2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

SegScope: Probing Fine-grained Interrupts via Architectural Footprints

Xin Zhang*[12], Zhi Zhang*[3], Qingni Shen†[12], Wenhao Wang[4], Yansong Gao[5], Zhuoxi Yang[12], Jiliang Zhang[6]
* Both authors contributed equally to this work
[1] School of Software and Microelectronics, Peking University
[2] PKU-OCTA Laboratory for Blockchain and Privacy Computing, Peking University
[3] The University of Western Australia [4] Institute of Information Engineering, CAS [5] Data61, CSIRO
[6] College of Semiconductors (College of Integrated Circuits), Hunan University
{zhangxin00, yangzx}@stu.pku.edu.cn, zhi.zhang@uwa.edu.au, qingnishen@ss.pku.edu.cn,
wangwenhao@iie.ac.cn, gao.yansong@hotmail.com, zhangjiliang@hnu.edu.cn

***Abstract*—Interrupts are critical hardware resources for OS kernels to schedule processes. As they are related to system activities, interrupts can be used to mount various side-channel attacks (i.e., monitoring keystrokes, inferring website visits, detecting GPU activities, and fingerprinting processes). Given that all these attacks rely on system file interfaces or architectural timers to probe interrupts, various countermeasures have been proposed to either remove the unprivileged access to the file interfaces or detect/cripple architectural timers.**

In this work, we propose SegScope, a new technique that abuses segment protection to provision *fine-grained* interrupt observations *without any timer*. As segment protection is widely used on x86, SegScope works across a wide range of Intel- and AMD-based CPUs. Particularly, we observe that while segment protection preserves the confidentiality of high privileged domain, it leaves a footprint via the data segment registers values when an interrupt occurs. With this key observation, SegScope is crafted by capturing the footprints. To show its security implications, we evaluate it in four case studies. *First*, SegScope has inferred website visits with a respective success rate of 92.4% on Chrome and 87.4% on Tor Browser in default system settings. *Second*, SegScope successfully extracts the keys from Cloudflare's Interoperable Reusable Cryptographic Library (CIRCL) v1.1. *Third*, SegScope steals DNN model architectures with an accuracy of over 80%. Last, SegScope effectively reduces the noise of interrupts to improve the performance of other side channels. As an example, SegScope reduces the error rate of Spectral side channel by 56×. Compared with existing timer-based interrupt-probing techniques, SegScope is fine-grained without introducing false-positives. Further, we leverage SegScope to craft a fine-grained timer, as regular timer interrupts as clock edges contain timestamps. Our evaluation shows that it achieves the same level of timing granularity as the high-resolution timer, i.e., `rdtsc` and `rdpru`. We then leverage the timer to break KASLR in about 10 seconds and mount a Flush+Reload based Spectre attack.

I. Introduction

Interrupts are critical to modern OSes, particularly for process scheduler [1], [22], [63]. An interrupt can switch current context into kernel, enabling the process scheduler to preempt a running process and perform essential tasks. Since interrupts are typically activated when a process requests system resources (e.g., network, system calls, etc) or when a user inputs data through devices, they have been exploited as a side channel, to monitor keystrokes [31], [51], [58], infer website visits [9], [77], detect GPU activities [39], [40], and perform process fingerprinting [12], [39], [40], [57].

These interrupt-based side channel attacks rely on either unprivileged *procfs* interface or architectural timers to probe interrupts. Specifically, interrupt statistics can be acquired by accessing */proc/interrupts* in Linux [12], [40], [57]. However, such attacks can be easily defeated by removing non-privileged access to the *procfs* interface [9], [75]. Besides, the query of the *procfs* interface is badly affected by context switches, typically with a sampling interval at the millisecond level, resulting in coarse-grained interrupt probing.

The other works [9], [31], [51], [58], [77] use architectural timers to probe interrupts. Considering that a jump in timestamp will occur if a process is interrupted, a high/low-resolution timer can be leveraged to observe the jump [9], [31], [51], [58]. Recently, Zhang et al. [77] have identified a pair of unprivileged instructions (i.e., `umonitor` and `umwait`) that can be used to probe interrupts. These instructions are only available to recent Intel microarchitectures since Tremont and Alder Lake. Particularly, a user process enters into a light-weight sleep state via the pair of instructions. When an interrupt occurs, the process will be awakened earlier than timeout. They observe such timing differences via an architectural timer.

To constrain the use of architectural timers, a number of countermeasures [23], [32], [33], [42], [45], [49], [62] have been proposed. Generally, these countermeasures either detect timers or cripple them. For example, MASCAT [23] and Oyama et al. [45] are static binary analysis tools, which can detect the use of timer-related instructions such as `rdtsc`. ARM-based CPUs (e.g., Apple M1) have removed the unprivileged fine-grained timers [25], [32], [49], [72]. For both Intel- and AMD-based CPUs, the defender can set the CR4.TSD bit to prevent unprivileged usage of the high-resolution timers (e.g., `rdtsc`, `rdtscp`, and `rdpru`) [1], [16], [22], [30], [77]. Last, AMD-based CPUs have reduced the resolution of timestamp counters since its Zen architecture [33].

† **Corresponding Author: Qingni Shen.** This work was supported by the National Natural Science Foundation of China (Grant No. 61672062).

In such a timer-constrained scenario, existing works [14], [32], [53], [55], [68] show that the attackers can build their own timers. Unfortunately, these timers are typically much noisier than architectural timers [15], [61], making it hard to use them to probe interrupts. To this end, we ask the following questions:

Is there a microarchitectural technique across x86 CPUs probing interrupts without any timers? If yes, what attacks can be mounted and what information can be leaked?

In this paper, we present SegScope, a new technique that abuses segment protection on x86 to acquire fine-grained interrupt observations without relying on any timer. Specifically, when an interrupt occurs and returns from kernel-space to user-space, CPUs with segment protection will check data segment registers (e.g., `DS`, `ES`, `FS`, and `GS`) and clear them if they contain kernel information or point to a null segment descriptor. We found that some non-zero selector values are considered as null segment selector. With this key observation, we assign a data segment register with a non-zero null value and this value will be reset to 0 when an interrupt switches context from kernel-space to user-space. By checking a pre-assigned data segment register, we can decide whether a current process has been interrupted. In a nutshell, while segment protection preserves the confidentiality of high privileged domain, it leaves a footprint via the value change in data segment registers when an interrupt occurs. With this footprint, SegScope is crafted without accessing the aforementioned pseudo-file interface or timers.

With fine-grained interrupt observations, SegScope is used as a side channel to infer website visits, similar to [9], [77]. Our experimental results show that SegScope fingerprints websites with a respective success rate of 92.4% on Chrome and 87.4% on Tor Browser in the default system setting. We further explore new security implications of SegScope via three more case studies. *First*, the attacker can use SegScope rather than architectural timers to construct a loop-counting program to monitor CPU frequency, which is related to the workload of CPU. We successfully extract the keys from Cloudflare's Interoperable Reusable Cryptographic Library (CIRCL) v1.1 [17] and steal DNN model architectures with an accuracy of over 80%. *Second*, because SegScope can distinguish the interrupted measurements, it can be used to effectively reduce the noise of interrupts to enhance other side channels which are noised by interrupt activities [77]. Taking the Spectral side channel as an example [77], SegScope has removed the impacts of interrupts and reduced its error rate by 56$\times$.

While existing timer-based interrupt probing techniques are unavailable in timer-constrained scenarios, two representatives (i.e., high-resolution timer based probing [51], [58] and loop-counting based probing [9], [31]) are compared against SegScope, results of which show that SegScope is fine-grained in probing each interrupt without any false positive involved.

Besides, we leverage SegScope to craft a fine-grained timer, as timer interrupts can be used as clock edges to build a clock interpolation scheme [53]. With an observation that different types of interrupts present distinguishable statistical characteristics, we use statistical methods (e.g., Z-score) to retain timer interrupts and filter out other interrupts.

To demonstrate the viability of the crafted timer, we perform a comprehensive evaluation on multiple Ubuntu OSes with Intel or AMD-based CPUs in either local or Amazon cloud environments. First, we evaluate its timing granularity and compare it against the high-resolution timers (i.e., `rdtsc` on Intel-based CPUs and `rdpru` on AMD-based CPUs [30]), results of which show that the SegScope-based timer achieves the same level of timing granularity as them. We then leverage this timer to break KASLR within about 10 seconds and perform a Flush+Reload [71] based Spectre attack.

Summary of contributions: The main contributions of this paper are as follows:

• We propose a technique, called SegScope, that abuses segment protection to precisely probe interrupts. To the best of our knowledge, this is the first effective probing technique without any timer. Since segment protection is supported on x86 by default, SegScope affects mainstream x86-based CPUs.

• The security implications of SegScope are demonstrated via 4 case studies. As a side channel, SegScope has been used to infer website visits, extract cryptographic keys, and steal DNN model architectures. Further, SegScope has significantly reduced the noise of interrupts to enhance other non-interrupt side channels.

• We further rely on SegScope to probe timer interrupts and thus craft a fine-grained timer. The crafted timer has been evaluated on multiple machines with Intel- or AMD-based CPUs in either local or Amazon cloud environments. The experimental results show that our timer has the same order of magnitude as `rdtsc` and `rdpru` in terms of timing granularity. Besides, it has been used to break KASLR within about 10 seconds and perform a Flush+Reload based Spectre attack.

Responsible disclosure: We have responsibly disclosed our findings, with proof-of-concept code[1], to Intel and AMD. Both vendors acknowledged our side-channel attacks.

II. Background and Related Work

A. Timer-based Interrupt Probing Techniques

The majority of existing interrupt side channel attacks [9], [31], [51], [58] rely on an architectural timer to probe interrupts, which are not applicable in timer-constrained scenarios. Specifically, some attacks [51], [58] use a high-resolution architectural timer to observe timestamp jumps. If a given process is interrupted, there will be a jump in its resulted timestamp, leaking interrupt information. The leaked interrupts can be induced by keystrokes, which indicate target victim's input. To this end, the attacks can infer target victim's password or secrets by correlating the interrupt information with keystrokes.128ge Alternatively, other works [9], [31] apply a

[1]The source code is released at https://github.com/zhangxin00/segscope.

low-resolution architectural timer to probe interrupts. Particularly, they build a loop counting program, inside which a low-resolution timer is invoked to sample a self-increment counter at fixed time intervals. If an interrupt occurs in the interval, the counter value will be lower, resulting in an interrupt detection. With such a loop-counting program, keystrokes are monitored [51] and websites are fingerprinted [9]. We compare SegScope with these probing techniques in Section III-B.

