Cross-VM and Cross-Processor Covert Channels Exploiting Processor Idle Power Management

Paizhuo Chen* Lei Li* Zhice Yang
School of Information Science and Technology, ShanghaiTech University
**Co-primary Authors*

Abstract

To achieve power-efficient computing, processors engage idle power management mechanisms to turn on/off idle components according to the dynamics of the workload. A processor's hardware components are classified and managed through the core and the uncore. The uncore is the supporting hardware shared by the cores, hence the decision of turning it on/off depends on the cores' activities. Such dependency implies a covert channel threat in multi-core platforms. Specifically, the power status of the uncore reflects the workload pattern of the active core, and it can be probed by any process running on the processor. This allows the process to infer the workload information of the active core. We show this covert channel can work across processors and violate VM isolation. We validate the channel in in-house testbeds as well as proprietary cloud servers.

1 Introduction

There is a persistent demand on increasing the power efficiency of computing platforms – for extending the battery life of mobile devices and reducing the cooling cost of servers. Ideally, power-efficient computing proportionally scales the power consumption with the computational workloads [21]. One complexity lying in front of this goal is the idle period, where no workload is waiting for execution but the hardware still consumes power.

Idle periods are ubiquitous in real computational tasks [48], hence processor idle power management [1,38] is proposed to conserve power in those periods. It progressively turns computing components, *e.g.*, the processor cores, off when there is no workload, and brings them back to life when workloads occur. Apart from the cores, a modern multi-core processor contains the uncore, which presents the supporting hardware components shared by the cores, *e.g.*, the last level cache (LLC), the memory controllers, *etc.* When all the processor cores are turned off, the uncore becomes a major source of idle power consumption [20, 30]. This raises an interesting problem for achieving power-efficient computing: on the one hand, it is necessary to have a scheme to effectively reduce the uncore idle power; on the other hand, as the uncore is shared by the cores, the scheme must handle the dependency carefully to avoid any performance degradation of the cores.

Today's processors adopt a straight-forward approach. The uncore is turned off if and only if its functionalities are no longer required. That is, when there is an active core, the uncore stays active to provide its functionalities; whenever there is no core active and the system configuration allows, the uncore is turned off to save power [1]. In other words, due to the nature of sharing, the power status of the uncore is determined by the activity of the cores.

In this paper, we show that, the above idle power management mechanism brings new covert and side channel vulnerabilities to current multi-core and multi-processor platforms. It can be exploited to break down security protection mechanisms relying on resource partition and isolation, *e.g.*, virtual machines (VMs). The revealed covert channel is based on two properties of the uncore power management logic:

- First, as the power status of the uncore is determined by the cores' activities, it in turn reflects the workload pattern of the processes residing on the processor.
- Second, the power status of the uncore affects the responsiveness of the processor, which can be probed by any process on the processor.

The above properties allow two isolated processes to covertly communicate. One process manipulates the uncore power status by applying dedicated workloads, while another process co-locating at the same processor probes the uncore power status to obtain the conveyed information. The properties also imply a side channel, which allows the attacker process to profile the workload information of the co-located processes.

While several vulnerabilities rooted in uncore (*e.g.*, cache [42], memory bus [66]) and processor power management [36, 58] have been separately studied, the uncore idle power management has not been explored and characterized. In this paper, we make the following contributions:

- We study the behavior of uncore idle power management. We empirically show that the uncore power status can be manipulated and probed.
- We leverage the uncore behavior to break down the isolation of VMs for covert communication. We characterize the covert channel through systemic evaluation. Results show that it can achieve up to 1200 bps when the VM host computer is lightly-loaded.
- We show that the channel can be used to profile the activities of the co-located VMs. We demonstrate that the network traffic intensity and the SSH keystroke activity can be correctly recognized.

2 Motivation and Overview

Power efficiency plays an important role in today's computing systems. Real computing tasks contain idle periods but computing hardware consumes power even in these periods. The idle power consumption is caused by various hardware factors [9]. A straightforward solution is to turn off computing resources when there are no tasks and bring them back to work when tasks appear. One factor that complicates this approach is the multi-core/processor architecture, where hardware computing components have functional dependencies.

2.1 Hierarchical Idle Power Management

A computing system may contain several sockets, each of which may contain a processor, each of which may contain multiple cores, each of which may contain multiple hardware threads. It is very typical that the computing units of the same level share some common resources. For example, the cores of a processor may share the LLC, and the processors with Non-Uniform Memory Architecture (NUMA) share their on-die memory controllers for remote memory access. When considering turning off a component, the dependency caused by the sharing must be carefully considered. For example, the LLC/memory controller should only be turned off when no core/processor sharing it is still active.

To resolve the dependency of sharing, it is natural to manage the power of the computing platform on a hierarchical basis [1]. As shown in Figure 1, a node in the tree denotes the shared hardware components at that level. When trying to reduce the idle power of a node, two factors should be considered. *First, to avoid significant performance degradation, a node (containing the shared resources of its descendants) should not turn off when any of its descendants are still active. Second, to reduce the power consumption, a node should always try to turn off whenever it is allowed and brings net energy savings.* We call the above rules **Idle Power Management Dependency (IPMD)**.

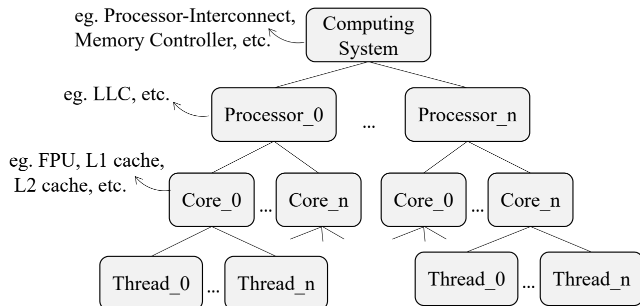


Figure 1: **Hierarchical Idle Power Management**. Multi-core/processor systems manage computing resources hierarchically. Computing units of the same level usually share some resources. A node in the tree represents components shared by its descendant nodes. To avoid performance degradation, the components represented by a node should not turn off if any of its descendant nodes are still active. As a consequence, the activity of a node can affect the power status of its ancestors.

2.2 Motivation of Exploiting IPMD for Covert and Side Channel

IPMD represents a high-level working principle for idle power management mechanisms. It fundamentally exists in general computing systems abiding by the hierarchical power management architecture as seen in Figure 1. Similar to other mechanisms managing shared resources [26], IPMD might also bring about covert/side channel vulnerabilities.

In this paper, we show that it is practically feasible to exploit IPMD for covert/side channels. We refer to this class of channels as the IPMD channel. Its intuition is that, through controlling the workload of a computing unit (*e.g.*, a core node in Figure 1), the idle power status of its ancestor nodes (*e.g.*, the processor node) can be manipulated accordingly. As the ancestor nodes are shared by other computing units (*e.g.*, the cores of the same processor), it is possible to probe the applied workload pattern by other units through probing the power status of their shared ancestors.

The IPMD channel we revealed is timely and unique. Today's power management mechanisms are becoming more and more efficient but complicated, and the balance between efficiency and security has not been well addressed. Very recently, some power management vulnerabilities have been reported [41,58], but the behavior of *idle* power management that the IPMD channel is based on is rarely disclosed [27]. The risks have not been well understood. In the following sections, we first give the necessary technical background (§3) and then characterize the IPMD behavior with in-house testbeds (§4). After that, we show the feasibility of exploiting IPMD for practical covert (§5) and side channel (§6) attacks in cross-processor and cross-VM situations. We also demonstrate them on Amazon AWS and Microsoft Azure cloud servers.

3 Technical Background

Modern processors incorporate several interfaces to govern the idle power. We follow Intel's terminology. A multi-core processor consists of cores and uncore. A core is a logically independent computing unit formed by ALUs, FPUs, and per-core caches. The uncore is shared by the cores and consists of supporting components, such as LLC, memory controller, processor-interconnect, *etc*. The cores' idle power is mainly managed by the core low power idle state (core C-state) mechanism. Compared with the cores, the uncore's idle power management is less transparent and determined by proprietary firmware.

3.1 Core Idle Power Management

When a core is idle, the processor turns it off to save power. In practice, as a core consists of several hardware components and saving power has a variety of ways [9], *e.g.*, turning down the clock frequency, reducing the working voltage, *etc*. Combinations of these approaches and components lead to several intermediate states between absolute on and off. Core C-states (CCs) are defined in the Advanced Configuration and Power Interface (ACPI) specification to describe these states [1]. Almost all modern processors implement this ACPI feature. A specific CC is denoted as CC*n* with an index *n*. CC0 is the active state, while CCs other than CC0 have components turned off/down. Deeper CCs (with larger *n*) save more power but take more time to become fully active [56], *i.e.*, *exit latency* [1]. Exit latency characterizes one of the overheads of switching to the idle states.

Core C-state only specifies which core components are turned off, but does not define when and which CC the processor should select. The control interface of CC is exposed to the software. The OS heuristically selects an appropriate CC and uses the interface to instruct the processor [3]. To balance power saving and performance, the selection algorithm takes two factors into consideration. The first is the statistics of the core's local workloads, which is used to predicate how long the core is likely to be idle in the near future. The second is the overhead. Manufacturers hard-code approximate exit latency values in the OS for the idle power management algorithm [13]. A core should not enter a C-state incurring latency longer than the estimated idle period. At a high level, the effect of the algorithm is when the workloads are intense and/or the interrupts are frequent, the core tends to stay in a light core C-state during its idle periods.

Specifically, the `cpuidle` subsystem [51] implements the above core idle power management mechanism. The low-level control interface is the CPU instructions. `MWAIT` and `HLT` are used to allow the core to enter certain CCs [60]. A userspace interface for configuring the core C-state selection is exposed by the `cpuidle` driver through `sysfs`. Its low-level interface is MSR C-state registers [7]. Several system-level QoS configurations such as `cpu_dma_latency` and BIOS configurations are also implemented through these MSR registers. The time spent in each CC can be monitored through the MSR C-state residency counters [7].

