SPECWANDS: An Efficient Priority-Based Scheduler Against Speculation Contention Attacks

Bowen Tang, Chenggang Wu, Pen-Chung Yew, *Life Fellow, IEEE*, Yinqian Zhang, Mengyao Xie, Yuanming Lai, Yan Kang, Wei Wang, Qiang Wei, and Zhe Wang

***Abstract*—Transient execution attacks (TEAs) have gradually become a major security threat to modern high-performance processors. They exploit the vulnerability of speculative execution to illegally access private data, and transmit them through timing-based covert channels. While new vulnerabilities are discovered continuously, the covert channels can be categorized to two types: 1) *Persistent Type*, in which covert channels are based on the layout changes of buffering, e.g., through caches or TLBs and 2) *Volatile Type*, in which covert channels are based on the contention of sharing resources, e.g., through execution units or issuing ports. The defenses against the persistent-type covert channels have been well addressed, while those for the volatile-type are still rather inadequate. Existing mitigation schemes for the volatile type such as *Speculative Compression* and *Time-Division-Multiplexing* will introduce significant overhead due to the need to stall the pipeline or to disallow resource sharing. In this article, we look into such attacks and defenses with a new perspective, and propose a scheduling-based mitigation scheme, called SPECWANDS. It consists of three priority-based scheduling policies to prevent an attacker from transmitting the secret in different contention situations. SPECWANDS not only can defend against both interthread and intrathread-based attacks but also can keep most of the performance benefit from speculative execution and resource-sharing. We evaluate its runtime overhead on SPEC 2017 benchmarks and realistic programs. The experimental results show that SPECWANDS has a significant performance advantage over the other two representative schemes.**

***Index Terms*—Resource contention, scheduling strategy, simultaneous multithreading (SMT), transient execution attack (TEA).**

Manuscript received 15 December 2022; revised 16 April 2023; accepted 26 May 2023. Date of publication 8 June 2023; date of current version 22 November 2023. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 61902374, Grant U1736208, and Grant 62272442; and in part by the Innovation Funding of ICT, CAS under Grant E161040. This article was recommended by Associate Editor S. Ghosh. *(Corresponding author: Zhe Wang.)*

Bowen Tang, Mengyao Xie, Yuanming Lai, Yan Kang, and Wei Wang are with SKLP, Institute of Computing Technology, CAS, Beijing 100190, China, and also with the University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: tangbowen@ict.ac.cn; xiemengyao@ict.ac.cn; laiyuanming@ict.ac.cn; kangyan@ict.ac.cn; wangwei2021@ict.ac.cn).

Chenggang Wu and Zhe Wang are with SKLP, Institute of Computing Technology, CAS, Beijing 100190, China, also with the University of Chinese Academy of Sciences, Beijing 100049, China, and also with Zhongguancun Laboratory, Beijing, China (e-mail: wucg@ict.ac.cn; wangzhe12@ict.ac.cn).

Pen-Chuang Yew is with the Computer Science and Engineering Department, University of Minnesota-Twin Cities, Minneapolis, MN 55455 USA (e-mail: yew@umn.edu).

Yinqian Zhang is with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: yinqianz@acm.org).

Qiang Wei is with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China (e-mail: prof_weiqiang@163.com).

Digital Object Identifier 10.1109/TCAD.2023.3284290

I. INTRODUCTION

SIMULTANEOUS multithreading (SMT), also known as hyper-threading, has become an important feature on modern high-performance processors. It allows multiple threads to run simultaneously on a physical core and share the resources in the instruction pipeline to cover the slack caused by the stalled threads, thereby improving the efficiency and throughput of the pipeline. However, due to the resource sharing on the pipeline, SMT can introduce new security vulnerabilities. Multiple threads may compete for shared resources and interfere with each other's execution under the common first-come–first-served (FCFS) policy. If an attacker can control one thread, he/she can figure out the execution state of other threads according to the difference in its execution time and then infer some private information, which is the so-called SMT contention-based side channel [1], [2], [3].

Such attacks can be mitigated by scheduling mutually distrusting threads on different physical cores. However, the recent transient execution attacks (TEAs) [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], such as the well-known Meltdown and Spectre, cannot be defended using such an approach. It is because an attacker can spawn and control multiple threads in some attack scenarios. The system cannot distinguish which ones are malicious when resource contention occurs.

Take an example from the work in [14]. It installs a malicious plug-in running on a sandboxed browser to launch such an attack. Assume two threads are both created by the plug-in and run on a physical core with SMT. One of them, called *Trojan*, leverages speculative execution to access a secret outside the sandbox. Trojan thread then issues a burst of requests to keep the resource busy if the bit to be transmitted is "1," and leaves the resource idle if it is "0" instead. The other thread, called *Spy*, then attempts to acquire the same resource and infer the secret value based on whether the resource is busy or not by measuring its acquisition time. In such an attack scenario, the resource contention is used as a covert channel to transmit the illegally obtained data from speculative accesses.

