Partial-SMT: Core-scheduling Protection Against SMT Contention-based Attacks

Xiaohui Wu[1,3], Yeping He[1,3], Qiming Zhou[3], Hengtai Ma[3], Liang He[3], Wenhao Wang[2], and Liheng Chen[1,3]

[1]University of Chinese Academy of Sciences, liheng@nfs.iscas.ac.cn
[2]SKLOIS, Institute of Information Engineering, CAS, wangwenhao@iie.ac.cn
[3]Institute of Software, CAS, {wuxiaohui, yeping, qiming, hengtai, heliang}@iscas.ac.cn

***Abstract*—Numerous recent works in side-channel attacks have experimentally shown that Simultaneous Multi-Threading (SMT) inherently has a broader attack surface as it exposes more microarchitecture components per-core than cross-core. Existing mechanisms that protect against these attacks either incur high execution costs or are ineffective against certain attack variants.**

In this paper, we propose Partial-SMT, a system based on core-scheduling that protects security-critical programs from all contention-based attacks due to SMT. Partial-SMT allocates some complete physical cores for the exclusive use of the individual applications and provides a user-level threading library linked into each application to control the placement of their threads on dedicated cores, thereby preventing the attacker from accessing shared CPU resources simultaneously on the victim's core. The key insight is that by limiting ourselves to SMT contention-based side channels, we can translate the protection into an allocation policy that allocates or frees computing resources with a granularity of one physical core. Security-critical applications can be implemented on-demand and coexist with existing applications. We demonstrate that Partial-SMT effectively defeats typical SMT contention-based attacks. We modify AES and SPEC 2006 to use Partial-SMT, and they all incur the slight negligible performance overhead.

***Keywords*-Simultaneous Multi-Threading, Core scheduling, Defenses, Side Channels**

I. Introduction

Microarchitectural side channels have been known for decades, with new techniques in computer infrastructure emerging every year [1], [2], they tend to be more difficult to mitigate. Simultaneous Multi-threading (SMT) allows multiple threads to execute on the same core simultaneously while sharing the same resources. It improves hardware resource utilization and system throughput with a modest increase in processor complexity, area, and power consumption [3]. These advantages have led to widespread adoption by microprocessor vendors.

However, numerous works demonstrate that SMT facilitates a set of eviction-based side channels, which mainly examine changes made by a victim's execution to the state of shared microarchitectural components such as caches [4]–[8], cache directories [9], and branch predictors [10], [11]. Furthermore, it comes with new contention-based attacks [12]–[17]. That is, measurable interference can occur across security boundaries on resources that are competitively shared, possibly leading to information leakage. This paper aims at understanding and addressing the security threats from contention-based side channels in the SMT processor, and deriving a novel defense mechanism.

Modern processors provide strong isolation guarantees at the architectural level, while none at the microarchitectural level. Defending against SMT contention-based side-channels is challenging, which even has motivated OpenBSD developers to abandon SMT completely. Countermeasures can operate at the hardware, the system, or the software level. Hardware-based mitigations address cache [18]–[21], branch predictor [10], [22], TLB logic [23] and execution logic [24] respectively. Countermeasures are as varied as the attacks themselves. For example, SMT-COP [24] eliminates execution logic side-channels in SMT processors by partitioning functional units and/or associated issue ports between threads running on the same core. These proposals are ineffective against certain attack variants such as Zomieload [12] and RIDL [13]. Furthermore, they are orthogonal and should be combined to create a comprehensively leakage-free SMT architecture.

At the software level, Wang [25] proposed to selectively disable SMT on processors running sensitive workloads but did not pursue implementation. Similarly, Zhang [26] designed DDM, which writes the processes' security requirements to the CPU register sets, and the Operating System (OS) calls the HLT instruction to dynamically turn on/off the hyperthreading according to the register values. These mechanisms focus on how to dynamically disable SMT during the execution of sensitive processes. However, turning off SMT is not an option, because it hampers applications' performance, making the system less suitable for data-intensive security computing.

Yarom [17] suggested only allowing threads from the same protection domain to share a physical core. To this end, Chen [27] and Oleksenko [28] proposed different approaches to verifying that two enclave threads are reliably scheduled on the same physical core in Intel Software Guard Extensions (SGX). If the threads were scheduled by OS on different cores, they concluded that the enclave was under attack and terminated it. The overhead could be high as the approaches are synchronized, i.e., two threads in a pair can make progress only if both are running. Otherwise, if one thread is descheduled, the other one has to stop and wait.