[1] "exit" means that the core exits from the idle state to the active state. Exit latency is caused by many factors [9], including the voltage ramp up, the PLL relock, the state restoration, and the control overhead.

3.2 Uncore Idle Power Management

When all the cores of a processor are idle, the uncore can be turned off to further reduce the idle power. Unlike the core idle power management, several mechanisms, which focus on different parts/aspects of the uncore, jointly govern the uncore idle power.

Package C-state (PC) is the most well-known one. It is also defined in the ACPI specification [1]. As the name suggests, the design philosophy of PC is almost identical to that of the core C-state. There are multiple levels of PCs. Deeper PCs except PC0 turn off more shared components to save power and incur larger exit latency. Unlike CC, the decision of which PC to enter is made by the hardware and firmware rather than the software. While the detailed implementation is not disclosed by vendors, according to the processor datasheet [60] and ACPI, it should follow the IPMD principle. The relation between PCn and CCn is architecture-dependent, but in general, PCs are driven by CCs. In Intel architectures starting from Haswell, the index of PC is always no larger than the smallest CC index. For example, when any core is in CC0, only PC0 is allowed. The numerical relation between PC and CC's indexes does not contain meaningful information except that it restricts the order of how the components are turned off, which reflects the dependency of the processor hardware. For example, the LLC, which is governed by PC6, is allowed to turn off only when all the per-core L1 and L2 caches are turned off, which are governed by CC3.

Package C-state only loosely regulates a part of the uncore idle power. The evidence can be observed through measurements. Briefly, we keep the processor idle and allow its cores into the deepest CC, so its uncore is also in the deepest PC. At the same time, we measure the power of the platform through a digital power meter. We adjust the deepest allowed PC through MSR registers, the power steps accordingly, which is as expected. However, we note that when both the CC and PC are kept unchanged, the power still steps up/down when other configurations related to the uncore are modified. For example, the Uncore Frequency Scaling (UFS) [31] automatically reduces the uncore frequency when the cores are idle [23]. UFS has an observable impact on the uncore power, but it can be configured independently with PC.

The uncore power is not determined by a single configurable factor. This is probably because the uncore consists of multiple components with diverse and independent functionalities, hence it is unnecessary to define a dozen stepping

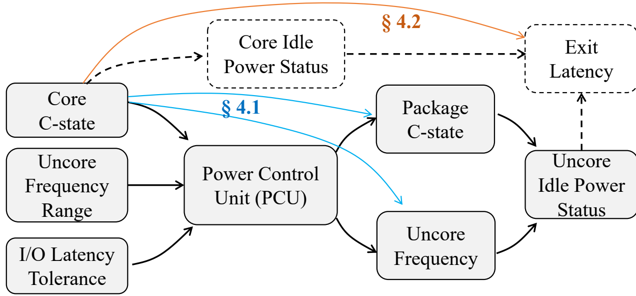


Figure 2: **Uncore Idle Power Management**. The uncore idle power is determined by several factors, including the core C-state, the uncore frequency range, the latency tolerance, *etc*. The PCU of the processor uses this information to select the appropriate package C-state and uncore frequency, which, probably along with other factors, determines the "off" level of the uncore components. The dependency between core and uncore idle power status can be observed indirectly via 1. the hardware statics of package C-state residency and the uncore frequency (the blue arrows, detailed in §4.1), and 2. the exit latency (the orange arrow, detailed in §4.2).

states to describe the "off" levels of the uncore components. The detailed uncore power management logic is implemented in the proprietary firmware in the processor Power Control Unit (PCU).

Current operating systems use various coarse-grained systems and BIOS profiles to control the uncore idle power. Figure 2 shows our understanding of how the uncore idle power is determined. We infer this from related patents [23], drivers [11], datasheets [60], and our measurements. The PCU takes the CC information, the uncore frequency range, and the latency constraints of I/O controllers from the processor. It decides upon an appropriate PC and uncore frequency accordingly, which then determines the power status of uncore components. However, as the detailed logic is not disclosed, it is hard for us to exhaust all the impacting factors in Figure 2. In the next section, we empirically show how cores and uncore interact to realize the IPMD principle in practice.

4 Behavior of Idle Power Management

To exploit IPMD for practical attacks, we face two basic questions. First, is IPMD really realized in practical computing systems? It might not even exist if power efficiency is not seriously emphasized by the computing system. Second, how can IPMD be precisely perceived by a program? We need a practically valid approach. To answer these questions, we conduct the following measurements.

Measurements in this section are conducted with controlled settings. Specifically, GUI and `SSH` are disabled to avoid the impact of the integrated graphics unit and the network traffic. The core frequency is fixed to the base frequency to isolate the impact of Dynamic Voltage and Frequency Scaling (DVFS) [32]. As I/O latency constraints, *i.e.*, Latency Tolerance Reporting (ltr) [11], affect the PC selection, they are all released. We use core_0,.., core_N to denote the physical cores of the processor.

Our tests are mainly carried out on Intel-based platforms, covering desktop, mobile, embedded, and server situations. We use the platform in the first row of Table 2 to present the study. Different platforms differ in the detailed values but the conclusion is the same.

4.1 Observing IPMD via Hardware Statistics

We first investigate how different CCs affect the uncore idle power. We note that although Intel processors provide sensors to measure the power of the chip components, it seems they are not designed for differentiating the core and uncore idle power [2]. An external power meter cannot differentiate the processor components either. Therefore, we use two indirect metrics – the PC index and the uncore frequency (the blue lines in Figure 2) to understand how different CCs affect the uncore idle power.

One method to precisely control CC is through the `cpuidle` sysfile. It controls the deepest allowed CC for each core. We keep the platform idle, hence the deepest CC is also the CC with the dominating residency time. Typically, the core stays at the deepest CC 99% of the time. Next, we dynamically adjust the deepest CC (from CC8 to CC0) of core_0 through the sysfile, and keep the other cores untouched (stay in CC8). By doing so, the dominating CC of core_0 is controlled by us. The reaction of the uncore idle power is observed through the PC residency time and the uncore frequency. The CC and PC residency are logged through the MSR register [7]. The uncore frequency is calculated by counting the uncore clock's ticks for 1 second [8].

As expected, we observe that the index of the dominating PC follows the CC index of core_0 timely and precisely, *i.e.*, the index of dominating PC = the index of core_0's dominating CC. This is because on the one hand, for this specific processor [60], the PC is adjusted to ensure that the PC index is no larger than the minimum CC index to meet the internal hardware dependency. On the other hand, the PC automatically goes into deeper idle states whenever it is allowed to save power. Moreover, the uncore frequency is inversely proportional to the CC index, meaning that the uncore turns down the frequency when more core components are turned off, and vice versa. The above behavior reflects the IPMD principle.

[2]Running Average Power Limit (RAPL) measures the power of the Intel processor in "planes", which cover the whole package or only the cores [7,12]. By definition, the uncore power could be obtained through subtracting the power of the core plane from the package plane. However, inconsistencies are observed. We found the core plane idle power changes when adjusting the PCs and uncore frequencies, which by definition should not affect the core idle power.

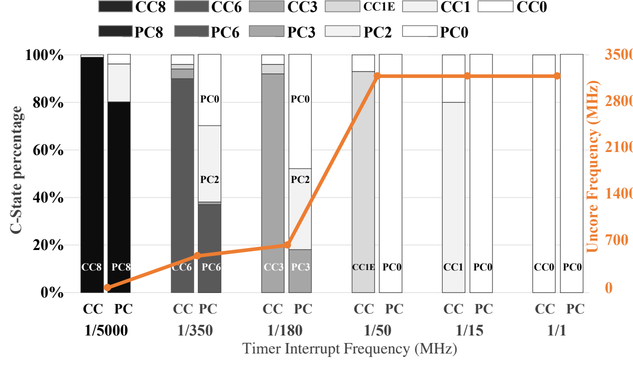


Figure 3: **Idle Power Management Dependency (IPMD) Observed via Hardware Statistics.** When the core C-state (CC) is changed by different interrupt workloads, the package C-state (PC) and the uncore frequency are accordingly adjusted by the processor's power management scheme.

To validate this observation in more practical settings, we reverse-engineer the `cpuidle` driver and leverage the information to accurately control CCs by generating specific workloads. We develop a periodical timer program (`TIMERSLACK` is set to 1 to disable the timer interrupt aggregation). When it is pinned to a core through CPU affinity, its timer interrupts can stimulate the core to stay in a certain CC.

Similarly, we keep the system idle and run the timer program on core_0. Cores other than core_0 are mainly in CC8 during the test. The residency time percentage of core_0's CC, PC, and the uncore frequency are shown in Figure 3. When core_0 is in deeper CCs, the percentage of deeper PCs increases and the uncore frequency decreases, and vice versa. That means the uncore tends to turn off/down components when the cores' components are turned off/down, and vice versa. This trend is identical to our test that directly controls CCs, and also coincides with the behavior described in the processor datasheet [60] and the ACPI specification [1]. All in all, the processor's core and uncore idle power management follows the IPMD principle.

4.2 Probing Uncore Idle Power via Exit Latency

An important implication of the IPMD principle is that the uncore idle power status reflects the power status of the cores sharing it. Although the hardware statistics directly show the status, accessing them generally requires permission or is just not possible. For example, in VM clients, only a few hardware registers are accessible. In this subsection, we show a more general approach to probing the uncore idle power.

Our insight is to take advantage of the overhead of turning off components. Deeper turning off generally leads to larger exit latency. We use the exit latency to probe the uncore idle power status. We assume the exit latency is composite and *independently* contributed by the core and uncore. We first use this assumption to build the latency model and then use measurements to validate it.