To block such covert channels, researchers have proposed a variety of defenses. One well-known approach is speculation compression (SC) [15], [16], [17], which delays the

speculative data to be propagated in the pipeline. That is, if an operand of an instruction comes from a speculative instruction, it will not be issued until the speculative instruction it depends on has become nonspeculative. This approach prevents the data of a speculative instruction from being transmitted via the covert channel. However, it will obstruct the speculative execution (e.g., from branch prediction) that has been the cornerstone of a modern CPU to improve its performance. Based on our own simulation results, its performance hit can be as high as 15% on a typical CPU with SMT. The other approach is to partition the resources in the spacial or temporal dimensions, such as time-division-multiplexing (TDM), to avert interthread interference [18], [19], [20]. Although this approach doesn't obstruct the speculative execution, it contradicts the original intention of resource sharing using SMT, and will also incur non-negligible performance overhead (e.g., more than 12% overhead according to our evaluation).

Besides the well-known resource contention in SMT, researchers have found the resource contention also exists in the single thread scenario, and it can be abused as a new convert channel [21], [22]. Such attacks exploit the contention caused by multiple instructions scheduled in the same scheduling window within a thread. These instructions have no dependence among them, and can be issued simultaneously in the pipeline that supports *multi-issuing* and *out-of-order* execution. The attacker can use the contention at the issuing port as a covert channel. Moreover, he/she only needs to launch one single thread without the need for interthread synchronization. This kind of attacks, which leverage intrathread covert channels, thus have a higher probability of success. TDM-based defense approaches are basically ineffective and are difficult to harden for this scenario.

In order to defend against the TEAs that exploit resource contention as a covert channel, which we call speculative contention attacks (SCAs), while minimizing the performance overhead, we proposed a secure scheduling scheme for shared resources, called SPECWANDS. The main idea behind SPECWANDS is to batching multiple operations into groups, and ensure that each group can share a resource without any interference from other groups in the same or different threads. To apply such an idea to different scenarios, we use the following three priority-based scheduling policies.

1) *Nonspeculative Operations have a higher Priority (NOP):* SPECWANDS assigns a higher priority to nonspeculative operations. They not only can be scheduled ahead of all other speculative operations but also allowed to preempt the speculative operations already occupying the resource. This policy can be used as a guideline for delimit the boundaries of each group, i.e., each nonspeculative operation is the header of each group.
2) *Last-owner-thread's Operations have a higher Priority (LOP):* When there are multiple speculative operations from different threads, SPECWANDS assigns a higher priority to the speculative operations whose owner thread used the resource most recently. This policy further clusters the speculative operations into groups based on their owner threads, i.e., their group header nonspeculative operations.
3) *Earlier Operations have a higher Priority (EOP):* Within the same SMT thread, when the operations that request a resource are all speculative, SPECWANDS assigns a higher and preemptive priority (similar to the NOP policy) to the operations that are issued earlier in program order. Different from NOP and LOP policies, its purpose is to prevent the backward contention within a group, i.e., to prevent a later operation from blocking an earlier operation in the group due to multi-issuing and out-of-order execution.

The above policies not only maintain the security principle of speculative noninterference (SNI) [23] but also facilitate the temporal locality of the shared resource usage. From our simulation results, SPECWANDS only introduces 1% and 5% performance overhead on realistic programs and SPEC 2017 benchmarks, respectively, which are much lower than those in STT [15] and SMT-COP [18], the representative work of the other two defenses.

To summarize, this article makes the following contributions.

1) We examine the deficiencies in existing defense approaches against SCAs, such as SC and TDM, and propose a novel mitigation scheme to reduce most delay between operations within the group, while blocking potential cover channels created in both interthread and intrathread modes.
2) We present a hardened instruction scheduler practicing above scheme, called SPECWANDS. It consists of three priority-based scheduling policies for different contention situations to divide the instructions into groups and ensure their security. Moreover, we analyze the performance impact of each policy based on the distribution of different contention situations.
3) We formally analyze the security of SPECWANDS, and evaluate its performance overhead and power consumption via detailed simulations. Compared with the other two state-of-the-art defenses [15], [18], SPECWANDS has promising advantages on some realistic programs and SPEC 2017 benchmarks.

II. BACKGROUND AND RELATED WORK

A. Modern CPU Pipeline and SMT

Fig. 1 shows the microarchitecture of a typical high-performance CPU, featuring branch speculation, out-of-order scheduler and various techniques to optimize the instruction-level parallelism (ILP). One such technique is SMT, which allows for multiple hardware-supported threads (HTs) to run on the pipeline concurrently, sharing critical resources such as L1-cache/TLBs, execution units (EUs) and issuing ports. This approach improves the pipeline throughput by hiding memory latency and maximizing resource utilization. Previous studies [24] have shown that a 2-context SMT processor can improve performance by up to 30% with some modest hardware cost. As a result, SMT has been widely deployed in PCs and warehouse servers.

