IdleLeak: Exploiting Idle State Side Effects for Information Leakage

Fabian Rauscher, Graz University of Technology, fabian.rauscher@iaik.tugraz.at
Andreas Kogler, Graz University of Technology, andreas.kogler@iaik.tugraz.at
Jonas Juffinger, Graz University of Technology, jonas.juffinger@iaik.tugraz.at
Daniel Gruss, Graz University of Technology, daniel.gruss@iaik.tugraz.at

***Abstract*—Modern processors are equipped with numerous features to regulate energy consumption according to the workload. For this purpose, software brings processor cores into idle states via dedicated instructions such as `hlt`. Recently, Intel introduced the C0.1 and C0.2 idle states. While idle states previously could only be reached via privileged operations, these new idle states can also be reached by an unprivileged attacker. However, the attack surface these idle states open is still unclear.**

In this paper, we present IdleLeak, a novel side-channel attack exploiting the new C0.1 and C0.2 idle states in two distinct ways. Specifically, we exploit the processor idle state C0.2 to monitor system activity and for novel means of data exfiltration, and the idle state C0.1 to monitor system activity on logical sibling cores. IdleLeak still works regardless of where the victim workload is scheduled, *i.e.*, cross-core, due to the low-level x86 design. We demonstrate that IdleLeak leaks significant information in a native keystroke-timing attack, achieving an F1 score of 90.5 % and a standard error on the timing prediction of only 12 µs. We also demonstrate website- and video-fingerprinting attacks using IdleLeak traces, pre-processed with short-time Fourier transforms, and classified with convolutional neural networks. These attacks are highly practical with F1 scores of 85.2 % (open-world website fingerprinting) and 81.5 % (open-world video fingerprinting). We evaluate the throughput of IdleLeak side channels in both directions in covert channel scenarios, *i.e.*, using interrupts and performance-increasing effects. With the performance-increasing effect, IdleLeak achieves a true capacity of 7.1 Mbit/s in a native and 46.3 kbit/s in a cross-VM scenario. With interrupts, IdleLeak achieves 656.37 kbit/s in a native scenario. We conclude that mitigations against IdleLeak are necessary in both personal and cloud environments when running untrusted code.

I. Introduction

Modern CPUs have various features to meet the energy consumption, and performance demands, of users and workloads. One of the most significant performance increases came with out-of-order execution. While out-of-order execution enables parallel use of multiple execution units for different instruction, most workloads do not fully utilize this parallelism. Hence, vendors introduced simultaneous multithreading (SMT), giving the software level the illusion of having more CPU cores, so-called logical cores. The CPU interleaves the instructions from multiple logical cores on one physical core. Consequently, CPUs achieve substantially higher utilization of execution units, improving performance by roughly 30 % [58]. The reason for this performance gain is that many on-core and off-core resources are competitively shared (e.g., caches and TLB), and others are inexpensive to split statically (e.g., load and store buffer) [54]. Thus, SMT can still negatively impact the performance of workloads if resources are statically split or partially occupied by other workloads.

Over the past three decades, researchers investigated the security implications of these optimizations and found numerous side channels in various microarchitectural elements. A particularly long line of research focused on caches and buffers [66], [3], [34], both in scenarios where attacker and victim share a physical core [49], [10], as well as cross-core attacks [66], [31]. Other works investigated contention on execution ports [1], [43] and scheduler queues [9]. All these works focused on contention and interference on specific microarchitectural elements. However, the increased necessity for energy and performance improvements led to more advanced efficiency features in recent CPUs. These go beyond single microarchitectural elements and operate on the level of entire logical cores, physical cores, or entire packages. In particular idle states can introduce substantial efficiency gains. However, until recently, idle state management was only possible from kernel space. Still, prior work showed that these optimizations may still have security implications [68], in particular when attacking Trusted Execution Environments (TEEs).

With the efficiency and performance goals for new microarchitecture designs, Intel recently introduced the user space sleep primitives `tpause`, a timed pause, and `umwait`, a timed pause with a memory trigger. These instructions allow efficient waiting for timeouts and memory accesses while the processor is idle to save energy. Prior work showed that `umwait` can be exploited for Spectre-type attacks without a timing-based side channel and interrupt monitoring in native code [68], similar to prior works showed for other side channels [42], [27], [48]. While the work [68] did show that the interrupt monitoring can be used for website fingerprinting in native code, they did not further investigate other undocumented interrupt wake-up triggers, the possible security implications of unprivileged modification of idle states, the behaviour of the idle states in VMs, or the related `tpause` instruction.

In this paper, we present IdleLeak, a novel attack exploiting control over the C0.1 and C0.2 idle states from user space and their undocumented behaviors. Our first attack technique, ActiveIdleLeak, shows that the performance-increasing effects of the C0.2 idle state yield unforeseen security implications

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24078
www.ndss-symposium.org

on workloads co-located on a sibling logical core. Our second attack technique, PassiveIdleLeak, shows that both the C0.1 and C0.2 idle state are woken up by some processor operations, including events from workloads running on different physical cores, and, in general, different activity on the same core or a sibling logical core. That is, instead of attempting co-location with the victim, PassiveIdleLeak can run on an arbitrary physical core where either of the logical cores is influenced by the victim workload due to the way x86 implements interrupts.

In our first attack, ActiveIdleLeak, the attacker uses the unprivileged `tpause` instruction to put a logical core into the C0.2 idle state. The C0.2 idle state relinquishes some on-core resources (e.g., load buffer, store buffer, and reorder-buffer entries) to the other logical core. Consequently, the single-threaded performance of the other logical sibling core increases. The attacker uses this performance side effect to construct a covert communication channel between separate security domains. We evaluate the capacity of the ActiveIdleLeak channel in native and cross-VM covert channels, yielding 7.1 Mbit/s ($\sigma_{\bar{x}}$ = 0.004 Mbit/s, n = 512) and 46.3 kbit/s ($\sigma_{\bar{x}}$ = 0.15 kbit/s, n = 370) respectively.

For our second attack, PassiveIdleLeak, we exploit both idle states to spy on activities on the same core and the sibling logical core. Both idle states are left, e.g., on interrupts, special instructions, user input, and scheduling behavior, on the same core. C0.1 is also influenced by such activity on sibling logical cores. Surprisingly, both idle states are also influenced by such activity in the host and co-located virtual machines (VMs). This unexpected behavior opens up attack vectors for leaking user activity information from host to guest.

We present an unprivileged keystroke-timing attack using PassiveIdleLeak, with an F1 score of 90.5 % and a standard error on the timing prediction of only 12 µs. Our keystroke-timing attack works across cores, regardless of where the victim is scheduled. This is possible since the attacker can occupy multiple cores, including logical cores that belong to the physical core receiving the key-down and key-up interrupts.

We demonstrate that PassiveIdleLeak running inside a VM can also be used to fingerprint website and video accesses by the host. We show that we can target either the interrupt handling of a victim workload running on another physical core, or the victim workload itself if it is co-located on a logical sibling core. In an open- and closed-world evaluation with 100 websites, we achieve F1 scores of 85.2 % and 93.1 % respectively on an independent test set. We then demonstrate that we can distinguish videos accessed by a user in a video-fingerprinting attack on popular video streaming platforms, with F1 scores of 90.2 % (closed-world) and 81.5 % (open-world) for YouTube, and 75 % (closed-world) and 70.5 % (open-world) for Pornhub using only the first 10 s of a video. Our attack is the *first* to demonstrate video fingerprinting based on interrupt detection via a side channel. The information gained by this attack could be used for extortion campaigns, illustrating the severe and previously unknown privacy implications of the novel idle states. We evaluate the PassiveIdleLeak channel in a native covert-channel scenario, yielding a true capacity of 656.37 kbit/s ($\sigma_{\bar{x}}$ = 0.63 kbit/s, n = 1 024).

Since interrupt scheduling does not adhere to side-channel aware scheduling policies like gang scheduling [21], systems are unprotected against our attack. Contrary to prior works, we focus on the undocumented behavior of the novel idle states and the new instructions unexpected behavior inside VMs compared to existing instructions. We discuss possible mitigations against IdleLeak and find that efficient mitigations are not trivial due to the nature of external interrupts and the ambiguity of the interrupt receiver.

To summarize, we make the following contributions:

- We analyze the security properties of the C0.1 and C0.2 idle states that can be manipulated by an unprivileged attacker, resulting in our novel attack IdleLeak.
- We show that IdleLeak is fast and robust, leaking up to 7.1 Mbit/s ($\sigma_{\bar{x}}$ = 0.004 Mbit/s, n = 512) in a native covert channel and 46.3 kbit/s ($\sigma_{\bar{x}}$ = 0.15 kbit/s, n = 370) in a cross-VM covert channel respectively.
- We demonstrate that IdleLeak can be used to monitor inter-keystroke timings with an F1 score of 90.5 % and a standard error on the timing prediction of only 12 µs.
- We demonstrate website fingerprinting IdleLeak attacks, with F1 scores of 93.1 % (closed world) and 85.2 % (open world) over the top 100 websites.
- We demonstrate video fingerprinting IdleLeak attacks on two video streaming platforms, with F1 scores of 90.2 % (YouTube) and 75 % (PornHub) over the top 20 videos using only the first 10 s of a video.

Outline: Section II provides background on SMT, side channels and power states. Section III presents the idea behind IdleLeak. Section IV evaluates native and cross-VM IdleLeak covert channels. Section V presents our keystroke timing attacks and Section VI our website- and video-fingerprinting attacks. Section VII discusses implications and mitigations. Section VIII discusses related work. Section IX concludes.

II. Background

In this section, we provide background on SMT and side-channel attacks. Finally, we discuss processor idle states, how the running software can influence them, and how they influence the performance and energy consumption.

A. Simultaneous Multithreading (SMT)

Modern CPUs have multiple execution units that execute different instructions simultaneously and out of order. While this speeds up the instruction throughput and can improve the wall-clock performance of workloads, for many workloads, the execution units are idling. Thus, to maximize performance and efficiency, modern CPUs have multiple execution threads (logical cores) that run separate instruction streams from different execution contexts on the same physical core. Intel calls this technique hyperthreading, more generally known as simultaneous multithreading (SMT). With SMT, many microarchitectural elements within the same physical CPU core can be shared (e.g., reorder buffer, load and store buffer, caches, the TLB) [54]. Similarly, processors achieve substantially higher utilization of execution units, improving performance by roughly 30 % [58], since, even on personal computers, many execution threads are constantly running in parallel. However, for single-threaded workloads, SMT can have a negative performance impact, as on-core resources are statically split or competitively shared, reducing the throughput for the single-threaded workload.

B. Side-Channel Attacks

Side-channel attacks on computer systems were first reported in the 1990s [22] and have since significantly influenced the system security area. These attacks obtain meta-data about a secret processed through a (possibly unintentional) channel and derive the secret fully or partially from this meta-data. Historically side-channel research focused on cryptographic primitives [35], [3] as they have a very well-defined threat model with a valuable secret, the secret key, and meta-data that obtainable by an attacker, such as execution time [22], [37], power consumption [24], [23], and EM radiation [39].