Besides, Zhang et al. [77] exploit two unprivileged instructions (i.e., `umonitor` and `umwait` that are only available to recent Intel microarchitectures) to probe interrupts. Particularly, a user process enters a light-weight sleep state via the two special instructions. When an interrupt comes, the crafted process is awakened earlier than timeout, resulting in more code executions in fixed time intervals. The timer interval is also fixed by a low-resolution architectural timer. With this interrupt probing technique, website visits can be inferred with an accuracy of 78%.

KeyDrown [51] probes keystroke-related interrupts by observing CPU cache state changes. When a keystroke event occurs, the keyboard interrupt handler will be executed and cached into its corresponding CPU cache sets. Thus, KeyDrown applies `Prime+Probe` [36] to monitor cache state changes in targeted CPU cache sets. Generally, `Prime+Probe` leverages an architectural timer to time access to targeted cache sets. To find targeted cache sets, they also need physical addresses of the keyboard interrupt handler, which is not feasible to an unprivileged attacker.

B. Segment Protection in x86

The segment protection is designed to improve fault tolerance and system security [1], [3], [22] , ensuring that a user process cannot access the kernel address outside controlled and well-defined interfaces. To achieve this, modern operating systems provide multiple privilege levels to access resources, which assign a value from 0 to 3 to key objects recognized by processors. The highest privilege level corresponds to 0.

To prevent a low-privilege program from accessing segments for a high-privilege data segment, each segment has a 2-bit *Descriptor Privilege Level* (DPL), indicating the minimum privilege required to access the segment. Correspondingly, each CPU core has a 2-bit *Current Privilege Level* (CPL) which reflects the privilege level of a currently running program. Furthermore, to temporarily lower the privilege level of a user process, each CPU core also stores a 2-bit *Requested Privilege Level* (RPL), which can switch privilege level flexibly. When currently running code attempts to access a data segment, its relevant CPL and RPL are checked against the DPL of the segment. Fig. 1 shows how the CPL and RPL are used to determine whether an access to a data segment is granted. Only when the CPL and RPL are both smaller than or equal to the DPL, can the corresponding data segment be accessed.

The x86 architecture defines segment registers [1], [22], including `CS`, `DS`, `ES`, `SS`, `FS`, and `GS` for memory segmentation, dividing main memory into segments or sections. Each

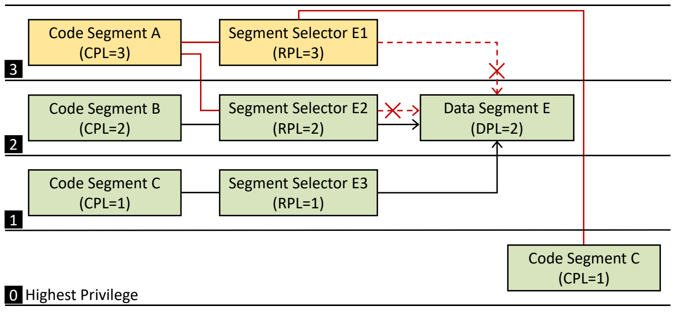


Fig. 1. Examples of accessing data segments from different privilege levels. Red arrows consists of two parts: a solid line which marks the control flow without segment protection check and a dashed line which marks the control flow that cannot be executed.

segment register consists of a visible part and a hidden part (also called shadow register or descriptor cache). The visible part is to store the logical base of a segment (called segment selector), and the hidden part is to store a segment descriptor which stores protection parameters (e.g., segment base and privilege levels) and cannot be read by programs.

When a program accesses a segment, its corresponding logical address is referenced as follows. First, as each address has a 16-bit segment selector and a 32-bit segment offset, the segment selector is loaded into the visible part of a relevant segment register. Second, the segment register is used to look up a corresponding segment descriptor for retrieving the protection parameters. Last, the aforementioned segment protection check (see Fig. 1) is performed. If the check succeeds, the logical address will be translated into its linear address. To facilitate subsequent accesses, the segment descriptor is cached into the hidden part of the segment register.

C. Timer Interrupts

In modern operating systems, timer interrupts are triggered by hardware, allowing OS to activate its kernel at a fixed time interval. In the x86 architecture, each CPU core is with an Advanced Programmable Interrupt Controller (APIC) that manages the timer interrupts. The APIC hardware allows user to program the number of timer interrupts generated per second (HZ), which commonly ranges between 100 and 1000 [50]. Thus the time intervals between every two timer interrupts are commonly at the millisecond level.

Existing timer interrupt based works assume a privileged user who controls the frequency of timer interrupts, in either attack or defense scenarios. Specifically, the majority of previous works [4], [11], [44], [52], [59], [60], [76] assume a privileged attacker that utilizes timer interrupts to periodically pause enclave programs [10], enabling precise control of the execution flow of a victim enclave. As such, they can carry out various side channel attacks to compromise the confidentiality of SGX enclave. On the defense side, Schwarz et al. [51] control the activation of timer interrupts to inject fake keystrokes, defeating side channel attacks of monitoring keystrokes.

III. Overview

In this section, we propose a new interrupt probing technique without relying on any timer. Built upon this technique, we further present a fine-grained timer. First, we present the threat model and assumptions. Second, we show that the segment protection can be used to probe fine-grained interrupts. Last, we craft a fine-grained timer via precisely probing timer interrupts.

Experimental setup: We evaluate SegScope on multiple machines as shown in Table I, including physical machines and cloud instances with different CPU models. Unless otherwise stated, we use the default system configuration.

TABLE I
System Configurations.

Machine	CPU	Kernel	OS	HZ
Xiaomi Air 13.3	Intel Core i5-8250U	5.15.0	Ubuntu 20.04.5	250
Lenovo Yangtian 4900v	Intel Core i7-4790	5.8.0	Ubuntu 20.04.1	250
Lenovo Savior Y9000P	Intel Core i9-12900H	5.15.0	Ubuntu 20.04.1	250
Honor Magicbook 16 Pro	AMD Ryzen 7 5800H	5.15.0	Ubuntu 20.04.1	250
Amazon t2.large (Xen)	Intel Xeon E5-2686	5.15.0	Ubuntu 22.04.1	250
Amazon c5.large (KVM)	Intel Xeon 8275CL	5.15.0	Ubuntu 22.04.1	250

A. Threat Model and Assumptions

In our threat model, we assume an unprivileged attacker who controls a user process with no special privileges. Thus, the attacker cannot make any modifications to a victim x86-based system such as disabling Dynamic Voltage and Frequency Scaling (DVFS), and Simultaneous Multithreading (SMT). We assume the system runs in either virtualized or bare-metal environment where architectural timers are constrained.

To fingerprint websites, a victim user is assumed to use popular browsers such as Chrome and Tor. To extract cryptographic keys, we assume that an unprivileged attacker can manipulate the input of the victim cryptographic model. To steal DNN model architectures, an attacker process and a victim model inference run in the separate CPU cores concurrently. To enhance Spectral, a victim program contains one or more Spectre gadgets. To break KASLR, the victim system has neither bugs nor vulnerabilities that allow the attacker to obtain a mapped kernel address.

B. Probing Fine-grained Interrupts

On x86, a global descriptor table (GDT) and a local descriptor table (LDT) are used to store segment descriptors. To detect access to unused segment registers, the first descriptor in `GDT` is reserved by CPUs. When a segment selector to the first entry (i.e., entry 0) of `GDT` (called *null segment selector*) is loaded into a data segment register (i.e., `DS`, `ES`, `FS`, and `GS`), it does not generate any exception. However, if memory access is performed using the register, it will cause a general-protection exception. Thus, by initializing data segment registers with *null segment selector*, accidental reference to unused segment registers guarantees an exception.

To prevent a low-privileged program from accessing high-privileged data segments, when returning to the outer-privilege level, x86 CPUs will check the segment registers and clear them if they contain high-privileged information. Function `Protected_Mode_Return` in Algorithm 1 defines how x86 CPUs perform the check. When CPUs return to outer privilege level, `CS.RPL` is greater than `CPL` (Line 5). For each of the 4 registers, its descriptor cache is assigned to `Des` (Lines 6-8). Lines 9-12 check if a segment selector satisfies one of two conditions. If either condition is satisfied, the selector will be reset to 0. In the first condition, CPUs check if the selector is a *null segment selector* (Line 9), which avoids the segment fault caused by reference to an unused segment register. If the first check fails, the register is in use. Thus in the second condition, CPUs access its descriptor cache, retrieve the protection parameters and check whether it points to a high-privileged segment (Line 10). If the first check succeeds, the second condition check is skipped and the null segment selector will be cleared (Line 11).

Algorithm 1: Segment protection check by x86 CPUs

```
1  Initially: $CS.RPL$ is the RPL of the $CS$ register, which
     represents the privilege level to return.
2  $CPL$ is the current privilege level.
3  Function $Protected\_Mode\_Return$
4      // If return to outer privilege level.
5      if $CS.RPL > CPL$ then
6          foreach $Reg \in (DS, ES, FS$ and $GS)$ do
7              // check the descriptor cache of $Reg$.
8              $Des \leftarrow Reg.Descriptor$
9              if $Reg.Selector == NULL$ or
10             $(Des.DPL < CPL$ and $Des.Type == Sensitive)$
                 then
11                 $Reg.selector \leftarrow 0$.
12             end
13         end
14     end
15     return
```

However, the null segment selector is not always 0. On x86, segment selector (Sel) consists of a 2-bit requested privilege level (RPL), 1-bit table indicator (TI) and 13-bit index. The index selects one of 8192 descriptors in a descriptor table (i.e., `GDT` or `LDT`). TI is a flag that specifies the descriptor table to use. If TI equals 0, the selector will use `GDT`. The RPL specifies an override privilege level of the selector (as discussed in Section II-B). For a given selector, if its value is 0x0000, 0x0001, 0x0002, or 0x0003, it points to the first entry of GDT, and its value is considered to be null. Thus, for non-zero segment selector values, they will be cleared when context switches from kernel space to user space, leaving an architectural footprint of an interrupt's occurrence.

