Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs

Fabian Rauscher
Graz University of Technology
Graz, Styria, Austria
fabian.rauscher@iaik.tugraz.at

Daniel Gruss
Graz University of Technology
Graz, Styria, Austria
daniel.gruss@iaik.tugraz.at

Abstract

Interrupts are fundamental for inter-process and cross-core communication in modern systems. Controlling these communication mechanisms historically requires switches into the kernel or hypervisor, incurring high-performance costs. To alleviate these costs, Intel introduced new hardware mechanisms to send inter-processor interrupts (IPIs) from user space without switching into the kernel and from virtual machines without switching into the hypervisor. However, it is unclear whether this direct, unsupervised interaction between unprivileged (or virtualized) workloads and the underlying hardware introduces a significant change in the attack surface.

In this paper, we present the IPI side channel, a novel side-channel attack exploiting the recently introduced user interrupts and IPI virtualization features on Intel Sapphire Rapids and the upcoming Intel Arrow Lake processors. The IPI side channel is the first cross-core interrupt detection side channel, allowing an attacker to monitor interrupts delivered to any physical core of the same processor. Our attack is based on precise measurements of the hardware delivery time of interrupts from user space and virtual machines. More specifically, we exploit that interrupts are delivered through a cross-core bus, leading to timing variations on the attacker's local IPIs. We present multiple case studies to compare the IPI side channel with the state of the art: First, we present an unprivileged cross-core covert channel with a native true capacity of 434.7 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) and a cross-VM capacity of 3.45 kbit/s (n=100, $\sigma_{\bar{x}}$=0.01). Second, we demonstrate a native inter-keystroke timing attack with an F_1 score of 97.9 %. Third, we present an open-world website fingerprinting attack on the top 100 websites, achieving an F_1 score of 89.0 % in a native scenario and an F_1 score of 71.0 % in a cross-VM (thin client) scenario. Furthermore, we discuss the broader context of the IPI side channels and categorize interrupt side channels and mitigations.

CCS Concepts

• **Security and privacy → Side-channel analysis and countermeasures**; **Operating systems security**; **Systems security**.

Keywords

side-channel attack, user interrupts, IPI virtualization, interrupt detection, website fingerprinting

This work is licensed under a Creative Commons Attribution International 4.0 License.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690242

ACM Reference Format:
Fabian Rauscher and Daniel Gruss. 2024. Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690242

1 Introduction

Modern computer systems are highly parallelized, with hundreds of tasks with different privileges and access permissions that are isolated from each other through process isolation or virtualization. Process isolation and virtualization are enforced by the hardware and configured by privileged software, e.g., operating system or hypervisor [33]. However, tasks still frequently need to communicate with each other, e.g., to exchange data or synchronize operations, the system software provides means for cross-process and cross-core communication, including shared memory, signals, pipes, and various interrupts. All of these mechanisms require interaction with the system software, in some cases, e.g., for signal and interrupts, even for every single instance. As this incurs high context switch overheads, Intel introduced two new features called *user interrupts* [35] and *IPI virtualization* on Intel Sapphire Rapids processors. While these features are currently only available on Xeon CPUs, they will be available on the next generation of Intel consumer CPUs called Arrow Lake. User interrupts and IPI virtualization are intended to minimize cross-process and cross-core communication overheads by allowing user tasks and virtual machines to directly send and receive interrupts without invoking the kernel or hypervisor.

Side-channel attacks exploit information channels that carry information derived from a secret value. In particular, timing side-channel attacks [40] gained a significant amount of attention as they can easily be mounted by an adversary controlling a piece of software on a victim system [63], or even remotely [4]. Besides caches as a popular attack target [25, 60, 104], also other microarchitectural components have been attacked [1, 16, 17, 20, 64]. Several works studied information leakage from interrupt timings, as they carry information about user input, e.g., mouse movements or keystroke presses on the keyboard, which typically trigger interrupts [13, 67, 80, 106]. If an attacker can accurately detect interrupts, they can infer the inter-keystroke timings, and consequently the written text, in a side-channel attack [81, 105]. Recent works also demonstrated these attacks from JavaScript [46]. Another common attack scenario using the interrupt channel is website- or video-fingerprinting. Cook et al. [13] used interrupt detection with a hot loop to mount a website fingerprinting attack. Zhang et al. [106] and Rauscher et al. [67] exploited interrupt detection to fingerprint

websites and videos. These previous works on interrupt side channels achieved high F_1 scores in fingerprinting scenarios. However, they require physical placement of the attacker on the specific physical core that receives the interrupts. Hence, if the attacker cannot run on the core receiving the interrupts, these attacks are thwarted.

Therefore, we ask the following questions:
How can an attacker observe interrupts received by other cores? Do the unprivileged user interrupts and the IPI virtualization features facilitate new interrupt side-channel attacks?

In this paper, we present the IPI side channel, the first cross-core cross-VM interrupt side channel: Our attack observes all interrupts irrespective of the interrupt target cores, *i.e.*, the victim can run on any other core, in a different process or virtual machine. We exploit the recently introduced user interrupts and IPI virtualization features on Intel Sapphire Rapids and the upcoming Intel Arrow Lake processors. User interrupts allow an attacker to send interprocessor interrupts (IPIs) to the attacker's own threads without any privileges and measure their delivery time. IPI virtualization allows an attacker to send IPIs inside of virtual machines without hypervisor intervention allowing for precise IPI time measurements. Our IPI side channel exploits that even when IPIs do not go to the attacker's own core, they run through a cross-core system bus [30]. Consequently, any activity on the system bus leads to timing variations on the attacker's local IPIs.

We evaluate the IPI side channel in multiple case studies and compare it with other state-of-the-art side channels: First, we present a cross-core covert channel between two unprivileged user processes using user interrupts and a cross-core covert channel between two virtual machines using virtualized IPIs. We achieve a true capacity of 434.7 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) in a native scenario and 3.45 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) in a cross-VM scenario. Second, we demonstrate an inter-keystroke timing attack using user interrupts with an F_1 score of 97.9 %. Third, we present a website fingerprinting side-channel attack on the top 100 websites. We evaluate our attack both in a closed-world native fingerprinting scenario, achieving an F_1 score of 91.7 %, and in an open-world native fingerprinting scenario with an additional class for other websites, achieving an F_1 score of 89.0 %. Furthermore, we demonstrate our website fingerprinting attack in a cross-VM attack achieving an F_1 scores of 80.4 % (closed-world) and 71.0 % (open-world).

Our IPI side channel is a significant improvement over prior interrupt side-channel attacks: Prior IPI side channels either relied on a software interface that is easy to constrain and already constrained on many systems [19], or on same-physical-core interrupt detection techniques [13, 46, 67, 74]. Unrestricted cross-core and cross-VM interrupt detection substantially shifts the threat model, enabling attacks regardless of where attacker or victim are scheduled and which core receives the interrupts. We discuss this context of our work as well as mitigations against interrupt side channels.

To summarize, we make the following contributions:

- We present the first cross-VM cross-core interrupt detection side-channel attack, based on direct access to IPIs from user space (user IPIs) and virtual machines (virtualized IPIs).
- We show that the IPI side channel can be used to leak up to 434.7 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) in a native cross-core covert channel and 3.45 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) cross-VM.
- We present a native inter-keystroke timing attack exploiting the IPI side channel with an F_1 score of 97.9 %.
- We present website-fingerprinting attacks with F_1 scores of 91.7 % (closed-world) and 89.0 % (open-world) with native user IPIs, and 80.4 % (closed-world) and 71.0 % (open-world) in a cross-VM scenario without attacker access to the core receiving the victim interrupts.

Outline. Section 2 provides background on side channels and interrupt detection. Section 3 explains user interrupts and IPI virtualization and discusses the basic idea of the IPI side channel. Section 4 evaluates the IPI side channel using native and cross-VM cross-core covert channel. Section 5 presents our inter-keystroke timing side-channel attack. Section 6 presents our website fingerprinting attack in a closed- and an open-world scenario. Section 7 presents our systematic comparison of published interrupt side-channel attacks and mitigations and discusses related work. Section 8 concludes.

Responsible Disclosure. We responsibly disclosed our findings to Intel (February 21, 2024) and shared a paper draft with them. Intel recommends software authors follow Intel's software guidance on side-channel resistance.

2 Background

In this section, we provide background on side-channel attacks, interrupt detection, and website fingerprinting attacks as a typical side-channel evaluation scenario.

2.1 Hardware-based Virtualization

Virtualization allows running a full system (kernel and userspace programs) inside a virtual environment. A hypervisor monitors and manages these virtual machines (VMs) and mediates access to the hardware. Initial virtualization technology was based on full system emulation and later para-virtualization [95]. To reduce the performance overheads of these purely software-based solutions, hardware vendors introduced instruction set extensions to drastically improve the performance of virtualization. Intel introduced their Virtualization Technology extension (VT-x) [89], featuring a new CPU execution mode allowing a hypervisor to run at a higher privilege level than the operating system (*i.e.*, VMX root operation) [33]. The hypervisor controls a VM control structure to configure the VM. Accessing hardware resources or interacting with the hypervisor is done through interrupts or `vmcall`, which trigger VM exit operations, handing over control to the hypervisor. However, context switches from and to a VM are expensive [44] as they require configuring the VM control structures and often the flushing of buffers and even caches [98]. Consequently, to optimize the performance of systems running VMs, it is crucial to avoid expensive context switch mechanisms [44]. Following this intuition, Intel also introduced more hardware extensions to facilitate the reduction of context switches. A recent addition in this direction is IPI virtualization [31], introduced with Intel Xeon Sapphire Rapids and Arrow Lake CPUs. IPI virtualization allows VMs to directly send IPIs without the expensive switch to the hypervisor [31].