One line of side-channel research focused on user input, mainly obtaining inter-keystroke timings but in some cases even precise key press timings for a small set of keys or even single keys [42], [11], [27], [57], [32], [50]. Inter-keystroke timings already contain a significant amount of information as the finger movements across the physical layout of a keyboard influence the inter-keystroke timings in unique ways. While any such timing differences depend on the specific user, previous work showed that written text can still be recovered from them, possibly with an initial learning phase [51], [52], [67], [48], using e.g., machine learning [27], [51], [52].

Another important direction of side-channel research investigates using side channels to establish covert communication channels. In this scenario, the victim and attacker collaborate and form a sender-receiver pair of a communication channel [64], [31], [45]. Covert channels have since been demonstrated on various microarchitectural elements [60], [62], [7], across VMs in the cloud [31], and in browsers [33], [44].

Side-channel research typically evaluates these basic attack targets in various attack scenarios, such as native code attacks [66], browser- or VM-based attacks [33], [38], or even attacks on and from TEEs such as Intel SGX [47], [63], [56], [59]. Each of these scenarios comes with different security properties and, hence, influences the applicability and impact of a potential side-channel attack. Another important aspect of these scenarios is the relative location of the attacker and victim. In many attacks, the attacker and victim run on the same core or two sibling logical cores [10], [1], [49], [46]. This is not surprising, as many of the targeted microarchitectural elements are not shared across cores, and thus, influences are only visible on the same core or sibling logical cores. Cross-core attacks are possible only for microarchitectural elements shared across the core [66], [30]. Consequently, research in the 2010s focused more on cross-core shared microarchitectural elements [38], [59] and less on private per-core microarchitectural elements. More recently, studies have focused more on these elements again [1], [28], [9].

C. Processor Power States (C-States)

C-States are processor power states defined in the Advanced Configuration, and Power Interface (ACPI) [55]. The ACPI defines power states C0 to Cn, where C0 is the running state in which the CPU executes instructions and C1 through Cn are idle states where the processor consumes less power. The time to enter and exit a C-State depends on the depth of the power state (with C1 being light sleep), where deeper idle save more power but have a higher cost to enter and exit.

Older Intel x86 processors only had the privileged `hlt` instruction to put the processor into the C1 idle state. To avoid the expensive system calls to briefly move the processor into an idle state, Intel subsequently introduced the `pause` instruction. The `pause` instruction for several processor generations was the most efficient way for user space programs to implement a busy wait while staying in C0. The *MONITOR* x86 ISA extension introduced the `mwait` instruction which allows waiting for a write to a memory location and provides a simple way to switch to deeper C-states than C1. Recently Intel introduced the *WAITPKG* x86 ISA extension with the Tremont and Alder Lake microarchitectures adding the `umwait` and `tpause` instructions on all privilege levels. While the privileged `mwait` instructions allows entering all C-States, `tpause` and `umwait` are restricted to two substates of the C0 running state (C0.1 and C0.2). The `umwait` instruction can be used to monitor a write accesses to a specific memory range. The `tpause` instruction allows to generically wait for a deadline specified in the `EDX:EAX` registers to optimize busy waits. Thus, `tpause` provides an efficient and fast way for user programs to switch to C0.1 and C0.2 and is now the most efficient option for short waits [17]. Thereby it reduces the energy consumption and, in the case of C0.2, increases the performance of sibling logical cores. Both of these idle states offer small power savings (5-13% compared to a busy wait) and a fast wake-up time (about 22% increase compared to a busy wait) [4]. Bityutskiy [4] found no significant latency advantage of C0.1 over C0.2, justifying the use of only the C0.2 in recent Linux kernel patches. For longer waiting times, the user can still rely on operating system support. Intel recommends the use of the `tpause` instruction for user-level busy polling, synchronization, and asynchronous I/O, to reduce energy consumption while maintaining a much lower wake-up latency than previously available methods [16].

The operating system can prevent user programs from setting excessively high deadlines by setting the maximum sleep time through the `IA32_UMWAIT_CONTROL` model-specific register. However, besides the time limit, `tpause` is also woken up by non-maskable interrupts, system management interrupts, machine check exceptions, and external interrupts, regardless if interrupts are enabled (`RFLAGS.IF`) [16], [18].

III. Idle State Side Effect Information Leakage

In this section, we present the attack primitives of IdleLeak. We first analyze the side effects of the C0.1 and C0.2 idle states and how they can be exploited. As these new idle states C0.1 and C0.2 are reachable from userspace, their behavior and effects can be observed by an unprivileged attacker. We then build two attack primitives, ActiveIdleLeak and PassiveIdleLeak. With ActiveIdleLeak, we use side effects of the C0.2 idle state to leak information from a restricted environment to an attacker-controlled environment. With PassiveIdleLeak, we use side effects of the C0.1 and C0.2 idle state to spy on co-located workloads, network activity, user input, and system activity. We also demonstrate both attack primitives in virtual machines and show how they can even be used for information leakage across a VM-host boundary. Our attacks demonstrate the negative security implications of adding user-controlled

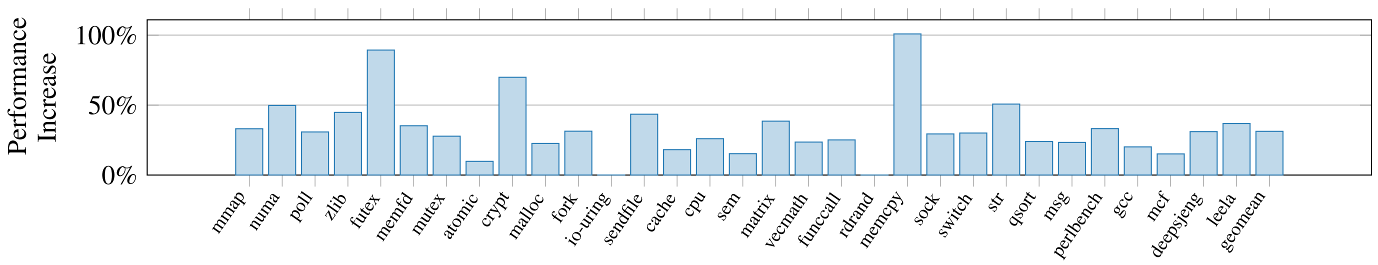


Fig. 1. Performance increase in the Phoronix Test Suite and SPEC CPU 2017 on a logical core while the sibling logical core is in the C0.2 idle state. We observe an average increase of 31 %.

idle states, as they allow observing various system-level events with a high accuracy, even in practical open-world attacks.

A. *Performance Side Effects of the C0.2 Idle State*

For ActiveIdleLeak, we investigate the side effects of the C0.2 idle state on the system performance, in particular the performance of co-located workloads. While performance gains are documented [4], their security implications are not. Therefore, we analyze the performance effects in more detail using micro-benchmarks and macro-benchmarks to infer how they can be used in side-channel and covert-channel scenarios.

As macro-benchmarks, we use SPEC CPU 2017[1] and Stress-NG from the Phoronix Test Suite. We run the benchmarks on an Intel i7-13900K on one core while the sibling logical core enters idle state C0.2 using `tpause`. We compare the benchmark results to a run with a busy wait on the sibling logical core. All benchmarks except for two show a significant performance increase of 31 % on average when the sibling logical core is in the C0.2 idle state (cf. Figure 1). Only two benchmarks from the Phoronix Test Suite show no significant change: `rdrand` and `io-uring`. The reason for these two outliers is that the performance limitations lie outside of the core: For `rdrand`, the corresponding random-number generator module is located outside of the processor core and shared across all cores [7]. For `io-uring`, testing the performance of the `io-uring` asynchronous I/O framework on Linux, the default setting is an interrupt-driven mode [25], *i.e.*, the performance of the test is largely not limited by core-internal resources. The benchmarks with the highest performance gains are primarily single-core compute-bound, e.g., `memcpy` at +100 %, `futex` at +89 %, and `crypt` at +70 %.

The performance is also influenced by the frequency of accesses to potentially uncached data, which induces pipeline bubbles and stalls. Hence, benchmarks like `memcpy`, `crypt`, and `str` show a much higher performance gain than benchmarks with a higher cache miss frequency, e.g., `cache` and `qsort`. Similarly, besides `io-uring`, also other benchmarks, e.g., `atomic` and `sem`, may require cross-core operations or the invocation of coherency protocols, leading to a lower performance gain than more compute-bound workloads.

These results already indicate that there are workload-dependent influences that an attacker could exploit. To understand the performance effects of C0.2 on an instruction level, we perform micro-benchmarks with specific operations we will subsequently use in our attacks. To determine the optimal instruction to construct a channel, we select nine candidate x86 instructions that we expect to be mainly influenced by core performance rather than external device or memory latency. Since we focus on the core performance influence, we test all instructions with register operands with randomized inputs.

[1]We excluded 648.exchange2, 620.omnetpp, 657.xz, 625.x264, and 648.xalancbmk as they did not compile on our test systems.

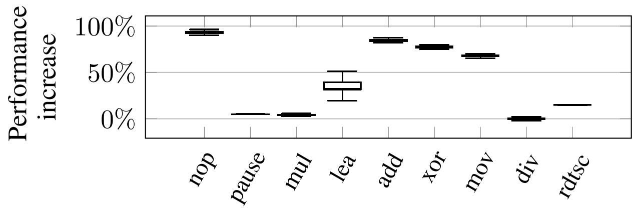


Fig. 2. Performance increase of a set of x86 instructions on an Intel i7-1260P, when the sibling logical core is in idle state C0.2 compared to a busy wait. We can see that for some instructions an increase of almost 100 % is possible, whereas other instructions see no performance gains.

We measure their change in execution speed when the sibling core is in C0.2 compared to a busy wait. Each instruction is measured 10 000 times with a busy wait on the sibling core and 10 000 times while the sibling core is in idle state C0.2. We repeat the target instruction 8 192 times to ensure that the CPU can take full advantage of the increased space of internal buffers and pipeline elements released by the idle sibling core. For each measurement, the target instruction executions are surrounded by `serialize` instructions, to avoid reordering of the measurement code.

The results of our measurements are shown in Figure 2. For fast instructions that take a single cycle to execute, such as `add`, `xor`, and `mov`, we observe a median speedup of roughly 80 %. We observe a speedup of less than 10 % for `pause`, `div`, and `mul`, even though they have no memory operands. This indicates that the throughput limitation for these instructions is not in any pipeline element or buffer shared between the sibling logical cores. The `lea` instruction receives a speedup of ≈40 % and `rdtsc` receives a speedup of ≈20 %. The `nop` instruction receives the most significant performance boost of 90 %. This is not surprising, as `nop` should only be limited by the throughput of the core's frontend and reorder buffer but no execution unit. Consequently, having the full reorder buffer available doubles the throughput. We use `nop` in combination with C0.2 to build a covert channel in Section IV.

B. *Interrupt Detection*

Zhang et al. [68] recently proposed the use of `umwait` for interrupt detection in native code as it is woken up on

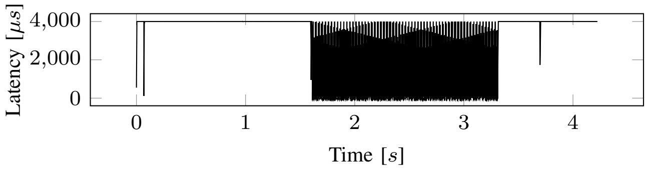
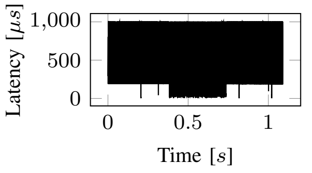
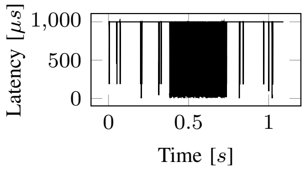


Fig. 3. Interrupts detected by a native attacker. The block with lower latencies in the middle, corresponds to a high number of interrupts caused by a touch pad.