4.2.1 The Model of Exit Latency

To ease the following discussion, we use the notations:

$$T^k_{total} := \text{total exit latency of core_k.}$$
$$C^k := \text{core_k's core C-state.}$$
$$T_{core}(\cdot) := \text{exit latency contributed by the core part.}$$
$$T_{uncore}(\cdot) := \text{exit latency contributed by the uncore part.}$$

According to our assumption, the exit latency observed in core_k, *i.e.*, T^k_{total}, is contributed by two parts. The core part $T_{core}(C^k)$ has different values when the core_k is in different core C-states. According to our study in §4.1, the uncore part follows the IPMD principle. Therefore, its power is determined by the core C-state of the most active core of the processor, so is the uncore exit latency. Therefore, the total latency can be decoupled as [3]:

$$T^k_{total} = T_{core}(C^k) + T_{uncore}(\min_{i \in \{0,\ldots,M\}} C^i), \tag{1}$$

where M is the maximum core index.

This formula has one important implication. Consider the quad-core processor in Figure 4 (b). Assume core_1, core_2, and core_3 are in CC8. When core_0 is in CC0, core_3's exit latency is:

$$T^3_{total} = T_{core}(C^3) + T_{uncore}(C^0) = T_{core}(CC8) + T_{uncore}(CC0).$$

Similarly, when core_0 is in CC8, core_3's exit latency is:

$$T^3_{total} = T_{core}(C^3) + T_{uncore}(C^3) = T_{core}(CC8) + T_{uncore}(CC8).$$

Note that if $T_{uncore}(CC8) \neq T_{uncore}(CC0)$, the two latencies of core_3 are different. In other words, when the CC of a core is unchanged, the changes in the exit latency observed at the core are contributed by the uncore and reflect the power status changes of the uncore. Next, we use measurement experiments to validate this model.

4.2.2 Measurement Study on the Exit Latency Model

The exit latency can be measured through a network interface card (NIC) having hardware timestamping ability [28]. When a packet arrives, the NIC timestamps it with its local timer as T_1. Then it issues an interrupt to the host computer. The NIC driver is pinned to a core to avoid dynamic interrupt routing. If the core and uncore are idle, the interrupt needs to wake them up for executing the NIC driver's interrupt service routine (ISR). The first place where the core resumes execution is at the `cpuidle` driver, exactly after the `MWAIT` instruction that puts the core into idle. Therefore, we request the NIC timestamp again after `MWAIT` as T_2. $T_{total} = T_2 - T_1$ is the exit latency observed at the core.

[3] Indexes of C-state, such as 1E, 7s, *etc.*, are not numbers. The order for numerical comparison intuitively follows the depth of the idle states they represent, *i.e.*, CC0 > CC1 > CC1E ... > CC7s > CC8.

Core C-state	T_{core}	T_{uncore}
CC0	0	0
CC1	2	0
CC1E	2	1
CC3	27	58
CC6	30	60
CC7s	30	107
CC8	30	240

Table 1: **Decoupled Exit Latency (μs).** The exit latency caused by the idle states is contributed by the core and uncore independently. The total exit latency observed at a core is described by Equation (1).

To determine the values of $T_{core}(C^k)$ and $T_{uncore}(C^k)$, our method is to measure T_{core} first. To do so, recall Equation (1), we make sure the uncore is active and does not contribute to T_{total}, *i.e.*, $T_{uncore} = 0$. This is achieved by forcing one core in CC0, *e.g.*, core_0. At the same time, T_{total} at different CCs is measured at another core, *e.g.*, core_3, by varying its workloads through the timer program in §4.1. Note that, as $T_{core} = T_{total}$, T_{core} at different CCs is measured and shown in Table 1. After obtaining T_{core}, T_{uncore} at a CC is measured by subtracting T_{core} from the total exit latency of that CC. Specifically, we vary one core's CC with the timer program and keep the remaining cores in the deepest CC. T_{uncore} = T_{total} - T_{core} is calculated in Table 1.

We also exhaust different combinations of CCs at different cores to compare the measured T_{total} and the computed one from the model by referring to the values in Table 1. They match exactly. For example, when core_0 is in CC3 and core_3 is in CC8, the measured exit latency at core_3 is 88 μs, which equals to the calculated one: $T^3_{total} = T_{core}(CC8) + T_{uncore}(CC3) = 30 + 58\ \mu s$.

Several interesting observations can be made with respect to Table 1. First, the exit latency contributed by the core increases when the CC index increases, which is under expectation. T_{core} has a sharp increase at CC3. This is because the L1 and L2 cache are flushed into the LLC. T_{core} does not increase any more with deeper CCs. This is because there are no more major core components for turning off at deep CCs. Second, the exit latency caused by the uncore is proportional to the CC index of the core. This trend is consistent with IPMD and the conclusion of the previous subsection §4.1. T_{uncore} only takes effect in deep CCs. This is because, due to the large exit latency, the processor only attempts to turn off the uncore when all the cores are deeply turned off, *i.e.*, in deep CCs.

4.3 The Idea of the IPMD Channel

Based on the above study, it is straightforward to come out with the idea of IPMD channel: the uncore idle power status can be stimulated by the core's activity (§4.1), and can be probed by other cores through measuring the exit latency (§4.2). The above allows the activity information of one core to be perceived by another core through their shared uncore. A concrete example is shown below.

CPU	Arch.	Deepest CC	Kernel	T_{uncore}
Core i5 6500	Skylake	CC8	5.4	240
			4.19	239
			4.14	239
Core i5 8500	Coffee Lake	CC10	5.4	225
Xeon E5 2630v4	Broadwell	CC6	5.6	23
Celeron J4105	Gemini Lake	CC10	4.19	211

Table 2: **Uncore Exit Latency of Different Platforms** (μs).

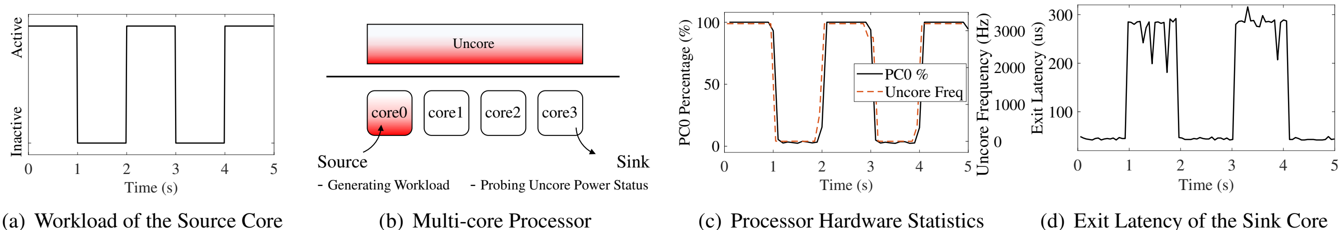
Figure 4 (b) is a quad-core processor (the architecture is symmetric and the four cores are identical, so the following core indexes are interchangeable). The workload is applied to core_0, while other cores are almost idle. core_0 is also called the Source core. core_3 is another core of the processor. core_3 and the Source core share the uncore of the processor. The Source core's workload can be probed at core_3. We call core_3 the Sink core. The method for probing is simply to measure the exit latency of the Sink core.

Figure 4 (d) shows the exit latency trace of the Sink core when the workload pattern of the Source core is shown in Figure 4 (a). The hardware statistics are shown in Figure 4 (c) for reference. Both Figure 4 (c) and (d) precisely reflect (a). When the Source core is active, *i.e.*, [0-1] s, [2-3] s, [4-5] s, the Sink core's exit latency is low. This is because the uncore is stimulated by the Sink core to be active (the residency time of PC0 and the uncore frequency is high). When the Source core is idle, the exit latency of the Sink core increases correspondingly, which is also reflected in the hardware statistics.

Through analyzing the exit latency, the workload pattern of the Source core can be inferred. This is a IPMD channel. Its efficiency depends on how much exit latency the uncore contributes to. As such, we measure $T_{uncore}(\cdot)$ of different hardware platforms and operating systems. Table 2 shows the uncore latency when all cores are in the deepest core C-state, indicating the maximum observable uncore exit latency. Due to the performance-oriented design, the value of the server processor is much smaller. In the following two sections, two practical attacks based on this IPMD channel are presented.

5 Cross-VM Covert Channel

Cloud vendors utilize virtualization technologies to allow multiple tenants to share the same physical machine. A major concern is that the sensitive information in the VM guest might be subject to theft attacks from the co-located VMs. Despite many efforts paid by the vendor, existing work has shown various attacks to break the VM isolation through exploiting the shared hardware resources [26]. However, it remains difficult to launch attacks under strict processor and memory isolation. In this section, we describe how the IPMD channel can be used for this purpose.



(a) Workload of the Source Core (b) Multi-core Processor (c) Processor Hardware Statistics (d) Exit Latency of the Sink Core

Figure 4: **Overview of the IPMD Channel**. In a quad-core processor in (b), all the cores are idle by default. When applying the workloads in (a) to the Source core, the activity of the Source core stimulates the idle power status of the uncore, which is reflected by the hardware statistic of the uncore in (c). It can then be further probed by the Sink core of the same processor through measuring the exit latency in (d), which allows the Sink core to infer the activity of the Source core.

5.1 Attack Model

We assume there are two entities in the attack. The attacker VM and the victim VM. We assume the victim VM is infected by the *Source* malware from the attacker. The infection can be achieved through social engineering [65], or VM image pollution, *e.g.*, publishing a modified Amazon Machine Image (AMI) in the AWS market [14]. Since the *Source* only contains several lines of code, it can be integrated into some useful applications without being noticed. We assume the *Source* can access sensitive information about the victim VM. For this reason, it might be subject to strict network firewall policies and information flow inspection [25], hence we assume it has no network connection with the attacker VM. The attacker VM is under the control of the attacker. We assume the attacker VM and the victim VM are co-located, which can be achieved through on purpose allocation [64, 67]. A patient attacker can also control many attacker VMs in different physical machines to passively wait for victim VMs allocated from migration or new instances. We assume the attacker VM has a *Sink* program to receive information from the *Source* through the IPMD channel once the co-location is established. We also assume the physical machine is not heavily loaded. Since the average utilization of cloud servers is about 40% [54], there are sufficient idle periods that can be leveraged in practice, *e.g.*, in non-busy hours.