Inspired by these works in the SGX threat model, we

978-1-6654-0392-4/20/$31.00 ©2020 IEEE
DOI 10.1109/TrustCom50675.2020.00059

present Partial-SMT, a system based on core-scheduling to allocate or free computing resources with a granularity of one core. A Partial-SMT protected application can have exclusive use of some physical cores. Meanwhile, threads from other applications can't be scheduled onto these cores. To the best of our knowledge, no prior work has taken the strategy of core scheduling to mitigate SMT contention-based attacks.

Partial-SMT changes the scheduling abstraction for applications from one based on threads to one based on cores. It consists of two components: the *core scheduler* is responsible for core management and schedules cores with allocation policies; the *user-level scheduler* provides a user-level thread library so that applications can control the placement of their threads on dedicated cores rather than being scheduled by the OS.

We evaluated the implementation and proved that Partial-SMT effectively defends against contention-based attacks due to SMT. We modified AES and SPEC 2006 to use Partial-SMT and performance evaluation showed they all incur slight negligible performance overhead.

To summarize, our contributions are as follows:

- We propose Partial-SMT, a new defense mechanism to mitigate SMT contention-based attacks, by leveraging core-scheduling to allocate dedicated physical cores to each application. This insight translates the protection into an allocation policy that allocates or frees computing resources with a granularity of one physical core.
- By only allowing threads from the same application to run simultaneously on a core, Partial-SMT retains the performance benefits of SMT in a better manner compared with these mechanisms which selectively turning SMT off.
- Partial-SMT needs no kernel modifications and protected applications can coexist with traditional applications.

II. Background

The work in this paper relies on the complex interplay between software and the underlying hardware. In the following, we provide the background information necessary to understand SMT contention-based attack requirements and the defense we proposed.

A. Simultaneous Multi-Threading (SMT)

SMT has become widespread via its introduction (under the name “Hyper-Threading”) into Intel Pentium 4 processors [3]. Hyper-Threading greatly increases the degree of instruction-level parallelism by having at least two logical cores per physical core. We consider the case of two logical cores in this paper because it is the most common case. Overall, the microarchitecture resources in a physical processor are owned in three ways.

- A few small data structures such as the architectural register file and the instruction pointer are replicated for each logical processor.
- Several buffers are shared by limiting the use of each logical processor to half the entries. These are partitioned resources.
- Most resources are fully shared to improve the dynamic utilization of the resource, including caches and all the execution units [29], as shown in Fig. 1. These resources are competitively shared between the logical processors.

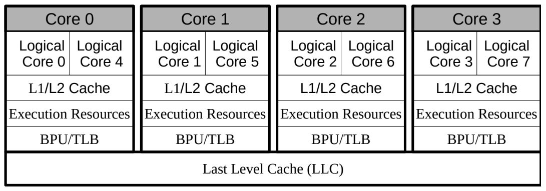

Fig. 1. Intel i7 core processor

B. Eviction- vs. Contention-based Attacks

Existing microarchitectural side-channel attacks are largely eviction-based. That is, they use evictions to bring the target component to a known state. After the secret operation, they can then examine the state to infer any changes that leak information about the secret operation. For example, in FLUSH+RELOAD, the attacker evicts a shared cache line and later checks whether it is reloaded by the victim in a secret operation. While attacks on CPU caches are the most common, attackers may equally target TLBs [23] and branch predictor [10].

Some recent attacks [14], [15] are contention-based. The attacker can also leak secret information by exploiting contention or conflicts on microarchitectural components. The attacker simply measures the bandwidth (i.e., operations per second) over time and, by observing its fluctuations, can infer information about the victim's operation. The high-level strategy in contention-based attacks is to observe the victim simultaneously, while that in eviction-based attacks is checking whether the microarchitectural state has been changed after the victim's operation. Contention-based side channels are unique to SMT settings.

The concurrent threads share a pool of functional units (FUs) that are allocated alternately every cycle. An attacker can create a timing side-channel by issuing a large number of instructions to a specific execution resource. By measuring the time required to execute a series of said instructions, the attacker can infer whether a victim executes instructions that use the same execution resource. PortSmash [14] and SMoTherSpectre [15] utilize this approach to extract private keys from OpenSSL. Yarom et al. [17] presented CacheBleed which exploits cache-bank conflicts on the Sandy Bridge microarchitecture. The authors exploit the fact that cache banks can only serve one request at a time. Later, Moghimi et al. [16] improved the previous work and exploited false dependency of memory read-after-write. Recently, there has been much press about Zomieload and RIDL, which leak data from the Line-fill Buffer (LFB) and Load Ports (LP) respectively. LFB and LP are competitively shared between hyperthreads. Evtyushkin et al. [10] exploit the observation that an adversary can create the branch target buffer (BTB) collisions which can impact the timing of the attacker's code. It allows the attacker to identify the locations of known branch instructions in the address space of the victim process or the kernel. These are typical SMT contention-based side channels.