(a) Recorded trace.



(b) Timer interrupts filtered.

Fig. 4. Touch-pad interrupts recorded by a VM-based attacker are clearly visible (indicated by the low latencies in the middle) when monitoring interruptions with `tpause` and filtering the timer interrupts.

external interrupts [17]. Furthermore, `umwait` also wakes up when the specified memory is written to. Since we do not require the memory monitoring for any of our attacks, we use `tpause`, as a less noisy alternative to `umwait`. We further show undocumented behavior that allows us to use `tpause` and `umwait` inside a virtual machine to detect interrupts of co-located virtual machines and the host system.

There are three reasons for `tpause` to wake up: First, the instruction continues if the specified deadline is reached. Second, it continues if the the operating system's deadline is exceeded. Finally, it wakes up if an interrupt occurs.

For PassiveIdleLeak interrupt monitoring, we set the `tpause` deadline to the maximum value. This high deadline ensures that we never wake-up due to reaching the deadline.[2] The operating system's deadline, setting the real maximum deadline, cannot be influenced by the user. However, a wake-up caused by the operating system's deadline sets the carry flag. A wake-up due to an interrupt does not set the carry flag. Thus, we can reliably detect interrupts using `tpause`.

To build our PassiveIdleLeak interrupt detection attack primitive, we repeatedly run `tpause` with the maximum sleep time until a wake-up occurs with the carry flag cleared. This is different to `umwait` which can also spuriously wake up due to memory activity, without revealing this wake-up reason to the user program, *i.e.*, effectively inducing noise into the channel.

Figure 3 shows an example of a PassiveIdleLeak interrupt trace recorded with the `tpause` instruction on an i7-1260P running a default-configured Ubuntu 22.04 (Linux 6.1.0). The y-axis represents the observed sleep time. Effectively, this is the time the logical core spent in the idle state. The x-axis shows the wall-clock time. One constant element the interrupt traces contains, is a continuous line, in this case at $4\,000\,\mu s$, representing regular timer interrupts generated by the operating system for context switches. Any deviation from this line indicates the occurrence of a different interrupt. The interrupt trace in Figure 3 consists mainly of timer interrupts at $4\,000\,\mu s$ and touchpad interrupts between $1.6\,s$ and $3.3\,s$. This trace already shows the possibility to use PassiveIdleLeak to spy on user input activity. A similar trace recorded in a VM environment is shown in Figure 4a. The VM runs a default-configured Debian 11 (Linux 6.2.0), on a default-configured KVM Ubuntu 22.04 host, with guest timer interrupts every $1\,000\,\mu s$. We observe wake-ups every $4\,000\,\mu s$ indicated by a drop in sleep time, despite the lack of received interrupts inside the guest. We find that these wake-ups are caused by the host's timer interrupts. The interrupts wake up the logical core from the idle state and let the execution return from `tpause` even though these interrupts are intended for the host. The interrupts cause an immediate VM exit and transfer control to the host. However, this still allows guests running in a VM to spy on interrupts intended for the host. We filtered the host timer interrupts by removing the regular interrupts every $4\,000\,\mu s$ shown in Figure 4b to highlight the other interrupts. This filtering results in a significantly clearer interrupt trace showing touchpad interrupts intended for the host between $0.4\,s$ and $0.7\,s$.

For C0.1, we further observe that the CPU not only wakes up on interrupts intended for the waiting core but also on interrupts intended for sibling logical cores even while running inside a VM. On our Ubuntu 22.04 (Linux 6.2.0) we further observed that exceptions, e.g., divide-by-zero, page fault, general protection fault, on a sibling logical core result in a wake up for C0.1. This allows an attacker to use C0.1 to spy on interrupts and exceptions intended for the sibling logical core and, in the case of a cloud environment, on interrupts and exceptions intended for different VMs. Importantly, this behavior does not induce a VM exit on the attacker's logical core but still allows observing the interrupt. This has a significant advantage over the scenario where the interrupt arrives on the attacker core. Since the attacker core does not execute the interrupt service routine, it does not execute a VM exit. Instead, it can re-enter C0.1 immediately after detecting an interrupt and, thus, is immediately ready to detect the next interrupt. Since interrupts do not adhere to security-aware scheduling policies [21] and the x86 architecture implies the assignment of interrupts to a specific cores, it is not trivial to mitigate this issue without significant performance cost, as we discuss in Section VII.

We did not observe the idle-state-interrupting behavior upon sibling-logical-core interrupts in idle state C0.2. Since the C0.1 and C0.2 idle states were both introduced for the user-mode instructions `tpause` and `umwait`, disabling access to C0.1 is not possible without fully disabling access to `tpause` and `umwait`. In Section V, we evaluate the PassiveIdleLeak attack primitive in attacks on user input. In Section VI, we show that PassiveIdleLeak can also be used for website and video fingerprinting, with high accuracies despite the victim running on a separate physical core.

IV. Covert Channel

We present two high-speed covert channels based on ActiveIdleLeak and on PassiveIdleLeak. The ActiveIdleLeak covert channel is based on the performance effects of the idle state C0.2 on different instructions. The PassiveIdleLeak covert channel is based on the wake-up of sibling logical cores from the idle state C0.1 when an interrupt or exception occurs.

[2]A spurious wake-up is possible when the time-stamp counter (TSC) overflows, which will not happen within 10 years of the last reset according to Intel [15].

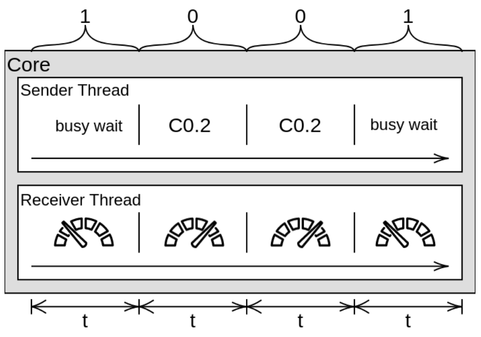


Fig. 5. Overview of a ActiveIdleLeak covert-channel transmission where t is the length of a time slice. For each time slice, the sender enters C0.2, which speeds up the receiver to send a '0'-bit or busy waits to send a '1'-bit.

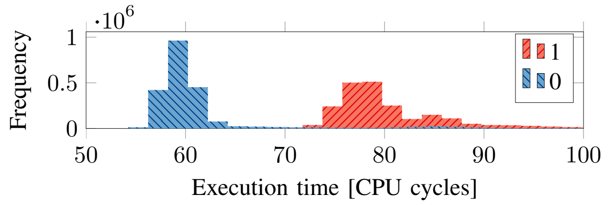


Fig. 6. The sender idling in C0.2 via `tpause` is used to encode a '0'-bit, busy-waiting is used to encode a '1'-bit.

A. Covert Channel Design

In this section, we explain the design of our two covert channels. For both the ActiveIdleLeak and the PassiveIdleLeak we use time slicing in combination with a primitive based on the corresponding attack to transmit data.

1) ActiveIdleLeak Transmission Primitive: In this scenario, we transmit a data stream bit-wise through the ActiveIdleLeak channel. The receiver repeatedly measures the execution time of a series of `nop` instructions as it showed the highest performance change in our micro-benchmark (cf. Section III). When the sender enters idle state C0.2, the execution time of the `nop` series will go down substantially. When the sender does not enter an idle state, the execution time of the `nop` series will not be affected. Thus, we build the covert channel on top of this timing difference, transmitting '0' and '1' bits.

Figure 5 presents the high-level overview of our covert channel. We use one receiver thread and one sender thread, each running on one logical core of the same physical core. In this example, the sender transmits a sequence of '0110'. For a '1'-bit, the sender performs a busy wait. Consequently, the receiver sees a low performance, *i.e.*, a higher execution time. For the next bit, a '0'-bit, the sender enters the C0.2 idle state via `tpause`, increasing the performance of the victim, *i.e.*, lowering the execution time of the `nop` series. The same operation follows for the next bit, another '0'-bit. For the fourth bit to transmit, a '1'-bit, the sender performs a busy wait, reducing the performance of the victim's `nop` series again. The receiver can now infer the full sequence '1001'.

Figure 6 shows the histograms for the two corner cases we use for transmission. Executing 512 `nop` instructions in the receiver takes 82 cycles ($\sigma_{\bar{x}}$ = 0.01 cycles, n = 2 151 841) when the sender performs a busy wait. In contrast, executing 512 `nop` instructions the receiver only takes 60.3 cycles ($\sigma_{\bar{x}}$ = 0.01 cycles, n = 2 151 841) cycles when the sender is in

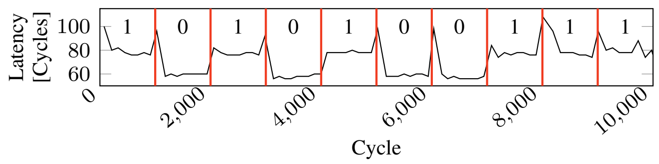


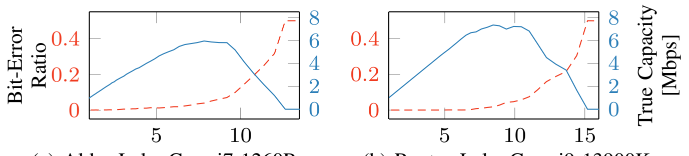
Fig. 7. The native ActiveIdleLeak covert channel transmission has a clearly visible difference in the latency between transmitting '0' and '1' bits.

state C0.2. While the timings of the two cases are very clearly separated, a small number of outliers can be observed in the '0'-bit case in Figure 6, at 80 to 90 cycles. These outliers result from the regular wake-ups from C0.2, caused by the operating system's deadline. Thus, for a short time, the sibling core is active before executing the next `tpause` to re-enter idle state C0.2. Since the observed execution time is then in the range of a busy wait, this effect introduces a small amount of noise into our side channel and our covert channel transmission.

2) PassiveIdleLeak Transmission Primitive: In this scenario, we transmit data bit-wise through the covert channel. We use one receiver and one sender thread, running on the logical cores of one physical core. The receiver measures the interrupt frequency of the sibling logical core with PassiveIdleLeak. The sender causes a wake-up of the receiver by generating exceptions. We use the divide-by-zero exception, as it can be triggered without a memory access. When the sender does not trigger exceptions, the receiver thread will detect fewer interrupts. Thus, we build the covert channel on top of this behavioral difference, transmitting '0' and '1' bits.

3) Synchronization: We synchronize our covert channels via the TSC. The transmission starts on a previously agreed-on TSC value and sends the bits in time slices of predefined length. For each time slice with our ActiveIdleLeak covert channel, the sender repeatedly executes `tpause` to stay in C0.2 until the end of the time slice to transmit a '0' or runs a busy wait to send a '1'. To compensate for noise, the receiver averages the execution times of all `nop` instruction sequences within a time slice at the end and determines the bit through a threshold. An example transmission with a time slice length of 1 000 cycles is shown in Figure 7. The time slices are indicated with vertical red lines. The ground truth is printed at the top of each time slice. The receiver's latency when receiving a '1' is around 80 cycles and for a '0' at 60 cycles. We can observe latency spikes of 100 cycles at the beginning of each time slice. These spikes are due to the sender determining which bit value should be sent in the upcoming time slice.