2.2 Side-Channel Attacks

Instead of exploiting software vulnerabilities, side channels exploit side effects of the implementation such as timing [40], power consumption [41], or radiation [66]. Many works focus on cryptographic primitives [4, 6, 40], leaking keys of vulnerable cryptographic implementations of e.g., AES [4, 60], RSA [6, 104], or ECDSA [103]. More recent works also focus on leakage on the system level, e.g., kernel information [29], user input [23, 57, 69], and other system activity [21]. Several works attempt to automate side-channel attacks, especially software-based side-channel attacks [7, 18, 23, 68, 78]. Among software-based side channels, especially cache side channels have taken a central role in the system security research community with generic techniques like Flush+Reload [104], Prime+Probe [50, 54, 60], and Flush+Flush [22]. Observing inter-keystroke timings is an interesting target for side-channel attacks, as it bypasses the security guarantees of any cryptographic algorithm applied by targeting the user's password instead [57]. Attacks on keystrokes are also difficult to mitigate as they run through an extensive code path: The software starts handling an interrupt in low-level kernel interrupt handler code and goes through kernel processing, library processing, and application processing until the keystroke shows a visual response to the user or is transmitted to the target buffer [74]. Inter-keystroke timings contain so much information that an attacker cannot only infer typed text [57] but also obtain privacy-related information, e.g., identify specific users [57]. Many works use machine learning to derive the secret input from the inter-keystroke timing trace [80, 81, 105].

In a more controlled setting, covert channels are a standard means to evaluate a side channel. In practice, covert channels can be relevant to exfiltrate secrets from co-located VMs [53, 55, 84, 101]. Covert channels are suitable to estimate the amount of noise and accuracy of a channel, and consequently, to provide a practical upper bounds for the leakage rate of the side channel [55]. Covert channels exploiting SMT, *i.e.*, two workloads share a physical core, and covert channels exploiting the cache, often reach the range of multiple megabytes per second [22, 70]. However, covert channels on other microarchitectural elements are often in the range of a few bytes per second [16, 101].

2.3 Interrupt Detection

Interrupts are commonly categorized into interrupts, faults, and traps [33]: Traps are intentionally configured to interrupt a process when a certain condition is reached. This can, for instance, be a certain memory access or reaching a specific location in the code. Faults occur when the processor cannot handle an issue in the instruction stream, handing over control to the operating system to decide what to do. Examples for faults are page faults, general protection faults, or a division by zero. Interrupts occur upon events that are not part of the implemented instruction stream of the program. For instance, keystrokes can occur at any time and need to interrupt the running workload. The same also holds for other external interrupt sources, including network interrupts, disk interrupts, or interrupts caused by other input devices, e.g., the mouse. Consequently, prior work showed that observing interrupts, inherently allows to spy on these events [15, 46, 80] or even learn about the interrupted instruction stream [92]. Hence, observing interrupts and mitigating their observability has been identified as a direct path to inter-keystroke attacks and their mitigation [74].

Schwarz et al. [74] exploited that jumps in the timestamps returned by the `rdtsc` instruction indicate whether an interrupt occurred. Lipp et al. [46] demonstrated a similar attack using a JavaScript-based counting thread. Zhang et al. [106] and Rauscher et al. [67] exploited the `umwait` and `tpause` instructions to detect interrupts. They evaluated their attacks in website fingerprinting attacks, video-fingerprinting attacks, inter-keystroke detection, and covert channels to measure the side-channel capacity.

2.4 Website Fingerprinting

Website fingerprinting is a common side-channel evaluation scenario, *i.e.*, it serves as a benchmark to compare performance side channels. Consequently, there is a broad range of different side channels that perform website fingerprinting attacks, reporting various accuracies: Spreitzer et al. [82] achieved an accuracy of 89 % on 100 websites using the data-usage statistics on Android. Jana et al. [37] exploited the memory usage statistics of browsers and reported an accuracy between 30 % and 50 % for the top 100 000 websites. Gulmezoglu et al. [26] used hardware performance events and achieved accuracy of 86.3 % on 40 websites. Qin and Yue [65] used the power side channel on Android to fingerprint websites, achieving an accuracy of 55 %. Shusterman et al. [79] reported a website fingerprinting accuracy between 45.4 % (on the Tor browser) and 91.4 % (in the best case). Cook et al. [13] reported an accuracy of up to 97.2 % in an open-world website fingerprinting scenario and up to 96.6 % in a closed-world website fingerprinting scenario with the top 100 websites, based on an interrupt timing side channel similar to that of Lipp et al. [46]. Zhang et al. [106] monitored interrupts using idle states from native code and reported an F_1 score of 70 % over the Alexa top 100 websites. In a similar attack, Rauscher et al. [67] achieved F_1 scores of 85.2 % and 93.1 % in an open- and closed-world evaluation over the Alexa top 100 websites.

3 IPI Side Channel

In this section, we present the attack primitive we use for cross-core and cross-VM interrupt detection. We provide an overview of user interrupts and how they can be triggered and received in userspace by an unprivileged attacker. We then provide an overview of IPI virtualization, allowing a VM to send inter-processor interrupts (IPIs) without hypervisor intervention, allowing for precise observation of their behavior and side effects. Lastly, we show the potential timing leakage of these new interrupt features to build an attack primitive. With these attack primitives, we can observe other interrupts on the same CPU to leak information from co-located workloads, such as network activity or keystrokes.

3.1 User Interrupts

User interrupts were introduced with the Intel Xeon Sapphire Rapids CPUs. The user interrupts feature **will** also be available on the upcoming Arrow Lake consumer CPUs. They allow the user to send and receive user inter-processor interrupts (user IPIs) using the `senduipi` instruction. Additionally, the system software can post user interrupts and send user-interrupt notifications. User IPIs, in particular, allow for fast inter-process communication.

At the time of writing, user interrupts are not officially supported by the Linux kernel. For our experiments, we use Linux with the official Intel implementation for user interrupt support [35]. While we use Intel's official patch, the exact software support implementation does not affect our results, as the attacks proposed in this paper rely mainly on the hardware implementation of user interrupts.

Both sending and receiving user IPIs is only possible in user space. If a user IPI is sent while the receiver thread is in kernel space or not running, the user IPI is buffered until the receiver thread is scheduled in user space. To allow for fast delivery when the receiver thread is running, the operating system reserves an interrupt vector in the advanced programmable interrupt controller (APIC) for user interrupts called the user-interrupt notification vector (UINV). The `senduipi` instruction sends an IPI with vector UINV to the core the receiver thread is running on. When the APIC receives an interrupt with the vector UINV, the CPU automatically performs checks for outstanding user interrupts instead of performing normal interrupt handling through the interrupt descriptor table (IDT). If the current thread has outstanding user interrupts, a user interrupt is triggered if the thread is in user space or set to pending if it is in the kernel. The CPU also checks for pending user interrupts when a thread state is restored, e.g., on a context switch.

While sending and receiving user IPIs can be done entirely in user space, user interrupts require support from the kernel. Among other things, the kernel manages the target cores for user interrupts, stores the handler address for the interrupts, determines which user interrupts are currently active, and which threads can receive which user interrupts. Specifically, in the setup process of user interrupts, support by the kernel is required. Intel proposes syscalls for registering and unregistering an interrupt handler as well as interrupt vectors for the receiver. Registering an interrupt vector as a receiver returns a file descriptor. Another thread can use this file descriptor to register as a sender for a given interrupt vector [35]. After the initial setup, the sender can trigger user IPIs with the `senduipi` instruction. Consequently, it is only possible for a thread to send user IPIs to threads that opt-in to receive them and only if the receiver shares the file descriptor with them.

The delivery delay of user IPIs and POSIX signals is shown in Figure 1. We measure the delay by executing `rdtsc` right before sending the POSIX signal or user IPI and as the first instruction in the handler function. The difference between the two TSC values is the minimum time between the signal or IPI being sent and the handler being executed from the user's perspective. User IPIs perform better by a factor of 4 with a delay of only 1256.4 cycles (n=10^6, $\sigma_{\bar{x}}$=0.05) compared to POSIX signals at 5179.6 cycles (n=10^6, $\sigma_{\bar{x}}$=0.29). When the sender and receiver are the same thread, user IPIs perform better by a factor of 3 with a delay of only 726.8 cycles (n=10^6, $\sigma_{\bar{x}}$=0.06) compared to POSIX signals at 2398.0 cycles (n=10^6, $\sigma_{\bar{x}}$=0.07). POSIX signals have a significant kernel overhead, as sending a signal involves a syscall, which then delivers the signal, possibly by sending a regular IPI or setting it pending, waiting for the receiver to detect it. User IPIs have no significant kernel overhead, resulting in fast delivery times. These results show that user IPIs are a valuable addition for fast inter-process communication.

3.2 IPI Virtualization

IPI virtualization was introduced with the Intel Xeon Sapphire Rapids CPUs and Intel Arrow Lake CPUs. Furthermore, IPI virtualization **will** be available on Arrow Lake CPUs, the next generation of consumer Intel CPUs. This feature allows a VM to post IPIs without generating a VM exit. Contrary to user IPIs, IPI virtualization is already part of the Linux kernel and is activated per default (if available) with KVM since Linux 6.0 [38].

While it was already possible for the host to send so-called virtual interrupts to running VMs on a different core using the process posted interrupts feature and for the guest to receive them through virtual interrupt delivery without a VM exit, a VM couldn't send IPIs without causing a VM exit. Similar to user IPIs, posted-interrupt processing uses an interrupt vector that the operating system can designate for virtual interrupts. When a core running a VM receives such an interrupt, the core will check for any open posted interrupts and trigger them immediately, if possible. Previously, when sending IPIs, writes to the corresponding advanced programmable interrupt controller (APIC) field would cause a VM exit. With IPI virtualization, these writes are virtualized and post interrupts directly, using the posted-interrupt processing mechanism. Thus, the guest can send IPIs between cores without VM exits while sender and receiver are running, reducing the overhead of IPIs in VMs significantly. According to an Intel engineer, this reduces the delivery time of IPIs inside of VMs by up to 22.21 % [24]. As operating systems regularly use IPIs for functionalities such as TLB shootdowns, faster IPI processing can have a significant performance impact on inter-processor communication. Furthermore, this feature allows VMs to also take full advantage of user IPIs, as user IPIs can use IPI virtualization to not cause VM exits.

3.3 Timing Behaviour

In this section, we discuss the timing behavior of user IPIs and IPIs sent through IPI virtualization.