Apart from the above, no further assumptions are made on the VMs or the hypervisor. Specifically, we do not assume any fixed or unfixed scheduling between physical cores and vCPUs [69]. We do not assume hyper threading [61], memory deduplication [34], HugePages [52], and cross-socket NUMA policies [52] in the VM guests or the hypervisor.

5.2 Measuring Exit Latency in the Guest VM

In § 4.2, we propose a method to probe the uncore activity via measuring the exit latency, which, however, is based on NIC's hardware timestamp and no longer possible in the VM situation. This is because most hypervisors virtualize NICs through connecting the VM guests to a virtual layer-2 switch and the VM guests cannot access the NIC hardware and low-level information. Our idea is to leverage the system timer. Similar to the NIC interrupts, the timer interrupts also wake up the idle cores, which can be refined for measuring the exit latency.

5.2.1 Basics of Timer

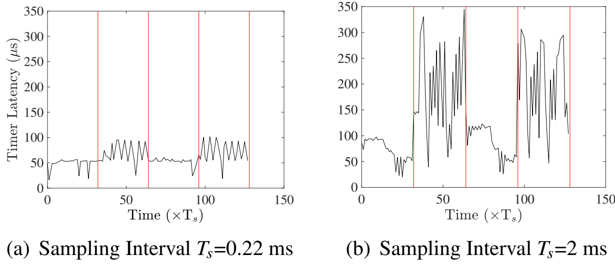
Software timers are implemented by the hardware programmable timer, which consists of a hardware counter and a reference clock. To arrange a timer event, the software timer assigns an "expire time" value to the hardware timer. The expire time represents how many clock ticks the hardware counter needs to wait before issuing an interrupt to the processor, *i.e.*, the timer interrupt, which is then handled by the timer event handler.

There are several hardware timers. Our experiments use the most common one: the Local Advanced Programmable Interrupt Controller (LAPIC) timer. LAPIC is a per-core hardware residing in each core. When its timer expires, the LAPIC triggers a timer interrupt to the core it binds to. The implementation details of the LAPIC timer are unclear, but on our platforms, it works properly even when the cores and uncore are in deep idle states. Modern LAPIC timers work closely with TSC clock [2], which probably indicates LAPIC timers use a clock source and power supply independent of the processor idle power management mechanism [17].

The timer interrupts in VMs are virtualized by the hypervisor. For example, the KVM hypervisor emulates a virtual LAPIC timer for every vCPU and the underlying hardware is still the hardware LAPIC. Specifically, when a guest VM sends a timer request, the KVM hypervisor sets the corresponding virtual LAPIC and writes the real LAPIC's registers via the Linux hrtimer subsystem. The hrtimer subsystem also takes charge of the reception of the timer interrupts and forwards them to the KVM hypervisor and then to the guest system.

5.2.2 Measuring Exit Latency via Timer Latency

From the above, the timer is able to generate interrupts at the scheduled time whether the processor is idle or not. Therefore,



(a) Sampling Interval T_s=0.22 ms (b) Sampling Interval T_s=2 ms

Figure 5: **Example of Timer Latency Trace.** The timer latency is measured in a VM guest pinned to a core. A square-wave workload is applied to another core as in Figure 4. The trace is recorded with different sampling intervals. The x-axis is indexed by the sample index. To convert to the absolute time, count each by one sampling interval.

a method to measure the exit latency is: at time T_{pre}, set a timer and expire it after T_s [4]. When the timer interrupt arrives, take the timestamp as T_{post}. Note that as the core might be idle when the timer fires, it takes time, *i.e.*, the exit latency, for the core to wake up to handle the timer interrupt. Therefore, if ignoring the overhead of the software, the timer latency, $T_{post} - T_{pre} - T_s$, is a good estimation for the exit latency T_{total}.

The feasibility of this approach can be visualized by an experiment similar to Figure 4. The settings are detailed in §5.4.1 (the desktop platform). We pin a process to a vCPU that is pinned to a physical core, and then apply a square wave workload at another core. We measure the timer latency through the process. Results in Figure 5 show an identical pattern to the workload. The amplitude of the "square wave" is affected by the sampling interval T_s. This is because more frequent timer interrupts result in shallower CCs. For example the workload is off at around 50, the core of (a) is still in CC3 while in (b) is mainly in CC8.

With the same setup, the decoupled exit latency can be measured by the timer latency similar to §4.2.2. The values are shown in Table 3. The latency is measured in both the VM host and guest. The HOST situation is close to Table 1, which again validates the timer measurement approach. Values in the GUEST column are slightly different to the HOST's, although their trend is similar. The difference is mainly caused by two reasons. The first is the virtualization overhead, where the timer latency is handled by an additional hypervisor layer, which contains more uncertainties. The second is the system configuration. The core frequency is fixed to the base frequency for comparing the HOST with Table 1, while in the GUEST, we adopt the default system configurations (not fixed) to show the default behavior.

[4] T_s is the **S**ampling duration from the measurement perspective.

	HOST Fixed		Guest Default	
Core C-state	T_{core}	T_{uncore}	T_{core}	T_{uncore}
CC0	9	0	12	1
CC1	15	0	20	11
CC1E	15	6	20	40
CC3	42	71	50	75
CC6	48	75	62	102
CC7s	48	111	62	146
CC8	48	260	62	304

Table 3: **Exit Latency Measured by Timer Latency (μs).** The Sink process is running in the VM guest to probe the exit latency of the physical processor. Values are slightly different from Table 1, which is caused by the overhead of VM and CPU DVFS configurations.

5.3 Communication Design

In this section, we present the complete communication design of the cross-VM covert channel. The *Source* conveys bits by generating/not generating the workload. The *Sink* detects the workload pattern of the processor through the timer latency. Their pseudocode is shown in Appendix.C.

The *Source* uses a special on-off keying scheme to modulate information. The scheme can be explained by Figure 4. To transmit a bit '1', it generates CPU load [5] to force a physical core in CC0 for a predefined time period T_{Sym}, representing the duration of a symbol. To transmit a bit '0', it sleeps for T_{Sym}.

Bits are packed into a structured frame containing two parts. The preamble is a special header with a predefined distinct bit pattern, which is used to determine the accurate start of a frame. The payload is a bit stream of a fixed length. Messages are packed into the payload for transmission.

The *Sink* process receives the frame through the covert channel. First, it continuously samples the timer latency of the host processor. It keeps a window of samples to detect the presence of a frame and find the start of the frame through correlating the window with the preamble pattern. Then, the payload part of the frame is extracted for decoding. The *Sink* takes the latency samples within a symbol period T_{Sym} to judge the bit. As the *Source* generates workload to represent '1', the *Sink* outputs '1' if the latency values are relatively low. Otherwise, it outputs '0'.

From the snapshot trace in Figure 5. The noise of bit '1' is much lower than that of bit '0'. This is because a deeper idle state would occasionally jump to a shallower one but CC0 will not. To account for this bias, we adopt a heuristic approach to select the differentiating threshold. Specifically, we first smooth the neighboring samples, and then jointly consider samples from several symbols to determine the threshold for the frame. In addition to that, the standard deviation of samples within a symbol is used as a decoding criterion.

[5] To keep the core active, we implement the *Source* by repeatedly assigning an already-expired time to an absolute-time timer, which returns immediately. After the timer returns, it polls TSC timestamps until reaching T_{Sym}.

5.4 Performance Evaluation

In this section, we evaluate the performance of the covert channel under practical settings with different platforms.

5.4.1 Testbed Setup

The experiments are conducted with in-house desktops and servers, and the public cloud.

Desktop: The hardware platform is an Intel Core i5-6500 (4 pCPU) with 8 GB RAM and Intel Q270 Chipset. The hyper-threading is disabled. The host OS is Debian 10 server with kernel 5.4.0. The guests use the same OS. The KVM modular is based on libvirt 5.0.0-4 and QEMU 1.3.1. Two VM guests are created in the host machine to emulate the attacker VM and the victim VM. Each guest is allocated with 2 GB RAM and 2 vCPUs. By default, the 4 vCPU are pinned to the 4 physical cores (pCPU). In the following scheduler evaluation, the impact of pin or unpin is discussed. The *Source* and *Sink* are created as normal processes without privileged permissions. They use `prctl` to set `TIMERSLACK` to 1 to increase the timer resolution [6]. All system configurations are default.

Server: The hardware platform is 2×Intel Xeon E5 2630v4 in 2 sockets (2×10 pCPU in total) with 64 GB RAM and Intel C610 Chipset. Software settings are the same as the desktop. 5 VM guests are created. One for the victim VM. One for the attacker VM, and the remaining is standby VMs. Each VM has 2 vCPU pinned to 2 pCPU and 4 GB RAM. By default, all the VMs are assigned to one socket. The impact of cross-socket is separately discussed in §5.4.2.

Cloud: The hardware platform is an Amazon EC2 c5 dedicated host with 2×Intel Platinum 8124M in 2 sockets (2×18 pCPU in total) and 144 GB RAM. Software settings are the same as the desktop situation. Since there is no socket control interface in the dedicated host, we use the following settings to manually force it. We create one c5.9xlarge instance (18 pCPU) to occupy one socket, and two c5.4xlarge (8 pCPU) instances to occupy the other. No NUMA hierarchy is observed (via `numactl`) in the 9xlarge instance. While there is no concrete evidence, we do not think the EC2 Nitro hypervisor will allocate memory and cores across NUMA nodes, so it is very likely the two 4xlarge instances are strictly isolated from the 9xlarge instance in terms of processor and memory.