For each time slice with our PassiveIdleLeak covert channel, the sender generates exceptions until the end of the time slice to transmit a '1' or runs a busy wait to send a '0'. The transmission starts with 16 '1'-bits for initialization. The receiver counts the number of interrupts detected with PassiveIdleLeak for the first 16 time slices, averages them, and divides them by 2 to compute a threshold value. For the following time slices, the sender transmits the data. For each time slice after initialization, the receiver counts the number of interrupts and compares them with the threshold value. If the number of interrupts is above or equal to the threshold value, a '1'-bit was transmitted. If the number of interrupts is lower than the threshold, a '0'-bit was transmitted.



(a) Alder Lake Core i7-1260P (b) Raptor Lake Core i9-13900K

Fig. 8. The raw capacity of our native ActiveIdleLeak channel, and the corresponding bit-error ratio and true capacity. We can see that the optimal true capacity is reached between 5 and 10 Mbit/s of raw capacity.

Relying on the TSC for synchronization has additional benefits for our IdleLeak covert channels. The wake-up deadline of `tpause` is specified by a TSC value at which the processor should leave the idle state again. Consequently, we inherently re-synchronize our covert channel through the used instruction and eliminate the need for additional synchronization logic.

B. Evaluation

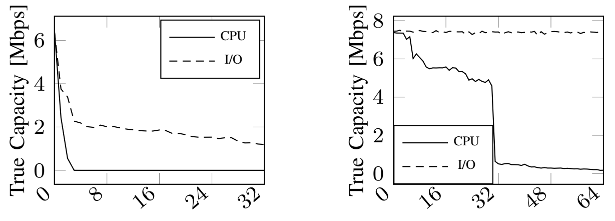
We evaluate both covert channels by sending data from `/dev/urandom` on an i9-13900K. We additionally evaluate our ActiveIdleLeak covert channel on an i7-1260P. We assume attacker and victim run in separate processes and that they can be scheduled on sibling logical cores of the same physical core. We assume they run native userspace code. Furthermore, there is no legitimate communication channel between the processes and no bugs could be exploited for communication.

For our initial tests, to obtain the optimal configuration parameters, we run the system idle without interfering workloads. Later on we evaluate the influence of different types of noise for the native ActiveIdleLeak channel. We evaluate different time slice lengths and record the raw capacity and bit-error ratio of the channel. Since our channels are based on time slices, the raw capacity is inversely linear in the time slice length. Thus, shorter time slices result in a higher transmission rate but due to the shorter time slices, more bit errors may occur. Consequently, while the raw bitrate increases with a time slice reduction, the actual channel capacity might decrease due to a higher bit-error ratio. To find the optimal transmission rate, we compute the true capacity based on the raw bitrate and the bit-error ratio.[3]

1) Native ActiveIdleLeak Covert Channel: Figure 8 shows the true channel capacity and bit-error ratio as a function of the raw capacity on our test systems. On the i9-13900K, shown in Figure 8b, the bit-error ratio stays below 1 % up to a raw capacity of 7.2 Mbit/s. The true capacity then peaks at a raw capacity of 8.9 Mbit/s with a bit-error ratio of 3.2 % ($\sigma_{\bar{x}}$ = 0.01 %, n = 512), corresponding to a true capacity of 7.1 Mbit/s ($\sigma_{\bar{x}}$ = 0.004 Mbit/s, n = 512). On the i7-1260P, shown in Figure 8a, the bit-error ratio is slightly higher. The true capacity here peaks at a raw capacity of 7.8 Mbit/s with a bit-error ratio of 4.4 % ($\sigma_{\bar{x}}$ = 0.18 %, n = 512), corresponding to a true capacity of 5.9 Mbit/s ($\sigma_{\bar{x}}$ = 0.05 Mbit/s, n = 512).

To determine the effect of system noise on the covert channel transmission rate, we run our channel while other workloads run on the CPU. We use the optimal parameters

[3]We use the binary symmetric channel model to 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.



(a) Stress threads on i7-1260P (b) Stress threads on i9-13900K

Fig. 9. Effect of I/O and CPU-related workloads simulated with the `stress` utility on the covert channel's true capacity. On the i9-13900K, the capacity reaches ~0 bit/s with more than 30 stress threads. I/O heavy workloads do not have a significant influence on the capacity. On the i7-1260P, capacity drops to ~0 bit/s with 4 CPU stress threads. I/O workloads decrease to 2 Mbit/s with 4 stress threads decreasing slowly with more threads.

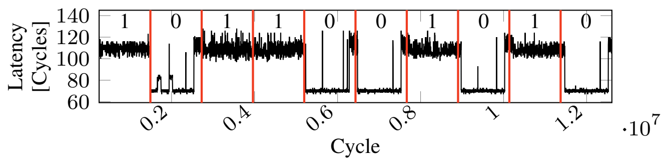
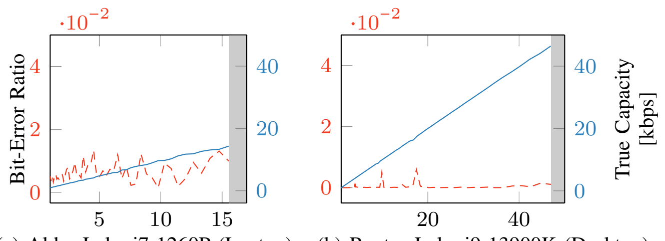


Fig. 10. The cross-VM covert channel transmission is more noisy but still has a clear difference in the latency between '0' and '1' bits.

for each CPU according to Figure 8. To simulate the system noise, we use the `stress` tool to run I/O and CPU workloads with different numbers of threads. For both of our tested CPUs (i7-1260P and i9-13900K), we run up to double the number of stress threads as there are logical cores for both I/O and CPU workloads. The results of our evaluation are shown in Figure 9. On the i9-13900K, I/O workloads do not significantly impact the covert channel, with the true capacity staying at 7.1 Mbit/s for all scenarios. With CPU workloads, the capacity decreases to 6 bit/s with 8 stress threads and to almost 0 Mbit/s with more than 30 stress threads. Together with our 2 threads taking part in the transmission, this results in 32 threads before the transmission breaks down, exactly the number of logical CPU cores. On the i7-1260P, I/O workloads have a significant effect on the transmission rate, dropping it from 6 Mbit/s to 2 Mbit/s with only 4 stress threads, decreasing slowly to ~1.8 Mbit/s with 32 threads after that. For CPU workloads, the covert channel drops to almost 0 bit/s with only 4 stress threads which are significantly less than the 16 logical cores the CPU has. During testing, we observed that the i7-1260P reached its thermal limits of 100 °C, causing it to thermal throttle. When fixing the fan speed to its maximum to avoid thermal throttling, we still observed high core frequency fluctuations throughout the experiment with an increasing number of stress threads. We presume these fluctuations to be the result of lower power limits and the stricter efficiency requirements of laptop CPUs.

2) Cross-VM ActiveIdleLeak Covert Channel: Our cross-VM covert channel evaluation is similar to our native code evaluation. Additionally to the separate processes for attacker and victim, we assume attacker and victim run in separate VMs on the same physical machine. Both VMs run a recent Debian 11 with Linux kernel 6.2.0 and can be scheduled on sibling logical cores. Attacker and victim have no means of communication besides the covert channel.

Figure 10 shows an example transmission of our cross-VM channel. As we are running in VMs, there is no shared



(a) Alder Lake i7-1260P (Laptop) (b) Raptor Lake i9-13900K (Desktop)

Fig. 11. The raw capacity (x-axis in kbps) of our cross-VM channel, and the corresponding bit-error ratio and true capacity. The optimal true capacity is reached at 46.9 kbit/s of raw capacity for the i9-13900K and at 15.5 kbit/s of raw capacity for the i7-1260P. At a higher raw capacity the synchronization mechanism fails leading to no data transmitted, marked by the gray area.

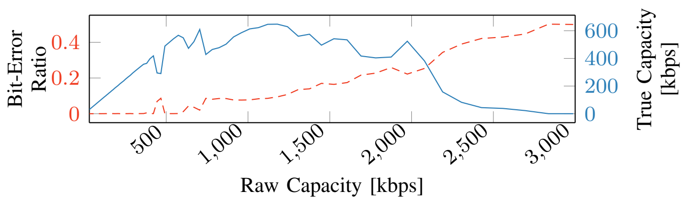


Fig. 12. The raw capacity of PassiveIdleLeak, and the corresponding bit-error ratio and true capacity on an i9-13900K. The optimal true capacity of 656.37 kbit/s ($\sigma_{\bar{x}}$ = 0.63 kbit/s, n = 1 024) is reached with a bit-error ratio of 9.22 % ($\sigma_{\bar{x}}$ = 0.02 %, n = 1 024) at 1 179 kbit/s of raw capacity.

time-stamp counter (TSC). We resolved this challenge with an initialization sequence of 5 alternations between '1' and '0', unlikely to be received through random noise. Random data follows after this sequence similar to Section IV-B1.

Figure 11 shows the true channel capacity and bit-error ratio as a function of the raw capacity on our test systems. On the i9-13900K, shown in Figure 11b, the bit-error ratio stays below 0.3 % up to a raw capacity of 46.9 kbit/s. The true capacity then peaks at a raw capacity of 46.9 kbit/s with a bit-error ratio of 0.22 % ($\sigma_{\bar{x}}$ = 0.07 %, n = 370), corresponding to a true capacity of 46.32 kbit/s ($\sigma_{\bar{x}}$ = 0.15 kbit/s, n = 370). On the i7-1260P, shown in Figure 11a, the bit-error ratio is slightly higher. The true capacity here peaks at a raw capacity of 15.5 kbit/s with a bit-error ratio of 2.1 % ($\sigma_{\bar{x}}$ = 0.39 %, n = 60), corresponding to a true capacity of 13.57 kbit/s ($\sigma_{\bar{x}}$ = 0.27 kbit/s, n = 60). We have no data for higher raw capacities for either CPU as the synchronization mechanism fails.

3) Native PassiveIdleLeak Covert Channel: Figure 12 shows transmission rate and bit-error ratio compared to the raw capacity of the PassiveIdleLeak channel. The true capacity peaks at 656.37 kbit/s ($\sigma_{\bar{x}}$ = 0.63 kbit/s, n = 1 024) with a bit-error ratio of 9.22 % ($\sigma_{\bar{x}}$ = 0.02 %, n = 1 024) at 1 179 kbit/s of raw capacity. After the true capacity peak, the time slice length is smaller than the time to trigger an exception and recover from it. We conclude that PassiveIdleLeak can be used to leak data at a high transfer rate from sibling logical cores.

4) Previous Work: Both our ActiveIdleLeak (7.1 Mbit/s) and PassiveIdleLeak (656 kbit/s) covert channels achieve comparable or faster transmission rates than previous work. Zhang et al. [68] built a covert channel based on detecting speculative writes with `umwait` and achieved a transmission rate of 200 kbit/s. Gast et al. [9] exploit scheduler contention to leak and transmit 2.7 Mbit/s. Saileshwar et al. [45] use cache contention and achieve a transmission rate of 14.4 Mbit/s.