3.3.1 User IPIs. To determine the impact of other interrupts on the delivery time of user IPIs, we ran a measurement thread that continuously sends user IPIs to itself and measures the time between the `senduipi` instruction and the interrupt service routine using `rdtsc`. An example of the measurement code is shown in Listing 1. We trigger different interrupt types to arrive on other cores. Figure 2 shows the result of these measurements. To provide a better overview, we filtered all measurements in the typical user IPI delivery range from the core to itself (< 900 cycles). We grouped the remaining interrupts into 10 million cycle large bins. Figure 2 shows how many unusually slow user IPIs were measured at a given time. We refer to this trace as an *interrupt trace*. We performed these measurements with a download in the background and the network interrupts arriving on a different core (Figure 2a), with a different thread sending user IPIs to another thread (Figure 2b), and a different thread sending regular IPIs (TLB shootdowns) to another thread (Figure 2c). In Figure 2a, the download starts at $\approx$ 10 billion cycles, resulting in a slight increase in unusually slow interrupts. The download ends at $\approx$ 20 billion cycles, resulting in a slight increase in unusually slow interrupts. In Figure 2b, the user IPIs start at $\approx$ 16 billion cycles, with a drastic increase in unusually slow interrupts. This increase is significantly higher than for network interrupts, as

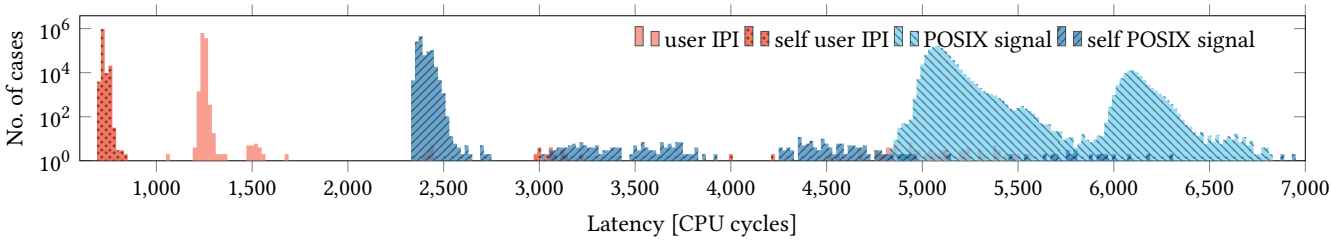


Figure 1: User IPI and POSIX signal delivery time in cycles. User IPIs are 4 times faster (3 in case of a self-interrupt).

```
volatile size_t end;

void __attribute__ ((interrupt))
handler(struct __uintr_frame *,
 -> unsigned long) {
  end = rdtsc();
}
void attack() {
  int ret =
 -> uintr_register_handler(handler, 0);
  int uintr_fd = uintr_create_fd(1, 0);
  stui();
  int uipi_handle =
 -> uintr_register_sender(uintr_fd, 0);

  for (;;) {
    end = 0;
    size_t start = rdtsc();
    senduipi(uipi_handle);
    while (!end);
    auto x = end-start;
    if (x < 900) continue;
    //<interrupt detected>
    //<further processing>
  }
}
```

Listing 1: User-interrupt side-channel measurement code.

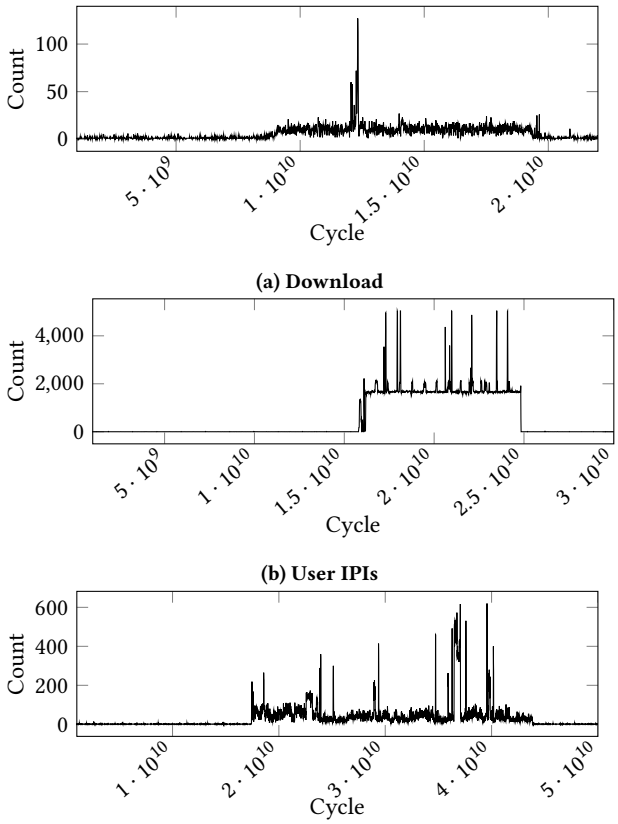


Figure 2: User IPI timings with different tasks generating interrupts on other cores in the background.

the user IPIs are sent at a significantly higher frequency than the typical arrival rate of network packets. The interrupt trace reaches almost 0 at ≈ 25 billion cycles when the user IPIs stop. In Figure 2c, the regular IPIs start at ≈ 20 billion cycles, resulting in a drastic increase in unusually slow interrupts. The interrupt trace reaches almost 0 at ≈ 45 billion cycles when the IPIs stop.

3.3.2 IPI virtualization. To determine the impact of other interrupts on the delivery time of IPIs inside VMs, we ran a measurement thread that continuously sends IPIs to itself inside of a VM (virtualized IPIs) and measures the time between the interrupt sent and the interrupt service routine using `rdtsc`. While the core sends IPIs to itself, we do not use self-IPIs. Self-IPIs are a special kind of IPI that allow a core to send an IPI to itself with low-performance overhead. As self-IPIs are not targeting other cores, they do not cause any contention on the shared system bus. Instead, we use regular IPIs, which allow for a target core to be specified, which we set to the core that sends the IPI. This will trigger an IPI that is sent out to the system bus and received by the core, making it affected by possible contention. Such measurements on an idle system are shown in Figure 3. An IPI from a core to itself takes 2148.1 (n=15 $*$ 10^6, $\sigma_{\bar{x}}$=68.3) cycles.

Our measurement code consists of a kernel module that registers a custom interrupt handler, repeatedly sends virtualized IPIs to itself, and measures the delivery time. We trigger different interrupt types to arrive on other cores. Figure 4 shows the result of these

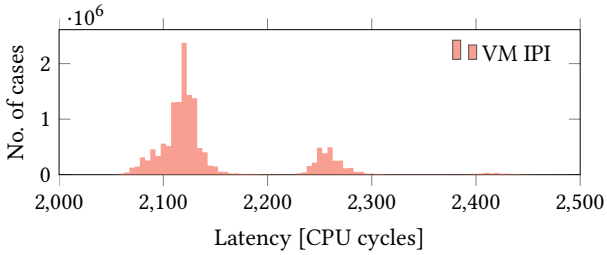
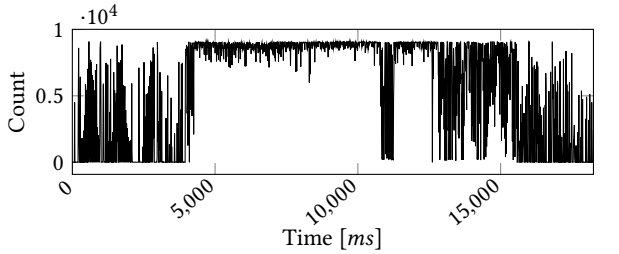
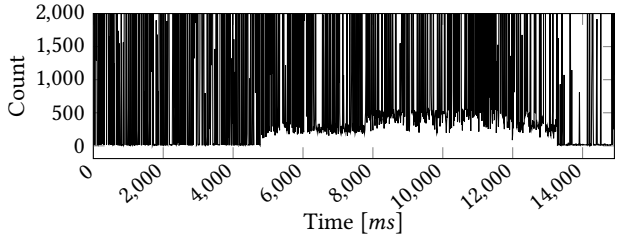


Figure 3: IPI timings of a core sending an IPI to itself.



(a) Download



(b) IPIs

Figure 4: IPI timings with different tasks generating interrupts on other cores inside of VMs in the background.

measurements. To provide a better overview, we filtered all measurements in the typical IPI delivery range from the core to itself (< 2200 cycles). We grouped the remaining interrupts into 10 ms large bins. Figure 4 shows how many unusually slow IPIs were measured at a given time. We performed these measurements with a download in a separate VM, and the network interrupts arriving on a different core (Figure 4a) and a different thread in a VM sending regular IPIs (TLB shootdowns) to another thread (Figure 4b). In Figure 2a, the download starts at 4 000 ms, drastically increasing the number of unusually slow interrupts. The download ends at 16 000 ms, resulting in a slight increase in unusually slow interrupts. In Figure 4b, the IPIs start at 5 000 ms, increasing the number of unusually slow interrupts. The interrupt trace goes back to almost 0 at 13 000 ms when the IPIs stop. The external interrupts from downloads, such as shown in Figure 4a, seem to have a significantly higher impact on the IPI latency than IPIs shown in Figure 4b.

3.3.3 Conclusion. These measurements show that it is possible to detect external interrupts and other IPIs, originating and targeting cores independent of the attacker. We assume this timing behavior results from contention on a shared bus used for delivering

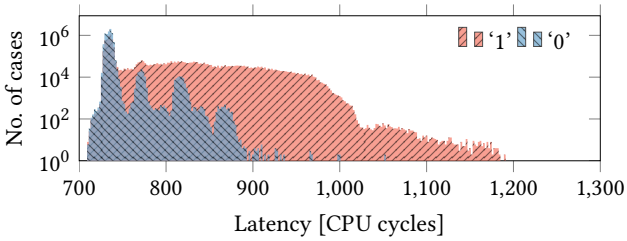


Figure 5: Difference between a '1' and a '0' for our user IPI covert channel.

interrupts. According to the Intel manual, this shared bus is the system bus on Xeon CPUs [32]. Despite this, we cannot detect other events that should result in an access to the system bus, such as cache coherency-related events between cores. Furthermore, we cannot detect other system activity, such as a high amount of cache evictions on other cores or from the shared L3 using the IPI side channel. This is not surprising, as cache evictions from other cores or the L3 should not significantly affect the attacker core due to the non-inclusive L3 of the Xeon CPU used. Therefore, even if the cache lines of the attacker are evicted from the L3 by other cores, they stay in the private L1 and L2 of the attacker's core. The IPI side-channel signal also disappears when replacing the IPI measurement code with a cache attacker or constant time code, or turning off IPI virtualization in case of the cross-VM attack. This further validates that the signal stems from the IPI delivery delay.