III. Threat Model

We consider the computer system with SMT processors. Here, we outline a threat model in which an adversary aims to extract sensitive data, such as cryptographic keys, from a victim program through contention-based side channels. The first goal of contention-based attacks is to ensure the co-residency of the victim and the attacker on the same physical core at the same time. The threads of each application are supposed to trust each other.

Since our focus in this paper is on the security risks of SMT, we assume the software system (like the OS kernel and the modules enforcing security policies) is free of software vulnerabilities. In this case, we assume that appropriate security policies can be enforced to prevent the attacker from accessing shared SMT resources during the victim's execution. These resources range from the shared cache, BPU to execution resources, and so on.

IV. Design

We investigate a different strategy for mitigating SMT contention-based side channels: adjusting core scheduling to dynamically isolate several physical cores for sensitive code to run on. With core-scheduling, it is possible to forbid mutually untrustworthy programs from sharing a physical core, without the permanent loss of parallel computations.

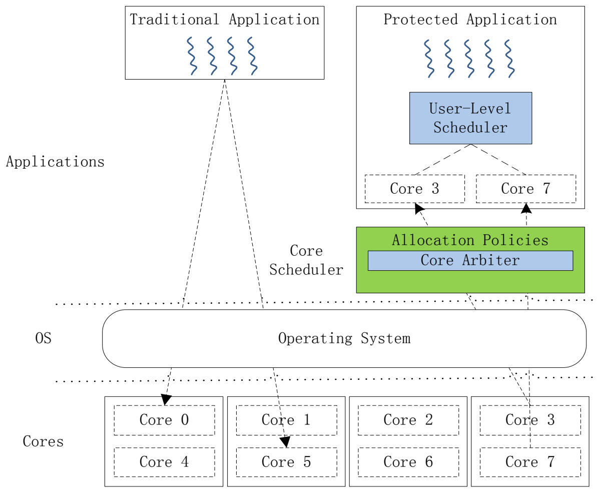

Fig. 2. The overview of Partial-SMT

A. Partial-SMT Overview

As shown in Fig. 2, protected applications can coexist with traditional applications. Under the traditional threading model, an application spawns multiple kernel threads and multiplexes user threads over all possible cores. Threads from different applications are considered equivalently when the OS makes scheduling decisions, so they may be scheduled on the same physical core.

The right part of Fig. 2 demonstrates the overall architecture of Partial-SMT. It consists of two components: the core scheduler is designed to manage cores and allocate dedicated physical cores to each application; the user-level scheduler allows applications to perform their own scheduling of threads onto these cores. These schedulers can leverage composable two-level scheduling frameworks such as Akaros [30] and Parlib [31], but they don't appear to have reached a level of maturity. Another example is Arachne [32], which allows each application to have exclusive use of a few cores. It shares our goal of allocating dedicated cores to applications, but we take a more special-purpose approach by coupling the core-scheduling to SMT side-channel defenses.

In the following, we will discuss Partial-SMT's two components, and present the design considerations.

B. The Core Scheduler

The core scheduler is responsible for allocating cores with the granularity of a physical core. It consists of a central core arbiter and allocation policies for a protected environment. The core arbiter explicitly allocates cores to an application, which makes the allocation step easier. What's more, we particularly focus on the following two issues.

Firstly, how to schedule a core simultaneously with the sibling logical core throughout its lifecycle managed by the core scheduler?

The core scheduler collects request information from each application about how many cores it needs and schedules cores with allocation policies, including allocating, preemption and re-allocation of cores, and releasing cores. The logic underlying this process is illustrated as a flowchart in Fig. 3. Steps marked in red stand for security improvements.