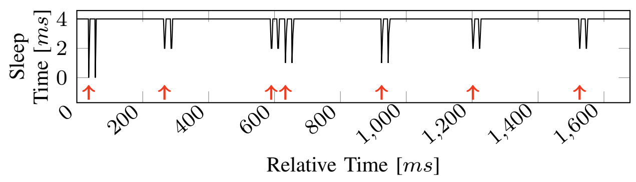


Fig. 13. Keystroke detection using PassiveIdleLeak on an i7-1360P. The downward spikes show the interrupts caused by key-down and key-up events, *i.e.*, where keystrokes are detected. The red arrows show the ground-truth.

V. KEYSTROKE DETECTION

In this section, we present our inter-keystroke timing attack. We exploit that keystrokes of USB keyboards generate interrupts and detect them with PassiveIdleLeak.

A. Threat Model

We assume the attacker has access to millisecond-accurate timers, e.g., `clock_gettime` or C++ standard clocks. We make no assumptions on the availability of high-resolution timers such as the TSC, as they can be manipulated. Linux assigns each external interrupt to one of the available cores. Interrupt-core assignments rarely switch between cores at runtime, as can be observed via `/proc/interrupts`. We assume the attacker can start multiple threads on different cores. Thus, we can assume that the attacker is eventually scheduled on the core that receives the keyboard device interrupts. We make no assumption on the core the victim code receiving the keystrokes is running on, as this is independent of the core that receives the keyboard device interrupts.

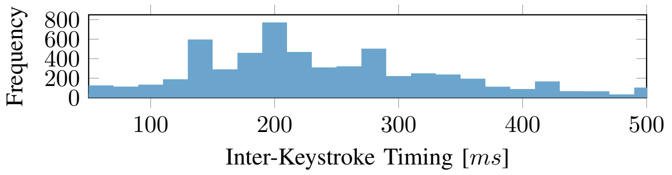
B. Attack Implementation

For our inter-keystroke timing attack, the attacker first records an interrupt trace using PassiveIdleLeak on the core that receives the USB interrupts. USB devices generate interrupts when sending data to the host system. A USB keyboard sends two types of interrupts for a single keypress: one for the key press (key down) and one for the key release (key up). Once recorded, the attacker analyzes the data and infers the precise inter-keystroke timings.

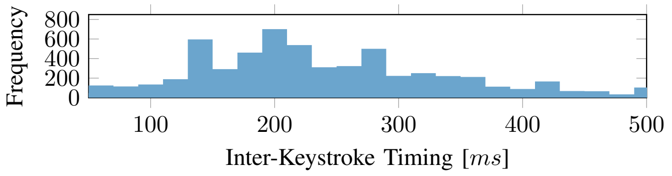
Figure 13 shows an interrupt trace recorded with PassiveIdleLeak. The key-down interrupts are marked with red arrows and result in a significant dip in sleep time. After each key-down interrupt, a key-up interrupt occurs. The time between two interrupts depends on the writing speed but is usually in the tens of milliseconds [20]. We use the key-up interrupt to distinguish keystroke interrupts from other interrupts that occur on the same core. We exclude unlikely interrupt pairs that are too low (<10 ms) or too high (>100 ms). Thus, for every interrupt in our trace, we search for a possible key-up interrupt that is at least 10 ms and at most 100 ms after the possible keystroke. We make no assumptions on overlapping key presses. If such an interrupt exists, we log the first interrupt as the key down event, remove the interrupt pair from the trace and continue with the next interrupt.

C. Evaluation

To evaluate PassiveIdleLeak, we use an ARM Mbed LPC-1768 μ-controller board. This controller board acts as a USB

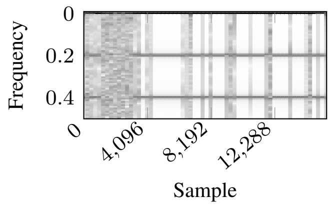


(a) Distribution of inter-keystroke timings in the ground-truth.

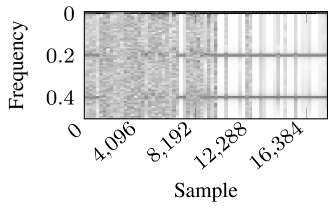


(b) Distribution of inter-keystroke timings recovered by PassiveIdleLeak on an i7-1260P.

Fig. 14. The distributions of inter-keystroke timings and of the ground-truth. The recovered and ground-truth inter-keystroke timings are very similar.



(a) google.com



(b) youtube.com

Fig. 15. The STFTs of the interrupt traces of different websites show distinct patterns, here with the examples (a) google.com and (b) youtube.com. Both traces were acquired in a VM-based attack, attacking a browser running on the host machine. (ploted values are amplified for readability)

keyboard device that we plug into our test machines to inject hardware keyboard interrupts. We use pre-recorded inter-keystroke timing data [29] covering a hundred participants typing an eight-letter word ten times. We use these 7 000 inter-keystroke timings and replay them using the μ-controller with a high-precision clock. Thus, we have highly accurate ground-truth data for the actual keystroke interrupts.

We evaluate our native-code inter-keystroke timing attack on an i7-1260P on Ubuntu 22.04 (Linux 6.1.0). We schedule the attacker on one of the two logical cores of the physical core that receives the keyboard interrupts. Our keystroke detection has a precision of 87.1 %, a recall of 94.1 %, and an F1 score of 90.5 %. For the timing, we measured and statically subtracted the average deviation, which was 15.2 µs, effectively minimizing the average deviation to 0. The detected and reference inter-keystroke timings of the correctly detected keystrokes are shown in Figure 14. The reference distribution (Figure 14a) and the detected distribution (Figure 14b) are almost identical. We achieve a standard deviation of 950 µs and a standard error of 12 µs for our correctly detected keystrokes. Therefore, we conclude that PassiveIdleLeak can accurately monitor interrupt-based singular events like keystrokes.

VI. Website and Video Fingerprinting

In this section, we present our website and video fingerprinting attacks using PassiveIdleLeak. We show that it is possible to determine the website a user accesses in a closed-world setting over the top 100 websites from the Alexa top 1 million list [2] and an open-world setting with the top 100 websites and an `other`-class for websites not in the top 100. Furthermore, we demonstrate an open-world and a closed-world video-fingerprinting attack on two popular video-streaming websites, YouTube and PornHub, to distinguish between the top 20 trending videos in the US at the time of writing for YouTube and between the most viewed videos of the 20 most popular categories on PornHub. In this scenario, we assume an attacker runs code in a VM on the same machine as the victim.

Network devices generate interrupts when sending or receiving data. When accessing a website or video streams, the computer sends and receives numerous network packages. The number of packages depends on the content, while the time between packages can, among other things, depend on the content, server location, and server software, resulting in unique interrupt patterns for most content. We use PassiveIdleLeak to infer the exact website accessed or video stream watched by the user, based on a convolutional neural network (CNN) we train to classify the interrupt patterns.

A. Threat Model and Attack Setup

We run our measurements on an i7-1260P CPU with Mozilla Firefox 113.0.2 running Ubuntu 22.04. We assume that a user wants to run untrusted code in a secure way and, hence, runs it in a VM. While running the code in a VM, the user browses the web. We assume the attacker has access to millisecond-accurate timers in the VM, e.g., `clock_gettime` or C++ standard clocks. We do not use and do not make assumptions about the availability of high-resolution timers, such as the timestamp counter, as they can be manipulated by the host. Linux assigns each external interrupt to one of the available cores. Typically, these assignments do not switch at runtime, as can be observed via `/proc/interrupts`. On Linux with KVM, VM threads are scheduled like other threads, irrespective of interrupt routing. Thus, it is reasonable to assume that the attacker VM is eventually scheduled on the core that receives the host's network device interrupts or on a sibling logical core. We do not make any assumption on the core the web browser is running on, as this is independent of the core that receives the network device interrupts.

B. Attack

1) Website Fingerprinting: Our attack consists of an online data-collection phase and an offline phase for processing and evaluation of the traces. The online phase consists of a user space program running inside a VM continuously running PassiveIdleLeak on the CPU core that receives the network device interrupts of the host as described in Section III-B. To measure the execution time, we use an accurate millisecond timer, as we only need to distinguish guest timer interrupts from other interrupts. In the offline phase of our attack, we analyze the collected traces. Since we previously determined that a timer interrupt in our VM setup takes ≥1 ms, as shown in Figure 4b, a sleep time of ≥1 ms means only timer interrupts occurred and an execution time of <1 ms means a different interrupt must have occurred.

In the next step, we search for the time frame in which the website access occurred. When there is no network traffic, the core rarely receives interrupts except for the regular guest timer interrupts, and the regular host timer interrupts. This results in a change in interrupt frequency whenever a website access occurs. We compute the short-time Fourier transform (STFT) of the interrupt trace with a window size of 256 to analyze the frequency change. The STFT of an access to `google.com` and an access to `youtube.com` are shown in Figure 15. The x-axis of the plots shows the sample number, and the y-axis shows the change in frequency. Since the time between samples can vary depending on the number of interrupts that occur in a given time frame and we do not require an exact sampling frequency for further processing, we do not have a unit of measurement for our frequency scale. Due to this lack of a consistent sampling frequency, we directly refer to the frequency value returned by the STFT without any unit of measurement. In case of no network traffic and a lack of other interrupts, the 0.2, 0.4, and 0 frequency components are high, while all other components are almost 0. An example of almost no network traffic is shown in Figure 15b in the last fourth of the trace since most of the website is already loaded. A website access starts if the 0.2, 0.4, and 0 frequency components decrease over multiple STFT windows. Following the detected website access, we use the following 512 STFT windows and forward them to our classifier. Examples can be seen between samples 0 and 4 096 in Figure 15a and between samples 1 024 and 10 240 in Figure 15b. While some of the tested websites take longer to load than the 512 STFT windows we use, we determined through manual testing that the used time frame is enough to uniquely identify a website.

After this pre-filtering step, we forward the 512 STFT windows to our convolutional neural network (CNN) for classification. We use the STFT instead of the interrupt trace since server response times can change slightly with each website access, shifting interrupt timings. The resulting shift in features in the time domain results in slightly different traces on every website access. Applying an STFT to the interrupt trace allows for efficient convolutions on the input rather than relying on less efficient, fully connected layers. Additionally, compared to a Fast Fourier Transform (FFT) over the whole trace, the STFT preserves part of the time domain by applying a Fourier transform on separate slices of the trace instead of the whole trace. This technique is well established in the field of signal classification [65], [6], [14]. Our CNN consists of 4 convolutional layers followed by 3 fully connected layers and outputs a match probability for each of the 100 websites.

2) Video Fingerprinting: For our video-stream fingerprinting attack, we measure the interrupt trace of the first 10 seconds of a video. Contrary to our website fingerprinting, we do not apply the STFT directly on the interrupt trace. As the interrupt trace of a video stream typically consists of a low number of interrupts, mainly from the timer interrupt, with short bursts of a high number of interrupts, an STFT directly on the interrupt trace becomes inefficient. Instead, for each millisecond, we count the number of interrupts that occurred, resulting in the number of interrupts per millisecond. We then perform an STFT on this transformed interrupt trace and feed the result into a CNN similar to our website fingerprinting attack. Our CNN for video-stream fingerprinting consists of 4 convolutional layers followed by 3 fully connected layers and outputs a match probability for each of the 20 videos.