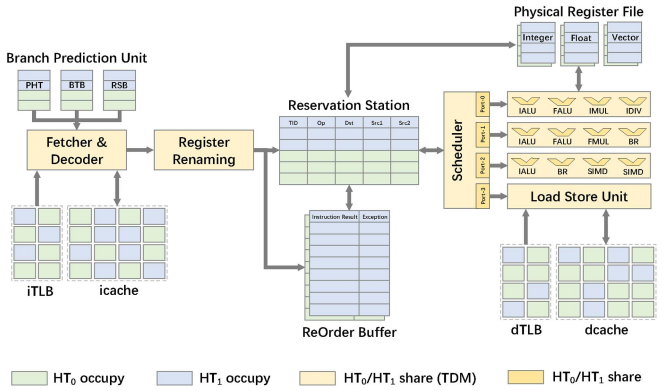


Fig. 1. Microarchitecture of a typical high-performance CPU that supports *speculative*, *out-of-order* execution with a 2-context SMT.

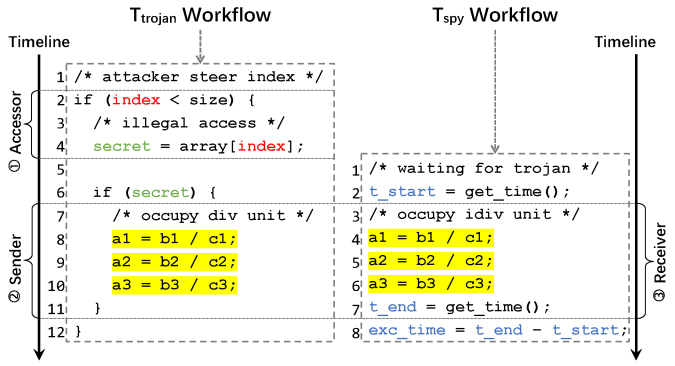


Fig. 2. PoC of an interthread SCA that exploits Spectre-V1 vulnerability. The "index" can be steered by an attacker, and the "secret" is the target to be leaked. The highlighted code can occupy an integer division unit and its corresponding issuing port.

B. Side-Channel Attacks on SMT Processors

While resources sharing can bring performance benefits, it also exposes the SMT processors to potential side-channel attacks. For the shared storage components, such as L1-caches/TLBs, the interference among sharing HTs can cause a subsequent access to be a hit or a miss. An attacker can exploit this side effect by measuring the difference in its access time, and infer some private information such as the encryption key or keyboard strokes through the access traces [25]. Because the placement and the measurement of the data layout in the side channel can be done asynchronously, such channels are called *persistent* channels. For shared computational components, such as issuing ports [3] and EUs [1], [2], the contention among threads can prolong the execution time of each HT, which can also be perceived by the attacker to reason about the private information related to the execution flow. Such side channels are referred to as *volatile* channels.

C. Transient Execution Attacks and Covert Channels

Side-channel attacks have not attracted attention of processor designers until the emergence of TEAs. The core logic of a TEA has three main steps: 1) *Accessor*; 2) *Sender*; and 3) *Receiver* [4], [26]. In the accessor step, the attacker illegally accesses the secret data through a staged speculative execution. There are two ways the attacker can set up the speculative attacks. In *Spectre-type* attacks, the attacker exploits hardware branch predictors on control flow transfer or memory disambiguation to bypass intended software defense codes, such as bound checking [13], data cleaning [8] or stack pivoting [9]. While in *Meltdown-type* attacks, the attacker exploits the bugs of access permission protection to break hardware isolation between domains, such as User_Space and Kernel_Space [], GuestVM and Hypervisor, or SGX_Enclave and Untrusted_Software_Stack [5], [12].

The attacker can then use the *sender* step and the *receiver* step to transmit the illegally accessed data through a covert channel. A covert channel works very similarly to that of a side channel. The only difference is that the sender and the receiver in a covert channel are both manipulated by the attacker. While in a side channel, the victim process is the sender and the receiver are controlled by the attacker. In this article, we use the term SCAs to describe the TEAs that are based on resource contention, i.e. the attacks that use *volatile covert channels*. Furthermore, we classify the existing SCAs into two categories.

Interthread SCAs: Fig. 2 shows a PoC code of the Spectre-V1 attack that exploits the resource contention on the issuing port and the EU on an SMT processor. In this attack, HT_{Trojan} and HT_{Spy} are both controlled by the attacker, with HT_{Trojan} acting as both the accessor and the sender, and HT_{Spy} acting as the receiver. First, HT_{Trojan} bypasses the bound check by manipulating the branch predictor at line 2, and reads the secret by out-of-bounds access at line 4 during the speculative execution. Then, HT_{Trojan} controls the execution of the subsequent division instructions according to the value of secret. Assume the secret has only one bit. If it is 1, the division will be executed. Meanwhile, HT_{Spy} will also perform division operations and measuring the time. If the time is shorter, it can infer that no contention has occurred and the secret is 0; otherwise, the secret is 1.

Intrathread SCAs: The above *interthread* SCAs can be extended and carried out in the same HT, i.e., an *intrathread* SCA. Some researchers have shown that, on a superscalar processor, multiple independent instructions in the same scheduling window can compete for the resources and thus can be used to form a covert channel [21], [22]. As Fig. 3 shows, the attacker combines the three steps in the same HT, and delays the receiver step (via line 2) to make it executed simultaneously with the sender step. This attack is also known as *SpecRewind* Attack [21].

