uncore-idle which is based on the uncore *idle* power management, is highly susceptible to noise: as long as one core in the entire system is fully active, all the uncores (in all the processors) are active and the covert channel no longer exists. In contrast, UF-variation demonstrates better reliability.

5 SIDE CHANNEL ATTACKS

In this section, we provide a preliminary study on exploiting UFS for side-channel attacks. The two factors that affect the uncore frequency (uncore utilization and core stalling) can both be used to construct side-channel attacks. Here we focus on the latter.

Attack methodology. As explained in Section 3.2, the uncore frequency is related to the proportion of active cores that are stalled. This proportion (and thus the frequency) may change depending on whether the victim's core(s) are active, reflecting the victim's core(s) utilization. Specifically, the attacker executes a stalling thread and a non-stalling thread. Then, when the victim's core(s) are inactive or minimally utilized, the uncore frequency stays at `freq_max`, since more than 1/3 of the active cores are stalled. However, if the victim's core(s) become active (but not stalled), the uncore frequency decreases because less than 1/3 active cores are stalled now. The core(s) activity may be related to some sensitive information of the victim, as shown in prior work [13]. Here we show two example attacks, file size profiling and website fingerprinting.

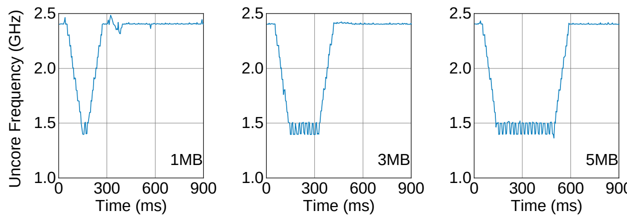


Figure 11: The uncore frequency traces captured while the victim compresses files with varying sizes.

File size profiling. In this case, the victim is executing a python program to compress a file. The total execution time of the program is correlated with the size of the file. As a result, if the attacker determines the execution time of this program by monitoring the uncore frequency, the attacker can deduce the file size and it might even be able to further infer which specific file the victim is compressing. The attacker collects the uncore frequency every 3 ms and the captured frequency traces are shown in Figure 11. When the victim is compressing a smaller file such as a 1MB file, the uncore frequency is only at `freq_min` for a brief period. In contrast, when compressing a larger file (e.g., 5MB), the uncore frequency stays at `freq_min` for much longer time. The attacker is able to distinguish the file sizes at a granularity of 300KB with an accuracy of over 99%. Note that the attacker is running the helper threads (the stalling/non-stalling threads, as explained above) while collecting the trace.

Website fingerprinting. The victim in this attack is a user browsing webpages in a browser, and the attacker aims to determine the website the victim is accessing through the uncore frequency trace. Similar to previous fingerprinting-based attacks [10, 13, 57], we leverage machine learning to develop an attack with two phases: the training phase and the attack phase. In the training phase, the attacker collects uncore frequency traces for 100 websites while accessing them. The attacker then uses these traces to train an RNN classifier. We use the same model and hyperparameters as [57]. During the attack phase, when the victim is accessing a website, the attacker collects the uncore frequency trace and feeds it to the RNN classifier. In both phases, the attacker records the uncore frequency every 3 ms. Note that in both phases the attacker also needs to run the helper threads (the stalling/non-stalling threads), as explained in the attack methodology before.

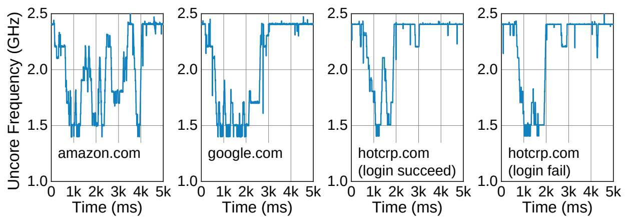


Figure 12: The uncore frequency traces captured while the victim is accessing varying domains.