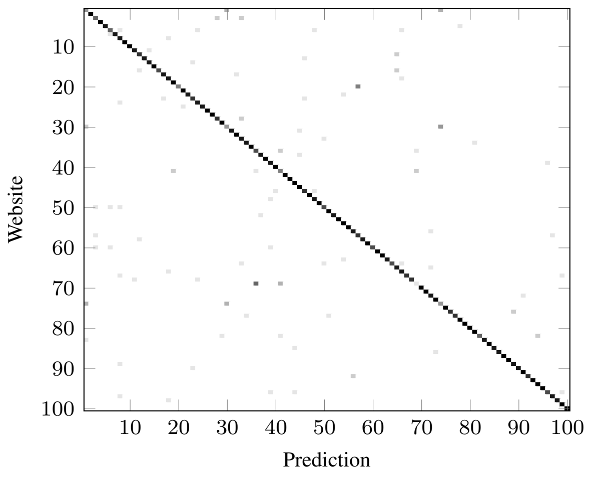


Fig. 16. The confusion matrix for our website-fingerprinting attack, with network interrupts arriving on the attacker's core. For the classification, 10 out of 50 samples serve as the test set, with the remainder as the training set, for each of the websites.

C. Evaluation

We evaluate our open-world website fingerprinting, closed-world website fingerprinting, and video fingerprinting attacks with the attacker VM on the sibling logical core of the core that receives the network interrupts. We additionally evaluate our closed-world website fingerprinting attack with the attacker VM on the core that receives the network interrupts. For each closed-world website fingerprinting scenario, we collected 5 000 traces (50 per website). For our open-world scenario, we collected 20 000 traces (200 per website) for each of the top 100 websites and 7 000 traces from 7 000 further websites from the Alexa 1 million list [2] (1 trace per website). For our closed-world video fingerprinting scenario on YouTube, we collected 2 900 10 s traces (145 per video) of the top `20` trending videos in the US at the time of writing; for PornHub, we collected 3 300 10 s traces (165 per video) of the most viewed videos in the top 10 default and gay categories. For our open-world video fingerprinting scenario on YouTube, we collected 1 500 10 s traces (75 per video) of the top `20` trending videos in the US at the time of writing; for PornHub, we collected 1 500 10 s traces (75 per video) of the most viewed videos in the top 10 default and gay categories. To represent the `other`-class we collected 1 000 traces of 1 000 random videos (1 trace per video) for both YouTube and PornHub.

For all attacks, we split the collected traces randomly into a test set (20 %) and a training set (80 %). Our CNN was trained with a validation split of 10 % of the training set.

1) Closed-World Same-Core Interrupt Website Fingerprinting: In this scenario, the attacker runs on the core that receives the network device interrupts. We tested our classifier on the test set which is not used for training and achieved an F1 score of 88.2 %. The full confusion matrix is shown in Figure 16. Each cell indicates the probability that our classifier labels the access to a website indicated by the row as a particular website indicated by the column. The high probabilities along the diagonal indicate that our classifier has a high accuracy

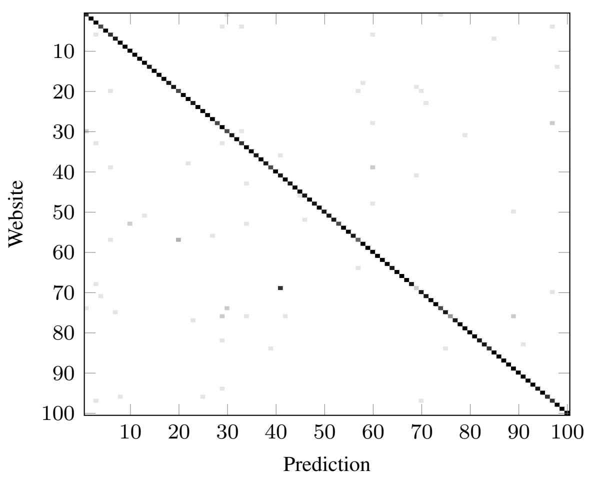


Fig. 17. The confusion matrix for our website-fingerprinting attack, with network interrupts arriving on a sibling logical core. For the classification, 10 out of 50 samples serving as the test set, with the remainder as the training set, for each of the websites.

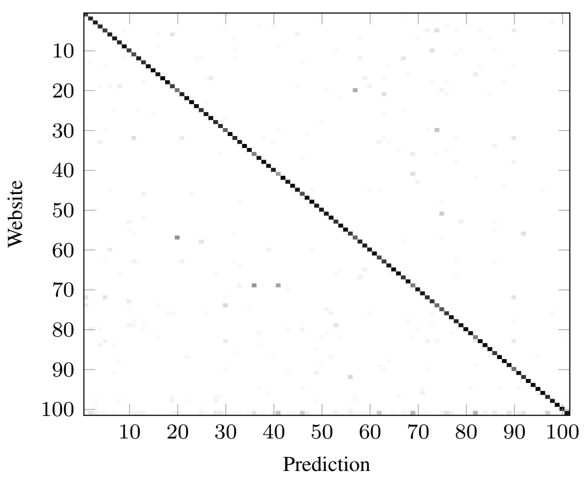


Fig. 18. The confusion matrix for our open-world website-fingerprinting attack, with network interrupts arriving on a sibling logical core.

with no website being labeled correctly for less than 30 % of the time, which is significantly higher than random guessing at 1 %. The websites with the worst accuracies are `twimg.com` (10 %), `google.com` (40 %), `google.co.in` (40 %), and `google.com.hk` (40 %). The `twimg.com` domain is used by twitter for images and videos and only serves a page on subdomains but not under the direct URL `twimg.com`, at the time of testing, resulting in the low score. All other websites have accuracies of at least 50 %. Websites that our classifier frequently confuses with each other include `google.com`, `google.com.hk`, and `google.co.in` since all three domains forward the browser to the same server. Grouping all Google domains into one class results in an overall F1 score increase from 88.2 % to 90 % for our model.

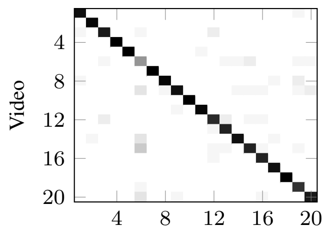
2) Closed-World Sibling-Logical-Core Interrupt Website Fingerprinting: In this scenario, the attacker runs on a sibling logical core of the core that receives the network device interrupts. We tested our classifier on a test set which is not used for training and achieved an F1 score of 92.4 %. The full confusion matrix is shown in Figure 17. Each cell indicates the probability that our classifier labels the access to a website indicated by the row as a particular website indicated by the column. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no website labeled correctly for less than 20 % of the time, which is significantly higher than random guessing at 1 %. The websites with the worst accuracies are `twimg.com` (20 %, we described this issue in Section VI-C1), `dzen.ru` (40 %), and `tencent.com` (60 %). All other websites have accuracies ≥70 %. Grouping all Google domains into one class results in an F1 score increase from 92.4 % to 93.1 % for our model.

Since PassiveIdleLeak has to use C0.1 to detect interrupts of the sibling logical core, the attacker detects all interrupts from both logical cores that are part of the physical core the attacker is running on. Despite the added noise from the higher interrupt frequency compared to Section VI-C1, this scenario performs significantly better, with an F1 score of 93.1 %. The increase in F1 score is the result of the higher accuracy PassiveIdleLeak has for detecting interrupts of sibling logical cores as they do not require the attacker logical core to execute interrupt service routines and do not result in VM exits for the attacker. Thus, the attacker is immediately ready to detect the next interrupt, decreasing the number of missed interrupts significantly. Despite the significant advantage of this approach, the added noise from the interrupts on the attacker core can negate this advantage if a device generates a high frequency and number of interrupts, resulting in a lot of noise. In such a high noise scenario, PassiveIdleLeak on the same core as the target interrupts using C0.2 (as evaluated in Section VI-C1) performs significantly better.

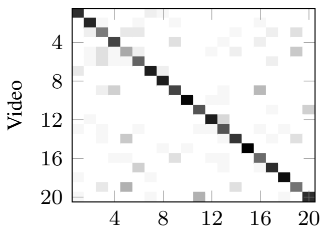
3) Open-World Website Fingerprinting: In this scenario, we add an `other`-class for websites not part of the top 100 from the Alexa top 1 million list [2] with the attacker running on a sibling logical core of the core that receives the network interrupts. For the training of the `other`-class, we use accesses to 5 600 websites from the top 1 million list. For testing of the `other`-class, we use accesses to 1 400 websites from the top 1 million list that are **not** in the training set. As the `other`-class test set websites have never been seen by our classifier during training, they result in a realistic accuracy measurement of our classifier on unknown websites.

Our classifier achieved a macro-averaged F1 score of 85.2 % on the test set. The `other`-class, in particular, has an accuracy of 87.4 % on the 1 400 test traces. The full confusion matrix is shown in Figure 18. Each cell indicates the probability that our classifier labels the access to a website indicated by the row as a particular website indicated by the column. The last column and row correspond to the `other`-class. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no website being labeled correctly for less than 37.5 % of the time, which is significantly higher than random guessing at 1 %. The websites with the worst accuracies are `microsoftonline.com` (37.5 %), as direct access to this domain does not resolve to an IP address, `t.co` (50 %), `twimg.com` (50 %) and `jianshu.com` (50 %). All other websites have accuracies of over 50 %.

4) Video-Stream Fingerprinting: In this scenario, the attacker fingerprints video streams using PassiveIdleLeak. The attacker runs on a sibling logical core of the core that receives the network device interrupts.

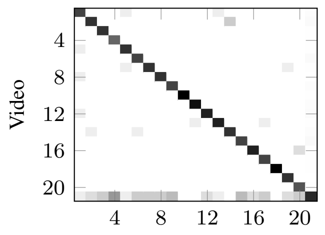


(a) Prediction (on youtube.com)

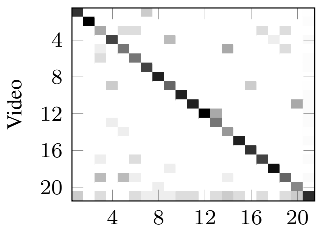


(b) Prediction (on pornhub.com)

Fig. 19. The confusion matrices for our video-stream fingerprinting attack with 20 videos, with network interrupts arriving on a sibling logical core performed on youtube.com and pornhub.com. For the classification, we use a 20 % test split, with the remainder as the training set, for each video.



(a) Prediction (on youtube.com)



(b) Prediction (on pornhub.com)

Fig. 20. The confusion matrices for our open-world video-stream fingerprinting attack with 20 videos and a seperate class for other videos, with network interrupts arriving on a sibling logical core performed on youtube.com and pornhub.com. For the classification, we use a 20 % test split, with the remainder as the training set, for each video.

For YouTube, our classifiers achieved macro-averaged F1 scores of 90.2 % (closed-world) and 81.5 % (open-world) on test sets which are not used for training. The full confusion matrix for the closed-world scenario is shown in Figure 19a. Each cell indicates the probability that our classifier labels streaming a video indicated by the row as a particular video indicated by the column. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no video being labeled correctly for less than 41.4 % of the time, which is significantly higher than random guessing at 5 %. The videos with the worst accuracies are `Video 6` (41.4 %), and `Video 19` (79.3 %). All other videos have accuracies of over 80 %. The full confusion matrix for the open-world scenario is shown in Figure 20a. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no video being labeled correctly for less than 60 % of the time, which is significantly higher than random guessing at 4.8 %. The videos with the worst accuracies are `Video 4` (60 %), and `Video 20` (66.7 %). All other videos have accuracies of over 70 %. The `other`-class in particular has a test accuracy of 83 % on videos never seen during the training phase.