We have tested all data segment registers (i.e., `FS`, `DS`, `ES` and `GS`) on the tested machine listed in Table I, results of which show that our SegScope crashes only when modifying `FS`, as `FS` is referenced by some programs such as glibc-based TLS, causing segment fault. As `GS` is available since 80386, it is less used by than `DS` and `ES`. In this paper, we pick `GS`.

We leverage the above mechanism to develop SegScope, a novel technique that can probe fine-grained interrupts without

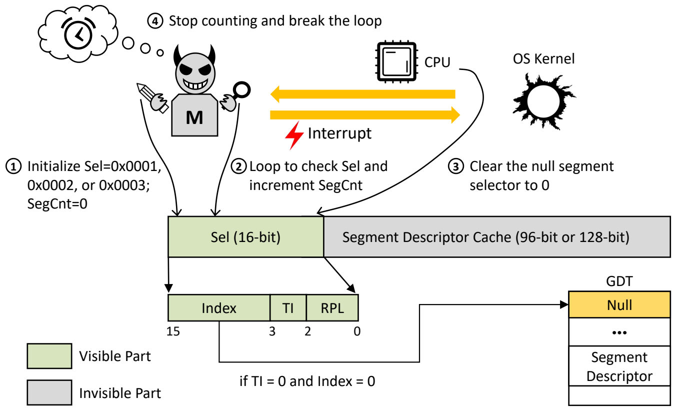

④ Stop counting and break the loop
CPU
OS Kernel
M
Interrupt
① Initialize Sel=0x0001, 0x0002, or 0x0003; SegCnt=0
② Loop to check Sel and increment SegCnt
③ Clear the null segment selector to 0
Sel (16-bit)
Segment Descriptor Cache (96-bit or 128-bit)
Index
TI
RPL
15
3
2
0
GDT
Null
•••
Segment Descriptor
Visible Part
Invisible Part
if TI = 0 and Index = 0

Fig. 2. SegScope exploits the footprints left by segment protection to detect interrupts and leverages a counter (i.e., SegCnt) to represent the time interval between two consecutive interrupts.

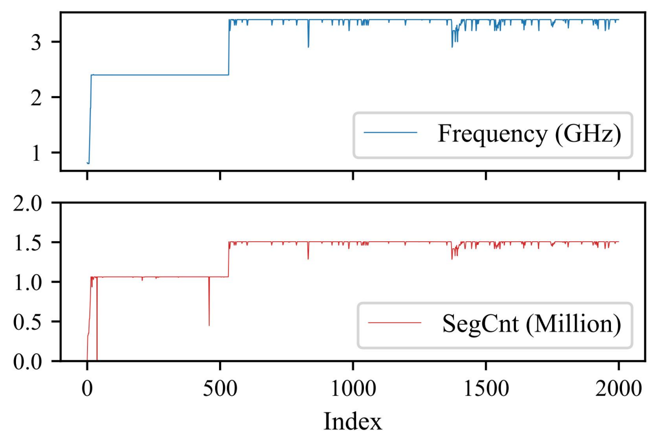


Fig. 3. SegCnt is linearly proportional to CPU frequency.

using any system statistics or timers. SegScope abuses the aforementioned footprints to detect interrupts and defines a counter to denote the time interval between two consecutive interrupts. Fig. 2 shows how SegScope exploits the footprints to probe interrupts. It has four main steps. *First*, we define a counter (coined as SegCnt), and set a data-segment selector (Sel) as non-zero null segment selector (i.e., 0x0001, 0x0002, or 0x0003). *Second*, a loop is defined, inside which we check the visible part of the segment selector and use SegCnt to measure the elapsed time. *Third*, during this period of time, if an interrupt occurs, current process context switches from user space (i.e., outer-privilege level) to kernel space. When the processor returns from an interrupt handler to user space, it will clear the null segment selector to 0. *Last*, upon detecting the selector's value change, we stop counting and break the loop. Thus, SegCnt denotes the number of executed loops, reflecting the elapsed time until an interrupt occurs. By repeating the above steps, we acquire fine-grained interrupt observations without relying on any timer.

Please note that after we pre-set a segment register, another process in the same system might accidentally sets the same segment register to another valid value rather than being cleared. In such cases, we still observe the value change of the monitored segment selector.

Optimizing SegCnt: As mentioned above, SegCnt is decided by the number of elapsed loops. Given that CPU frequency affects the number of instructions that can be executed within a certain time period, it has direct impacts on SegCnt. On top of that, when an interrupt occurs, preempting SegScope, context switch occurs and an interrupt handler routine is invoked. Thus, SegCnt stops self-increment until context switches back to user space. To this end, we present an equation below to formulate the relationship among SegCnt, the CPU frequency *Freq* and the time cost of an interrupt handler routine *w*.

$$\text{SegCnt} = \frac{Freq}{k} \times \left\lfloor \text{Time}_{interrupt} - w \right\rfloor, \tag{1}$$

where SegCnt is linearly proportional to $\frac{Freq}{k}$ for a given

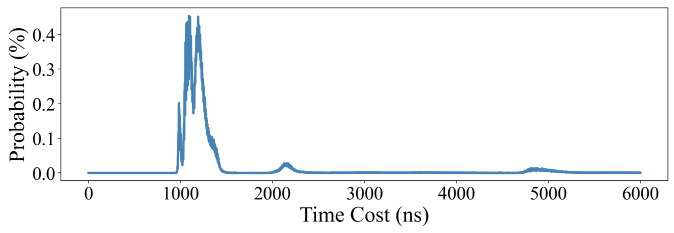


Fig. 4. The distribution of time costs (i.e., w) incurred by an interrupt handler routine. More than 90% of w are between 1 µs and 1.5 µs.

interval in userspace (k is a constant dependent on specific CPU architectures). Besides, considering that $\text{Time}_{interrupt}$ denotes the interval of two consecutive interrupts which is not all used by processes, we use $\text{Time}_{interrupt} - w$ to represent the interval in userspace.

Specifically, modern processors dynamically adjusts its frequency upon its computing workload at runtime [43]. A higher CPU frequency will increase the speed of executed instructions, introducing a higher SegCnt. To experimentally analyze their relationship, we use SegScope to probe 2,000 interrupts at runtime using the Lenovo machine listed in Table I. In the meantime, SegCnt and CPU frequency are recorded, generating Fig. 3. Clearly, SegCnt is linearly proportional to CPU frequency (with a few outliers).

Based on their relationship, we can significantly improve SegCnt to better reflect the elapsed CPU cycles between two consecutive interrupts. To retrieve *Freq*, an unprivileged Linux user interface (i.e., `scaling_cur_freq`) is available [13], [35], [47], [65]. Also, we can decide a good timing (so-called warm-up) to conduct our side-channel attacks by checking if the processor frequency is relatively stable.

To time the execution time of an interrupt handler routine (i.e., w), we develop an `eBPF` tool following [8], [9]. We run this tool for about 40 seconds to collect 1,000,000 measurements on the Lenovo Yangtian machine listed in Table I. Please note that `eBPF` used here is for analysis only. Fig. 4 shows the distribution of time costs incurred by an interrupt handler routine. All the measurements of time costs for an interrupt handler routine are less than 6 µs and 90.7% are between 1.0 µs and 1.5 µs. Thus, w is at the level of microseconds, and 3 orders of magnitude less than $\text{Time}_{interrupt}$ that is at the level of milliseconds, indicating a negligible impact on SegCnt.

Comparing SegScope with existing interrupt probing techniques: While SegScope works in timer-constrained scenarios where existing interrupt probing techniques do not work, we compare its performance on detecting/probing interrupts with that of two representative existing techniques, i.e., high-resolution timer based probing [51], [58] and loop-counting based probing [9], [31][2]. Please note that SegCnt is used to capture the time interval of two consecutive interrupts. Thus, SegScope does not need it in this comparison, and leverages the aforementioned footprint only.

To reproduce the first probing technique, we follow Schwarz et al. [51] that invoke `rdtsc` to observe the timestamp jumps. Regarding the second probing technique, we follow Lipp et al. [31] that defines a loop-counting program where an architectural timer with a resolution of $1ms$ is used to sample the self-increment counter at an interval of $5ms$. Besides, the techniques require an empirically-determined threshold to distinguish timestamp jump and counter plunge, respectively. To select an appropriate threshold, we follow [9] that uses `eBPF` to decide whether a collected timestamp or counter value is interrupted. As such, we determine a threshold of $1,000$ for the high-resolution timer based probing and $274,550$ for loop-counting based probing, respectively.

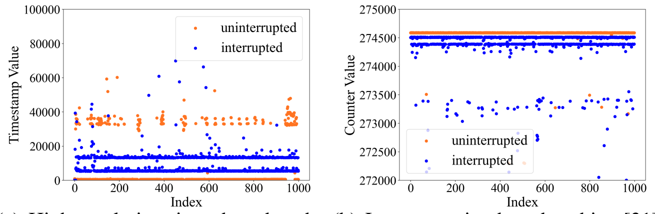
To benchmark their effectiveness in probing interrupts, we first configure `HZ` of CPU frequency, ranging from 100 to 1000, indicating the frequency of timer interrupts. We then use `isolcpus` to isolate a core for running a probing technique, with the aim to significantly reduce the noise from system activities. For each technique, we run it for 10 seconds to probe enough interrupts and repeat 100 times. As a baseline, $10 \times HZ$ timer interrupts and 3 performance monitoring interrupts are observed based on `eBPF`.

TABLE II
A COMPARISON OF SEGSCOPE AND EXISTING TECHNIQUES.

Methods	HZ = 100	HZ = 250	HZ = 1000
SegScope	1003.1 ± 0.3	2503.7 ± 0.6	10003.1 ± 0.4
Schwarz et al. [51]	1170.5 ± 51.1	2740.3 ± 62.7	10224.6 ± 52.3
Lipp et al. [31]	1038.8 ± 20.9	2000 ± 0	2000 ± 0

As shown in Table II, SegScope have achieved fine-grained interrupt probing with a precise number of $10 \times HZ + 3$ interrupts and a small variance under various settings. For the high-resolution timer based probing [51], it has much more false-positives with a significantly larger variance for three different frequencies. The loop-counting based probing [31] has a similar problem at the frequency of 100 and is only able to detect up to 2,000 interrupts at higher frequencies. This is

[2]`Prime+Probe` based probing proposed by [51] is not feasible for an unprivileged attacker.