Figure 12 shows two examples of the collected traces. The top-1 accuracy of our attack is 82.18%, i.e., the attacker has an 82.18% chance of correctly predicting which website a trace corresponds to. The top-5 accuracy, the rate at which the correct website is one of the classifier's top five predictions, is 91.48%. In addition to identifying the accessed website, the attacker can also learn how the website is used. For example, the attacker is able to differentiate between successful and unsuccessful login attempts on hotrcrp.com.

6 DISCUSSION

6.1 Countermeasures

Fixing the uncore frequency. The prerequisite of UF-variation is that the uncore frequency is dynamically adjusted with UFS (based on the running workloads). Thus, to prevent this covert channel, the system software can disable UFS. That is, the system software can set the minimum and maximum uncore frequencies to be the same (cf. Figure 1), forcing the uncore to operate at a fixed frequency (`freq_fix`). However, it may be difficult to determine the value of `freq_fix`. Using a high frequency increases the energy consumption. For example, for graph analytics applications [19], fixing the uncore frequency at `freq_max` increases the energy consumption by 7%. In contrast, using a low uncore frequency reduces the performance. A more desirable method is to randomize the uncore frequency: instead of always using a particular uncore frequency, every certain period of time, the system software randomly selects a frequency (from within the allowed frequency range) to set as the uncore frequency (i.e., `freq_fix`). This can guarantee security while maintaining a balance between the performance and energy consumption.

Restricting the frequency range for UFS. Using a smaller frequency range for UFS, compared to the default range, can mitigate the side-channel attacks. From our experiments, limiting the range for UFS to no larger than 0.2 GHz (e.g., from 1.5 GHz to 1.7 GHz) makes it very difficult to distinguish the uncore frequency traces for different websites. However, this method cannot stop the covert channel (UF-variation). When using a smaller frequency range for UFS, the temporal resolution of UFS remains the same as before (i.e., 10 ms). Further, some of the conditions for triggering frequency increase/decrease also remain the same. For example, with more than 1/3 active cores being stalled, the uncore frequency still increases by 0.1 GHz every 10 ms, until reaching the highest frequency allowed. Thus, the channel capacity of UF-variation remains the same as long as the maximum and minimum frequencies for UFS are not set to be equal (cf. Figure 1).

Maintaining high uncore utilization. The UFS-based covert/side channels can also be prevented by maintaining high uncore utilization: one can use a background thread that is always stressing the uncore to make it stay at `freq_max`.

6.2 Related Work

We have already introduced existing uncore covert channels in Section 2.3. In this section, we discuss prior covert channels (and side channel attacks) based on the core frequency variations.

Channels based on Turbo Boost. Kalmbach et al. [37] discovered that when Turbo Boost is enabled, the maximum core frequency (on Intel processors) is selected based on the number of active cores. Based on this, they built a new covert channel where information is encoded into the maximum core frequency by placing load on a certain amount of cores. Additionally, Wang et al. [60] discovered that with Turbo Boost, the core frequency adjustments depend on the power consumption, which is data dependent. This indicates that the core frequency adjusts based on the data it is processing, resulting in different performance with different data. They built an attack that undermines constant-time cryptography algorithms based on this observation.

Channels based on frequency throttling. Khatamifard et al. [39] found that CPU power management systems dynamically adjust the CPU core frequencies to prevent exceeding the power limits, i.e., the core frequencies are related to the available power headroom. They used this principle to construct new covert channels named POWERT channels. In addition, Liu et al. [41] showed that a privileged adversary that can reduce the power limits can extract AES-NI keys (from an enclave) based on frequency variations.

7 CONCLUSION

In this paper, we presented the first covert channel based on UFS. We showed that the uncore utilization and the proportion of active cores that are stalled are two key factors affecting the uncore frequency. Based on this observation, we developed a new covert channel, UF-variation, with a channel capacity of 46 bit/s. We further showed that although UF-variation has lower capacities than previous uncore covert channels, it remains functional even with uncore partitioning in place, while previous channels do not. Finally, we demonstrated that it is possible to build side channel attacks utilizing UFS.