For PornHub, our classifiers achieved macro-averaged F1 score of 75 % (closed-world) and 70.5 % (open-world) on test sets which are not used for training. The full confusion matrix for the closed-world scenario is shown in Figure 19b. Each cell indicates the probability that our classifier labels streaming a video indicated by the row as a particular video indicated by the column. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no video being labeled correctly for less than 33.3 % of the time, which is significantly higher than random guessing at 5 %. The videos with the worst accuracies are `Video 5` (33.3 %), `Video 3` (51.5 %), `Video 19` (54.5 %), and `Video 16` (57.8 %). All other videos have accuracies of over 60 %. The full confusion matrix for the open-world scenario is shown in Figure 20b. The high probabilities along the diagonal indicate that our classifier has a high accuracy with no video being labeled correctly for less than 33.3 % of the time, which is significantly higher than random guessing at 4.8 %. The videos with the worst accuracies are `Video 3` (33.3 %), `Video 14` (40 %), and `Video 20` (46.7 %). All other videos have accuracies of over 53 %. The `other`-class in particular has a test accuracy of 82 % on videos never seen during the training phase.

The F1 scores for PornHub (75 % closed world, 70.5 % open world) are significantly lower than for YouTube (90.2 % closed world, 81.5 % open world), mainly for two reasons: First, Pornhub has a lower default video resolution of 720p (in some cases even 480p) on our system, whereas YouTube has 1080p resulting in fewer network packages within the same time frame. Second, platform- or company-specific intros at the beginning interfere with our fingerprinting. Both issues can be addressed by using more than the first 10 s for classification.

D. Previous Work

Our website fingerprinting attack achieves an F1 score of 93.1 % in a closed-world and 85.2 % in an open-world fingerprinting scenario, over the top 100 websites, which is on par and in most cases even better than previous work. Spreitzer et al. [53] achieved an accuracy of 89 % on 100 websites using the data-usage statistics on Android. Jana et al. [19] exploited the memory usage statistics of browsers and reported an accuracy between 30 % and 50 % for the top 100 000 websites. Gulmezoglu et al. [13] used hardware performance events and achieved accuracy of 86.3 % on 40 websites. Zhang et al. [68] performed website fingerprinting on the top 100 pages of the Alexa Top 1M list by monitoring interrupts using `mwait` and reported an F1 score of 70 % on an Intel CPU. Based on our F1 score of 93.1 %, we can conclude that our attack has a much higher accuracy.

Our video fingerprinting attack achieves an F1 score of 90.2 % in a closed-world fingerprinting scenario over the top 20 videos. Reed et al. [40] exploit the change in throughput of Dynamic Adaptive Streaming over HTTP (DASH) throughout a video to fingerprint videos streamed from Netflix through a wireless network with an accuracy greater than 90 % in less than 5 min. Gu et al. [12] built a bitrate-independent video fingerprinting attack on DASH by monitoring network traffic throughput and matching video fingerprints and achieved an accuracy of 90 % after 3 min on custom videos. Reed et al. [41] use passive network traffic analysis to fingerprint 20 min long Netflix videos transmitted through HTTPS with an accuracy of 99.5 %. Based on our F1 score of 90.2 % on YouTube videos, our results are in line with existing attacks, with significantly shorter measurement times using only package frequency information through network interrupts.

VII. Discussion and Mitigations

Our work shows previously unknown security implications of idle states, in particular on the cross-process and cross-VM confidentiality of highly sensitive privacy-related information. The `tpause` instruction offers a fast and energy-efficient alternative to unprivileged busy waits. While prior

TABLE I. LIST OF PROPOSED MITIGATIONS, THEIR PERFORMANCE IMPACT MEASURED WITH THE STRESS-NG BENCHMARK SUITE, AND IF THE MITIGATION PARTIALLY (◐) OR FULLY (●) MITIGATES OUR ATTACKS.

Mitigation	Perf. Change	Security
Isolated IRQ Core	−4.8 %	◐
IRQ Randomization	~0 %	◐
Disable `WAITPKG` (VMX)	−2 %	◐
Hardware Change	`unknown`	●

work already used `mwait` to detect interrupts [68], our work shows that the underlying root cause is the idle state, revealing a more generic problem, including the performance-related information leakage of C0.2, as shown in Section III-A.

Our work shows that the potential security risks of idle states have previously not been fully understood. Consequently, apart from disabling the instruction set extension, the mitigations discussed by Zhang et al. [68] do not resolve the root cause of the security issue. Moreover, secure scheduling policies, e.g., core scheduling [21], [26], [8], do not protect against our attacks as the core assignment for interrupt handling is independent of these policies for threads, processes, and VMs. In Table I, we provide an overview of possible mitigations, their performance impact measured through the Stress-NG benchmark suite, and their effectiveness in mitigating the attacks proposed in this paper. We propose the following mitigations:

a) Disabling WAITPKG: Currently, there is no option to disable only the `WAITPKG` instruction set extension completely. It is possible to disable `WAITPKG` for the user space, but only together with all other `TSC` related instructions, including `rdtsc` by setting the `TSD` bit in the `CR4` register [17]. This is not feasible as numerous user space applications rely on the availability of the `rdtsc` instruction. It is only possible to disable the C0.2 through the `IA32_UMWAIT_CONTROL` MSR forcing `umwait` and `tpause` to fall back to C0.1 if C0.2 is requested [17], which does not mitigate the security issues present in C0.1. Contrary to the native case, it is possible to disable `WAITPKG` for virtual machines. While most user space applications do not use `umwait` or `tpause`, as they are new instructions, the Linux kernel uses `tpause` with C0.2 for short delays and falls back to a busy wait if the instruction is not supported. Disabling the instruction set extension for security would mean sacrificing its potentially substantial performance gains, as shown in Section II-C, which is not a practical solution for most systems. For disabling `WAITPKG` in virtual machines, we measured a performance degradation of 2 % on our i7-1260P with the Stress-NG benchmark.

b) Isolated IRQ Core: To avoid a possible detection of external interrupts, the operating system can isolate at least one physical core for handling external interrupts. This interrupt isolation would make it no longer possible to detect external interrupts through `umwait` or `tpause` with C0.2 on the same core or with C0.1 on the sibling logical core, as the attacker can not be scheduled on the physical core that receives the interrupts. While this mitigates attacks targeting external interrupts, the undocumented wake-ups of C0.1 on exceptions and in/out-port instructions of sibling logical cores are still exploitable. Furthermore, isolating physical cores for interrupt handling comes with a significant performance impact, especially on multithreaded workloads. Specifically for larger server systems, multiple cores are needed to handle the interrupt load fully. On our test system with an i7-1260P, the performance in the Stress-NG benchmark decreased by 4.8 % when dedicating one physical core for interrupt handling.

c) IRQ Randomization: By regularly randomizing the core assignments of external interrupts in short periods of time, the operating system can introduce a significant amount of noise to fingerprinting attacks. A higher reassignment frequency causes more noise for the attacker but also a higher performance and energy overhead. While this mitigation makes attacks monitoring external interrupts significantly more challenging, it does not entirely mitigate them, as an attacker can still monitor interrupts. Furthermore, exception monitoring by the attacker through the C0.1 idle state is still possible. On our i7-1260P, we did not observe a significant performance overhead in the Stress-NG benchmark with random interrupt affinity reassignment every 0.5 s.

d) Hardware Changes: Completely removing wake-up on interrupts is not viable as software, such as the Linux kernel, relies on this functionality. We propose removing the undocumented wake-ups of C0.1 on exceptions and external interrupts of sibling logical cores. Furthermore, to completely mitigate interrupt monitoring from VMs, a solution would be to adopt the behavior of the `hlt` instruction, which does not wake up in case of a VM exit. These changes would completely resolve the interrupt-related security issue of `umwait` and `tpause` for the scenario of a virtual-machine-based attacker and limit the attack surface in a native scenario.

e) Other Potential Mitigations: We conclude that this issue requires a hardware mitigation, as in software we can only fully disable `umwait` and `tpause` for virtual machines and other mitigation techniques are not sufficient and impose a possible negative performance impact. Removing wake-up reasons may be a viable approach for the cases where the sibling logical core is woken up. However, e.g., in our attack in Section VI-C1, we exploited interrupt handling on the same core, which is a wake-up reason that cannot be eliminated. The older privileged `mwait` instruction, which is similar to the unprivileged `umwait`, allows the CPU to switch into deeper sleep states, and wakes up when a VM exit occurs [18]. It is, therefore, reasonable to assume that this is also the intended behavior for `umwait` and `tpause`. However, currently, this behavior is not documented in the Intel instruction manual and can be the source of security issues regarding virtual-machine-based software isolation.

Besides the interrupt-related issues of idle states, we also show how the performance gain results in security problems. As we can expect more applications start to incorporate `umwait` and `tpause` in the near future, the attack surface will further increase where attackers can also infer, e.g., control-flow information on other applications.

Generic solutions include trapping idle-state instructions in virtual machines, and injecting fake keystrokes for noise against our inter-keystroke timing attacks [48]. However, these approaches come with performance and energy costs that might not be justified compared to just removing the idle state control. Moreover, for other attacks, e.g., our covert channel and the website and video fingerprinting, noise reduces the performance but does not fully mitigate the leakage.

Finally, prior work discussed the detection of side channels using performance counters [36]. However, as Zhang et al. [68] already noted, there are no performance counters tracking the use of idle-state controlling instructions so far. In contrast to many prior microarchitectural attacks, IdleLeak does not induce a negative performance impact in the victim, that could be detected by the victim. Instead, IdleLeak rather improves the performance of the victim workload. However, the victim cannot detect this irregularity as malicious behavior, as this legitimately happens, with identical idle state choices, when the corresponding processor core is legitimately idling.

In conclusion, our work highlights the necessity for future work on effective mitigation of idle-state side channels.

VIII. Related and Future Work

a) Interrupt Detection & Keystroke Attacks: Interrupt detection has been used in several previous works. Ristenpart et al. [42] used interrupt detection to synchronize attacker and victim. Schwarz et al. [48] presented an interrupt-detection-based attack in native code using high-resolution timers such as the x86 instruction `rdtsc`. They observed that `rdtsc` values have larger jumps when an interrupt occurs, as the attacker is not scheduled in this time frame. They further proposed a countermeasure to keystroke timing attacks which injects uniform high-frequency stream of fake keyboard interrupts. Lipp et al. [27] demonstrated a keystroke timing attack from JavaScript using a counting thread instead of a high-resolution timer. Prior to these works, keystroke timing attacks have been performed based on cache attacks [42], [11], smart phone sensors [5], and remote timing measurements [52].