(a) High-resolution timer based probing [51]
(b) Loop-counting based probing [31]

Fig. 5. The results of using high-resolution timer based probing [51] and loop-counting based probing [31] to probe interrupts.

because its sampling period is $5ms$ with up to 200 interrupts per second detected.

The reason why SegScope is much accurate than the other two in probing interrupts is that it reports an interrupt only when an interrupt caused the footprint. For the other two techniques, their detection relies on a pre-determined empirical threshold, which varies from machines to machines and is unreliable. Fig. 5 shows $1,000$ interrupted measurements and $1,000$ uninterrupted measurements for each technique. Clearly, it is impossible to set a threshold that can distinguish interrupted timestamp values or counter values.

Besides, as SegScope can precisely detect the interrupts that occur outside of its execution period via the footprint, it can be used to enhance other non-interrupt side channels. Specifically, before a non-interrupt side channel measurement, we set a segment register. After that, we perform the value check of the register. As such, we can determine whether the measurement is interrupted. If so, it will be filtered out. As the two previous techniques require continuous execution, they have not been demonstrated to reduce the impact of interrupts for other side channels.

C. *SegScope-based Timer*

Timer interrupts are critical hardware resources for OS kernel to schedule processes. As the time interval between two consecutive timer interrupts is fixed, they can be used as clock edges to build a clock interpolation scheme [53]. To construct such clock edges, we must filter timer interrupts.

To this end, we quantitatively analyze the impact of different types of interrupt on SegCnt. Specifically, an `eBPF` tool is developed to map collected SegCnt to an interrupt type. The analysis is performed on the Lenovo Yangtian machine listed in Table I. Our experimental results show that the top 3 probed interrupts are timer interrupts (994,748 corresponding SegCnt are obtained), rescheduling interrupts (962 corresponding SegCnt are obtained), and performance monitoring interrupts (872 corresponding SegCnt are obtained). Fig. 6 shows the statistical distribution of SegCnt corresponding to each type of interrupts. Particularly, different types of interrupts present distinguishable statistical characteristics regarding SegCnt. For example, SegCnt corresponding to timer interrupts is concentrated, as timer interrupts are activated at fixed time intervals. Thus, we pick a statistical method (i.e., Z-score) to retain SegCnt that correspond to timer interrupts.

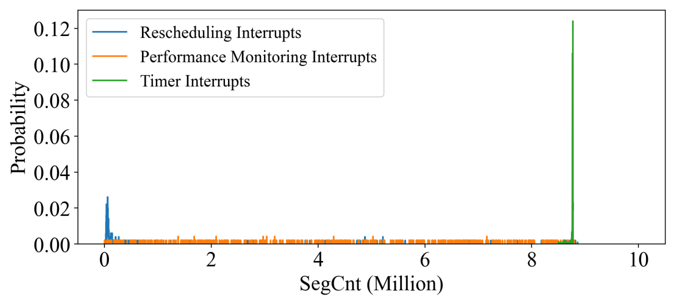


Fig. 6. The impact of specific interrupts on SegCnt. There is a significant statistical difference between timer interrupts and other interrupts.

With this key observation, we have three steps to build SegScope-based timer. *First*, we leverage SegScope to precisely probe interrupts. *Second*, we build a clock interpolation scheme by incrementing SegCnt between two consecutive interrupts. *Last*, we retain those SegCnt corresponding to timer interrupts by applying a simple statistical method (i.e., Z-score). Eq. 2 shows how Z-score represents how many standard deviations from the average a measurement is. In this paper, we filter out outliers for which Z-score are not within the range [-2,2].

$$\text{Z-score}(X) = \frac{X-\mu}{\delta} \tag{2}$$

where μ is the average and δ is the standard deviation. X is a measurement (i.e., SegCnt in our technique).

To construct the clock interpolation scheme, we define a loop-counting program in our timer. Specifically, in one loop, SegCnt is self-incrementing. When a timer interrupt is probed, the loop stops and current SegCnt is initialized to 0. Subsequently, another same loop starts and stops until a subsequent timer interrupt comes. Thus, the second SegCnt denotes the time between two consecutive timer interrupts. As the time interval between two consecutive timer interrupts is fixed, SegScope can time the other piece of code that shares the time interval.

Fig. 7 illustrates how the SegScope-based timer is used to measure the execution time of attacker-controlled code. Two pieces of code share a fixed time interval termed as T. One piece is attacker-controlled code (e.g., a code snippet that accesses a kernel address in breaking KASLR) that needs to be timed. The other piece is SegScope-based timer. When our timer statistically probes a timer interrupt and stops at t_1, the attacker-controlled code starts to execute and finishes at t_a. Thus, SegScope-based timer continues until a subsequent timer interrupt comes at t_2. Thus, the time that the attacker-controlled code takes is derived from the fixed time interval and the time SegScope takes from t_a to t_2.

Comparing SegScope-based timer with existing timers: Lipp et al. utilize a counting thread as a high-resolution timing source [32]. Particularly, a dedicated thread self-increments a global counter, which is read as timestamps. Based on their work, Schwarz et al. [55] propose an optimized asm

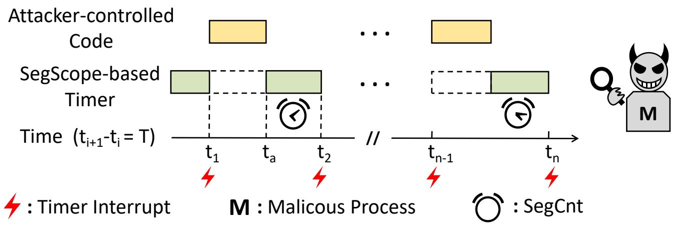


Fig. 7. An illustration of using SegScope-based timer to measure the execution time of attacker-controlled code.

version. We compare SegScope-based timer with the optimized counting thread [55] regarding their timer granularity and stability. Specifically, granularity refers to the cost of CPU cycles for one increment of a counter (either the counting-thread based counter or SegCnt). The stability is the degree to which each method is affected by system noise.

We follow [55] to reproduce the optimized counting thread. To achieve better stability, we pin the counting thread and the monitored code on different logical cores of the same physical core for execution.

To evaluate the granularity of the two timers, we apply SegScope to probe 10,000 consecutive interrupts for the test machines in Table III and use SegScope-based timer and counting thread to time the interval of two consecutive interrupts. *First*, when an interrupt is probed, we acquire its corresponding SegCnt and the counter value of counting thread, respectively. As a baseline, a high-resolution timer (i.e., `rdtsc` on Intel-based CPUs and `rdpru` on AMD-based CPUs [30]) is used to measure the elapsed CPU cycles. *Second*, Using distinguishable statistical characteristics of timer interrupts against SegCnt, we apply Z-score to filter out outliers and retain SegCnt corresponding to timer interrupts. *Last*, we measure the ratio of SegCnt or counting-thread based counter to CPU cycles as the granularity.

To evaluate their stability, we use each timer to measure a fixed time of 1 million CPU clock cycles. First, we implement two pieces of code: attacker-controlled code and timing code. The attacker-controlled code loops to use a high-resolution timer to read the timestamp counter and breaks when it increases by more than 1 million. The timing code uses either our timer or counting thread to repeatedly measure the execution time of the attacker-controlled code, and finally obtain 10,000 measurements for each. Second, we use Z-score to filter out outliers for both the two timers. Third, we compute the standard deviation (std) of either timer as their stability. Last, for a fair comparison, the computed standard deviations are multiplied by their corresponding granularity and converted to CPU cycles.

As shown in Table III, SegScope-based timer achieves one increment every 1.29 cycles on average, at the same granularity level of `rdtsc`, `rdpru`, and the optimized counting thread. Besides, the stability of the SegScope-based timer is at the same order of magnitude as the optimized counting thread. Both are less stable than `rdtsc` and `rdpru`.

Combing the granularity and stability, our timer's resolution

TABLE III
A COMPARISON OF SEGSCOPE-BASED TIMER AND COUNTING THREAD [55] TO NATIVE TIMESTAMP COUNTER AS THE BASELINE. SEGSCOPE HAS THE SAME ORDER OF MAGNITUDE WITH COUNTING THREAD.

		SegScope	**Counting thread (optimized asm)**	**rdtsc / rdpru (Baseline)**
Xiaomi Air 13.3	Granularity	0.93	0.54	1
	Std (Cycles)	1675.9	44.2	4.9
Lenovo Yangtian 4900v	Granularity	1.56	0.93	1
	Std (Cycles)	4577.9	2408.7	6.8
Honor Magicbook 16 Pro	Granularity	1.02	1.06	1
	Std (Cycles)	5109.8	3721.4	27.7
Amazon t2.large	Granularity	1.48	0.86	1
	Std (Cycles)	5585.8	18962.6	8.5
Amazon c5.large	Granularity	1.47	0.84	1
	Std (Cycles)	3106.4	10678.1	2.4
Average	Granularity	1.29	0.85	1
	Std (Cycles)	4011.2	7163.0	10.1

is at the level of thousands of CPU cycles. To further validate its resolution, we perform a Flush+Reload based Spectre attack in Section IV-F, which amplifies the timing difference to about 4000 CPU cycles, resulting in a secret-string leakage with a success rate of 100%.

IV. CASE STUDIES

In this section, we demonstrate how SegScope can be leveraged to mount end-to-end side channel attacks, including fingerprinting websites, extracting cryptographic keys, stealing DNN model architectures, enhancing Spectral attack, and breaking KASLR.

A. *Fingerprinting Websites*

A user's website visit history is sensitive, which shows personal interests, economic situation, political views, etc. Existing work [7], [9], [56], [67], [78] demonstrates that an unprivileged attacker can use various side channel techniques to fingerprint websites that have been visited. In this section, we show how SegScope can be used for website fingerprinting.

Specifically, as different websites can trigger different network and graphics activities, inducing featured device interrupts. These website-induced interrupts can be exploited for performing website fingerprinting attack [9]. Relying on this observation, an attacker can use SegScope to probe a certain number of interrupts, resulting in a time-series trace of probed interrupts.

Aligned with existing work [7], [9], [13], [56], [67], [78], we assume that the attacker leverages deep neural networks (DNNs) as classifiers to fingerprint websites. Specifically, in the offline phase, we use SegScope to collect enough interrupt traces incurred by website visiting for model training. In the online phase, we use the well-trained model to predict the website that has been visited. As presumed in [13], [24], [29], [78], the attacker puts her malicious code in the victim's computer, so she can access local system resources.