8 ACKNOWLEDGEMENT

We thank the anonymous MICRO 2023 reviewers for their insightful feedback. We would also like to thank Andrew Zigerelli for his valuable comments. This work is supported in part by US National Science Foundation #2154973, #1910413, #2011146.

REFERENCES

[1] Onur Acıiçmez. 2007. Yet another microarchitectural attack: Exploiting I-cache. In *ACM workshop on Computer security architecture.*

[2] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting secret keys via branch prediction. In *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference.*

[3] Samira Mirbagher Ajorpaz, Daniel Moghimi, Jeffrey Neal Collins, Gilles Pokam, Nael Abu-Ghazaleh, and Dean Tullsen. 2022. EVAX: Towards a Practical, Pro-active & Adaptive Architecture for High Performance & Security. In *55th IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[4] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the linux symposium.*

[5] Zelalem Birhanu Aweke and Todd Austin. 2018. Øzone: Efficient execution with zero timing leakage for modern microarchitectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE).*

[6] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2021. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *USENIX Security Symposium.*

[7] Malini K Bhandaru, Ankush Varma, James R Vash, Monica Wong-Chan, Eric J DeHaemer, Christopher Allan Poirier, Scott P Bobholz, et al. 2016. Dynamically controlling interconnect frequency in a processor. US Patent 9,323,316.

[8] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. 2020. Reload+Refresh: Abusing cache replacement policies to perform stealthy cache attacks. In *USENIX Security Symposium.*

[9] Paizhuo Chen, Lei Li, and Zhice Yang. 2021. Cross-VM and Cross-Processor Covert Channels Exploiting Processor Idle Power Management.. In *USENIX Security Symposium.*

[10] Jack Cook, Jules Drean, Jonathan Behrens, and Mengjia Yan. 2022. There's always a bigger fish: a clarifying analysis of a machine-learning-assisted side-channel attack. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA).*

[11] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. 2022. Don't Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects. In *USENIX Security Symposium.*

[12] Shuwen Deng, Bowen Huang, and Jakub Szefer. 2022. Leaky frontends: Security vulnerabilities in processor frontends. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA).*

[13] Debopriya Roy Dipta and Berk Gulmezoglu. 2022. DF-SCA: Dynamic Frequency Side Channel Attacks are Practical. *arXiv preprint arXiv:2206.13660* (2022).

[14] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *USENIX Security Symposium.*

[15] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 1–21.

[16] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2021. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).*

[17] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[18] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 1–23.

[19] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Acm sigplan notices* 47, 4 (2012), 37–48.

[20] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2016. Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller. In *IEEE International Symposium on High*

Performance Computer Architecture (HPCA).

[21] Peter Garraghan, Paul Townend, and Jie Xu. 2013. An analysis of the server characteristics and resource utilization in google cloud. In *IEEE International Conference on Cloud Engineering (IC2E).*

[22] Lukas Giner, Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss. 2023. Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks. In *IEEE Symposium on Security and Privacy (SP).*

[23] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures.. In *NDSS.*

[24] Ben Gras, Kaveh Razavi, Herbert Bos, Cristiano Giuffrida, et al. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks.. In *USENIX Security Symposium.*

[25] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).*

[26] Yanan Guo, Xin Xin, Youtao Zhang, and Jun Yang. 2022. Leaky way: a conflict-based cache covert channel bypassing set associativity. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[27] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. 2021. Ivcache: Defending cache side channel attacks via invisible accesses. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI (GLSVLSI).*

[28] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. 2022. Adversarial Prefetch: New cross-core cache side channel attacks. In *IEEE Symposium on Security and Privacy (S&P).*

[29] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. 2015. An energy efficiency feature survey of the intel haswell processor. In *IEEE international parallel and distributed processing symposium workshop.*

[30] Jawad Haj-Yahya, Lois Orosa, Jeremie S Kim, Juan Gómez Luna, A Giray Yağlıkçı, Mohammed Alser, Ivan Puddu, and Onur Mutlu. 2021. IChannels: exploiting current management mechanisms to create covert channels in modern processors. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).*

[31] Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd Austin, and Mohit Tiwari. 2019. Cyclone: Detecting Contention-Based Cache Information Leaks Through Cyclic Interference. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 57–72.