5.4.2 Channel Capacity

We measure the channel capacity to quantify the throughput performance of the IPMD covert channel.

Metrics: Channel capacity is the theoretical throughput upper bound of a communication channel, which is independent of the error correction schemes. We follow the methodology in [36], and treat the IPMD channel as a binary asymmetric channel [7]. The channel capacity is determined by the symbol error rate (which is also the bit error rate in our case). If every symbol is correct, then the capacity of one symbol C_{Sym} is 1 bit. Due to the noise, C_{Sym} is less than 1 bit. For example, when the symbol error rate is 10%, C_{Sym} is about 0.5 bits. The symbol error rate is empirically measured from 10 frames, each of which contains 8k bits (An example of the raw error rate is shown in Table 4). According to the error rate, C_{Sym} can be calculated from the information theory model. The capacity of the channel is determined by the frequency of the symbols and the per symbol capacity: $C = C_{Sym}/T_{Sym}$ bps. We vary the symbol duration T_{Sym} and the sampling interval T_s to obtain the results in Figure 6.

Sym	No Load		20%		40%		60%	
Dur	1→0	0→1	1→0	0→1	1→0	0→1	1→0	0→1
2	3.8%	18.4%	5.9%	46.5%	27.5%	36.2%	34.0%	24.4%
4	7.1%	7.4%	5.3%	43.1%	10.2%	46.4%	35.1%	23.7%
8	0.4%	2.2%	4.9%	36.6%	11.4%	46.0%	45.6%	32.8%
16	0.2%	0.8%	0.1%	35.1%	33.3%	38.9%	45.2%	31.5%
32	0.1%	0.6%	0.3%	41.0%	8.9%	47.0%	49.3%	31.1%

Table 4: **Raw Bit Error Rate**. Values are from the experiments of Figure 9 (a). $1 \to 0$ and $0 \to 1$ denote the bit flip errors of 1 to 0 and 0 to 1, respectively. The background load tends to flip 0 to 1 (20%, 40%). As we use a dynamic threshold to judge bits, when the load is intense (60%), the threshold is biased by the load and increases the error rate of 1 to 0.

Desktop: As shown in Figure 6 (a), a larger T_{Sym}, results in smaller channel capacity. This is because, although the symbol error rate is smaller (not depicted), each symbol takes more time to transmit. One abnormal phenomenon is that the channel capacity consistently increases when adopting fewer samples to represent a symbol, which generally implies the channel noise is still very small and the capacity can be even higher by using a larger sampling rate. However, this is not possible on this platform. When we further decrease T_s from 0.22 ms to 0.2 ms, no bits can be extracted from the latency trace. The reason has been mentioned in §5.2.2. When the frequency of the probing timer of the *Sink* is high enough, the core will be stimulated to stay in CC0. As a result, the T_{uncore} in Equation (1) does not introduce any observable latency differences when the load of other cores changes. This property limits the capacity and the time resolution of the IPMD channel.

Server: The results are shown in Figure 6 (b). We highlight the difference from the desktop results. The maximum channel capacity is reduced to 200 bps. The main reason is that the latency traces are much nosier. This is reflected by the peak located at $T_{Sym} = 4 \times T_s$, meaning that 4 samples are

[6] In side channel experiment, only the *Sink* uses this option.

[7] This approach is only correct for our current binary modulation, since one can take more latency levels introduced by different c-states to represent information (*i.e.*, the channel is not binary). For general channel capacity estimation, one must take the signal bandwidth into consideration [22].

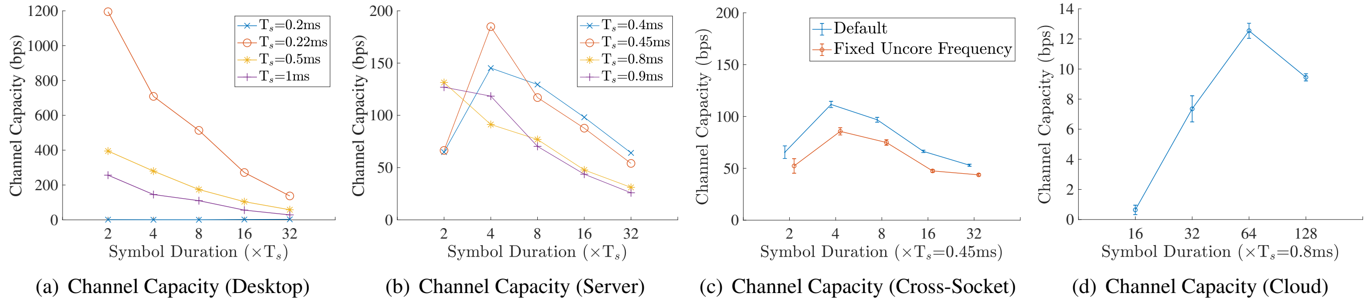


Figure 6: **Channel Capacity.** Measured with different sampling interval T_s and symbol duration T_{Sym}. The maximum capacity 1200 bps is achieved with the desktop platform. The corresponding raw throughput is $1/(2 \times T_s)$=2.3 kbps with 10% error rate.

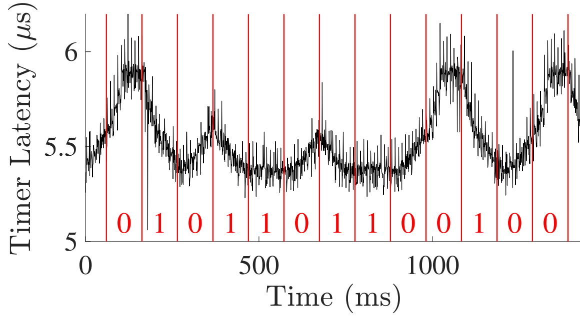


Figure 7: **Raw Latency Trace of Covert Transmission (Amazon EC2).** The trace is sampled at the *Sink* with T_s=0.8 ms at the attacker VM, while the *Source* transmits `0101101100100` at the victim VM with T_{Sym}=128×T_s=102 ms. When the *Source* is busy-waiting to transmit "1", the timer latency at the *Sink* decreases. When the *Source* is idle to transmit "0", the timer latency increases. A full switch between the two states takes around 150 ms, which limits the capacity of the IPMD channel.

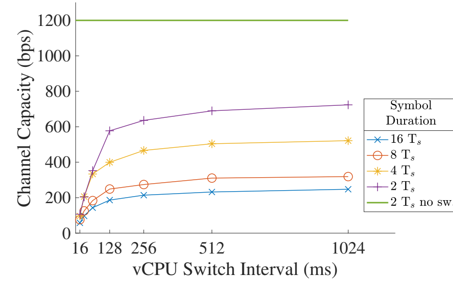
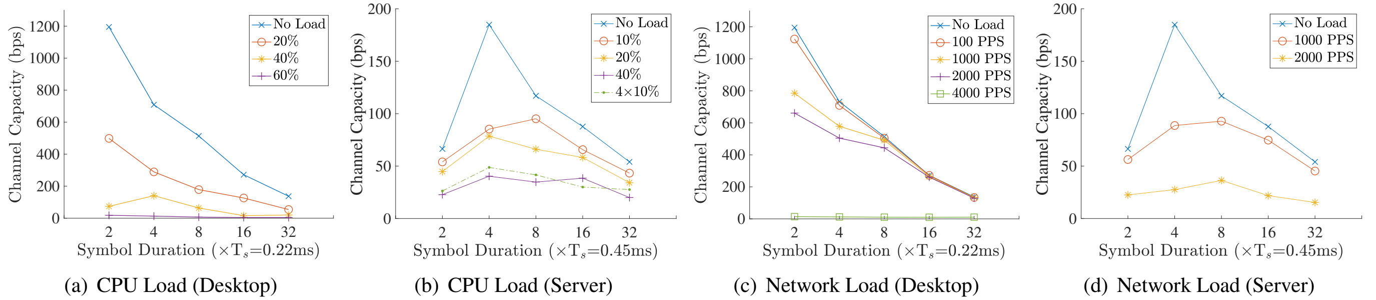


Figure 8: **Impact of the VM Scheduler.** Frequently scheduling vCPUs to different physical cores, although not common, hurts efficiency of the IPMD channel. "no sw." denotes the fixed case without core switching.

better than 2 samples for combating the noise. Additionally, the minimum sampling interval is also larger than the desktop, otherwise the uncore will always be active. There are also other minor reasons related to the processor architecture, *e.g.*, its timer is noisier than the desktop, and the latency difference caused by the uncore is smaller in (see Xeon in Table 2).

Cross-Socket: With the server platform, we conduct the same test by assigning the victim VM to a socket and leave the remaining VMs in another socket. VMs are launched with the NUMA `strict` policy to avoid cross-socket memory access. As shown in Figure 6 (c), the IPMD channel still exists. We next explain the reason. In the NUMA architecture, if one socket is active, the uncores of other sockets must be active to keep their memory controllers and interconnect links active to support remote memory access from other NUMA nodes, which directly leads to the IPMD channel. The channel remains even if remote memory access is not allowed. This is because current power management mechanisms have not taken into account this factor. Therefore, unlike existing cross-processor channels [52], the IPMD channel requires no cross-processor memory share. We further decouple the impacting factors by first fixing the uncore frequency, which has a negative impact on the channel capacity. Then, we disable the PC by setting PC = PC0 in the BIOS, and the cross-socket channel is neutralized (not depicted). The phenomenon coincides with our analysis in §3.2.