Experimental setup: We carry out our experiments on the Xiaomi machine listed in Table I, using both Chrome and Tor Browser. For the default setting, we use *taskset* to pin the malicious process and browser process to the same logical core. Moreover, to test the robustness of our method, we also test our method in other settings including pinning the two tasks to different logical cores, disabling frequency scaling, and disabling hyper-threading. In the different cores setting, we pin the attacker process to one logical core 1 and the victim browser to logical core 2 to avoid scheduling contention. In the disabling frequency scaling setting, we use the Linux command `cpufreq-set` to fix the CPU frequency at 2.5 GHz.

TABLE IV
CLASSIFICATION ACCURACY (AVG±STD) ACROSS 10-FOLD CROSS VALIDATION FOR WEBSITE FINGERPRINTING.

Setting	Chrome 109		Tor Browser 12	
	Top-1 Acc	Top-5 Acc	Top-1 Acc	Top-5 Acc
Default	92.4% ± 0.4	98.4% ± 0.2	87.4% ± 1.4	97.3% ± 0.4
Different cores used	91.0% ± 0.8	98.1% ± 0.4	83.3% ± 1.4	96.3% ± 0.2
Frequency scaling disabled	94.6% ± 0.5	98.9% ± 0.3	87.4% ± 0.9	96.5% ± 0.3
Hyper-threading disabled	94.5% ± 0.7	98.8% ± 0.3	89.5% ± 0.8	97.2% ± 0.3

Since Amazon has shut down Alexa ranking site[3], we use the same website list as [9], which has 100 websites. However, some of them have been shut down or changed domain names, so we finally use 95 of the 100 websites. For each setting, we record 100 traces for each of the 95 websites in Chrome and Tor Browser. For each trace, we use SegScope to sample 5000 SegCnt successively. To mirror typical use behavior, the browser's cache is not cleared before accessing each website.

We feed the collected SegCnt traces without any preprocessing into Long Short-Term Memory (LSTM) model consisting of 32 units[4] . First, we split the traces into 10 folds and select one fold as test set. Then, the remaining traces are split into a training set with 81% of all the traces and a validation set with 9% of all the traces. We repeat this procedure for each fold and compute the average accuracy across the 10 folds as the final accuracy.

Experimental results: The accuracy of our LSTM classifier is shown in Table IV. In the default setting, our attack's top-1 accuracy on Chrome is 92.4%, and the top-5 accuracy (an attacker uses no more than five guesses to find a website that is actually visited) is 98.4%. While Tor Browser introduces a strong security mechanism against side channel attacks, the top-1 accuracy of our attack is 87.4% and the top-5 accuracy is 97.3%, much higher than random guess of 1%. Moreover, the accuracy is above 80% in all settings, which validates that our method can be used to construct information-rich time-series traces in various settings.

B. *Extracting Cryptographic Keys*

In this case study, we show that an unprivileged attacker can steal cryptographic keys by using SegScope to probe interrupts. Specifically, the attacker uses SegScope rather than ar-

[3]https://www/alexa.com/topsites

[4]The LSTM (32 units, sigmoid activation) model can be found at https://github.com/jackcook/bigger-fish. We use the same model and hyper-parameters as [9], [56].

chitectural timer to construct a loop-counting program, which is affected by CPU frequency (as mentioned in Section III-B). We use SegScope to repeatedly probe every interrupts and sample a self-incremented SegCnt when each interrupt comes. After each sample, the SegCnt will be set to 0 and increment itself until the next interrupt comes.

Wang et al. [64] has shown that a timer-based loop-counting program can be used to mount realistic frequency side channel attack (i.e., pixel stealing). However, they rely on an architectural timer and can be easily defeated by adding noise to the timer. As discussed in Section III-B, when the interrupts are activated regularly (e.g., timer interrupts), incrementing speed of SegCnt will be proportional to the CPU frequency. We can use SegScope to obtain frequency information from an unprivileged process, without access to the `cpufreq` interface or any architectural timer.

We verify that the workload can effect our SegCnt. Aligned with [65], we use SegScope to attack the CIRCL which spawns 300 concurrent goroutines and uses 10 randomly generated 378-bit keys. For each target bit i in a secret key m, if $m_i \neq m_{i-1}$, the crafted challenge ciphertext will trigger an anomalous 0 value. If $m_i = m_{i-1}$, there will be no challenge ciphertext that can trigger the anomalous 0 value. We conduct our ciphertext attack on our Lenovo Yangtian machine. Fig. 8 shows the distribution of SegCnt when the challenge ciphertext introduces an anomalous 0 or not. A correct key-bit guess causes the processor to execute at a higher frequency than an incorrect key-bit guess does, resulting in higher SegCnt. Once we have the ability to distinguish if $m_i \neq m_{i-1}$, we can group the bits. Finally, we only need to guess whether the first bit is 0 or 1 to extract all the bits (the search space is significantly reduced to 2).

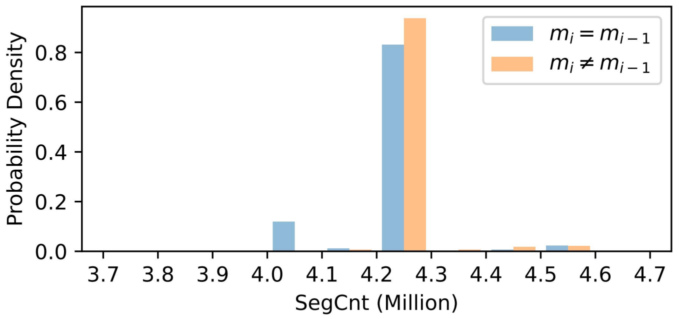


Fig. 8. Distribution of SegCnt when the challenge ciphertext introduces an anomalous 0 ($m_i \neq m_{i-1}$) or not ($m_i = m_{i-1}$).

C. Stealing DNN Model Architectures

To design an effective DNN model, developers need to spend significant time and energy in searching and fine-tuning model architectures. The optimized model structures are always considered as confidential and high-value property [41], [66], [69], [73], [74]. Furthermore, a known DNN architecture can also help to mount adversarial transfer attacks [37], [46], in which the attacker requires a substitute model to generate their adversarial examples. Their attack success rate depends on the similarity between the victim model and the substitute model. To this end, previous work has exploited various side channels to steal DNN model architectures, such as EM signal [41], computer bus [20], [79], cache side channel [70], and Rowhammer [48].

Aligned with previous work [20], [41], [48], [70], [79], our attack has two phases: an *offline preparation* phase and an *online classification* phase. We assume that the victim runs a DNN model to inference and the attacker uses SegScope to perform her attack. In the *offline preparation* phase, we collect enough SegCnt traces to build a series of well-trained classifiers, which can translate a SegCnt trace to its corresponding layer types. In the *online classification* phase, we query a black-box target model running on the victim machine to trigger its model inference and use a SegScope process to collect SegCnt traces. Then we use the offline-trained classifiers to recover the model structure.

Experimental setup: We carry out our experiments in our Lenovo Yangtian machine. The victim model uses PyTorch 1.13.0 with Python version 3.9.13 as its underlying deep learning framework. The attacker process and the victim process run on separate physical cores. We consider 2,000 network structures for training and 500 network structures for test, including various model architecture families (e.g., AlexNet, VGG, and random architectures). The input size used by these models is assumed as 3×224×224. The batch size is selected from [32,64,96]. [5] We embed the `torch.autograd.profiler` into each DNN model to automate the annotation.

We use an optimized BiLSTM as classifier to segment the SegCnt trace corresponding to different layers of an accurate model layer sequence. The classifier[6] is trained in a machine with an NVIDIA GeForce RTX 3090 GPU (24 GB video memory), Intel i9-10920X CPU (24 logical cores) and 128 GB DRAM memory. The deep learning framework used here is PyTorch 2.0.1 with Python version 3.8.13. Aligned with previous work [41], we use two metrics to evaluate the structure recovering performance, i.e., Levenshtein Distance Accuracy (LDA) and Segment Accuracy (SA). LDA represents the similarity between a predicted structure and a ground-truth structure. SA represents the percentage of sampling points that are correctly predicted into their ground-truth layer types.

Experimental results: As shown in Fig. V, LDA for all the layer types is about 87.2%. The layers that require intensive computation achieve better SA than those layers with less intensive computation because the latter has a smaller number of sampling points. The sampling frequency is dependent on the frequency of timer interrupts, which is 250 by default, so we believe that our attack can perform better in systems with larger HZ (e.g., 1000).

[5] The batch size is constrained by the heat generated by the model inference which will cause a huge number of thermal event interrupts to noise our attack. We decrease the maximum batch size to control the heat when the model size is increasing. Exploring the security implications of thermal event interrupts is left as future work.

[6] The classifiers can be found at https://github.com/LearningMaker/DeepTheft.

TABLE V
CLASSIFICATION ACCURACY OF NETWORK STEPS (BN = BATCH NORMALIZE, MP = MAX POOLING, AP = AVERAGE POOLING).

Layer	Conv	BN	ReLu	MP	AP	Linear	Overall
SA (%)	98.2	77.8	58.6	85.2	50.4	52.8	97.7
LDA (%)	87.7	86.0	85.6	85.6	86.5	86.9	87.2

TABLE VI
THE ARCHITECTURAL STATES WHEN DIFFERENT MICROARCHITECTURAL EVENTS OCCUR.

Microarchitectural events	Architectural states	
	EFLAGS.CF	Selected data segment register
Timeout	1	1
Cacheline writes	0	1
Interrupts	0	0

D. *Enhancing Spectral Attack*

Spectral (Spectre with architectural leakage) attack [77] exploits the new ISA extension (i.e. `umonitor` and `umwait`) introduced by Intel to resurrect Spectre attack in a timer-constrained scenario. To achieve this, an unprivileged attacker needs to control two processes that run on separate cores and share a cache line. The monitoring process executes the `umonitor` instruction with the cache line as the target and executes `umwait` to enter a light-weight sleep mode. The other process mistrains the branch predictor and make CPU load the value with a secret offset into cache.