- On receiving the request from an application, the number of cores needed is expected to be even. If not, the core scheduler issues a warning and refuses to allocate any cores. This policy can prevent the misbehaving application from threatening other protected applications.
- If there are some idle cores, the core scheduler gradually allocates cores to the application with the best effort. To ensure that a physical core is allocated as a unit, we first pick a core and make sure the sibling core is idle. Otherwise, we have to repeat this process.
- If there is no idle core, the core scheduler should decide whether the preemption is required using a simple priority mechanism. If required, the core scheduler asks the low priority application to release cores but does not unilaterally preempt cores. The preempted application's threads are suspended. Similarly, we treat a physical core as a unit when re-allocating cores to applications. If no preemption is required, the new application has to wait for idle cores.
- When a core is no longer needed by any Partial-SMT application, the core scheduler checks the sibling core's

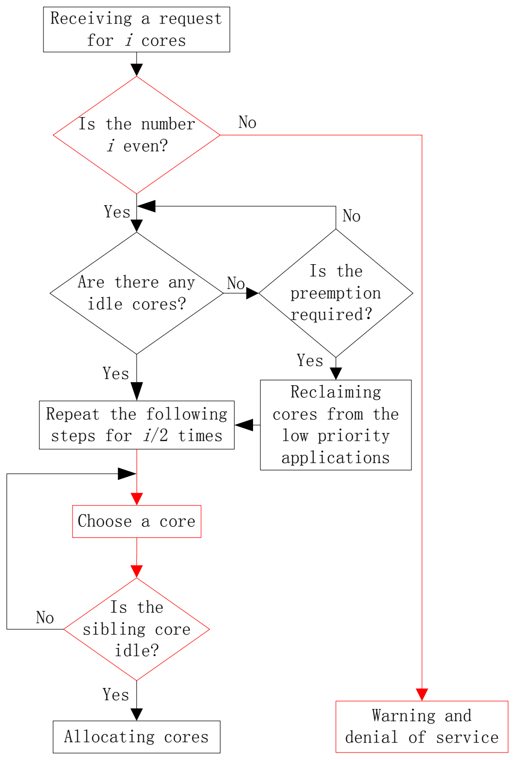
Receiving a request for i cores
Is the number i even?
No
Yes
Are there any idle cores?
No
Is the preemption required?
No
Yes
Yes
Reclaiming cores from the low priority applications
Repeat the following steps for $i/2$ times
Choose a core
Is the sibling core idle?
No
Yes
Allocating cores
Warning and denial of service

Fig. 3. Core-scheduling with allocation policies

state and does not add the pair of cores back to Linux kernel until the physical core is idle.

- Applications can request several cores, not designated physical cores. The main reason is that our method only needs to monopolize a physical core. No matter which one. Specifying physical core id may cause latency and affect performance.

Secondly, the defense mechanism guarantees that threads from any other application can't be scheduled onto these cores. Different from core allocation, defense mechanisms require exclusive use by an application - a separate allocation step is still needed to actually hand them out. Once cores are allocated to an application, they are completely isolated from other processes.

The core scheduler uses Linux *cpuset* mechanism to manage and allocate cores; only the kernel threads created by Partial-SMT run on these cores. Unmanaged cores are scheduled by OS, so traditional applications are supposed to use the unmanaged cores.

A frequently proposed approach to pinning threads to cores is setting thread affinities, thereby avoiding kernel multiplexing. However, one application can't pin its threads by setting thread affinities to the managed cores. According to the description of cpuset in Linux Programmer's Manual, if a process's cpuset placement conflicts with the sched_setaffinity scheduling affinity mechanism, cpuset placement is enforced. This can result in sched_setaffinity calls returning an error. It is verified in the security evaluation in Section 6.

According to the allocation policies, a Partial-SMT application requests an appropriate number of cores, and the core scheduler selects pairs of cores that are idle. Therefore, hardware resources on a physical core are shared only by threads from the same application. If the attacker uses Partial-SMT, he can just request and use the resources on his own physical core. This scheme also eschews the security problem of whether an application is trusted.

C. The User-level Scheduler

Controlling over cores introduces new problems for applications that they must decide which thread to place on which core. It is common to all two-level scheduling frameworks [31], [32]. Therefore, applications can employ the user-level scheduler known from prior works to multiplex their own set of threads on top of those cores.

The Partial-SMT application can implement any custom thread placement except creating new kernel threads outside Partial-SMT. For example, using std::thread, as these threads will run on the unmanaged cores and co-scheduling is violated. For single-threaded applications, the user-level scheduler creates a shadow kernel thread running on the co-located logical core.

V. Implementation

Partial-SMT is implemented on Arachne, a user-level thread management system that provides applications visibility into cores and control over the cores they are using. Besides these benefits, we make two security enhancements: allocation policies and custom core policy.