4 Covert Channel

In this section, we present a cross-core and a cross-VM covert channel based on the IPI side channel. The covert channel is based on the performance impact other interrupts on the system have on user IPIs and virtualized IPIs. Covert channels are the most commonly used scenario to evaluate new side channels [85].

4.1 Covert Channel Design

In this section, we provide a high-level overview of our user-interrupt covert channel. We use time-slicing in combination with the IPI side channel shown in Section 3.3 to transmit data.

We transmit data by either performing user IPIs (native) or virtualized IPIs (cross-VM) in the sender or by busy waiting and measuring the time IPIs take in the receiver. When a '1'-bit is sent, the sender sends IPIs to itself, and when a '0'-bit is sent, the sender busy waits. The receiver sends IPIs to itself and measures the timings.

The timings for the '1' and '0' cases for user IPIs are provided in Figure 5. A '1'-bit results in measured user IPI timings of 785.4 cycles (n=8 · 10^9, $\sigma_{\bar{x}}$=0.023) and a '0'-bit results in 735.7 cycles (n=8 · 10^9, $\sigma_{\bar{x}}$=0.003) making them clearly distinguishable. Despite this clear difference in the average IPI time, both cases significantly overlap, as shown in Figure 5. This overlap results in a noisy but still functional covert channel.

The timings for the cross-VM covert channel in a sample transmission are shown in Figure 6. A '1'-bit results in frequent spikes in the transmission time, while there are no spikes when a '0'-bit

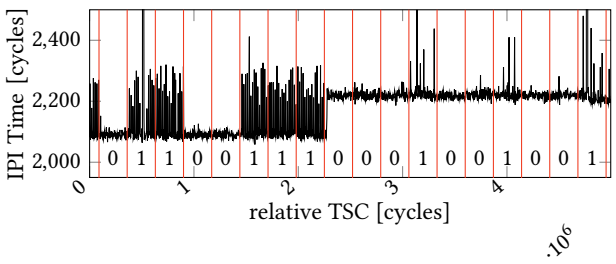


Figure 6: Example transmission of our cross-VM covert channel using virtualized IPIs.

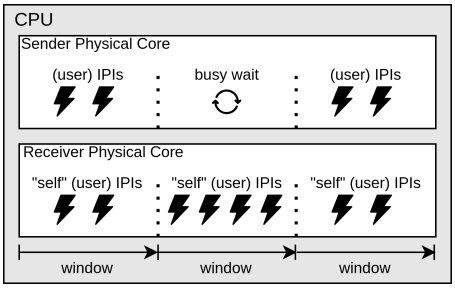


Figure 7: Overview over a short sample transmission using user IPIs. When a '1' is sent, the sender runs self-user IPIs, slowing down the receiver's user IPIs. When a '0' is sent, the sender busy waits, which does not affect the receiver's user IPIs.

is transmitted. The base latency of IPIs inside of VMs changes frequently, as shown in Figure 6, where the base latency is at ≈2 100 cycles for the first half of this example and at ≈2 250 cycles for the second half. This frequent change in the base latency is most likely due to other system events. A higher base latency also lessens the effect other IPIs have on the receiver's IPI latency. While the frequently changing base latency does make it more challenging to extract the sent bits, they are still distinguishable.

To synchronize the covert channel, we use the processor's time stamp counter (TSC) with the `rdtsc` instruction. The transmission for the native covert channel starts at a fixed TSC value or TSC modulo overflow. This reliably synchronizes sender and receiver. There is no direct communication between the sender and receiver. In the cross-VM covert channel, the initial synchronization occurs through the sender transmitting an initialization sequence, as the two VMs do not share the same TSC. Despite this, the TSC still increments at the same rate, allowing us to use it for further synchronization after establishing the beginning of the transmission.

The transmission itself is divided into fixed-sized transmission windows. Within each transmission window, one bit is sent. To send a bit, the sender either sends IPIs to itself throughout the transmission window or busy waits. The receiver continuously sends the IPIs to itself. After the transmission window is finished, the receiver determines the bit received using the timings of all its IPIs within this window.

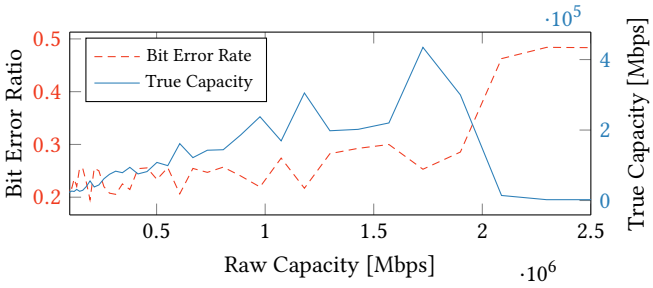


Figure 8: The raw capacity and corresponding bit error ratio of our covert channel, as well as the resulting true capacity. The optimal true capacity is reached at a raw capacity of 1 726.6 kbit/s and a bit error ratio of 25.3 % (n=100, $\sigma_{\bar{x}}$=1.4).

Figure 7 provides an overview of a typical transmission. In the first transmission window, the sender continuously sends IPIs to itself, slowing the IPIs of the receiver down to transmit a '1'. In the second window, the sender busy waits, not affecting the receiver, to transmit a '0'. Finally, in the third window, the sender, again, sends IPIs to itself, slowing the IPIs of the receiver down to transmit a '1'. This results in the transmission of the bit sequence '101'.

4.2 Evaluation

We evaluate our covert channel using random data on an Intel Xeon Silver 4410T. We tested two scenarios, native using user IPIs and cross-VM using virtualized IPIs. The sender and receiver run in separate processes, or VMs in the case of cross-VM, and are scheduled on two separate physical cores. Furthermore, we assume that there is no legitimate communication channel between them.

To determine the optimal transmission speed, we evaluate the covert channel for different transmission window lengths and record the raw capacity and bit error ratio. As our channel is based on time slices, decreasing the transmission window length increases the raw capacity. With a decrease in window length, the bit error ratio may increase, as there is less time for the receiver to determine the sent bit correctly. This decreases the true-channel capacity if the window length is too short due to the higher bit error ratio. To determine the optimal window length, we compute the true capacity of our channel using the binary symmetric channel model.[1]

Native. We take the average for the true capacity and the bit error ratio over 100 runs for each transmission window length. The results of our optimization are shown in Figure 8. The user-interrupt covert channel is noisy due to the low attack margin (see Figure 5). Furthermore, our covert channel is affected by all external interrupts and IPIs on the same CPU, resulting in a high bit error ratio. The optimal transmission speed of 434.7 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) is reached at a raw capacity of 1 726.6 kbit/s and a bit error ratio of 25.3 % (n=100, $\sigma_{\bar{x}}$=1.4). With a raw capacity higher than 1 726.6 kbit/s, the bit error ratio increases significantly, leading to a lower true capacity.

[1]We compute the true channel capacity T as $T = C \cdot (1 + ((1-p) \cdot \log_2(1-p) + p \cdot \log_2(p)))$ where C is the raw bit-rate and p the bit-error probability.

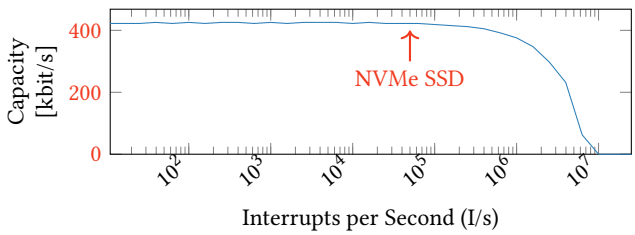


Figure 9: The channel capacity of our native covert channel with an increasing amount of interrupt noise using IPIs. The capacity is unaffected until a interrupts noise of ≈600 kI/s interrupts per second and drops to ≈0 bit/s at ≈10 MI/s. As a reference point, the NVMe SSD that is part of our test system can generate up to ≈50 kI/s.

Cross-VM. We optimized the cross-VM covert channel by hand due to the additional challenge of the initial synchronization of sender and receiver and the constantly changing base IPI latency in a VM scenario. This covert channel is affected by all external interrupts and IPIs on the same CPU, as well as further noise from the VMs, resulting in a high bit error ratio. Our cross-VM covert channel has a true capacity of 3.45 kbit/s (n=100, $\sigma_{\bar{x}}$=0.01) and a bit error ratio of 18.9 % (n=100, $\sigma_{\bar{x}}$=0,01). This lower capacity compared to the native scenario is the result of further noise introduced by the VM scenario, as well as the constantly changing base IPI latency, which makes bit extraction more challenging.

Noise Resilience. To determine the noise resilience of the IPI side channel, we ran our native covert channel with a variety of stressors. We ran the stressors and the attack on separate cores to rule out a drop in capacity due to the attacker not being scheduled, as we want to focus on the noise directly generated by the stressors. We used the stress-ng CPU benchmark to generate compute-intense noise and the stress-ng I/O benchmark to generate noise through constant disk accesses and, therefore, NVMe SSD interrupts. As expected, the CPU benchmark does not negatively influence the channel capacity when all cores run the stress-ng CPU stressor except for the sender and receiver cores. The stress-ng I/O stressor also does not influence the channel capacity. While the stress-ng I/O benchmark does constantly generate disk accesses, even in this high I/O scenario, our NVMe SSD only generates ≈50k interrupts per second (I/s), which is significantly lower than the covert channel's ≈2 MI/s. The interrupt stress generated by the NVMe SSD is too low to impact the attack significantly.

To determine the interrupt frequency required to affect the channel capacity, we ran additional measurements using a custom interrupt stressor that generates IPIs on other cores at various frequencies. The results of these measurements are shown in Figure 9. The channel capacity starts to be affected by the other interrupts at ≈600 kI/s, dropping from 434.7 kbit/s to 391.9 kbit/s, slowly decreasing with more interrupt noise. The channel reaches a capacity of ≈0 bit/s at ≈10 MI/s. Our NVMe SSD was only able to generate ≈50 kI/s in our tests, which is significantly lower than the ≈600 kI/s required to measurably affect the channel.