Closely related to IdleLeak is the recent work by Zhang et al. [68], exploring the `umonitor` and `umwait` for side-channel attacks, primarily to translate microarchitectural states into architectural states in transient-execution attacks. They evaluate their side channel, continuously executing `umonitor` and `umwait`, in interrupt detection scenarios counting the number of wake-ups in fixed time periods and making deductions on system activity, similar to prior works. Clearly, interrupt detection attacks are not new but they are important for comparability with prior work. Thus, we follow their best practice and also evaluate our idle-state side channel in interrupt detection scenarios for comparability. Furthermore, we demonstrate the *first* video fingerprinting attack based on interrupt detection with the IdleLeak side channel. This attack illustrates the severe and previously unknown privacy implications of the novel idle states that IdleLeak uncovered: Information about sensitive online video consumption may be used for instance for extortion campaigns [61].

One further difference to Zhang et al. [68] is our attack technique: Zhang et al. [68] use a documented feature of an instruction to wake up on interrupts and can only be used in a native non-VM scenario. While they focused on the behavior of these instructions, they did not explore the behavior of the idle states nor the problem around the interrupt core affinity. In contrast, we discovered the undocumented effect that the `tpause` instruction wakes up upon several events, including interrupts but also other system-level events (hardware exceptions, e.g., page faults, divide by zero, VM exits, inport and outport operations), which is orthogonal to the findings of Zhang et al. [68] compared to prior works [42], [27], [48]. Our findings are surprising as the spurious wake-up behavior of `tpause` on VM exits due to, e.g., host interrupts is inconsistent with other sleep instructions such as `hlt`, which does not wake up on VM exits. While not the main focus of their work, they also briefly evaluated their approach in a website-fingerprinting scenario. In contrast to their work, we focused on the broader security implications of idle states in general and demonstrated different attack scenarios (same core, sibling core) and new attacks (e.g., the first interrupt-detection-based video fingerprinting). We investigated the performance-enhancing effects and discovered further wake-up causes, including interrupts of unrelated workloads even in other virtual machines and the host, and exceptions and interrupts of sibling logical cores. Therefore, with our discoveries, the mitigations proposed by Zhang et al. [68] are not sufficient anymore and future work needs to find mitigations that are effective but maintain an acceptable efficiency.

IX. Conclusion

The new idle states, C0.1 and C0.2, introduce novel leakage that can be used to monitor system activity, in particular interrupts, in the case of C0.1 even interrupts arriving on logical sibling cores. Since interrupts on x86 are scheduled regardless of the corresponding workload, the attacker can spy on victims running on separate physical cores, by focusing on interrupt activity instead. We evaluated both our techniques ActiveIdleLeak and PassiveIdleLeak with covert channels, achieving true capacities of $7.1\,\text{Mbit/s}$ ($\sigma_{\bar{x}} = 0.004\,\text{Mbit/s}$, $n = 512$) and $656.37\,\text{kbit/s}$ ($\sigma_{\bar{x}} = 0.63\,\text{kbit/s}$, $n = 1\,024$) in native code. In a cross-VM scenario, we still achieves $46.3\,\text{kbit/s}$ with ActiveIdleLeak. We demonstrated native keystroke-timing attacks, website- and video-fingerprinting attacks, all with high F-Scores, and a low standard error on the timing. The highly sensitive information that an attacker can acquire through these attacks, potentially exposing even sexual preferences to an attacker, can be used in different ways such as extortion campaigns. While mitigations against IdleLeak may be expensive due to the way interrupts are implemented on x86, we conclude that further research on efficient and effective mitigations is necessary to thwart the exploitation of this side channel.

Acknowledgments

We thank the anonymous reviewers and our anonymous shepherd, for their guidance, comments, and suggestions. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF project NeRAM I-6054-N and FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided by generous gifts from AWS, Google, Intel and Red Hat. 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] Aldaya, A.C., Brumley, B.B., ul Hassan, S., García, C.P., Tuveri, N.: Port Contention for Fun and Profit. In: S&P (2019)

[2] Alexa Internet, Inc.: The top 1 million sites on the web (5 2023), https://www.alexa.com/topsites

[3] Bernstein, D.J.: Cache-Timing Attacks on AES (2005), http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[4] Bityutskiy, A.: Sapphire Rapids C0.x idle states support (2023), https://lwn.net/Articles/925863/

[5] Cai, L., Chen, H.: TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In: USENIX HotSec (2011)

[6] Chen, Z., Xu, Y.Q., Wang, H., Guo, D.: Deep STFT-CNN for spectrum sensing in cognitive radio. IEEE Communications Letters (2020)

[7] Evtyushkin, D., Ponomarev, D.: Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In: CCS (2016)

[8] Faggioli, D.: Re: [RFC PATCH v3 00/16] Core scheduling v3 (2019), https://lore.kernel.org/lkml/277737d6034b3da072d3b0b808d2fa6e110038b0.camel@suse.com/

[9] Gast, S., Juffinger, J., Schwarzl, M., Saileshwar, G., Kogler, A., Franza, S., Köstl, M., Gruss, D.: SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In: S&P (2023)

[10] Gras, B., Razavi, K., Bos, H., Giuffrida, C.: Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In: USENIX Security (2018)

[11] Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security (2015)

[12] Gu, J., Wang, J., Yu, Z., Shen, K.: Walls have ears: Traffic-based side-channel attack in video streaming. In: INFOCOM (2018)

[13] Gulmezoglu, B., Zankl, A., Eisenbarth, T., Sunar, B.: Perfweb: How to violate web privacy with hardware performance events. In: ESORICS (2017)

[14] Huang, J., Chen, B., Yao, B., He, W.: ECG arrhythmia classification using STFT-based spectrogram and convolutional neural network. IEEE access (2019)

[15] Intel: Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide (2019)

[16] Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual (2023)

[17] Intel: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z (2023)

[18] Intel: Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide (2023)

[19] Jana, S., Shmatikov, V.: Memento: Learning Secrets from Process Footprints. In: S&P (2012)

[20] Killourhy, K.S., Maxion, R.A.: Comparing anomaly-detection algorithms for keystroke dynamics. In: IEEE/IFIP International Conference on Dependable Systems & Networks. IEEE (2009)

[21] Kim, T., Peinado, M., Mainar-Ruiz, G.: StealthMem: system-level protection against cache-based side channel attacks in the cloud. In: USENIX Security (2012)

[22] Kocher, P.: Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO (1996)

[23] Kocher, P., Jaffe, J., Jun, B., Rohatgi, P.: Introduction to Differential Power Analysis. Journal of Cryptographic Engineering (2011)

[24] Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: CRYPTO (1999)

[25] Lerner, A., Bonnet, P.: Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In: International Conference on Management of Data (2021)

[26] Linux Kernel Documentation: Core Scheduling (2022), https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/core-scheduling.html

[27] Lipp, M., Gruss, D., Schwarz, M., Bidner, D., Maurice, C.m.t.n., Mangard, S.: Practical Keystroke Timing Attacks in Sandboxed JavaScript. In: ESORICS (2017)

[28] Lipp, M., Hadžić, V., Schwarz, M., Perais, A., Maurice, C., Gruss, D.: Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In: AsiaCCS (2020)

[29] Loy, C.C.: Keystroke100 Dataset (2021), http://personal.ie.cuhk.edu.hk/~ccloy/downloads_keystroke100.html

[30] Maurice, C., Neumann, C., Heen, O., Francillon, A.: C5: Cross-Cores Cache Covert Channel. In: DIMVA (2015)

[31] Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., Römer, K.: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS (2017)

[32] Monaco, J.: SoK: Keylogging Side Channels. In: S&P (2018)

[33] Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS (2015)

[34] Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA (2006)

[35] Page, D.: Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Cryptology ePrint Archive, Report 2002/169 (2002)

[36] Payer, M.: HexPADS: a platform to detect "stealth" attacks. In: ESSoS (2016)

[37] Percival, C.: Cache Missing for Fun and Profit. In: BSDCan (2005)

[38] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security (2016)

[39] Quisquater, J.J., Samyde, D.: ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In: E-smart (2001)

[40] Reed, A., Klimkowski, B.: Leaky streams: Identifying variable bitrate dash videos streamed over encrypted 802.11 n connections. In: CCNC (2016)

[41] Reed, A., Kranch, M.: Identifying https-protected netflix videos in real-time. In: CODASPY (2017)

[42] Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS (2009)

[43] Rokicki, T., Maurice, C., Botvinnik, M., Oren, Y.: Port Contention Goes Portable: Port Contention Side Channels in Web Browsers. In: AsiaCCS (2022)

[44] Rushanan, M., Russel, D., Rubin, A.D.: MalloryWorker: Stealthy Computation and Covert Channels Using Web Workers. In: International Workshop on Security and Trust Management (2016)

[45] Saileshwar, G., Fletcher, C.W., Qureshi, M.: Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In: ASPLOS (2021)

[46] van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue In-flight Data Load. In: S&P (2019)

[47] Schwarz, M., Gruss, D., Weiser, S., Maurice, C., Mangard, S.: Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA (2017)

[48] Schwarz, M., Lipp, M., Gruss, D., Weiser, S., Maurice, C., Spreitzer, R., Mangard, S.: KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS (2018)

[49] Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS (2019)

[50] Schwarzl, M., Kraft, E., Gruss, D.: Layered Binary Templating. In: ACNS (2023)

[51] Simon, L., Xu, W., Anderson, R.: Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards. PETS (2016)

[52] Song, D.X., Wagner, D., Tian, X.: Timing Analysis of Keystrokes and Timing Attacks on SSH. In: USENIX Security (2001)

[53] Spreitzer, R., Griesmayr, S., Korak, T., Mangard, S.: Exploiting data-usage statistics for website fingerprinting attacks on Android. In: ACM Conference on Security & Privacy in Wireless and Mobile Networks (2016)

[54] Taram, M., Ren, X., Venkat, A., Tullsen, D.: SecSMT: Securing SMT Processors against Contention-Based Covert Channels. In: USENIX Security (8 2022)

[55] UEFI Forum, Inc.: Advanced Configuration and Power Interface (ACPI) Specification Release 6.5 (2022)

[56] Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., Strackx, R.: Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In: USENIX Security (2017)

[57] Vila, P., Köpf, B.: Loophole: Timing Attacks on Shared Event Loops in Chrome. In: USENIX Security (2017)

[58] Walton, S.: How Screwed is Intel without Hyper-Threading? (2019), https://www.techspot.com/article/1850-how-screwed-is-intel-no-hyper-threading/

[59] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., Gunter, C.A.: Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In: CCS (2017)

[60] Wang, Z., Lee, R.B.: Covert and Side Channels due to Processor Architecture. In: ACSAC (2006)

[61] Winder, D.: Has A 'Hacker' With Your Password Really Recorded You Watching Porn? (2022), https://www.forbes.com/sites/daveywinder/2022/11/28/has-a-hacker-with-your-password-really-recorded-you-watching-porn/

[62] Wu, Z., Xu, Z., Wang, H.: Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. ACM Transactions on Networking (2014)

[63] Xu, Y., Cui, W., Peinado, M.: Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P (2015)

[64] Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., Schlichting, R.: An exploration of L2 cache covert channels in virtualized environments. In: CCSW (2011)

[65] Yao, S., Piao, A., Jiang, W., Zhao, Y., Shao, H., Liu, S., Liu, D., Li, J., Wang, T., Hu, S., et al.: Stfnets: Learning sensing signals from the time-frequency perspective with short-time fourier neural networks. In: The World Wide Web Conference (2019)

[66] Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security (2014)

[67] Zhang, K., Wang, X.: Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In: USENIX Security (2009)

[68] Zhang, R., Kim, T., Weber, D., Schwarz, M.: (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In: USENIX Security (2023)