A. Arachne Overview

Fig. 4 demonstrates the overall architecture of Arachne. There are mainly three components working together. The core arbiter consists of a stand-alone user process plus a small library linked to each application. The Arachne runtime and core policies are libraries linked into applications.

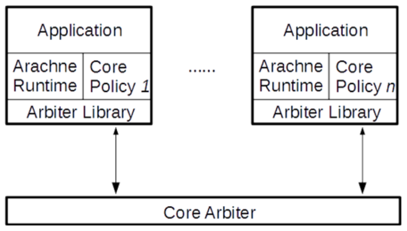

Fig. 4. The overall architecture of Arachne

1) The Core Arbiter: The core arbiter is responsible for claiming control over the system's cores and allocating them to applications. It divides cores into two groups: managed cores and unmanaged cores. Managed cores are allocated by the core arbiter, only the kernel threads created by Arachne run on these cores. Unmanaged cores continue to be scheduled by Linux. The core arbiter reserves at least a core as the unmanaged core. This scheme allows Arachne applications to coexist with traditional applications whose threads are managed by

Linux kernel. Arachne applications receive preferential access to cores.

When the core arbiter starts up, it creates one cpuset for unmanaged cores (the *unmanaged cpuset*) and places all of the system's cores into that set. It then assigns every existing kernel thread (including itself) to the unmanaged cpuset; any new threads spawned by these threads will also run on this cpuset.

The core arbiter collects request information from applications and divides the available cores among competing applications from highest priority to lowest. To allocate cores to an Arachne application, the arbiter removes cores from the unmanaged cpuset and creates one managed cpuset corresponding to each core, which initially has no threads assigned to it. Once cores are allocated to an application, they belong to the application and will not be preempted from the application without warning. When application requirements change, the core arbiter adjusts the core allocations.

2) The Arachne Runtime and Core Policy: The Arachne runtime contains user thread abstraction similar to thread packages based on kernel threads, including thread creation and deletion, locks, and condition variables. The Arachne runtime creates several kernel threads and uses them to implement user threads, which are used by Arachne applications.

Arachne uses kernel threads as a proxy for cores. Each kernel thread created by the runtime executes on a separate core and has exclusive access to that core when it is running. When the arbiter allocates a core to an application, it unblocks one of the application's kernel threads on that core; when the core is removed from an application, the kernel thread running on that core blocks. The Arachne runtime runs a simple dispatcher in each kernel thread, which multiplexes several user threads on the associated core. It does not preempt a user thread once it has begun executing.

The core policy working together with the Arachne runtime determines how cores are used by that application. To help applications request an appropriate number of cores for handling their current load, the Arachne runtime gathers performance information and a core load estimator is needed. Core policies precisely control how threads are scheduled onto these cores.

B. Security Enhancements

Arachne provides two particular features that make our implementation more efficient. First, it implements core scheduling, which significantly simplifies dedicated-cores allocating. We use the core arbiter to schedule cores with allocation policies to achieve the goal of allocating cores with the granularity of a physical core. Second, it contains a core policy framework that provides interfaces for applications to precisely control how threads are scheduled onto dedicated cores. Custom core policy is required to be consistent with allocation policies.

In summary, the core scheduler in Partial-SMT consists of the core arbiter with allocation policies. Custom core policy, the load estimator, and original Arachne runtime work together to implement the user-level scheduler.

1) Allocation Policies: Assuming the system has N physical cores, the core topology is similar to Fig. 1. A pair of logical cores composes one physical core. Core *0* and its sibling core are at least reserved for unmanaged cpuset. The core scheduler allocates cores from the highest priority to the lowest, so low-priority applications may receive no cores. If there are not enough cores for all requests at a particular level, the core scheduler randomly chooses one from the requesting applications. It's different from the original Arachne arising from our core allocation policies. We analyzed the source code of Arachne and implemented allocation policies by modifying the code related to core-scheduling.

2) Custom core policy: The core policy framework provides interfaces and functions for writing new core policies. Each application can choose its own core policy or the default one. The number of cores required by a Partial-SMT application is determined by security requirements rather than the workloads, thus we make some adjustments to the core load estimator. If the number of cores the application needs turns out to be odd, the application except for the single-threaded application will request one less. Partial-SMT makes a reasonable trade-off between security and resource utilization.

C. Safeguarding Security-critical Programs

From the programmer's perspective, the application code has to be modified to have exclusive use of physical cores. Partial-SMT performs fine-grained core allocation at the OS level and provides the programmers with a small user-thread library written in C++. A programmer calls these APIs explicitly. In the typical programming model, the program firstly determines how many cores it needs; then it initializes the library, adds a bunch of threads, and controls the placement of its threads on those cores. In the end, the program waits for all threads to terminate. Programmers can protect important control-flows, such as encryption algorithms.