Previous Work. Saileshwar et al. [70] exploit contention on shared hardware resources on the CPU and achieve a cross-core capacity of 14.4 Mbit/s using shared addresses. Gruss et al. [22] time the `clflush` instruction to detect if a cache line is present in shared memory and achieve a cross-core capacity of 3.4 Mbit/s. Liu et al. [50] use Prime+Probe on the L3 with a capacity of 600 kbit/s, only slightly faster than our attack. While some of these attacks are faster, they require either information about physical memory, addressing functions, or shared memory with the main goal of observing cache accesses. Our attacks require access to IPIs with the main goal of observing the delivery of other IPIs and interrupts. The most closely related covert channel is by Rauscher et al. [67], using the new `tpause` instruction to detect interrupts and other events on the same physical core, achieving a true capacity of 656 kbit/s with an error rate of 9.2 %. While our attack is slightly slower with 434.7 kbit/s, we can detect interrupts from other physical cores.

5 Keystroke Detection

In this section, we present our inter-keystroke timing attack using user interrupts. Contrary to previous interrupt detection-based attacks, our attacker is not required to run on the physical core that receives the victim's keyboard interrupts. Instead, we measure the IPI delivery time of IPIs from the attacker core to itself to detect them. We do not directly infer text from these measurements but instead use them to determine the channel quality by comparing our measurements with the keystroke-timing ground truth. Few works recover text from timings, e.g., Song et al. [81], as this has become a standard but training-intense machine learning task.

5.1 Threat Model and Attack Setup

We run our measurements on an Intel Xeon Silver 4410T CPU with Ubuntu 22.04 and the Linux 6.0 kernel published by Intel [35] that supports user interrupts. We assume that the attacker can run unprivileged code on the victim system and has access to a high-precision timer, e.g., `rdtsc`. The attacker may be running on an isolated core that does not receive or handle any hardware interrupts. Finally, we assume that a user is providing input to the system via keystrokes while the attacker program is running.

5.2 Attack Evaluation

For our inter-keystroke timing attack, the attacker continuously sends user IPIs to itself. By monitoring the time until the user IPI is handled, we can observe whether other system events, e.g., keyboard interrupts.

As the raw user IPI timings can be noisy, we apply a moving minimum filter, which returns the minimal timing observed in a 400 sample window. Figure 10 shows a trace of the filtered user IPI timings while a user is typing on the system. We can observe a distinctive pattern with 3 upward ticks in the user IPI delay. Based on this pattern, we can detect keystrokes using a similarity measure with a sliding window over a trace, e.g., with a window size of 200 million cycles.

We evaluated our attack with a human typing on the keyboard into a program measuring the ground truth. Based on this ground truth, we computed the false negative and false positive rate, as well as the temporal deviation from the ground truth. Overall, we

Table 1: Keystroke Detection Rates and F_1 Scores reported in other works.

Paper	Co-location	Detection Rate	F_1 Score	Temporal Standard Deviation
our work	cross-core	98.2 %	97.9 %	6.15 ms
Schwarz et al. [74]	same core	100 %	94 %	≈1 ms
Rauscher et al. [67]	same core	94.1 %	90.5 %	0.95 ms
Lipp et al. [46]	same core	81.75 %	n/a	n/a

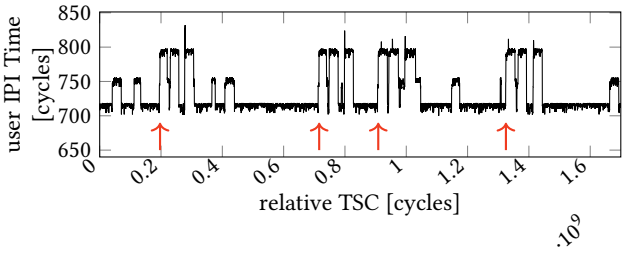


Figure 10: Keystroke detection using user IPIs. For visualization purposes, we filtered the noise by only plotting the minimum delay of 400 measurements each. At every keystroke, indicated by red arrows, there are 3 distinct upward ticks in the user IPI delays, making them distinguishable from background noise.

have a ground-truth trace with 571 keystrokes. The trace recorded with our IPI side channel contains 575 keystrokes. However, 14 of these keystrokes were false positives, meaning that we also had 10 false negative keystrokes, *i.e.*, only 561 true positive keystroke detections. Consequently, we have a precision of 97.6 % and a recall of 98.2 %. Based on this, we compute an F_1 score of 97.9 %. This is slightly higher than the F_1 scores and identification rates reported in other works, as shown in Table 1.

On the temporal scale, we observe a slightly higher standard deviation of 6.15 ms, which is expected as our attack is a cross-core attack in contrast to the other same-core attacks. Still, our temporal standard deviation is significantly lower than the average inter-keystroke interval of 120 ms and standard deviation of 11 ms for fast typists [14] and in the same order of magnitude as same-core inter-keystroke timing attacks.

6 Website Fingerprinting

In this section, we present a website fingerprinting attack using user interrupts and virtualized IPIs. We show a cross-core native website fingerprinting attack in both a closed-world and open-world scenario on the top 100 websites from the Alexa top 1 million list [2] using user IPIs to detect network interrupts. For open-world website fingerprinting, we use a separate `other` class for all websites not in the top 100. Furthermore, we present a cross-core cross-VM website fingerprinting attack in both a closed-world and open-world scenario on the top 100 websites from the Alexa top 1 million list [2] using virtualized IPIs to detect network interrupts. Contrary to previous interrupt detection-based fingerprinting attacks, our attacker is not required to run on the physical core that receives the victim browser's network interrupts.

6.1 Threat Model and Attack Setup

In this section, we discuss the thread model and overall setup of our attack.

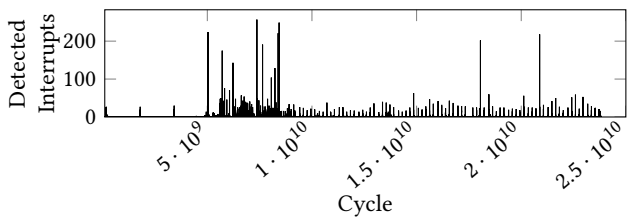
Native. We run our measurements on an Intel Xeon Silver 4410T CPU with a default-configured Google Chrome 121.0. We assume that the attacker can run code on the victim system and has access to a high-precision timer, e.g., `rdtsc`. While the attacker code is running, the victim browses the web. We make no assumptions on whether the attacker is able to run on the core that receives the network interrupts, as the interrupts could be rerouted to a separate isolated core as proposed to mitigate interrupt detection attacks by previous works [67]. The attacker is not able to monitor interrupts through any system interfaces such as `/proc/interrupts`. Furthermore, we do not make any assumptions about the core the web browser is running on.

Virtual Machine Attack Scenario. We assume a thin client scenario, where attacker and victim run co-located on the same VM host. Thin client devices have limited computing capability and only access a VM containing a full desktop environment. Thin clients offer companies lower initial hardware and maintenance costs compared to traditional desktops and are offered by major cloud providers [39]. While the attacker code is running, the victim browses the web.

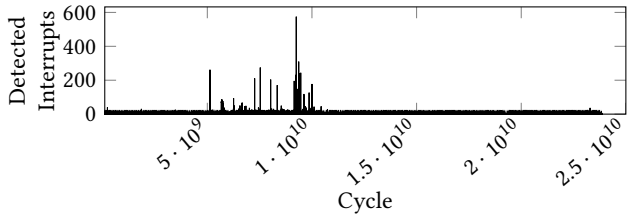
We run our measurements on an Intel Xeon Silver 4410T CPU with default-configured Google Chrome 121.0. For the hypervisor we use KVM with disk caching disabled, as recommended by Red Hat [27], on a stock Ubuntu 22.04. No custom kernel is required for this attack, as IPI virtualization is part of the Linux kernel and enabled by default. We assume that the attacker and victim run inside separate VMs in the cloud and are co-located on the same CPU, but both are running on separate physical cores. We assume that the attacker has root access inside its own VM and can load kernel modules, allowing for precise measurements of IPI timings. We make no assumptions on whether the attacker is able to run on the core that receives the network interrupts, as the interrupts could be rerouted to a separate isolated core, as proposed to mitigate interrupt detection attacks by previous works [67], or on a core the attacker does not run on.

6.2 Attack

Our attack consists of a data-collection phase and an offline phase for processing and classification of collected traces. The collection

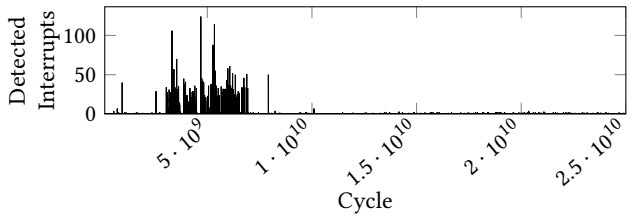


(a) `google.com`

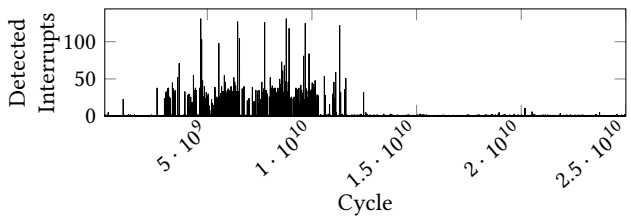


(b) `youtube.com`

Figure 11: Interrupt traces of `google.com` (Figure 11a) and `youtube.com` (Figure 11b) measured by the attacker with user IPIs.



(a) `google.com`



(b) `youtube.com`

Figure 12: Cross-VM interrupt traces of `google.com` (Figure 11a) and `youtube.com` (Figure 11b) measured by the attacker with virtualized IPIs.

phase consists of the attacker running on the victim system collecting interrupt traces through either user IPIs, in the case of the native scenario, or virtualized IPIs, in the case of the cross-VM scenario. Such interrupt traces for `google.com` and `youtube.com` using user IPIs are shown in Figure 11 and for the cross-VM scenario using virtualized IPIs in Figure 12. The x-axis represents the CPU cycle relative to the beginning of the trace, and the y-axis represents the number of interrupts detected through our measurements. For the native user interrupts scenario (Figure 11) `google.com` and `youtube.com` have distinct traces, with `google.com` having a larger amount of interrupts at the beginning and multiple interrupts regularly throughout the trace and `youtube.com` having most interrupts occur at the start of the website access with one exceptionally high spike at 10 million cycles. For the cross-VM scenario (Figure 12) `google.com` and `youtube.com` also have distinct traces, with `google.com` having a large number of interrupts for a short period of time with distinct peaks, and `youtube.com` having a similar amount of interrupts over a longer period. As seen in both Figure 11 and Figure 12, there are more detections in our cross-VM scenario from the additional noise introduced by the VMs.