D. Existing Mitigations

In most cases, mitigations for the attacks via persistent channels have been well established. They include domain partitioning [27], [28], index/replacement randomization [29], [30], and footprint-based detection [31], [32]. In particular, for TEAs through persistent covert channels, schemes that use *Invisible Speculation* [33], [34], [35], [36] extends the squashing mechanism in the pipeline to cleanup and rollback the side effects in the cache memory for the mis-predicted speculative execution with modest performance and hardware overhead.

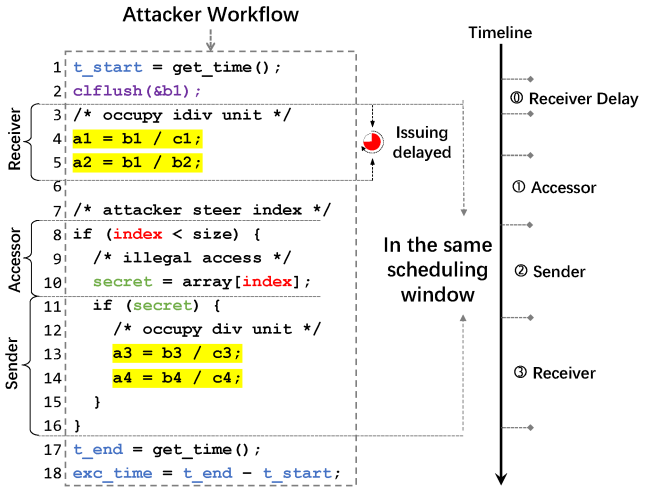


Fig. 3. PoC of an intrathread SCA, also called *SpecRewind* Attack, which also exploits Spectre-V1 vulnerability and leverages the intrathread contention as covert channel.

However, these methods cannot be generalized and applied to volatile channels. Currently, the only secure solution is to disable SMT for security-sensitive HTs [37], or scheduling mutually untrusted threads on different physical cores [38], [39]. However, this approach only works against traditional side-channel attacks when the protected target, such as the thread executing a cryptographic computation or code inside a secure enclave, can be identified by the programmer before running. But for SCAs, any thread with vulnerable speculative code can be exploited by an attacker. It is hard to distinguish an HT_{Trojan} from an HT_{Spy} as they are often from the same user group and supposed to be trustworthy.

Another solution is to use TDM scheme on the shared resources [18]. It allocates time slices for each HT to avoid resource contention. However, this approach violates the original intent of SMT to share unused resources when available. It can incur significant performance overhead due to its rigidity in resource sharing (more than 12% in our evaluation). Although, some approaches try to adaptively allocate time slices to improve resource utilization [19], the adaptive measure can inevitably be used to become another potential covert channel. Other schemes [20], [40] adopt asymmetric allocation strategies for threads with different trust levels to ensure that highly trusted threads can obtain more time slices. However, the trust level needs to be specified by the programmer through some annotations. So, it comes back to the earlier question of "which threads can we trust?".

Another type of defenses against SCAs is SC [15], [16], [17]. Instead of blocking covert channels, it disallows data or other potential microarchitectural side effects from propagating to the downstream instructions by stalling/blocking some dependent instructions during the speculative execution until the execution has reached some safe points. The advantage of such schemes is that they can block both persistent and volatile covert channels of TEAs comprehensively. But, due to the use of stalling and blocking of the dependent instructions, they often incur a significant performance overhead (up to 15% according to our evaluation).

TABLE I
VULNERABILITY AND ACCESS CAPABILITY OF EXISTING SCAS . **AC**: WHETHER IT CAN ACCESS THE DATA CROSS HARDWARE PROTECTION DOMAINS

Attack	Vunerability	AC
Spectre-PHT/BTB/RSB [13]	Control Flow Prediction	✗
Speculative Store Bypass [8]	Memory Disambiguiation	✗
Speculative Load Disorder [41]	Memory Order Speculation	✗
SWAPGS [9]	Out-of-Order Execution	✓
Rogue System Register Read [42]	Buggy #NM Exception Handler	✓
Meltdown [4], [12]	Buggy #GP Exception Handler	✓
L1TF (Foreshadow) [4], [5]	Buggy Terminate Fault Handler	✓
MDS [6], LVI [7]	Buggy Assist/Abort Data Forward	✓

III. THREAT MODEL

Table I lists the vulnerabilities exploited by existing SCAs, as well as the capabilities for their illegal accesses. We assume a powerful attacker can launch any SCA listed in Table I within or across domains. For example, the attacker can inject an attack code through malicious Javascript scripts, or malicious browser plug-ins. The code can exploit Spectre-PHT vulnerability [13] to bypass the bound check of browser's sandbox and access some private keys and cookies. Or, the attacker can exploit Meltdown vulnerability [12] by crossing the hardware domain of the kernel and accessing some critical data structures. The attacker can then transmit the stolen secrets through a covert channel based on resource contention as shown in Figs. 2 and 3. Furthermore, the attacker can launch a malicious virtual machine (VM) in the cloud and exploit Foreshadow vulnerability [5] to access the data in other victim VMs residing on the same physical core. The malicious VM can also transfer the stolen data through the fabricated contention covert channel.