Cloud: Both cross-socket and in-socket are tested in the cloud server. For in-socket, the two 4xlarge instances are used as the attacker and victim respectively. For the cross-socket experiment, the 9xlarge is the attacker and one of the 4xlarge instance is the victim. The remaining 4xlarge is applied with a 400 requests per minute (RPM) HTTP traffic load (see §6). Unlike the server situation, the performance of the two settings is identical, so only the cross-socket results are shown in Figure 6 (d). We highlight the difference from the server and desktop cases. First, the boundaries of the symbols are not "sharp" (see the raw trace in Figure 7), we guess it is because the uncore takes a significant period of time (around 150 ms) to change its power states. This is the major performance bottleneck and is related to the hypervisor power management schemes and the processor architecture. Second, the timer latency is significantly lower than the desktop and server cases. This is because the EC2 instance adopts a performance-oriented power plan by default [15], where the minimum CC



(a) CPU Load (Desktop) (b) CPU Load (Server) (c) Network Load (Desktop) (d) Network Load (Server)

Figure 9: **Impact of CPU and Network Load.**

is CC1. However, as the uncore frequency is not fixed, the IPMD channel still exists but with a much lower capacity of several bps.

5.4.3 Impact of VM Scheduler

In KVM, vCPUs are realized by threads, and assigned to physical cores by the CFS scheduler [24]. A vCPU can be scheduled to different cores to balance the per-core load. To ensure cache efficiency, repinning a vCPU is not frequent in practice. In our settings, with enough load, we observe that the scale is around several seconds. Therefore, we fix the mapping between cores and vCPUs by default.

To understand the worst-case performance due to the scheduler. We consider two cases. First, both the *Source* and *Sink* are executed by the SAME core. From Equation (1), this case does not introduce much difference, except the core T_{core} part is different when transmitting '1' and '0'. Its throughput is almost identical to the default case, *i.e.*, two processes are executed by different cores. Second, the *Source* is executed by a fixed core, and the *Sink* is executed by different cores. To achieve this, the *Sink* is pinned to a vCPU, which is pinned to all the available cores cyclically, including the one executing the *Source*. The time of staying in one core is called the vCPU switch interval. We measure the performance in Figure 8. The green line denoted with "no sw." shows the reference for the performance without scheduling. Fast scheduling has an obvious negative impact on the channel. Errors appear when symbols encounter vCPU switching.

5.4.4 Impact of Workload

In previous experiments, the computing platform is almost idle. This section studies the impact of the workload.

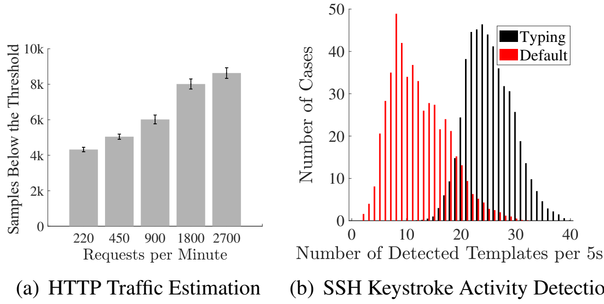
First, on the desktop platform, we use `stress-ng` to apply the CPU load with the minimum load slice to one of the two vCPUs of the victim VM and raise the load to different levels. Figure 9 (a) shows the channel does not work when the CPU load is more than 40%. The reason is that the uncore is already fully active, hence the load changes introduced by the *Source* cannot be perceived by the *Sink*. In Figure 9 (b), the server has a similar trend. The load of 10%, 20% and 40% is applied to the victim VM only. The 4×10% load is evenly divided among the 4 VMs except the attacker VM, *i.e.*, a 10% load is applied to one of the two vCPUs of the 4 VMs. The results of 40% and 4×10% are close, this is because the four 10% VM tasks are not synchronized. This means that the more VMs or unsynchronized per-core tasks there are, the harder it is to launch the covert channel.

Second, we developed a network stress tool to generate UDP traffic at different rates in units of packet per second (PPS). The UDP payload is 64 Bytes and the packet interval is almost even with a small random variation. Note that we use the small payload size to narrow down the impacting factors. The results with a larger payload size are similar. The traffic is generated by another machine towards the victim VM. The Ethernet controller of the host machine is Intel i219. Figure 9 (c) and Figure 9 (d) show that a light network load does not affect the throughput, especially when using a longer symbol duration. Similar phenomena are observed in §5.4.2 when measuring the capacity of the cloud. This is because the major impacting factor of the network packets is their interrupts. When the PPS is low, network interrupts are like a single noise event. Larger network loads begin to affect the performance, and completely disables the channel at 4000 PPS. The reason is that the interrupt frequency of the 4000 PPS traffic is close to the timer interrupt at 0.2 ms in Figure 6 (a), where the uncore is already active and no uncore latency can be detected.

6 Cross-VM Activity Profiling

This section presents a preliminary study on the feasibility of exploiting the IPMD channel for side channel attacks. Unlike the covert channel, the side channel leaks information about the victim without a cooperative malware. The intuition is that the uncore power status reflects the activity of cores, so we use the timer latency in the attacker VM to infer the activities of the co-located VM.

Network Traffic Intensity Estimation. We consider the attack scenario in a typical cloud setting, where the victim VM hosts a web server and the co-located attacker VM is waiting to spy on the network traffic rate of the victim VM.



(a) HTTP Traffic Estimation (b) SSH Keystroke Activity Detection

Figure 10: **IPMD Side Channel.** The attacker VM uses the IPMD channel to (a) probe the network traffic and (b) detect whether there is SSH keystroke activity in the co-located victim VM or not.

With this information [55], the attacker can infer the number of visitors or even which web page is being (mostly) visited. The intensity of the network traffic is correlated to the core activities, because the cores must be active in serving the NIC interrupts, the TCP stack processing, and various I/O and computation tasks related to web queries, all of which might be reflected in the uncore power status.

We use the EC2 dedicated host to emulate this case. The c5.9xlarge instance uses the `nginx` to host a web page of 3 MB. One c5.4xlarge instance is the attacker VM. Another c5.4xlarge instance is idle. We create another t2.xlarge instance in the same EC2 region to act as the visitors. `JMeter` is used to emulate 20 concurrent visitors to fetch the 3 MB file repeatedly from the web server. The idle period of each visitor is adjusted to evaluate different query frequencies, *i.e.*, the HTTP traffic load.

Similar to the covert channel, when the victim VM serves HTTP queries, the latency caused by the uncore reduces. We use an approach like the communication decoding to count the latency samples below a threshold to estimate the network activity. The attacker probes the traffic for 1 min excluding the ramp-up period and repeats for 10 times. Results are shown in Figure 10 (a). Even with such a simple implementation, the traffic rate can be determined in the resolution of around 100 RPM. We observe that the scheme cannot differentiate traffic higher than 4000 RPM, this is because the cores and hence the uncore are busy all the time. As a result, the attacker VM cannot observe the latency difference.

SSH Keystroke Activity Detection. `SSH` is widely used in Linux-based servers for remote access. In the interactive mode, the `SSH` client sends the input key to the server without aggregation. The timing attacks with `SSH` keystrokes are pioneered by Song *et. al.* [59]. We study the feasibility of launching such an attack with the IPMD channel. In this scenario, the attacker VM wants to probe the precise keystroke timing of the `SSH` session of the co-located VM. The intuition is that, the `SSH` traffic causes processing load of the network and encryption stack, which might be reflected in the uncore power status.

As we measured in §5.4.2, the time granularity of the IPMD channel in EC2 servers is quite restricted due to their performance-oriented power policies. For servers adopting power-efficient plans, a finer granularity can be expected. Such choices are not unusual, as the default power plans of the in-house desktops and servers we encountered are all power-efficient. To understand the potential risks, *e.g.*, with not-so-sophisticated server administrators, the desktop platform is used as the reference, as its uncore responds to the workload changes in a timely manner. The attacker VM uses T_s=0.22 ms to sample the timer latency. The victim VM is idle except for an `SSH` session from a remote `SSH` client within the same LAN. We develop a lightweight routine in the victim VM to timestamp the keystrokes from `SSH` via `stdin` as the ground-truth. A script is used to type dummy characters in the `SSH` client to generate the keystroke input.

As shown in Figure 12 in Appendix.B, while the latency trace is noisy, there is an obvious downwards spike at every keystroke, which is caused by the `SSH` processing activity. We identify `SSH` keystrokes according to three simple rules based on the mean and width of the spike, which actually define a spike template. The `SSH` input lasts for 30 mins. We count the number of detected templates every 5 seconds, and its histogram is shown in Figure 10 (b). The red bars are detected templates when there is no typing, which is contributed by the system background activities similar to the keystrokes'. We choose 20 spikes per 5 seconds as the threshold to differentiate whether the user is typing or not. The attacker misses 4.9% of typing cases, and the false positives account for 8.3% of the detected ones. The F1-score is 0.93. We note that although the resolution of the keystroke timing is enough, it remains challenging to guess the exact keystroke due to the false positives. It may require sophisticated and powerful rules such as learning algorithms to differentiate the keystrokes from the system noise.

7 Related Work

This section reviews existing covert and side channels stemming from computing architecture and power management mechanisms. We focus on software-based attacks relying on no additional hardware, *e.g.*, power probes [39], RF receivers [58], *etc.*

Architectural Covert and Side Channel. Shared microarchitectural components are naturally suitable for building covert/side channels. A lot of such attacks have been explored by the existing literature [26]. While the IPMD channel is a general approach to leak and steal information, the attack examples in this paper target virtualized environments, hence we list several representative cross-VM covert/side channels in Table 5 for comparison. We classify them according to the computing hierarchy, *i.e.*, single core, processor (cross-core), and system (cross-processor). Higher-level hierarchy

Scale	Article	Shared Components	Rate (Error)		Side Channel	Cross-VM	NUMA `strict`
Core	Ristenpart *et. al.* [55]	L2 Cache	0.2bps	(N/A)	●	●	○
	ZombieLoad [57]	Memory Order Buffer	2.0kbps	(2.5%)	●	●	○
Processor	Maurice *et. al.* [47]	Last Level Cache	600.0kbps	(1%)	●	● [47]	○
	CrossTalk [53]	Staging Buffer	24.0kbps	(5%)	●	Local	○
	POWERT [36]	Power Budget	0.1kbps	(N/A)	N/A	Local [35]	○
System	Wu *et. al.* [66]	Memory Bus	0.4kbps	(0.39%)	○	●	N/A
	DRAMA [52]	DRAM Row Buffer	2600.0kbps	(8.7%)	●	Local	○
	IPMD Channel (this)	Uncore	2.3kbps	(10%)	●	●	●

Table 5: **Comparison of Cross-VM Covert and Side Channels.**

generally means a looser sharing connection but a higher probability of launching practical attacks in VMs or clouds, since core and processor level attacks can be eliminated through exclusively assigning physical cores or sockets to VM guests.