In the offline phase, these traces are first preprocessed with a short-time Fourier transform (STFT). The STFT performs multiple Fourier transforms on short windows of the trace, resulting in 2D data consisting of the frequency information for each window on one axis and the time on the other axis. The preprocessing through an STFT allows us to perform convolutions on our data, making it possible to use a convolutional neural network (CNN) for the classification. This is a well-established signal processing technique [11, 28, 67, 102]. Our CNN consists of 4 convolutional layers and 3 fully connected layers and outputs a probability for each website and one for the `other` class in the case of the open-world scenarios. The output is the probability that the input belongs to a given website.

6.3 Evaluation

We evaluate our attack on closed-world and open-world scenarios with the attacker running on a physical core that does **not** receive network interrupts or run the browser. We collected 200 traces for each of the explicitly classified websites (20 000 in total). For the open-world scenario, we additionally collected the traces of `5000` additional websites (one per website) from the Alexa top 1 million list [2], which are not in the top 100. To evaluate our attack, we randomly split our data for each class into 5 equally large parts and performed 5-fold cross-validation. The test set for each run does `not` overlap with the training set. Due to this, in the open-world scenarios, website traces in the test set of the `other` class belong to websites that the model has never seen during training. We train our CNN with a validation split of 10 % of the training set.

6.3.1 Native Closed-World Website Fingerprinting. In this scenario, we only classify the top 100 websites in a native scenario using user IPIs. Our classifier achieved an F_1 score of 91.7 %. The confusion matrix is shown in Figure 13. Each cell represents the probability that our classifier detects a trace from a website (y-axis) as a given label (x-axis). The clear diagonal shows the high accuracy of our classifier. The worst-performing websites are `google.com.hk` at 42 %, `google.co.in` at 59 %, and `dzen.ru` at 68 %. All other websites have accuracies of above 70 %. The `google.com.hk` and `google.co.in` sites, in particular, are often misclassified as other Google domains.

6.3.2 Native Open-World Website Fingerprinting. In this scenario, we classify the top 100 websites and use an `other` class for all other websites in a native scenario using user IPIs. Our classifier achieved a macro averaged F_1 score of 89.0 %, showing a high accuracy across

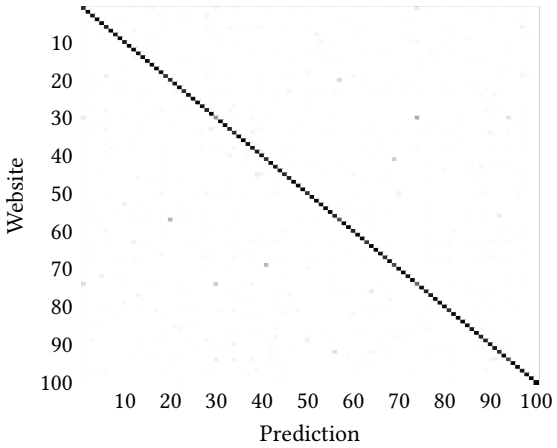


Figure 13: Native closed-world website fingerprinting confusion matrix.

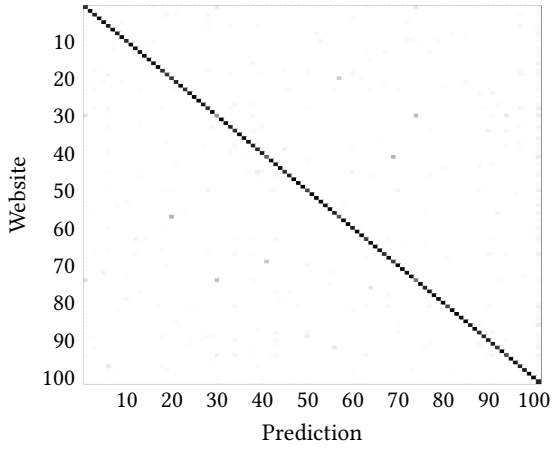


Figure 14: Native open-world website fingerprinting confusion matrix.

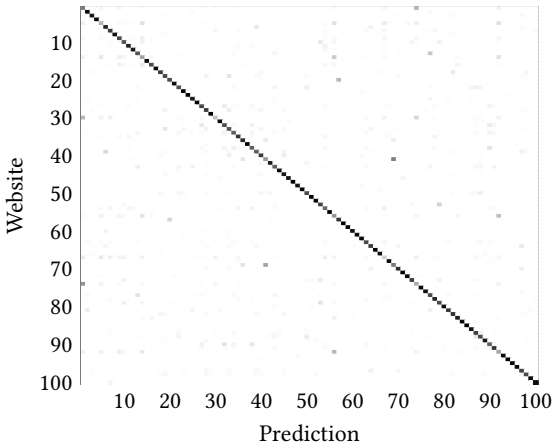


Figure 15: Cross-VM closed-world website fingerprinting confusion matrix.

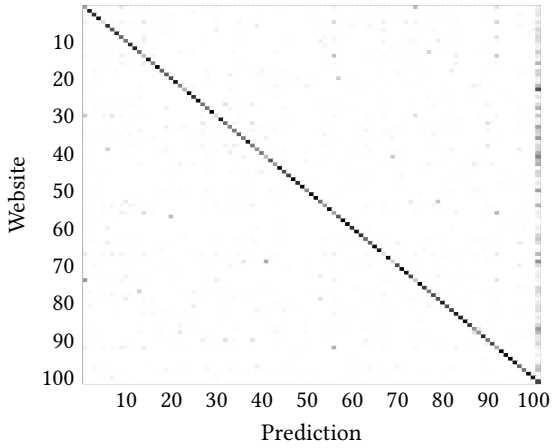


Figure 16: Cross-VM open-world website fingerprinting confusion matrix.

all classes, and an accuracy of 85.2 % on the `other` class. The confusion matrix is shown in Figure 14. Each cell represents the probability that our classifier detects a trace from a website (y-axis) as a given label (x-axis). The clear diagonal shows the high accuracy of our classifier. The worst-performing websites are `google.com.hk` at 41.5 %, `google.co.in` at 54.5 %, `microsoftonline.com` at 63 %, and `dzen.ru` at 67 %. All other websites have accuracies above 70 %. Similar to the closed-world scenario, the `google.com.hk` and `google.co.in` websites, in particular, are often misclassified as other Google domains. The `microsoftonline.com` domain performs poorly relative to other websites, as only sub-domains of it are accessible, e.g., `login.microsoftonline.com`, while we only tested the exact domains listed in the Alexa top 1 million list [2].

6.3.3 Cross-VM Closed-World Website Fingerprinting. In this scenario, we only classify the top 100 websites in a cross-VM scenario using virtualized IPIs. Our classifier achieved an F_1 score of 80.4 %. The confusion matrix is shown in Figure 15. Each cell represents the probability that our classifier detects a trace from a website (y-axis) as a given label (x-axis). The clear diagonal shows the high accuracy of our classifier. The worst-performing websites are `indeed.com` at 18 %, `google.com.hk` at 24 %, and `facebook.com` at 29.5 %. All `indeed.com` and `facebook.com`, in particular, are fast-loading websites when already cached, making them more challenging to distinguish with the added noise from the VMs. The Google domains are, similar to the other scenarios, often misclassified as other Google domains.

6.3.4 Cross-VM Open-World Website Fingerprinting. In this scenario, we only classify the top 100 websites in a cross-VM scenario using virtualized IPIs. Our classifier achieved a macro averaged F_1 score of 71.0 %, showing a high accuracy across all classes, and an accuracy of 71.7 % on the `other` class. The confusion matrix is shown in Figure 16. Each cell represents the probability that our classifier detects a trace from a website (y-axis) as a given label

Table 2: Overview of Different Interrupt Side-Channel Works

Attack	Software Interface	Hardware-Based Same-Core	Hardware-Based Cross-Core
Inter-Keystroke Timing	[74, 105]	[46, 74, 87, 88]	**our work**
Website Fingerprinting	[51]	[13, 67, 106, 107]	**our work**
Application Fingerprinting	[15, 51, 86]		
Cryptographic Key Leakage		[107]	
DNN Model Stealing	[51]	[107]	

Table 3: Mitigations for Interrupt Side-Channel Attacks

Approach	Mitigations
Constrain interfaces	[13, 105]
Constrain co-location	[67]
Constrain timers	[36, 45, 47, 48, 52, 61, 94, 106]
Non-leaking timers	[3, 42, 52, 94]
Inject Noise	[74]

(x-axis). The clear diagonal shows the high accuracy of our classifier. Similar to the closed-world scenario, the worst-performing websites are `indeed.com` at 11 %, `google.com.hk` at 13.5 %, and `facebook.com` at 13.5 %. The Google domains are, similar to the other scenarios, often misclassified as other Google domains.

6.4 Previous Work

Our closed-world attack achieves an F_1 score of 91.7 % (native) and 80.4 % (cross-VM), while our open-world attack achieves an F_1 score of 89.0 % (native) and 71.0 % (cross-VM), performing similarly to previous works while not requiring to run on the core where the network interrupts arrive. Zhang et al. [106] achieved an F_1 score of 78 % on the top 100 sites using the `mwaitx` instruction to detect interrupts on the core that receives the network interrupts. Gulmezoglu et al. [26] exploit performance counters, achieving an accuracy of 86.3 % on 40 websites. Spreitzer et al. [82] use data-usage statistics on Android to classify 100 websites with an F_1 score of 89 %. Rauscher et al. [67] exploit the `tpause` instruction in VMs to classify 100 websites with an F_1 score of 93.1 %.