Out-of-Scope: We do not consider TEAs through persistent covert channels, e.g., cache or TLB. As explained in Section II-D, existing defenses are effective and efficient to cover these attacks, and they are orthogonal to our scheme. We also exclude nontransient side-channel attacks because of their limited threat. Some effective defenses against those attacks have been introduced in Section II-D.

IV. DESIGN OF SPECWANDS

A. Overview

We first revisit the handicaps of two existing defense approaches, i.e., TDM and SC. For TDM, the strict partitioning of time slices is too rigid for most workloads that may have unbalanced resource requirements. For example, if we have two active HTs in the system, with one being more computation-intensive and the other more memory-intensive. Using TDM, almost half of the computation and memory resources may be wasted. For SC, when branch instructions are issued frequently and the average branch resolution time is long (nearly 20 cycles for SPEC benchmarks according to our evaluation), the aggregated issuing delay will incur a significant performance overhead. Fig. 4(a) shows the timelines of two HTs that execute the code snippet shown in the box. In each iteration, the division operation within the loop body need to be delayed until the guarding branch is resolved, i.e.,

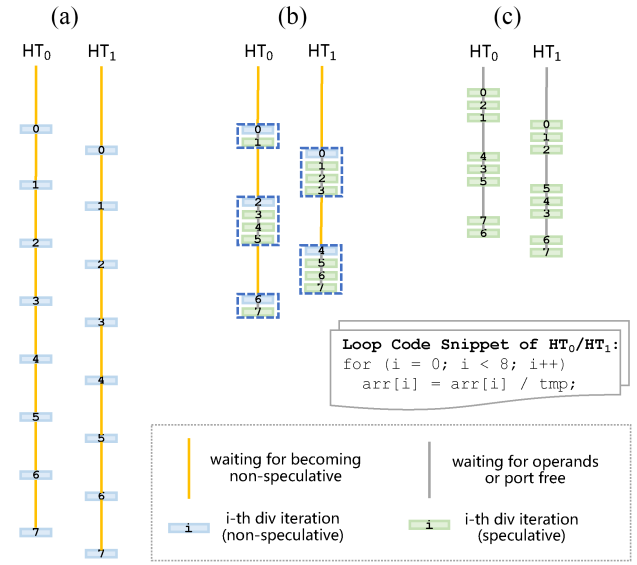


Fig. 4. Timelines of loop iterations in the code snippet shown in the figure under Speculative Compression (SC), Grouping (i.e., the policy of SPECWANDS), and native FCFS policies, respectively. (a) Loop iteration under SC policy. (b) Loop iteration under grouping policy. (c) Loop iteration under FCFS policy.

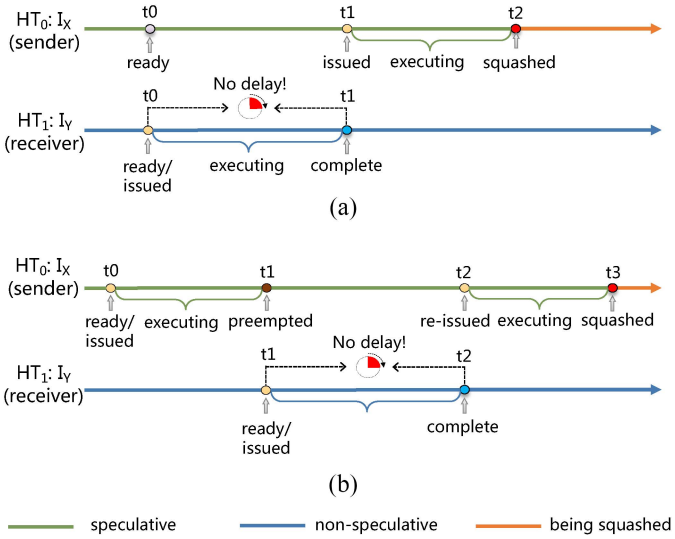


Fig. 5. Workflow of the NOP policy and its defense mechanism in an interthread scenario. It is also the same in an intrathread scenario. (a) NOP workflow when operations are ready at the same time. (b) NOP workflow when the nonspeculative operation is ready later.

until the operation becomes nonspeculative. If the loop iterates more than eight times, the overhead can reach 8× as long for SC policy.

Our Insight: The above observation leads us to conclude that *if the scheduler can batch multiple speculative operations from the same HT into a group and allow them to exclusively occupy the resource for a short period of time (as most of the speculative execution windows are relatively short), then it can avoid the delay on each group member to improve performance.* As shown in Fig. 4, compared to the timeline under SC policy in (a), the timeline of grouping-based policy in (b) greatly reduces the number of delays and is close to the timeline under the native FCFS policy in (c).