VI. Evaluation

In this section, we evaluate the security and performance properties of Partial-SMT. All experiments were conducted on a ThinkCentre machine. Table 1 describes the configuration of the machine.

TABLE I
CONFIGURATION OF THE MACHINE

Processor model	Intel (R) Core(TM) i7-6700
Microarchitecture	Skylake
Clock frequency	3.4GHz
Physical cores	4
Core Number	8
Kernel version	Kernel-4.15.0 (Ubuntu16.04)
SMT support	support

A. Security Evaluation

Firstly, we launched a dummy Partial-SMT application as the victim and verified how effective Partial-SMT is at ensuring exclusive use of a physical core, as shown in Fig. 5.

Then we pretended to be an attacker and tried to co-locate with the victim. A frequently proposed solution to running an adversary sibling thread on the same physical core is setting thread affinities. In the experiment, the attacker got such an error "taskset: failed to set pid 0's affinity: Invalid argument" during the victim's execution, which means the attacker can't pin its threads to the cores dedicated to the victim. The attacker using *cpuset* also failed.

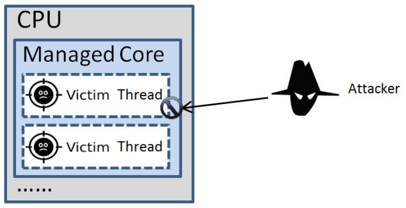

Fig. 5. Prototype verification

Based on the critical verification, we modified the victims to use Partial-SMT in typical SMT contention-based attacks such as PortSmash, SMoTherSpectre, and ZombieLoad. As the first requirement of the attacks is not satisfied, there is no possibility for the attackers to perform any concurrent attacks on the core's resources shared between hyperthreads. The above attacks in user-space are all mitigated.

B. Performance Evaluation

1) Cryptographic Library: AES is one of the most popular block ciphers used in cryptography. As Partial-SMT is primarily targeted at multi-threaded applications, we modified AES-ECB to use Partial-SMT in a parallel mode of operations of the cipher. The performance is measured on varying numbers of cores and threads. All results were compared with Linux Pthread. The reported results were averaged over 10 runs. We show the trending values for large buffers (from 1Mbyte to 100Mbyte) in terms of cycles per byte from two aspects: thread scheduling and total execution time of the application. Cycles per byte (CPB) is a unit of measurement which indicates the number of clock cycles a microprocessor will perform per byte of data processed in an algorithm. Therefore, smaller is better.

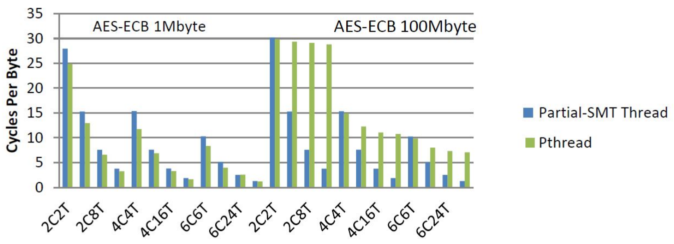

Fig. 6. Thread scheduling of different modes

Fig. 6 contains two parts whose values differ from the size of buffers: 1Mbyte is on the left, 100Mbyte is on the right.

When the buffer is 1Mbyte, the execution time of a single thread on average is less than 20ms. We can see that the trending values for Pthread and Partial-SMT are the same, and Partial-SMT thread is slightly worse. It does not cause a significant slowdown because the core allocation policies are relatively small, including the surrounding code for the associated operations.

When the buffer is 100Mbyte and the number of cores is fixed, the thread duration of Pthread does not decrease when the number of threads increases. The main reason is that Linux thread scheduling mechanism is based on time-slice, which is 5ms-800ms. As a result, the thread duration is longer than its actual execution time. As for Partial-SMT, the thread duration is actually its execution time. Partial-SMT does not preempt a user thread once it has begun executing, so the thread can finish its execution as soon as possible. This feature is critical for defending against time-sharing side-channels.