As shown in Table VI, there are three reasons for waking up the monitoring process. If the monitored process sleeps until the maximum sleep time is reached (default is 100,000 cycles), the carry flag will be set (`EFLAGS.CF` = 1). Otherwise, the carry flag will be cleared (`EFLAGS.CF` = 0). However, the original Spectral attacker cannot distinguish cache line writes and interrupts because both the two architectural events clear the carry flag. Even in an idle system without extra interrupts caused by applications, the interrupts are still considered to be an unavoidable noise for original Spectral attack, causing a bit error rate of nearly 1%.

In this case study, we show that a Spectral attacker can use SegScope to filter out the interrupted measurements, thereby enhancing his attack. Specifically, the attacker sets the segment register as a non-zero null value (e.g., 0x1) at the monitoring core and then executes the Spectre attack. After the monitoring process is awakened, the attacker checks the segment register and `EFLAGS.CF` at the same time. If the preset segment register is cleared, she can conclude that the monitored process is awakened by an interrupt rather than a cache line write.

Experimental setup: We use a simple bit-wise Spectre gadget [2], [19], [38], [54], [77] to implement Spectral attack. We evaluate our enhanced Spectral on the Lenovo Savior machine. The victim code containing a Spectral gadget runs on different physical cores with the monitoring code. We follow the mistraining strategy used by Zhang et al. [28], [77], which provides 5 in-bound indices for every out-of-bound index. To increase the chance of inducing mis-speculation, we call the gadget 12 times for every bit to leak.

Experimental results: With the default `mwait` timeout of 100,000 cycles, our enhanced Spectral achieves an average leakage rate of 53,114 bit/s with a 0.01% error rate, while the original Spectral achieves an average leakage rate of 53,889 bit/s with 0.56% error rate. Figure 9 shows the error rate for different `umwait` timeouts from 20,000 to 200,000 cycles. Even in an idle system, where there are almost no interrupts actively triggered by processes, the interrupt noise still causes an error rate of up to 1%. However, SegScope can be used to distinguish and filter out the interrupt-induced error bits.

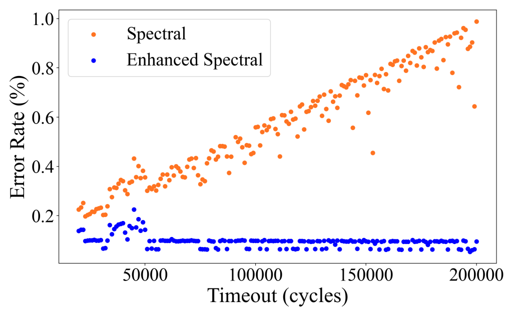


Fig. 9. Error rate w.r.t. timeouts. SegScope can remove the impacts of interrupts and significantly reduce the error rate of Spectral.

E. *Breaking KASLR*

Kernel address-space layout randomization (KASLR) is a defense mechanism against memory corruption exploits, which randomizes the base address of text segment at every boot time. In Linux, the text segment is mapped to an address range of 1GB size and aligned with 2MB, so there are 512 possible base addresses [5], [6]. Prior work [18], [21], [30] has shown that there is a timing difference when accessing or executing *prefetch* instructions on a memory address depending on whether that address is mapped or not. Recently, without reliance on architectural timers, some studies utilize their side channels, such as temperature [26], power [34], and CPU frequency [65], to distinguish between mapped address and unmapped address. The idea of breaking KASLR using SegScope is to build a SegScope-based stealthy timer to measure the execution time of attacker's operations, obviating usage of architectural timer.

For the attack, we repeatedly access or prefetch the first byte of each of the possible base addresses for the kernel text segment and use SegScope to measure its execution time [18], [21]. When using the memory access method, we register a segment fault handler in the userspace to handle segment faults, avoiding process crashes [21]. We run our measurement on the 512 possible addresses. Finally, to distinguish the access or `prefetch` instructions to mapped addresses from access or `prefetch` instructions to unmapped addresses, we use SegScope to measure the execution time of these operations.

Experimental setup: There are two configurable parameters in our prototype: a loop number (K) before each timing and a timing number (C) for each possible address. Specifically, before each timing, we access or prefetch data from a possible address K times first. After that, we use SegScope to measure the overall execution time. The the above steps are repeated C times for each possible address. We note that timing side channel attacks always require multiple sampling to obtain a stable result.

We first investigate the ground truth when executing memory accesses and `prefetch` instructions for a mapped address and unmapped address separately on the Xiaomi Air 13.3 machine. Second, we directly access each possible address and use various timers to distinguish the mapped address from unmapped addresses on our Lenovo Yangtian machine for 1000 trials. Last, we evaluate the success rate when executing `prefetch` instructions to access each possible address on four tested machines listed in Table I for 1000 trials.

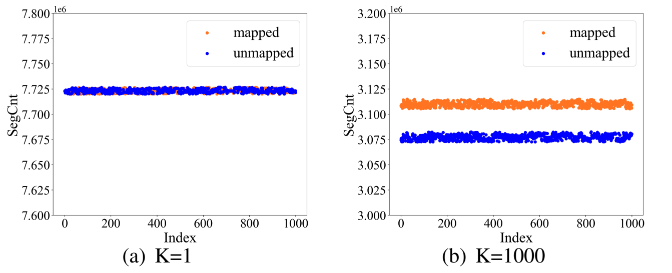


Fig. 10. The impact of K on SegCnt when accessing a mapped or unmapped kernel address (segment faults that will be incurred are dealt with by a pre-defined userspace handler).

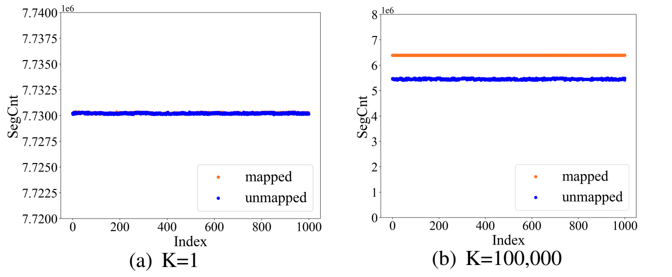


Fig. 11. The impact of K on SegCnt when accessing a mapped or unmapped kernel address via `prefetch` (no segment faults will be incurred).

Experimental results: Fig. 10 shows the difference of SegCnt when performing 1 and 1000 continuous memory accesses for each timing. It can be seen that the attacker can amplify the timing difference by performing multiple memory accesses continuously and using the sum of their execution time to distinguish the mapped address. We also use `prefetch` instruction to repeat the steps above. As shown in Fig. 11, attacker can distinguish the `prefetch` instructions to mapped address from the `prefetch` instructions to unmapped address with a proper K.

As shown in Table VII, it is hard for the architectural timer with a granularity of 1 ms and counting thread to distinguish the timing differences between mapped and unmapped addresses. Compared with our timer without any denoising technique, the three denoised SegScope-based timers are much more effective and achieve high success rate when we repeatedly attack 10 times (C = 10).

Table VIII shows the results of using `prefetch` to break KASLR on various environments. SegScope successfully derandomizes KASLR with 100% accuracy within only approximately 10 seconds.

TABLE VII
COMPARISONS OF USING VARIOUS TIMERS TO BREAK KASLR WHEN DIRECTLY ACCESSING THE POSSIBLE ADDRESSES. PLEASE NOTE THAT THESE ARCHITECTURAL TIMERS ARE UNAVAILABLE IN OUR THREAT MODEL.

Timer	Param. C	Time (s)	Top-1 Acc	Top-5 Acc
Our timer without any denoising	1	2.06	2.7%	96.1%
	10	20.41	0.3%	1.3%
Our timer with Z-score (default)	1	2.05	1.5%	96.9%
	10	20.33	99.6%	99.8%
Our timer with frequency	1	2.05	71.1%	89.8%
	10	20.43	83.7%	92.4%
Our timer with Z-score and frequency	1	2.06	71.9%	88.6%
	10	20.31	100%	100%
Counting thread [32], [55]	1	1.27	0.3%	1.3%
	10	12.61	0.3%	0.9%
Architectural high-resolution timer (i.e., rdtsc)	1	1.27	96.9%	97.6%
	10	12.56	93.8%	94.9%
Architectural timer (1us)	1	1.26	97.2%	98.2%
	10	12.50	93.8%	94.1%
Architectural timer (1ms)	1	1.27	0	0
	10	12.64	0	0

TABLE VIII
EVALUATION OF USING SEGSCOPE TO BREAK KASLR ON VARIOUS ENVIRONMENTS.

Machine	Param. C	Time (s)	Top-1 Acc	Top-5 Acc
Xiaomi Air 13.3	1	2.14	63.7%	98.4%
	5	10.28	100%	100%
Lenovo Yangtian 4900v	1	2.05	96.1%	100%
	5	10.24	100%	100%
Amazon t2.large	1	2.05	83.0%	99.7%
	5	10.21	100%	100%
Amazon c5.large	1	2.06	87.2%	99.2%
	5	10.31	100%	100%

F. Leaking Memory with Spectre

Spectre is a CPU vulnerability that allows an attacker to cross memory boundary and read any target secrets in memory [27]. As there are multiple Spectre variants, we choose the Spectre-V1 of bounds check bypass in this evaluation.

Specifically, it has a bounds-check gadget, which has conditional branch instructions to check if an array-index candidate is within a valid range. We first feed the gadget with multiple legal array indexes to mistrain a branch predictor. After that, an out-of-bound index is fed to the gadget, bypassing the bounds check and performing a speculative memory access. The access loads an attacker-chosen secret from memory into cache, which is leaked by previous side channel techniques such as Flush+Reload [71] with an architectural timer.

Experimental Setup: To validate the resolution of SegScope-based timer, we use it to replace the architectural timer in

Flush+Reload. As one byte has 256 possible values, only the value loaded speculatively into cache is the secret and thus has shorter access latency, resulting in higher SegCnt. For other values, they have lower SegCnt. When performing one round of Flush+Reload to infer a secret, we use a number of same gadgets rather than 1 to amplify the timing difference between cache hits of the secret value and cache misses of other 255 values. When the number of gadgets reaches 200, the timing difference is amplified to about 4,000 CPU cycles and our timer can be effectively used to infer the secret. The attack is performed in the Xiaomi machine listed in Table I.