To achieve the assumption, we propose a priority-based scheduler named SPECWANDS. It can efficiently slice and batch the instructions into groups by determining their priority based on their speculative status and the history of resource occupation. SPECWANDS incorporates three scheduling policies: 1) nonspeculative operations have higher priority over speculative operations (denoted as NOP); 2) last-owner-thread's operations have higher priority among speculative operations from different HTs (denoted as LOP); and 3) earlier operations have higher priority within the same HT (denoted as EOP). As shown in Fig. 4(b), the NOP policy enforce each group header operation to wait until it becomes nonspeculative; LOP policy allows inner speculative operations within a group to inherit resource until another group header operation (from the other HT) become nonspeculative; EOP policy inhibits the disorder of inner speculative operations within a group.[1]

[1]It should be noted that the grouping of SPECWANDS is solely determined by the scheduler, without involving compiler assistance as is the case with Intel Itanium processors.

The security concept behind them is the principle of SNI [23], i.e., *the observable states of a speculative execution should be indistinguishable from those when the same code sequence is executed nonspeculatively.* Focusing on resource contention-based covert channels, the SNI principle means that the machine states as the result of the resource allocation policies should be independent of whether the code sequence is executed speculatively or not. We will detail the workflow, defense mechanism and performance impact of these policies in the following sections.

B. NOP: Nonspeculative Operations Have Higher Priority

In SPECWANDS, each nonspeculative operation is treated as the header of a group. When a group header operation compete for resources together with another group inner speculative operations, it should be assigned higher and preemptive priority.

This policy ensures that nonspeculative operations from a potential receiver can access the shared resource immediately, and will not observe the interference caused by speculative operations staged by a potential sender. Fig. 5 shows such an example with two instructions from two HTs, one as a sender and the other as a receiver, competing for an issuing port. The policy takes effect in the following two scenarios.

If the port is free when the contention occurs, the nonspeculative instruction will access the port immediately. As Fig. 5(a) shows, instruction I_X from HT_0 and instruction I_Y from HT_1 compete for the port at t_0. The status of I_X is speculative, and I_Y is not. According to the NOP policy, I_Y can occupy the port immediately, and I_X needs to wait until the port is free. In this scenario, if HT_0 acts as the sender and HT_1 as the receiver, owing to the NOP policy on HT_1, no information can be transmitted.

If the port is currently occupied by a speculative instruction, the nonspeculative instruction can preempt it immediately. As

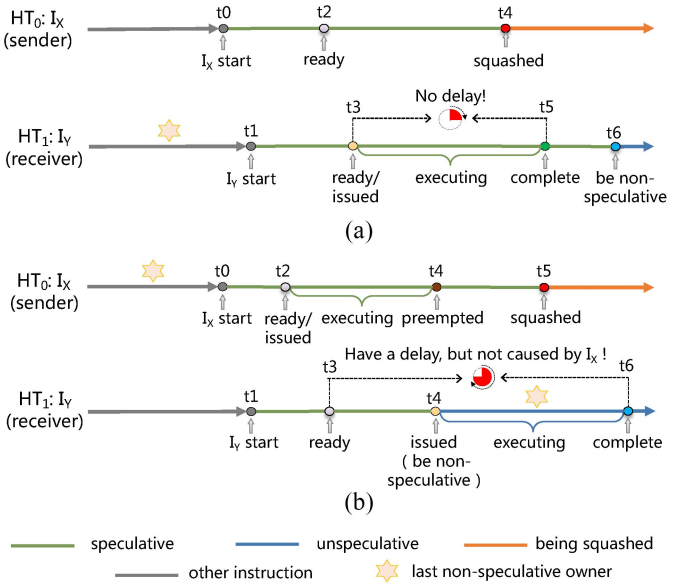


Fig. 6. Workflow of the LOP policy and its defense mechanism. (a) LOP workflow when last nonspeculative owner is $\text{HT}_{\text{receiver}}$. (b) LOP workflow when last nonspeculative owner is $\text{HT}_{\text{sender}}$.

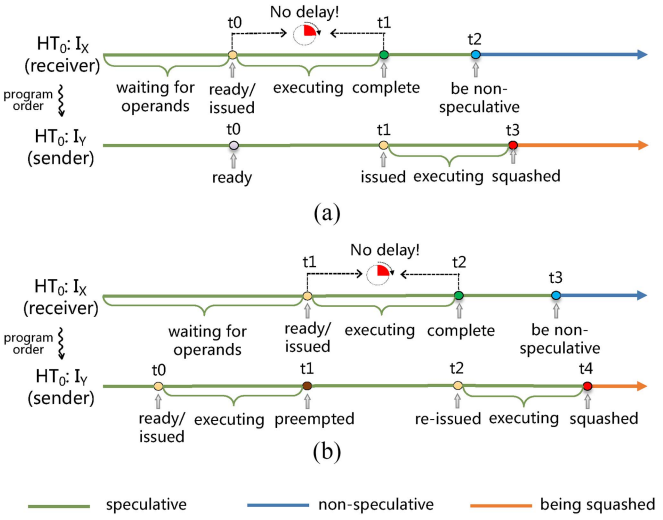


Fig. 7. Workflow of the EOP policy and its defense mechanism. (a) EOP workflow when operations are ready at the same time. (b) EOP workflow when the earlier operation is ready later.

shown in Fig. 5(b), speculative instruction I_X from HT_0 (acting as the sender) is occupying the port exclusively at t_0; At t_1, instruction I_Y (acting as the receiver) is ready and becomes nonspeculative. According to the NOP policy, it can preempt the port, and I_X needs to wait for being reissued until the completion of I_Y at t_2. In this case, the sender cannot interfere with the receiver, thus no information can be transmitted either.