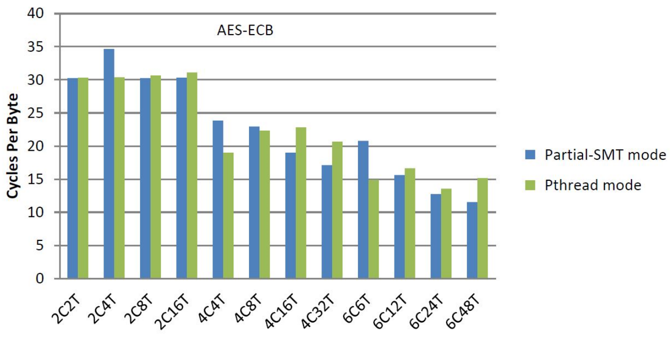

Fig. 7. The execution time of AES-ECB in both modes

As indicated in Fig. 7, with the number of cores increasing, the cycles/byte decreases accordingly in both modes. When the number of threads is small, the cycles/byte in Partial-SMT mode is slightly higher than that of Pthread, because the load balance mechanism without time-slice is slightly worse. When the number of threads exceeds twelve, the Partial-SMT mode is better.

2) Comparison between Partial-SMT and other methods: DDM [26] is the latest research on mitigating SMT side channels. Fig. 8 demonstrates applications executing with DDM and Partial-SMT respectively. With DDM, security-critical applications request to set the other logical core to the HALT state, so both applications are nonparallel and require more time to complete. In the figure below, Partial-SMT allows threads from the same application to share a core simultaneously, thus execution time are reduced. Partial-SMT achieves security isolation without disabling SMT. For single-threaded programs, Partial-SMT is equivalent to DDM.

To evaluate the performance of DDM and Partial-SMT, we randomly created mixed workloads from combinations of the SPEC 2006 benchmarks compiled for the Alpha architecture. We pair one floating-point (fp) and one integer-intensive (int) benchmark. The fp and int workloads are expected to use different sets of functional units, so the processors running mixed workloads can achieve efficient pipeline utilization and higher instruction throughput. Evaluation results in Fig. 9 show

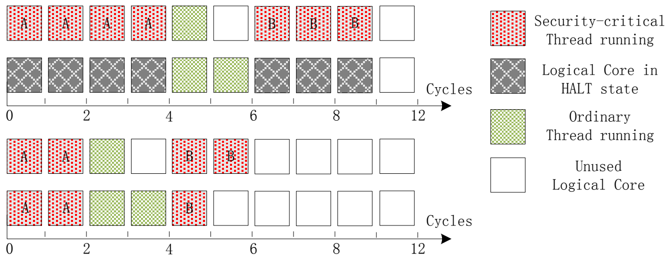

Fig. 8. Top: applications executing with DDM. Bottom: security-critical applications executing with PartialSMT.

that Partial-SMT improves performance by 12% compared with the DDM mechanism.

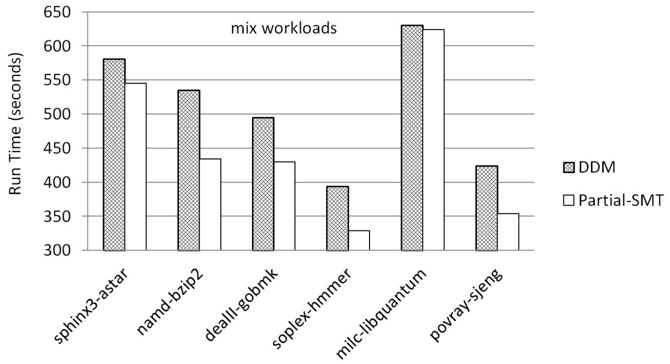

Fig. 9. Performance of SPEC 2006 benchmarks with DDM and Partial-SMT.

C. Extensions

While Partial-SMT was shown to be practical to mitigate SMT contention-based attacks, it may not be effective at protecting eviction-based attacks. When control passes from the yielding thread to another runnable thread on the same core quickly, an attacker may observe its residual state in the cache, branch predictor, TLB, or other hardware structures. We investigate incorporating a selective per-core state-cleansing into thread scheduling as supplementary, which was proposed as a method for OS to protect themselves by periodically cleansing a fraction of the I-cache, D-cache, and branch predictor states [33].

VII. Related Work

A. Soft Isolation in Clouds

Cloud computing relies on resource sharing to achieve high resource utilization, which also introduces the threat of malicious VMs abusing the scheduling of shared resources. The straightforward solution is hard isolation that dedicates hardware to each VM. However, this comes at the cost of reduced efficiency. They investigate the principle of soft isolation: reduce the risk of sharing through better scheduling.