Cache is arguably the most widely exploited components for covert/side attacks. Ristenpart *et. al.* [55] exploit per-core caches to leak information from the Amazon cloud, which achieves 0.2 bps. Along with this insight, attacks on per-core caches are proposed with the improved performance [62, 69], but they rely on hyper-threading or the scheduler for time sharing. For this reason, subsequent attacks are based on LLC to work across cores [29, 33, 42, 46, 47, 68], but they are limited to the same physical processor. Irazoqui *et. al.* [34] and Armageddon [40] leverage the shared memory to make cache attacks across processors, but their approaches are not applicable to practical VMs, where memory deduplication is usually disabled by default [33].

In addition to caches, other per-core components, such as Memory Order Buffer (MOB) [61], are exploited too. Recent research into vulnerabilities of speculative execution has led to powerful covert/side attacks [37, 57, 63], *e.g.*, ZombieLoad [57] exploits the MOB in the core logic to leak the data or to communicate with the co-located hardware thread. While Spectre and Meltdown-like attacks usually operate in the single core scale, CrossTalk [53] traps the staging buffer shared among cores to mount cross-core attacks.

Caches are restricted to an individual processor, but memory is shared. Memory and memory bus are leveraged for cross-processor covert/side-channel attacks. Wu *et. al.* [66] uses atomic memory operations to force memory bus contention to achieve cross-processor communication, but as atomic operations are not used by general programs, the related side channel might not be possible. DRAMA [52] reverse-engineers the DRAM mapping to construct contentions in the DRAM row buffer. They further demonstrate a side channel for logging the key strokes timings. DRAMA is superior to the IPMD channel in terms of throughput and resolution, but IPMD is verified in the public cloud and works with the NUMA `strict` policy, which is the default memory pin policy in KVM to avoid VM performance degradation due to remote memory access. Further, DRAMA also depends on HugePages, which is not the default configuration of VMs [5].

Power-Management-Related Covert and Side Channel. To pursue power efficiency, modern computing platforms expose rich interfaces for software power management and monitoring. This trend has led to several vulnerabilities.

Thermal channels [22, 43, 45] transmit information through executing workloads to increase the heat of the core, which can then be probed by the temperature sensor. Miedl *et. al.* [50] shows that, the workload can also be probed through the processor power sensors. Recently, PLATYPUS [41] leverages the Intel RAPL interface to launch powerful software-based power analysis, which can even spy on cryptographic keys. The above approaches require accessing certain processor registers, which are blocked by the VM hypervisor.

In a computing system with a limited power budget (to prevent overheating), an intensive workload reduces the performance of other workloads. This property is utilized by the POWERT [36] to build a workload-based covert channel. The property is also related to the DVFS mechanism, where core frequency is adjusted according to the intensity of workloads. The covert channel attack can be mounted by accessing the CPU frequency registers [19, 49] or counting the performance [35]. IPMD channel and DVFS or POWERT channels are based on different and complementary mechanisms. IPMD occurs when cores are idle and it is effective across the whole system, while DVFS occurs when cores are active and affects the single processor only.

It is worth remarking that the IPMD channel is a completely new covert channel functioning across cores and processors. Like exiting covert channels, it is based on shared resources, *i.e.*, uncore, but its information is not conveyed through competing for the shared resource, *i.e.*, cache lines, memory bus, power budget, *etc.*, but through mutual-boosting, *i.e.*, active cores heating the computing platform improve (but not reduce) the latency performance of co-located cores, which is a very distinct behavior.

8 Countermeasures

We have disclosed our findings to the security teams of Intel, Amazon AWS, and Microsoft Azure. The IPMD channel is more like a design trade-off rather than a flaw, hence it can

be easily eliminated through configurations but at a cost elsewhere: 1. Disabling idle power management mechanisms. *e.g.*, fixing the uncore frequency and fixing PCs to PC0 (see Figure 6). The drawback is the power efficiency, *e.g.*, increasing the maintenance cost of servers, contradicting all the engineering efforts in enabling power-efficient computing. A more feasible solution is to only disable the uncore power management in sensitive scenarios. 2. Reducing the timer resolution. High resolution timer and ticks (`RDTSC`) have been blamed for enabling various timing covert/side channels. Blurring timer resolution [44], such as the Windows timer (either unintentionally or intentionally) can effectively reduce the channel capacity and granularity (see Figure 11). The cost is the determinism of the system and the applicability of time-sensitive applications. 3. Scheduling. The VM scheduler can pin VMs to pCPUs at a high frequency (see Figure 8). However, frequent scheduling not only makes cache deficient but also increases the probability of core or processor co-resistance, which instead eases many other attacks. 4. Auditing. The system administrator can observe an abnormal interrupt rate due to the timer latency measurement. A patient attacker might choose a low sampling rate and long symbol duration to hide its footprint.

9 Conclusion

Since the power efficiency is more and more important, it is worth understanding the security risks of corresponding mechanisms. This paper reveals a cross-processor and cross-VM covert channel stemming from the core and uncore sharing dependency in the processor idle power management mechanism. This paper primarily focuses on breaking the isolation of VMs. Theoretically, it might be possible to extend the insight to launch attacks in browsers and trusted computing environments.

Acknowledgements

We sincerely thank our shepherd Mohammad A. Islam and anonymous reviewers for their valuable comments and suggestions. We thank Shauna Dalton for the proofreading. We thank Tianming Wen from ShanghaiTech for helping us with the server platforms. This work is supported (in part) by the ShanghaiTech Startup Fund, the Shanghai Sailing Program 18YF1416700, the “Chen Guang" Program 17CG66 supported by Shanghai Education Development Foundation and Shanghai Municipal Education Commission, and NSFC 62002224.

References

[1] Advanced configuration and power interface (acpi) specification. https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf, Feb. 2021.

[2] Apic timer. https://wiki.osdev.org/APIC_timer, Feb. 2021.

[3] Cpu power saving methods for real-time workloads. https://elinux.org/images/8/86/CPU_idle.pdf, Feb. 2021.

[4] "ENLIGHTENING" KVM HYPER-V EMULATION. https://archive.fosdem.org/2019/schedule/event/vai_enlightening_kvm/, Feb. 2021.

[5] "HugePages Configuration in Amazon EC2. https://aws.amazon.com/premiumsupport/knowledge-center/configure-hugepages-ec2-linux-instance/, Feb. 2021.

[6] Hyper-V Configuration. https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/role/hyper-v-server/configuration, Feb. 2021.

[7] Intel 64 and ia-32 architectures software developer manuals. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html, Feb. 2021.

[8] Intel 6th core product family uncore perfomrance monitoring guide. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/6th-gen-core-family-uncore-performance-monitoring-manual.pdf, Feb. 2021.

[9] Intel Atom Processor. http://caxapa.ru/thumbs/135047/2008_07_01_(1)_Atom_Day_45nm%26Markets%26Ato.pdf, Feb. 2021.

[10] Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization. https://www.intel.com/software/pcm, Feb. 2021.

[11] Intel pmc core. https://01.org/blogs/rajneesh/2019/using-power-management-controller-drivers-debug-low-power-platform-states, Feb. 2021.

[12] Intel rapl. https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl, Feb. 2021.

[13] intel_idle.c. https://elixir.bootlin.com/linux/latest/source/drivers/idle/intel_idle.c, Feb. 2021.

[14] Mitiga recommends all aws customers running community amis to verify them for malicious code. https://medium.com/mitiga-

io/security-advisory-mitiga-recommends-all-aws-customers-running-community-amis-to-verify-them-for-5c3e8b47d2d8, Feb. 2021.

[15] Processor state control for your EC2 instance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html, Feb. 2021.

[16] Testing the Windows Subsystem for Linux. https://docs.microsoft.com/en-us/archive/blogs/wsl/testing-the-windows-subsystem-for-linux, Feb. 2021.

[17] Tsc frequency variations with temperature. https://community.intel.com/t5/Software-Tuning-Performance/TSC-frequency-variations-with-temperature/td-p/1126518, Feb. 2021.

[18] WSL 2 Post BUILD FAQ. https://devblogs.microsoft.com/commandline/wsl-2-post-build-faq/, Feb. 2021.

[19] M. Alagappan, J. Rajendran, M. Doroslovački, and G. Venkataramani. Dfs covert channels on multi-core platforms. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. IEEE, 2017.

[20] S. Balasubramanian, T. Thomas, S. Shrimali, and B. Ganesan. Reducing power consumption of uncore circuitry of a processor, Nov. 18 2014. US Patent 8,892,924.

[21] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[22] D. B. Bartolini, P. Miedl, and L. Thiele. On the capacity of thermal covert channels in multicores. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[23] M. K. Bhandaru, A. Varma, J. R. Vash, M. Wong-Chan, E. J. Dehaemer, C. A. Poirier, S. P. Bobholz, et al. Dynamically controlling interconnect frequency in a processor, Apr. 26 2016. US Patent 9,323,316.

[24] J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, and J. Sopena. The battle of the schedulers: Freebsd ULE vs. linux CFS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 85–96, 2018.

[25] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[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*, 8(1):1–27, 2018.

[27] N. Gholkar, F. Mueller, and B. Rountree. Uncore power scavenger: A runtime for uncore power conservation on hpc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–23, 2019.

[28] V. Govtva. Intel xeon server cpu maximum wake latency measurement. 2019.