C. LOP: Last-Owner-Thread's Operations Have Higher Priority

If the group header operation has occupy a resource, the rest of inner speculative operations within the group can inherit the ownership without waiting for becoming nonspeculative, until it is preempted by another group header operation. In other words, when multiple speculative operations from different HTs compete for a resource, SPECWANDS will give a higher priority to the competitor whose HT is the most recent nonspeculative owner of the resource.

Note that this policy is *nonpreemptive*, which means the owner HT can exclusively occupy the resource during the current period, regardless of whether it needs it or not, until the other HT preempts the resource, i.e., switches the ownership, via the NOP policy.

Combined with the NOP policy, this policy guarantees that the resource allocation across multiple HTs is independent of any speculative operation. It prevents the accessor of an SCA from modulating the resource to setup a covert channel. Fig. 6 shows the workflow of the LOP policy in more details and discuss how it can defeat interthread SCAs.

In Fig. 6(a), we assume HT_1 is a more recent nonspeculative owner of the port. At t_2, the instruction I_X of HT_0 is ready, but according to the LOP policy, it still cannot occupy the port even though the port is free. At t_3, when the instruction I_Y of HT_1 is ready, it can be issued immediately. If HT_0 acts as the sender and HT_1 as the receiver, nothing can be transmitted because HT_1 observes no delay.

On the contrary, in Fig. 6(b), HT_0 is the most recent nonspeculative owner and its subsequent instruction I_X (acting as the sender) can occupy the port immediately at t_2. Thus, the instructions I_Y from HT_1 (acting as the receiver) needs to wait until it becomes nonspeculative at t_4. *In this case, although a delay occurs, the receiver cannot attribute it to the contention created by the sender, because no matter I_X exists or not, HT_1 still needs to wait as it is not the most recent nonspeculative owner. The delay only depends on the previous nonspeculative instructions of HT_0, which cannot be dependent to the speculative accessor of any SCA.*

D. EOP: Earlier Operations Have Higher Priority

The above policies does not aim at intrathread SCAs, i.e., the contention of speculative operations within the group, due to multi-issuing and out-of-order execution. Therefore, we need more information to identify potential receivers and senders and keep a receiver from observing the interference created by a sender within the same HT.

With a closer look at the PoC in Fig. 3, we can notice that the receiver must be earlier than the sender in the program order. Otherwise, the completion time of the receiver will depend on the resolution time of the branch instruction at line-8, instead of its own execution time. That is why intrathread SCAs are called *SpecRewind Attacks* [21]. Based on this observation, SPECWANDS assigns a higher and preemptive priority to the earlier speculative operations than other speculative operations within the same HT.

Fig. 7 shows the EOP workflow using an example with two speculative instructions from HT_0. We assume the earlier one (I_X) serves as the receiver and the later one (I_Y) as the sender. There is no data dependence between them, and the issuing of receiver is delayed because its operands are not available until it encounters the sender within the scheduling window.

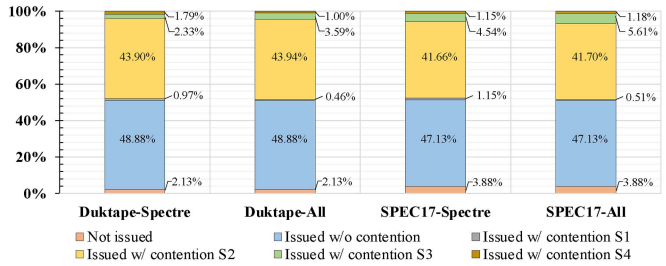


Fig. 8. Distribution of various contention scenarios on a "vanilla" SMT processor. **S1**: one competitor is nonspeculative and the other is speculative; **S2**: both competitors from different HTs are speculative, and the competitor from the most recent owner HT is the first to occupy the resource; **S3**: same as S2, except the competitor is not from the most recent owner HT that occupies the source first; and **S4**: both competitors are speculative and from the same HT.

As shown in Fig. 7(a), if the port is idle at t_0 and two instructions are both ready to be issued, the earlier receiver will be scheduled first. However, as shown in Fig. 7(b), if the sender is issued at t_0 and the receiver becomes ready at t_1, it will immediately preempt the issuing port.

E. Estimation on Performance Impact

We collect some statistics on various contention scenarios on an insecure native SMT system to estimate the performance impact of those three policies. The configuration and the methodology of the experiments are the same as those described in Section VII-A, and the results are presented in Fig. 8.