7 Related Work & Discussion

A series of works have investigated side-channel attacks exploiting interrupts or their effects. The information channels can coarsely be divided into three categories (cf. Table 2):

The **first** category of attacks uses software interfaces to obtain information about interrupts, e.g., `/proc/interrupts` [15, 51, 74, 86, 105]. While this system information is provided architecturally without noise and regardless of the attacker's core scheduling, they are trivial to mitigate by making the corresponding software interface privileged and these interfaces are generally not available from within VMs for the host system. Prior work demonstrated inter-keystroke timing [74, 105], website-fingerprinting [51], and application-fingerprinting [15, 51, 86] attacks. Some works have even demonstrated leakage of DNN models [51] and cryptographic keys [107]. Still, by constraining the software interface, all of these attacks can be mitigated in practice. The other two categories use side channels to monitor hardware behavior.

Concretely, the **second** category is hardware-based same-core attacks that typically exploit that the interrupt has to be executed by the core under attack. Several attacks use busy loops of timing measurements and measure when they are interrupted as timer jumps, *i.e.*, latency spikes [13, 46, 74, 87, 88]. More recent works exploit CPU features that do not depend on busy-looping the SMT thread under attack or a co-located SMT thread on the same physical core [67, 106, 107]. Hence, all of these attacks depend on being co-located on the same core as the attacker. In this setting, powerful attacks are possible, including inter-keystroke timing attacks [46, 74, 87, 88], website-fingerprinting attacks [13, 67, 106, 107], as well as leakage of DNN models and cryptographic keys [107].

The **third** category is cross-core hardware-based attacks that do not exploit system interfaces and also do not require a busy loop on the victim core. Our work is the first attack to demonstrate this possibility by exploiting both the user interrupts feature as well as the virtualized inter-processor interrupts feature. Our attack generalizes in the virtualized setting to any VM that can send and measure the latency of inter-processor interrupts.

Mitigations. To mitigate the IPI side channel effectively and efficiently, hardware changes may be necessary. We believe that the current implementation sends IPI messages and lets cores check and decide which IPI messages they accept. However, it is unclear why the system bus transmitting interrupts between cores has to be used for local and I/O interrupts that only affect a single core. This approach inherently allows us to probe the corresponding system bus and, thereby, the interrupt activity of other cores. A different design of this system bus could affect our attack. Similarly, changing how cores check whether they should receive an interrupt would also have the potential of mitigating our attack.

While our conclusion is that closing the IPI side channel requires re-designing the corresponding interrupt handling hardware, we still want to discuss mitigations proposed by the academic community (cf. Table 3) against prior interrupt-driven attacks:

The **first** category is to remove the `/proc` interface or make it privileged [13, 105]. Virtual machines also cannot access the host's `/proc` interface for security reasons. This approach has been implemented on various systems [74, 107], leaving only other channels open to mitigate. However, with the `/proc` interface disabled or unavailable inside a VM, our attack still works. For user interrupts, we could also limit the attack surface by only allowing a selected number of trusted applications to use user interrupts. As user interrupts require kernel support to register a receiver and a sender thread, we suggest to restrict these kernel interactions to certain user groups or applications, e.g., specific drivers. This restriction would no longer allow an attacker to take advantage of the user-interrupt side channel while making user interrupts available for applications that need them. We do not consider entirely disabling user interrupts a viable option due to their low delay times

compared to POSIX signals. This restriction is not possible for IPI virtualization, as modern operating systems require IPIs for fast inter-processor communication, and disabling IPI virtualization for all untrusted VMs would remove the performance improvements gained by IPI virtualization for these VMs.

The **second** category of mitigations is to constraint either the co-location of the attacker workload with the victim or its placement on the interrupt-receiving core [67]. However, as we can detect interrupts on any core, this approach does not affect our attack, even when isolating interrupts to a separate physical core. Also, randomizing interrupt core assignments does not affect our attack.

The **third** category is to constrain timers, e.g., by making them privileged, as has been discussed in numerous works [36, 45, 47, 48, 52, 61, 94, 106]. Similarly, the **fourth** category is to modify timers in a way that they do not depend on the secret information anymore [3, 42, 52, 94]. Both approaches have limited effect in practice as the community has found many ways to bypass them, e.g., using counting threads [47, 73, 76, 100] and timeless methods [93].

A **fifth** category is to introduce noise [74]. Noise has been studied as a mitigation against power side channels as well [43]. However, from this context, it is also known that noise only reduces the side channel signal but cannot eliminate it [43]. Consequently, adding noise also can only reduce the signal, which, according to Schwarz et al. [74], is sufficient against inter-keystroke timing attacks. However, it is unclear whether similar approaches to inject noise could be sufficient to mitigate interrupt side-channel attacks in other attack scenarios such as website- and application fingerprinting, or leakage of cryptographic keys or DNN models.

Generic Side-Channel Mitigations. A generic side-channel mitigation covering interrupt side channels as well, is side-channel detection, e.g., using performance counters [62]. Intel suggests that user interrupts can be tracked through architectural Last Branch Records (LBRs) and Intel Processor Trace, which record user interrupts the same way as normal interrupts [33]. Despite this, it is difficult to distinguish a benign workload using the IPI side channel for high-frequency message passing between processes and a malicious workload using user IPIs or virtualized IPIs for an attack.

Beyond Side Channels Exploitation of Inter-Processor Interrupts. Inter-processor interrupts provide multi-core and multi-processor systems with a means to communicate and synchronize across cores. A common example is the invalidation of a virtual memory mapping, which is cached by the TLB, requiring a so-called TLB shootdown, a coordinated operation across multiple processors or processor cores to invalidate the corresponding TLB entries across all cores. Wang et al. [96] exploit this behavior to spy on the accessed bit in the page-table entry of an SGX enclave. Zhang et al. [108] exploit IPIs to preempt a victim, amplifying their Prime+Probe attack. Zhang and Reiter use IPIs to continuously flush the caches of other cores to mitigate cache attacks on these [109]. However, none of these works studies the timing of IPIs themselves.

Trusted Execution Environments. Considering the emerging concept of trusted execution environments (TEEs) such as Intel TDX, Intel SGX, and AMD SEV-SNP, interrupt side channels may pose a relevant attack vector. We consider **two** attack scenarios.

The **first** category is the case of a **malicious host**. With SGX-Step [91] (Intel SGX) and SEV-Step [99] (AMD SEV-SNP), the host sets up the APIC timer to trigger an external interrupt shortly after entering the protected guest to single- and zero-step the guest. This can be used to determine instructions executed [99], amplify power side channels [49], or assist microarchitectural attacks [5, 56, 75, 90, 92]. Intel TDX includes a mitigation against single- and zero-stepping [34] to prevent these kinds of attacks. As the host has to be able to inject interrupts, e.g., for device emulation, TDX and AMD SEV-SNP allow the host to arbitrarily inject interrupts. Schlüter et al. [72] showed that a malicious host can inject interrupt vectors typically used for software interrupts, such as syscalls, to change register values on AMD SEV-SNP and Intel TDX. WeSee[71] injects interrupts with the interrupt number reserved for the virtualization exception leading the guest to assume that such an exception occurred on AMD SEV-SNP. Sridhara et al. [83] send signals to SGX enclaves to modify the enclave state. Constable et al. [12] propose a hardware ISA extension to make SGX enclaves interrupt aware, allowing an enclave to detect and mitigate interrupt-based attacks. While our IPI side channel does work in a malicious host scenario, the virtual machine host receives all external interrupts and only forwards them to the guest if necessary. Hence, the host already knows which interrupts occur at a given time without requiring a side channel.

The **second** category is the case of a **malicious guest**. As the code executed inside of an SGX enclave, a TDX guest, or an SEV-SNP guest, is not directly accessible by the host, nefarious activity by the guest can be challenging for the host to detect. While there are no interrupt-related attacks from inside a TEE yet, there are multiple works discussing and showing the threat of malicious enclaves in Intel SGX [73, 77]. There are also multiple works that try to detect or defend against malicious SGX enclaves [58, 97, 110]. Similar to existing interrupt-detection-based attacks, an attacker could use `rdtsc` [74] or a counting thread [73] to detect interrupts on the same core in TDX and SEV-SNP or the `tpause` instruction to detect interrupts on a sibling logical core in TDX [34, 67]. Counting threads have also been explored in SGX enclaves as a defense against interruption-based attacks [8–10, 59]. The IPI side channel does not work inside AMD SEV-SNP, Intel TDX, or Intel SGX at the time of writing. Both AMD SEV-SNP and Intel TDX require the host to handle the routing and injection of IPIs sent from inside guests. The feature set inside Intel SGX is even more limited and excludes sending of user or regular IPIs. Hence, it is not possible to mount the IPI side channel from inside these TEEs.

8 Conclusion

While user interrupts and IPI virtualization drastically reduce the performance cost of cross-process and cross-core signaling, our work shows that these new features can be exploited to detect interrupts delivered to any core in native and cross-VM scenarios. We used the IPI side channel for cross-core covert communication between two processes that send IPIs to themselves, with a native true capacity of 434.7 kbit/s (n=100, $\sigma_{\bar{x}}$=0.03) and cross-VM true capacity of 3.47 kbit/s (n=100, $\sigma_{\bar{x}}$=0.01). We presented an inter-keystroke timing attack with an F_1 score of 97.9 % and a standard deviation of 6.15 ms. Furthermore, we demonstrated a cross-core

website fingerprinting attack that achieves an F_1 score 89.0 % in an open-world native scenario and 71.0 % in an open-world cross-VM scenario, highlighting the security and privacy implications. While there are mitigations against interrupt side-channel attacks, the change in the attack scenario (*i.e.*, cross-core cross-VM) also bypasses several mitigations. We conclude that bringing interrupts to userspace and providing VMs with low latency access to IPIs can have unforeseen side effects, resulting in an increased attack surface for security- and privacy-related applications.