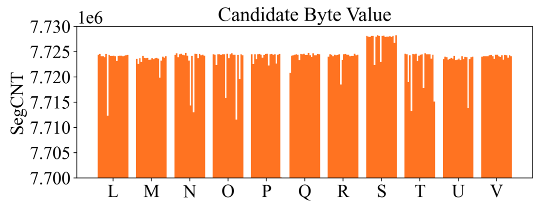


Fig. 12. Reading out arbitrary address in application memory. The candidate byte value with the highest SegCnt is the one stored at the given offset.

Experimental Results: We have successfully leaked a secret string of 1000 characters with a success rate of 100% and a leakage rate of 0.15 B/s on average. Taking secret 'S' as an example, Fig. 12 shows possible candidate values with their corresponding access latency, denoted by SegCnt. Clearly, 'S' is the secret as it has the highest SegCnt and all other candidate values have lower SegCnt.

V. Discussion

Other security implications of SegScope: With fine-grained interrupt probing, SegScope can be used to perform other interrupt side channels, such as monitoring keystrokes [31], [51], [58] and inferring PDF document contents [39]. As SegCnt is consistent with CPU frequency, SegScope can also be used to demonstrate other frequency-based attacks such as building covert channels [47], [65] and extracting AES-NI keys [35]. Besides, SegScope can be used to improve the performance of micro-benchmark programs. Particularly, if a well-defined benchmark program is interrupted, its measured execution time will not be accurate. As such, SegScope can significantly filter out the interrupted measurements, similar to the enhanced Spectral attack.

Modifying OS kernel/x86 CPU architecture to mitigate SegScope: Section III-B discusses that SegScope exploits segment protection check to clear non-zero null segment selectors when transitioning from kernelspace to userspace. The check is enforced by x86 CPUs via intended instructions, e.g., `IRET`. The recent Linux kernels do not save and restore `DS`, `ES`, and `FS`, but support `GS` via `SWAPGS`. However, `GS` is still cleared when the context returns to userspace. Thus, if kernel is modified to save and restore `DS`, `ES`, and `FS`, they will still be cleared by the CPU when the context switches to userspace, rendering the kernel-based mitigation ineffective.

A potential strategy of mitigating SegScope is to keep the values of the segment registers unchanged in future architectures, which however introduces a new covert channel: one process manipulates the values of the segment registers to transfer data, while another process, running on the same logical core, monitors the changes in the register values. Another possible mitigation is a hardware-software co-design: when context switch occurs, OS kernels save and restore the segment registers for every process, and CPUs preserve the non-zero segment selectors as-is. However, this mitigation is unable to protect legacy hardware, and needs cooperation between x86 processor and OS vendors. Also, the updates to kernels introduce additional overhead during context switching.

Restricting/monitoring access to segment registers: As SegScope leverages access to data segment registers to probe interrupts, restricting the access to root users can be effective. However, this restriction is likely to badly affect benign programs that need to access these registers. Alternatively, MASCAT [23] can be extended to detect SegScope in a binary check, as SegScope requires frequent accesses to certain segment registers. However, the attacks may be encapsulated and concealed to bypass the detection, e.g., via Intel SGX, known as SGX malware [55]. Besides, frequent setting of a segment register is not always needed. In our enhanced Spectral attack, for one side-channel measurement, SegScope is used twice to check whether it is noised by interrupts.

Mitigating SegScope-based timer: As our crafted timer relies on the fixed time interval between two consecutive timer interrupts, a possible mitigation is to dynamically change the interval between consecutive timer interrupts. This requires modifying the process scheduler and recompiling the OS kernel. Another solutions is to enable *tickless mode*. In this mode, no timer interrupts will be activated and sent to CPU cores that are idle or have a single running process, as no process switch is needed. Thus, when our timer is scheduled to run on these cores, it will be mitigated. However, we can render this mode ineffective by co-locating SegScope with a computing-intensive process in the same logic core.

Limitations of SegScope-based timer: As discussed in Section III-B, SegScope-based timer is badly affected by system noise, including CPU frequency and kernel routine. While the timer achieves the same granularity as the high-resolution timer such as `rdtsc`, it is much less stable, rendering its resolution lower (at the level of thousands of CPU cycles). Besides, when the execution time of the attacker-controlled code is longer than the time interval between two consecutive timer interrupts, our timer can only acquire the remainder from the execution time modulus the fixed time interval. Fortunately, this is still sufficient for our timer-based attacks, because the remainders can distinguish the access to mapped addresses from unmapped addresses in breaking KASLR, and cache hits from cache misses in performing Spectre. Our timer does not work for an attack that happens to have the same remainders for different cases of measurements, which is unlikely to occur.

VI. Conclusion

In this paper, we introduce SegScope, a new technique that abuses segment protection on x86 to acquire fine-grained interrupt observations without relying on any timer. To show its security implications, SegScope has been used to demonstrate different case studies without any timers, such as inferring website visits, extracting cryptographic keys, stealing DNN model architectures, and enhancing Spectral attack. Compared with existing timer-based interrupt-probing techniques, SegScope is fine-grained without introducing false-positives. Besides, we rely on SegScope to craft a fine-grained timer, as regular timer interrupts as clock edges contain timestamps. The granularity of our crafted timer is at the same order of gratitude as the high-resolution architectural timer, i.e., `rdtsc` on Intel-based CPUs and `rdpru` on AMD-based CPUs. To demonstrate the viability of the crafted timer, we replace the architectural timer with our timer to mount realistic timing side channel attacks. Experimental results show that KASLR is cracked within about 10 seconds and Flush+Reload based Spectre attack is successfully performed.

References

[1] AMD, Inc., "Amd64 architecture programmers manual volume 2: System programming," 2019.

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

[3] S. Bratus, P. C. Johnson, A. Ramaswamy, S. W. Smith, and M. E. Locasto, "The cake is a lie: privilege rings as a policy resource," in *ACM workshop on Virtual Machine Security*, 2009, pp. 33–38.

[4] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution," in *USENIX Security Symposium*, 2018.

[5] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *ACM SIGSAC Conference on Computer and Communications Security*, 2019, p. 769–784.

[6] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break it, fix it, repeat," in *Asia Conference on Computer and Communications Security*, 2020, p. 481–493.

[7] Y. Cohen, K. S. Tharayil, A. Haenel, D. Genkin, A. D. Keromytis, Y. Oren, and Y. Yarom, "Hammerscope: Observing dram power consumption using rowhammer," in *ACM SIGSAC Conference on Computer and Communications Security*, 2022, p. 547–561.

[8] I. community, "BPF compiler collection (bcc)," 2023. [Online]. Available: https://github.com/iovisor/bcc

[9] J. Cook, J. Drean, J. Behrens, and M. Yan, "There's always a bigger fish: A clarifying analysis of a machine-learning-assisted side-channel attack," in *International Symposium on Computer Architecture*, 2022, p. 204–217.

[10] V. Costan and S. Devadas, "Intel sgx explained," *Cryptology ePrint Archive*, 2016.

[11] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai, "Smashex: Smashing sgx enclaves using exceptions," in *ACM SIGSAC Conference on Computer and Communications Security*, 2021, p. 779–793.

[12] W. Diao, X. Liu, Z. Li, and K. Zhang, "No pardon for the interruption: New inference attacks on android through interrupt timing analysis," in *IEEE Symposium on Security and Privacy*, 2016, pp. 414–432.

[13] D. R. Dipta and B. Gulmezoglu, "Df-sca: Dynamic frequency side channel attacks are practical," in *Annual Computer Security Applications Conference*, 2022, p. 841–853.

[14] S. B. Dutta, H. Naghibijouybari, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Leaky buddies: cross-component covert channels on integrated cpu-gpu systems," in *International Symposium on Computer Architecture*, 2021, p. 972–984.

[15] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss, "Rapid prototyping for microarchitectural attacks," in *USENIX Security Symposium*, 2022, pp. 3861–3877.

[16] S. Fan, Z. Hua, Y. Xia, H. Chen, and B. Zang, "Isa-grid: Architecture of fine-grained privilege control for instructions and registers," in *International Symposium on Computer Architecture*, 2023.

[17] A. Faz-Hernández and K. Kwiatkowski, "Introducing CIRCL: An advanced cryptographic library," *Cloudflare*, 2019. [Online]. Available: https://github.com/cloudflare/circl.

[18] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *ACM SIGSAC Conference on Computer and Communications Security*, 2016, p. 368–379.

[19] L. Hetterich and M. Schwarz, "Branch different - spectre attacks on apple silicon," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2022, p. 116–135.

[20] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood, and Y. Xie, "Deepsniffer: A dnn model extraction framework based on learning architectural hints," in *Architectural Support for Programming Languages and Operating Systems*, 2020, p. 385–399.

[21] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *IEEE Symposium on Security and Privacy*, 2013, pp. 191–205.

[22] Intel, Inc., "Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c," 2019.

[23] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Mascat: Preventing microarchitectural attacks before distribution," in *ACM Conference on Data and Application Security and Privacy*, 2018, p. 377–388.

[24] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *IEEE Symposium on Security and Privacy*, 2012, pp. 143–157.

[25] J. Kim, S. van Schaik, D. Genkin, and Y. Yarom, "Ileakage: Browser-based timerless speculative execution attacks on apple devices," in *ACM SIGSAC Conference on Computer and Communications Security*, 2023, p. 2038–2052.

[26] T. Kim and Y. Shin, "Thermalbleed: A practical thermal side-channel attack," *IEEE Access*, vol. 10, pp. 25 718–25 731, 2022.

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

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

[29] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing webpages rendered on your browser by exploiting gpu vulnerabilities," in *IEEE Symposium on Security and Privacy*, 2014, pp. 19–33.

[30] M. Lipp, D. Gruss, and M. Schwarz, "AMD prefetch attacks through power and time," in *USENIX Security Symposium*, 2022, pp. 643–660.

[31] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard, "Practical keystroke timing attacks in sandboxed javascript," in *European Symposium on Research in Computer Security*, 2017, pp. 191–209.

[32] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *USENIX Security Symposium*, 2016, pp. 549–564.

[33] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of amd's cache way predictors," in *Asia Conference on Computer and Communications Security*, 2020, p. 813–825.

[34] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "Platypus: Software-based power side-channel attacks on x86," in *IEEE Symposium on Security and Privacy*, 2021, pp. 355–371.