[29] 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*, pages 279–299. Springer, 2016.

[30] V. Gupta, P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan, and G. Srinivasa. The forgotten 'uncore': On the energy-efficiency of heterogeneous cores. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 367–372, 2012.

[31] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. An energy efficiency feature survey of the intel haswell processor. In *2015 IEEE international parallel and distributed processing symposium workshop*, pages 896–904. IEEE, 2015.

[32] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[33] 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*, pages 591–604. IEEE, 2015.

[34] 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*, pages 353–364, 2016.

[35] M. Kalmbach, M. Gottschlag, T. Schmidt, and F. Bellosa. Turbocc: A practical frequency-based covert channel with intel turbo boost. *arXiv preprint arXiv:2007.07046*, 2020.

[36] S. K. Khatamifard, L. Wang, A. Das, S. Kose, and U. R. Karpuzcu. Powert channels: A novel class of covert communicationexploiting power management vulnerabilities. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 291–303. IEEE, 2019.

[37] 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)*, pages 1–19. IEEE, 2019.

[38] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power management of datacenter workloads using per-core power gating. *IEEE Computer Architecture Letters*, 8(2):48–51, 2009.

[39] P. Lifshits, R. Forte, Y. Hoshen, M. Halpern, M. Philipose, M. Tiwari, and M. Silberstein. Power to peep-all: Inference attacks by malicious batteries on mobile devices. *Proceedings on Privacy Enhancing Technologies*, 2018(4):141–158, 2018.

[40] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[41] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss. Platypus: Software-based power side-channel attacks on x86. 2021.

[42] 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*, pages 605–622. IEEE, 2015.

[43] Z. Long, X. Wang, Y. Jiang, G. Cui, L. Zhang, and T. Mak. Improving the efficiency of thermal covert channels in multi-/many-core systems. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1459–1464. IEEE, 2018.

[44] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129. IEEE, 2012.

[45] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun. Thermal covert channels on multi-core platforms. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 865–880, 2015.

[46] C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64. Springer, 2015.

[47] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer. Hello from the other side: Ssh over robust cache covert channels in the cloud. In *NDSS*, volume 17, pages 8–11, 2017.

[48] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. *ACM SIGARCH Computer Architecture News*, 37(1):205–216, 2009.

[49] P. Miedl, X. He, M. Meyer, D. B. Bartolini, and L. Thiele. Frequency scaling as a security threat on multicore systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2497–2508, 2018.

[50] P. Miedl and L. Thiele. The security risks of power measurements in multicores. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1585–1592, 2018.

[51] V. Pallipadi, S. Li, and A. Belay. cpuidle - do nothing, efficiently... https://www.kernel.org/doc/ols/2007/ols2007v2-pages-119-126.pdf, Feb. 2021.

[52] 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)*, pages 565–581, 2016.

[53] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE Symposium on Security & Privacy*, 2021.

[54] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*, pages 1–13, 2012.

[55] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[56] R. Schöne, D. Molka, and M. Werner. Wake-up latencies for processor idle states on current x86 processors. *Computer Science-Research and Development*, 30(2):219–227, 2015.

[57] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.

[58] N. Sehatbakhsh, B. B. Yilmaz, A. Zajic, and M. Prvulovic. A new side-channel vulnerability on modern computers by exploiting electromagnetic emanations from the power management unit. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 123–138. IEEE, 2020.

[59] D. X. Song, D. A. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium*, volume 2001, 2001.

[60] 6th generation intel® processor families for s-platforms, datasheet, volume 1 of 2. https://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-datasheet-vol-1.html, Feb. 2021.

[61] D. Sullivan, O. Arias, T. Meade, and Y. Jin. Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in iaas clouds. In *NDSS*, 2018.

[62] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[63] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.

[64] 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)*, pages 913–928, 2015.

[65] I. S. Winkler and B. Dealy. Information security technology? don't rely on it. a case study in social engineering. In *USENIX Security Symposium*, volume 5, pages 1–1, 1995.

[66] 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*, 23(2):603–615, 2014.

[67] Z. Xu, H. Wang, and Z. Wu. A measurement study on co-residence threat inside the cloud. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 929–944, 2015.

[68] 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)*, pages 888–904. IEEE, 2019.

[69] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316, 2012.

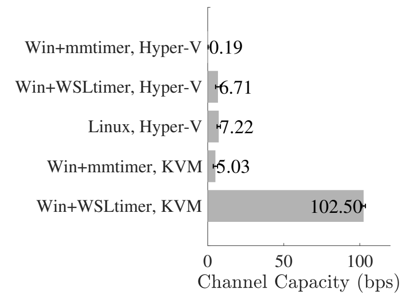


Figure 11: **IPMD Channel in Windows-based Platforms.** Labels are denoted in (left, right). The left label denotes the victim VM and timer configuration. The right label denotes the the hypervisor. The OS of the attacker VM is Linux. Experiments with Hyper-V hypervisor is conducted in the Microsoft Azure Cloud.

Appendix

A. Attacks on Windows Platforms

We validate the IPMD channel in the Windows OS and the Hyper-V hypervisor. A major difference from Linux-based settings is that the resolution and accuracy of the Windows timers are much lower, which limits the capacity of the IPMD channel.

We first use the server platform to validate the IPMD channel with the Windows guests on the KVM hypervisor. The guest OS is the Windows Server Standard 2019 v1809. The win server is assigned with 8 GB RAM. `hypervclock` is enabled [4] in KVM. Other settings are the same as in §5.4.1. Results are shown in Figure 11. Bars in Figure 11 denote the largest channel capacity achieved through varying different T_s and T_{Sym} combinations.

Compared with Figure 6 (b), the capacity (Win+mmtimer, KVM) drops by an order. This is mainly because of the win timer. According to our tests, its finest accuracy in our platform is no better than 1 ms (through `mmtimer` API), which is much coarser than the `hrtimer` in Linux, resulting in timing jitters in the symbols. The impact of the timer mechanism can also be validated through comparing with the (Win+WSLtimer, KVM), where WSL is short for the Windows Subsystem for Linux 1 or WSL 1 [8], which emulates Linux system calls through a translation layer in Windows [16]. The accuracy of WSL timers is around 0.5 ms, which is better than the native win timers. As a result, the IPMD channel achieves comparable capacity as the Linux OS. We guess the WSL timer is probably based on lower-level and independent mechanisms.

We set up the Hyper-V hypervisor in two cases. The in-house one is our server with Hyper-V server 2019. The cloud one is based on the Dsv3_Type2 (Intel Platinum 8171M,

[8] WSL 2 is based on the Hyper-V architecture [18]

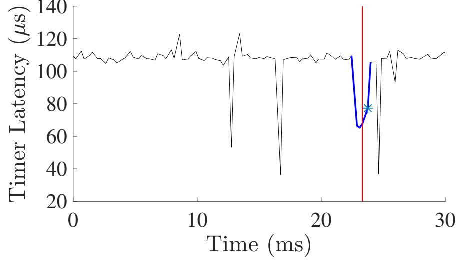


Figure 12: **Raw Latency Trace of SSH Keystrokes (Desktop platform).** The trace is sampled in the attacker VM when the victim VM is accessed through `SSH`. The red line denotes the timing of the keystroke from the `stdin` of the victim VM. The blue curve outlines the spike caused by the `SSH` processing. We define the spikes as the downwards areas below the average latency. We identify the keystroke spikes according to three heuristic rules: 1. the width of the bottom of the spike is within 3 to 6 samples, *i.e.*, (0.66, 1.32) ms; 2. the mean latency of the spike is between 60 μs and 80 μs; 3. The last-second sample (the blue star) is greater than 30 μs.

Skylake-SP, and 504 GB RAM) dedicated host in Microsoft Azure Cloud. As their performance is close, Figure 11 only depicts the Azure case.

The guest software settings are the same as KVM case. Three D2s_v3 instances are launched on the dedicated host. One for the attacker VM. The other two for the Linux VM and Windows VM respectively. Each D2s_v3 instance is assigned with two vCPU.

When using the Hyper-V hypervisor, due to the inaccuracy of the underlying win timer, the performance of most cases are close to the (Win+mmtimer, KVM) case. The (Win+mmtimer, Hyper-V) is the worst as there are two layers of win timers.

Another impacting factor is the Windows power plan, which specifies the behavior of the hypervisor power management mechanism [6]. We can only conduct this test with our in-house servers. There is no much difference among different plans in terms of the capacity. Compared with the default Balanced plan, the High Performance plan does not fix CC to CC0, nor does it disable PCs, but the cores tend to operate with higher frequencies. The above is observed through the Intel `pcm` [10] tool.

B. Raw Trace of SSH Keystrokes

Figure 12 illustrates the raw latency trace collected in the attacker VM when the co-located victim VM is accessed through `SSH` from a remote host. Both the `SSH` server and client are OpenSSH_7.9p1. We observe that the raw traces of different ciphers are slightly different. The results of the default cipher `chacha20-poly1305@openssh.com` are the most stable. This is probably because the ChaCha algorithms have not been fully hardware accelerated, which incurs a larger software footprint.

C. *Source* and Timer Latency Measurement in *Sink*

Algorithm 1 Source Process: send frame via IPMD channel

```
1: procedure SENDFRAME(frame, frameLen, T_Sym)
2:     for i ← 1, frameLen do
3:         if frame[i] = 1 then
4:             Execute something for T_Sym
                        ▷ to keep the occupied core active
                        ▷ T_Sym is the duration of a symbol
5:         else
6:             sleep(T_Sym)
7:         end if
8:     end for
9: end procedure
```

Algorithm 2 Measure exit latency via timer latency

```
1: procedure MEASURE EXIT LATENCY(T_s)
2:     while true do
3:         record T_pre
4:         sleep (T_s)
5:         record T_post
6:         T_total[i++] ≈ T_post − T_pre − T_s
7:     end while
8: end procedure
```