Varadarajan et al. [33] proposed a minimum run time (MRT) guarantee for VM virtual CPUs that limits the frequency of preemptions that can effectively prevent existing Prime+Probe cache-based side-channel attacks. They also integrate a simple per-core CPU state cleansing mechanism to provide further protection.

Liu et al. [34] reveal that the root cause of such attacks is the constant sharing patterns of hardware resources between VMs. They proposed a scheduling-based mechanism called Shuffler schedulers, which distributes CPU time to vCPUs with equal probability, would reduce the overall vulnerable probability of the system.

In order to avoid asynchronous cache side channels between hyperthreads, STEALTHMEM [35] did not assign the logical cores of the same physical core to different VM. Some widely used hypervisors such as Hyper-V already implement this policy. Partial-SMT is a different mechanism to achieve the isolation and scheduling of applications at a finer granularity.

B. Core Scheduling

Numerous works have been developed to achieve high throughput and low latency, including dividing the cores so that each application has exclusive use of a few cores. Scheduler activations [36] are similar to Arachne in that they allocate processors to applications to implement user-level threads efficiently. Akaros [30] and Parlib [31] follow in the tradition of scheduler activations. Akaros is a new operating system that allocates dedicated cores to applications and makes all blocking system calls asynchronous; Parlib is a framework for building user schedulers on top of dedicated cores. Akaros offers functionality analogous to Arachne's core arbiter. Similar to Arachne, these works addressed performance rather than security. Negotiating over cores would be easier for applications to coordinate their parallelism, but they do not eliminate possible side-channels.

VIII. Conclusion

In this paper, we present Partial-SMT, a system based on core-scheduling that enables applications to request cores rather than threads. It also guarantees that an application with security requirements can have exclusive use of physical cores and threads from other applications can't be scheduled onto these cores. The security analysis and empirical evaluation prove that Partial-SMT effectively mitigates SMT contention-based attacks. Performance evaluation with AES and SPEC 2006 shows that implementing them in software is possible with reasonable system overhead.

Acknowledgment

We are grateful to Jingzheng Wu, Qiuming Tao, Henry Qin, Xuanyu Liu, Tianlin Huo, Shaofeng Lin for their helpful comments. This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences under Grant No. XDA-Y01-01. Wenhao Wang was supported by National Natural Science Foundation of China (Grant No.61802397).

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

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

[3] O. Aciicmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*. IEEE, 2007, pp. 80–91.

[4] Y. Yarom and K. Falkner, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.

[5] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+ abort: A timer-free high-precision l3 cache attack using intel tsx," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 51–67.

[6] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

[7] N. Lawson, "Side-channel attacks on cryptographic software," *IEEE Security & Privacy*, vol. 7, no. 6, pp. 65–68, 2009.

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

[9] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 888–904.

[10] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[11] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 693–707, 2018.

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

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

[14] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 870–887.

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

[16] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 538–570, 2019.

[17] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[18] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2016, pp. 406–418.

[19] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2008, pp. 83–93.

[20] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 347–360.

[21] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 775–787.

[22] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "Brb: Mitigating branch predictor side-channels." in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 466–477.

[23] S. Deng, W. Xiong, and J. Szefer, "Secure tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 346–259.

[24] D. Townley and D. Ponomarev, "Smt-cop: Defeating side-channel attacks on execution units in smt processors," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 43–54.

[25] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 2006, pp. 473–482.

[26] Y. Zhang, Z. Zhu, and D. Meng, "Ddm: A demand-based dynamic mitigation for smt transient channels," *arXiv preprint arXiv:1910.12021*, 2019.

[27] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 178–194.

[28] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting sgx enclaves from practical side-channel attacks," in *2018 USENIX Annual Technical Conference (USENIXATC 18)*, 2018, pp. 227–240.

[29] D. Manual, "Intel 64 and ia-32 architectures software developers manual," 2016.

[30] B. Rhoden, K. Klues, D. Zhu, and E. Brewer, "Improving per-node efficiency in the datacenter with new os abstractions," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011, pp. 1–8.

[31] K. A. Klues, "Os and runtime support for efficiently managing cores in parallel applications," Ph.D. dissertation, UC Berkeley, 2015.

[32] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: core-aware thread management," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 145–160.

[33] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-vm side-channels," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 687–702.

[34] L. Liu, A. Wang, W. Zang, M. Yu, M. Xiao, and S. Chen, "Shuffler: Mitigate cross-vm side-channel attacks via hypervisor scheduling," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2018, pp. 491–511.

[35] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 189–204.

[36] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 53–79, 1992.