[35] C. Liu, A. Chakraborty, N. Chawla, and N. Roggel, "Frequency throttling side-channel attack," in *ACM SIGSAC Conference on Computer and Communications Security*, 2022, p. 1977–1991.

[36] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[37] Y. Liu, X. Chen, C. Liu, and D. Song, "Delving into transferable adversarial examples and black-box attacks," 2017.

[38] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "Dolma: Securing speculation with the principle of transient non-observability," in *USENIX Security Symposium*, 2021, pp. 1397–1414.

[39] H. Ma, J. Tian, D. Gao, and C. Jia, "Walls have ears: Eavesdropping user behaviors via graphics-interrupt-based side channel," in *Information Security*, 2020, pp. 178–195.

[40] H. Ma, J. Tian, D. Gao, and C. Jia, "On the effectiveness of using graphics interrupt as a side channel for user behavior snooping," *IEEE Transactions on Dependable and Secure Computing*, pp. 3257–3270, 2021.

[41] H. T. Maia, C. Xiao, D. Li, E. Grinspun, and C. Zheng, "Can one hear the shape of a neural network?: Snooping the gpu via magnetic side channel," in *USENIX Security Symposium*, 2022.

[42] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *International Symposium on Computer Architecture*, 2012, pp. 118–129.

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

[44] D. Moghimi, J. V. Bulck, N. Heninger, F. Piessens, and B. Sunar, "CopyCat: Controlled Instruction-Level attacks on enclaves," in *USENIX Security Symposium*, 2020, pp. 469–486.

[45] Y. Oyama, "How does malware use rdtsc? a study on operations executed by malware with cpu cycle measurement," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2019, pp. 197–218.

[46] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Asia Conference on Computer and Communications Security*, 2017, p. 506–519.

[47] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Dvfsspy: Using dynamic voltage and frequency scaling as a covert channel for multiple procedures," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022, pp. 654–659.

[48] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories," in *IEEE Symposium on Security and Privacy*, 2022, pp. 1157–1174.

[49] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "Pacman: Attacking arm pointer authentication with speculative execution," in *International Symposium on Computer Architecture*, 2022, p. 685–698.

[50] S. Schildermans, K. Aerts, J. Shan, and X. Ding, "Paratick: Reducing timer overhead in virtual machines," in *International Conference on Parallel Processing*, 2021, p. 1–10.

[51] M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. Maurice, R. Spreitzer, and S. Mangard, "Keydrown: Eliminating software-based keystroke timing side-channel attacks," in *Network and Distributed System Security Symposium*, 2018.

[52] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *ACM SIGSAC Conference on Computer and Communications Security*, 2019, p. 753–768.

[53] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript," in *Financial Cryptography and Data Security*, 2017, pp. 247–267.

[54] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *European Symposium on Research in Computer Security*, 2019, p. 279–299.

[55] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017, p. 3–24.

[56] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *USENIX Security Symposium*, 2019, pp. 639–656.

[57] X. Tang, Y. Lin, D. Wu, and D. Gao, "Towards dynamically monitoring android applications on non-rooted devices in the wild," in *ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2018, p. 212–223.

[58] J. Trostle, "Timing attacks against trusted path," in *IEEE Symposium on Security and Privacy*, 1998, pp. 125–134.

[59] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A practical attack framework for precise enclave execution control," in *Workshop on System Software for Trusted Execution*, 2017.

[60] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic," in *ACM SIGSAC Conference on Computer and Communications Security*, 2018, p. 178–195.

[61] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *IEEE Symposium on Security and Privacy*, 2019, pp. 88–105.

[62] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in xen," in *ACM Workshop on Cloud Computing Security Workshop*, 2011, p. 41–46.

[63] U. Walter and V. Oberle, "μ-second precision timer support for the linux kernel," 2001.

[64] Y. Wang, R. Paccagnella, A. Wandke, Z. Gang, G. Garrett-Grossman, C. W. Fletcher, D. Kohlbrenner, and H. Shacham, "DVFS frequently leaks secrets: Hertzbleed attacks beyond SIKE, cryptography, and CPU-only data," in *IEEE Symposium on Security and Privacy*, 2023, pp. 2306–2320.

[65] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning power Side-Channel attacks into remote timing attacks on x86," in *USENIX Security Symposium*, 2022, pp. 679–697.

[66] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Faruque, "Leaky DNN: Stealing deep-learning model secret with gpu context-switching side-channel," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2020, pp. 125–137.

[67] S. Wu, J. Yu, M. Yang, and Y. Cao, "Rendering contention channel made practical in web browsers," in *USENIX Security Symposium*, 2022, pp. 3183–3199.

[68] H. Xiao and S. Ainsworth, "Hacky racers: Exploiting instruction-level parallelism to generate stealthy fine-grained timers," in *Architectural Support for Programming Languages and Operating Systems*, 2023.

[69] L. Xie, C. Li, Z. Wang, X. Zhang, B. Chen, Q. Shen, and Z. Wu, "Shisrcnet: Super-resolution and classification network for low-resolution breast cancer histopathology image," *International Conference on Medical Image Computing and Computer Assisted Intervention*, 2023.

[70] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," in *USENIX Security Symposium*, 2020, pp. 2003–2020.

[71] Y. Yarom and K. Falkner, "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack," in *USENIX Security Symposium*, 2014, pp. 719–732.

[72] J. Yu, A. Dutta, T. Jaeger, D. Kohlbrenner, and C. W. Fletcher, "Synchronization storage channels (S2C): Timer-less cache Side-Channel attacks on the apple m1 via hardware synchronization instructions," in *USENIX Security Symposium*, 2023, pp. 1973–1990.

[73] S. Zhai, Y. Dong, Q. Shen, S. Pu, Y. Fang, and H. Su, "Text-to-image diffusion models can be easily backdoored through multimodal data poisoning," *ACM Multimedia*, 2023.

[74] S. Zhai, Q. Shen, X. Chen, W. Wang, C. Li, Y. Fang, and Z. Wu, "Ncl: Textual backdoor defense using noise-augmented contrastive learning," *arXiv preprint arXiv:2303.01742*, 2023.

[75] K. Zhang and X. Wang, "Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems," in *USENIX Security Symposium*, 2009, p. 17–32.

[76] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based fault injection using selective state reset," in *USENIX Security Symposium*, 2024.

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

[78] Z. Zhang, S. Liang, F. Yao, and X. Gao, "Red alert for power leakage: Exploiting intel rapl-induced side channels," in *Asia Conference on Computer and Communications Security*, 2021, p. 162–175.

[79] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes attack: Steal dnn models with lossless inference accuracy," in *USENIX Security Symposium*, 2021.

Appendix

A. Abstract

As discussed in the paper, we proposed SegScope, a new technique that abuses segment protection on x86 to acquire fine-grained interrupt observations without relying on any timer. To reveal its security implications, we leverage SegScope to mount different attacks. First, SegScope, as a side channel, is used to fingerprint websites, extract DNN model architectures, and steal cryptographic keys. Second, SegScope has enhanced existing non-interrupt side channel attacks (e.g., Spectral attack). Last, SegScope has been leveraged to craft a fine-grained timer, which serves for KASLR de-randomization and Spectre, respectively. In this artifact, we choose website fingerprinting (SegScope serves as an interrupt side channel), enhanced spectral attack, and KASLR de-randomization (SegScope serves for a timing side channel).

B. Artifact check-list (meta-information)

- **Experiments:**
  We describe three attacks, i.e., fingerprinting websites, enhancing Spectral attack, and breaking KASLR.
- **How much disk space is required (approximately)?:**
  About 100 MB.
- **Publicly available?:**
  The code of three aforementioned attacks is publicly available. The other code of this paper is available upon request.
- **Code licenses (if publicly available)?:**
  General Public License v3.0.

C. Description

1) How to access: The source code is available here: https://doi.org/10.6084/m9.figshare.24658953.v1.

2) Hardware dependencies: As segment protection is supported by x86 processors, SegScope is expected to work on all x86 CPUs. We have validated SegScope on multiple x86 CPUs (i.e., Intel Core i5-8250U, Intel Core i7-4790, Intel Core i9-12900H, and AMD Ryzen 7 5800H). To demonstrate an enhanced Spectral attack, recent CPUs supporting UMWAIT/UMONITOR instructions are required such as Intel Core i9-12900H in our paper.

3) Software dependencies: We recommend Ubuntu 20.04.5 with Linux kernel 5.15 by default. In this environment, build tools (i.e., gcc, make) and Python 3 are needed.

D. Major Claims

1) C1: SegScope can probe fine-grained interrupts without relying on any timer. We successfully leverage it to perform an interrupt side channel attack by fingerprinting which website a user has opened.

2) C2: As SegScope can distinguish side channel measurements that have been interrupted, it effectively reduces the noise of interrupts on other non-interrupt side channels. Taking the Spectral side channel as an example, SegScope has removed the impacts of interrupts and significantly reduce its error rate.

3) C3: We apply SegScope to build a fine-grained timer, which achieves the same level of timing granularity as the high-resolution timer, i.e., `rdtsc` and `rdpru`. Based on the timer, we successfully break KASLR within about 10 seconds.

E. Experiments

1) E1: Fingerprinting websites

How to: We apply SegScope to probe the website-induced interrupts while opening a website.

Results: As shown in the Table IV in the paper, SegScope has inferred website visits with a high success rate (92.4% on Chrome and 87.4% on Tor Browser in our tested machine).

2) E2: Enhancing Spectral

How to: As the Spectral side channel attack is noised by interrupts, we exploit SegScope to effectively filter the noised measurements and discard them.

Results: As shown in Figure 9 in the paper, SegScope removes the impacts of interrupts and significantly reduces the error rate of Spectral by 56×.

3) E3: Breaking KASLR

How to: We leverage SegScope to craft a fine-grained timer, as timer interrupts can be used as clock edges to build a clock interpolation scheme. We further use the crafted SegScope-based timer to measure the timing difference when accessing/prefetching a mapped/unmapped memory address.

Results: As shown in Table III in the paper, SegScope-based timer achieves the same level of timing granularity as the high-resolution timers, i.e., `rdtsc` and `rdpru`. With this timer-enabled timing side channel, KASLR has been derandomized within about 10 seconds as shown in Table VII in the paper.

Notes: If SegScope-based timer does not work well, please check the grub configuration and ensure that it does not enable the tickless-full mode. This mode is disabled on Ubuntu systems by default.