Acknowledgments

We thank the anonymous reviewers and our anonymous shepherd for their guidance, comments, and suggestions. We would also like to thank our DIMVA reviewers for their feedback on an earlier version of this paper. Furthermore, we thank Andreas Kogler for his feedback and insightful discussions. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF project NeRAM I6054). Additional funding was provided by generous gifts from Red Hat, and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2018. Port Contention for Fun and Profit. In *S&P*.
[2] Alexa Internet, Inc. 2023. The top 1 million sites on the web. https://www.alexa.com/topsites
[3] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. 2010. Determinating timing channels in compute clouds. In *CCSW*.
[4] Daniel J. Bernstein. 2005. Cache-Timing Attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
[5] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security*.
[6] Elad Carmon, Jean-Pierre Seifert, and Avishai Wool. 2017. Photonic Side Channel Attacks Against RSA. In *HOST*.
[7] Sebastien Carre, Victor Dyseryn, Adrien Facon, Sylvain Guilley, and Thomas Perianin. 2019. End-to-end automated cache-timing attack driven by Machine Learning. *Journal of Cryptology* (2019).
[8] Guoxing Chen, Mengyuan Li, Fengwei Zhang, and Yinqian Zhang. 2019. Defeating Speculative-Execution Attacks on SGX with HyperRace. In *Dependable and Secure Computing (DSC)*.
[9] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. 2018. Racing in hyperspace: closing hyper-threading side channels on SGX with contrived data races. In *S&P*.
[10] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *AsiaCCS*.
[11] Zhibo Chen, Yi-Qun Xu, Hongbin Wang, and Daoxing Guo. 2020. Deep STFT-CNN for spectrum sensing in cognitive radio. *IEEE Communications Letters* (2020).
[12] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. 2023. {AEX-Notify}: Thwarting Precise {Single-Stepping} Attacks through Interrupt Awareness for Intel {SGX} Enclaves. In *USENIX Security*.
[13] 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 *ISCA*.
[14] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. 2018. Observations on typing from 136 million keystrokes. In *CHI Conference on Human Factors in Computing Systems*.
[15] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In *S&P*.
[16] Dmitry Evtyushkin and Dmitry Ponomarev. 2016. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In *CCS*.
[17] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*.
[18] Anders Fogh. 2016. Covert Shotgun: automatically finding SMT covert channels. https://cyber.wtf/2016/09/27/covert-shotgun/
[19] Google Issue Tracker. 2017. Android O prevents access to /proc/stat. https://issuetracker.google.com/issues/37140047
[20] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*.
[21] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. 2019. Page Cache Attacks. In *CCS*.
[22] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.
[23] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*.
[24] Zeng Guang. 2022. KVM Commit "[v9,0/9] IPI virtualization support for VM". https://patchwork.kernel.org/project/kvm/cover/20220419153155.11504-1-guang.zeng@intel.com/
[25] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P*.
[26] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. 2017. PerfWeb: How to violate web privacy with hardware performance events. In *ESORICS*.
[27] Jiri Herrmann, Yehuda Zimmerman, Dayle Parker, and Scott Radvan. 2019. *Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide*.
[28] Jingshan Huang, Binqiang Chen, Bin Yao, and Wangpeng He. 2019. ECG arrhythmia classification using STFT-based spectrogram and convolutional neural network. *IEEE access* (2019).
[29] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*.
[30] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture.
[31] Intel. 2022. Intel Architecture Instruction Set Extensions and Future Features.
[32] Intel. 2023. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z.
[33] Intel. 2024. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide.
[34] Intel. 2024. Intel Trust Domain Extensions Module Base Architecture Specification. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html
[35] Intel. 2024. UINTR Linux Kernel. https://github.com/intel/uintr-linux-kernel
[36] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2018. MASCAT: Preventing microarchitectural attacks before distribution. In *CODASPY*.
[37] Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning Secrets from Process Footprints. In *S&P*.
[38] Linux Kernel. 2022. https://cdn.kernel.org/pub/linux/kernel/v6.x/ChangeLog-6.0. In *Linux Kernel Change Log 6.0*.
[39] Simon Kissler and Owen Hoyt. 2005. Using Thin Client Technology to Reduce Complexity and Cost. In *ACM SIGUCCS conference on User services*.
[40] Paul Kocher. 1996. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*.
[41] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *CRYPTO*.
[42] David Kohlbrenner and Hovav Shacham. 2016. Trusted Browsers for Uncertain Times. In *USENIX Security*.
[43] Itamar Levi, Davide Bellizia, David Bol, and François-Xavier Standaert. 2020. Ask Less, Get More: Side-Channel Signal Hiding, Revisited. *IEEE Transactions on Circuits and Systems* 67, 12 (2020), 4904–4917.
[44] Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2015. Reducing World Switches in Virtualized Environment with Flexible Cross-world Calls. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 375–387.
[45] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD Prefetch Attacks through Power and Time. In *USENIX Security*.
[46] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. 2017. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS*.
[47] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security*.
[48] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *AsiaCCS*.
[49] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*.

[50] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.

[51] Haoyu Ma, Jianwen Tian, Debin Gao, and Chunfu Jia. 2021. On the Effectiveness of Using Graphics Interrupt as a Side Channel for User Behavior Snooping. *Transactions on Dependable and Secure Computing* 19, 5 (2021), 3257–3270.

[52] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News* (2012).

[53] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: Cross-Cores Cache Covert Channel. In *DIMVA*.

[54] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *RAID*.

[55] 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*.

[56] Daniel Moghimi. 2023. Downfall: Exploiting Speculative Data Gathering. In *USENIX Security*.

[57] John Monaco. 2018. SoK: Keylogging Side Channels. In *S&P*.

[58] Soo Jung Moon, Hoorin Park, and Wonjun Lee. 2021. Preventing enclave malware with intermediate enclaves on semi-honest cloud platforms. In *BigComp*.

[59] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *USENIX ATC*.

[60] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.

[61] Yoshihiro Oyama. 2019. How does malware use RDTSC? A study on operations executed by malware with CPU cycle measurement. In *DIMVA*.

[62] Matthias Payer. 2016. HexPADS: a platform to detect "stealth" attacks. In *ESSoS*.

[63] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan*.

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

[65] Yi Qin and Chuan Yue. 2018. Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7. In *TrustCom/BigDataSE*.

[66] Jean-Jacques Quisquater and David Samyde. 2001. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In *E-smart*.

[67] Fabian Rauscher, Andreas Kogler, Jonas Juffinger, and Daniel Gruss. 2024. IdleLeak: Exploiting Idle State Side Effects for Information Leakage. In *NDSS*.

[68] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. 2017. Automated website fingerprinting through deep learning. In *NDSS*.

[69] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*.

[70] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. 2021. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *ASPLOS*.

[71] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. 2024. WeSee: Using Malicious# VC Interrupts to Break AMD SEV-SNP. *arXiv preprint arXiv:2404.03526* (2024).

[72] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. Heckler: Breaking Confidential VMs with Malicious Interrupts. In *USENIX Security*.

[73] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*.

[74] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*.

[75] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.

[76] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*.

[77] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical Enclave Malware with Intel SGX. In *DIMVA*.

[78] Martin Schwarzl, Erik Kraft, and Daniel Gruss. 2023. Layered Binary Templating. In *ACNS*.

[79] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through The Cache Occupancy Channel. In *USENIX Security*.

[80] Laurent Simon, Wenduan Xu, and Ross Anderson. 2016. Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards. *PETS* (2016).

[81] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. 2001. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security*.

[82] Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard. 2016. Exploiting data-usage statistics for website fingerprinting attacks on Android. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks*.

[83] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, and Shweta Shinde. 2024. SIGY: Breaking Intel SGX Enclaves with Malicious Exceptions & Signals. *arXiv preprint arXiv:2404.13998* (2024).

[84] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. 2018. Microarchitectural Minefields: 4K-aliasing Covert Channel and Multi-tenant Detection in IaaS Clouds. In *NDSS*.

[85] Jakub Szefer. 2016. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *Cryptology ePrint Archive, Report 2016/479* (2016).

[86] Xiaoxiao Tang, Yan Lin, Daoyuan Wu, and Debin Gao. 2018. Towards Dynamically Monitoring Android Applications on Non-rooted Devices in the Wild. In *WiSec*.

[87] Albert Tannous, Jonathan T. Trostle, Mohamed Hassan, Stephen E. McLaughlin, and Trent Jaeger. 2008. New Side Channels Targeted at Passwords. In *ACSAC*.

[88] Jonathan T Trostle. 1998. Timing Attacks Against Trusted Path. In *S&P*.

[89] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56.

[90] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.

[91] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Workshop on System Software for Trusted Execution*.

[92] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS*.

[93] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. 2020. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In *USENIX Security*.

[94] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. 2011. Eliminating fine grained timers in Xen. In *CCSW*.

[95] VMware. 2007. Understanding Full Virtualization, Paravirtualization, and Hardware Assist.

[96] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*.

[97] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. 2019. SGXJail: Defeating Enclave Malware via Confinement. In *RAID*.

[98] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. https://foreshadowattack.eu/foreshadow-NG.pdf

[99] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. 2024. SEV-Step: A Single-Stepping Framework for AMD-SEV. *TCHES* (2024), 180–206.

[100] John C Wray. 1992. An Analysis of Covert Timing Channels. *Journal of Computer Security* (1992).

[101] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security*.

[102] Shuochao Yao, Ailing Piao, Wenjun Jiang, Yiran Zhao, Huajie Shao, Shengzhong Liu, Dongxin Liu, Jinyang Li, Tianshi Wang, Shaohan Hu, et al. 2019. Stfnets: Learning sensing signals from the time-frequency perspective with short-time fourier neural networks. In *The World Wide Web Conference*.

[103] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. *Cryptology ePrint Archive, Report 2014/140* (2014).

[104] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.

[105] Kehuan Zhang and XiaoFeng Wang. 2009. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security*.

[106] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. 2023. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In *USENIX Security*.

[107] Xin Zhang, Zhi Zhang, Qingni Shen, Wenhao Wang, Yansong Gao, Zhuoxi Yang, and Jiliang Zhang. 2024. SegScope: Probing Fine-grained Interrupts via Architectural Footprints. In *HPCA*.

[108] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS*.

[109] Yinqian Zhang and MK Reiter. 2013. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*.

[110] Zeyu Zhang, Xiaoli Zhang, Qi Li, Kun Sun, Yinqian Zhang, Songsong Liu, Yukun Liu, and Xiaoning Li. 2021. See through Walls: Detecting Malware in SGX Enclaves with SGX-Bouncer. In *AsiaCCS*.