[32] Intel. 2023. Intel® 64 and IA-32 architectures optimization reference manual. Available at https://cdrdv2.intel.com/v1/dl/getContent/671488.

[33] Intel. 2023. Intel® 64 and IA-32 architectures software developer's manual. Available at https://cdrdv2.intel.com/v1/dl/getContent/671200.

[34] Intel. 2023. Intel® Xeon® Processor Scalable Family Technical Overview. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html.

[35] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S $ A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *IEEE Symposium on Security and Privacy (S&P).*

[36] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross processor cache attacks. In *ACM on Asia conference on computer and communications security (Asia CCS).*

[37] Manuel Kalmbach, Mathias Gottschlag, Tim Schmidt, and Frank Bellosa. 2020. Turbocc: A practical frequency-based covert channel with intel turbo boost. *arXiv preprint arXiv:2007.07046* (2020).

[38] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2016. A high-resolution side-channel attack on last-level cache. In *Annual Design Automation Conference (DAC).*

[39] S Karen Khatamifard, Longfei Wang, Amitabh Das, Selcuk Kose, and Ulya R Karpuzcu. 2019. Powert channels: A novel class of covert communicationexploiting power management vulnerabilities. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[40] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A. L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path Confidence Based Lookahead Prefetching. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[41] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. 2022. Frequency throttling side-channel attack. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS).*

[42] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (S&P).*

[43] Mulong Luo, Andrew C Myers, and G Edward Suh. 2020. Stealthy tracking of autonomous vehicles with cache side channels. In *USENIX Security Symposium (USENIX Security).*

[44] Mulong Luo, Wenjie Xiong, Geunbae Lee, Yueying Li, Xiaomeng Yang, Amy Zhang, Yuandong Tian, Hsien-Hsin S Lee, and G Edward Suh. 2023. Autocat: Reinforcement learning for automated exploration of cache-timing attacks. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA).*

[45] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS.*

[46] John D. McCalpin. 2018. Address hashing in Intel processors.

[47] Samira Mirbagher-Ajorpaz, Gilles Pokam, Esmaeil Mohammadian-Koruyeh, Elba Garza, Nael Abu-Ghazaleh, and Daniel A Jiménez. 2020. Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[48] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming* (2019).

[49] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference.*

[50] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. 2021. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *USENIX Security Symposium.*

[51] Colin Percival. 2005. Cache missing for fun and profit.

[52] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium.*

[53] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS).*

[54] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. 2021. I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).*

[55] Daniel Townley, Kerem Arıkan, Yu David Liu, Dmitry Ponomarev, and Oğuz Ergin. 2022. Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks. In *31st USENIX Security Symposium.*

[56] Tarunesh Verma, Achilleas Anastasopoulos, and Todd Austin. 2022. These Aren't The Caches You're Looking For: Stochastic Channels on Randomized Caches. In *IEEE International Symposium on Secure and Private Execution Environment Design (SEED).*

[57] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. 2022. MeshUp: Stateless cache side-channel attack on CPU mesh. In *IEEE Symposium on Security and Privacy (SP).*

[58] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. 2014. Timing channel protection for a shared memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA).*

[59] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2016. SecDCP: secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference.*

[60] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power {Side-Channel} Attacks Into Remote Timing Attacks on x86. In *31st USENIX Security Symposium.*

[61] Hassan MG Wassel, Ying Gao, Jason K Oberg, Ted Huffmire, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. 2013. SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 583–594.

[62] Wenjie Xiong and Jakub Szefer. 2020. Leaking information through cache LRU states. In *IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[63] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE Symposium on Security and Privacy (S&P).*

[64] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are Coherence Protocol States Vulnerable to Information Leakage?. In *IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[65] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium.*

[66] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: A timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* (2017).