For NOP: The NOP policy can affect the performance in two ways. It can positively eliminate the contention caused by the wrong-path speculation. Also, it can negatively preempt the speculative execution on the correct path by some nonspeculative operations, and force some of its operations to be re-executed. However, as shown in Fig. 14, almost 50% of instructions are not issued (3%) or issued without contention (47%). For these two situations, SPECWANDS will not incur any overhead. Only about 1% of the issued instructions encounter a contention with one nonspeculative and the other speculative competitor. This situation can lead to preemption under the NOP policy. Thus, we believe that the NOP policy will incur only modest overhead.

For LOP: The LOP policy is designed not only to break the *speculative dependence* required by all SCAs but also to utilize the temporal locality of resource occupancy to minimize the performance impact. As shown in Fig. 8, scenarios S2 and S3, i.e., both competitors are speculative and from different HTs, constitute 46% of total cases. Among them, 42% are in scenario S2, i.e., the first arriving request are from the last owner HT; while only 4% cases are in scenario S3, i.e., the first arriving request is not from the last owner HT. The results show that the efficiency of the LOP policy used in SPECWANDS approximates to the FCFS policy used in the insecure native SMT processor.

For EOP: From Fig. 8, we can see that only about 1% of the cases are in scenario S4, i.e., both competitors are speculative and from the same HT, which may violate the EOP policy and cause preemption and re-execution with a negative impact on performance. This result indicates that the EOP policy will also have a minimal impact on the overall performance.

Overall, the performance impact of all three scheduling policies are quite small. And subsequent performance evaluation on a simulated system, as described in Sections VII-C and VII-D, also confirm our assertions.

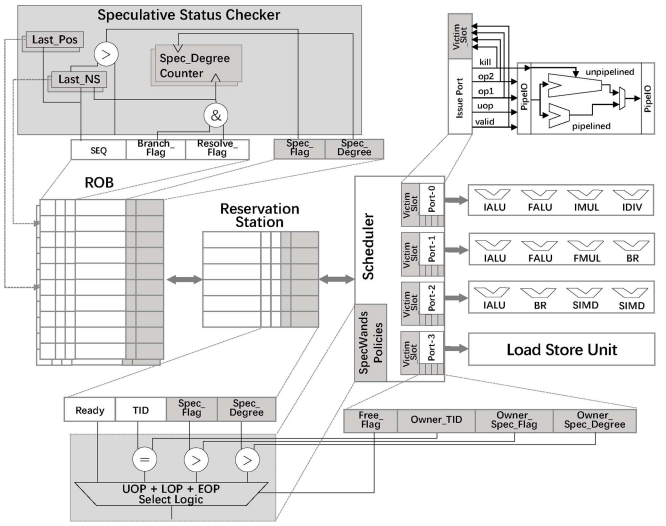


Fig. 9. Framework of SPECWANDS. The shaded boxes are those introduced by SPECWANDS.

V. KEY IMPLEMENTATION ASPECTS

Fig. 9 presents the framework of SPECWANDS on a typical SMT microarchitecture. Before introducing its implementation, we need to clarify the definition on when a speculative instruction becomes "nonspeculative." It can determine the defense capability and the impact on performance. Similar to earlier work, such as [15], [33], and [34], SPECWANDS has two operating modes that correspond to two variants of definition on when an instruction has become nonspeculative.

1) SPECWANDS-*Spectre:* This operating mode only defends against SCAs that exploit vulnerabilities from branches, e.g., Spectre-PHT/BTB/RSB. Here, a speculative instruction is considered to have become *nonspeculative* when all its previous branch instructions have been resolved and predicted correctly. This mode has a small performance overhead because the speculative instructions from the correctly predicted path can become nonspeculative quickly and free to compete for resources as the instructions in the unsecured native processor. But its defense scope is more restrictive, and should be deployed with other mitigation schemes to form a more comprehensive defense system.
2) SPECWANDS-*All:* This variant can defend against all SCAs that include existing SCAs and any future SCA. Here, an instruction is considered to be nonspeculative only when it reaches the head of ROB without triggering an exception. This variant has a relatively higher performance overhead due to longer waiting time for becoming nonspeculative. But it has a much wider defense scope so that it does not rely on any other mitigation schemes.

Based on the above definition, we can also give two variants of the definition on the instruction ordering in the EOP policy. In the SPECWANDS-ALL mode, we adhere to the conventional definition of instruction ordering, i.e., an "earlier" instruction means an "older" instruction in the original program order. In the SPECWANDS-Spectre mode, we define instruction ordering between two instructions using their relative *speculative degree*, which means the number of control flow predictions (branches) exercised by the instruction. From the view of dynamic control flow graph, the instruction in the deeper basic block (dominated by more branches) has larger speculative degree. Taking the PoC in Fig. 3 as an example, the division instruction at line-13 has 2-more speculative degree than the division instruction at line-4, since before line-13 enters the pipeline, the speculation of previous branches at line-8 and line-11 must have been exercised. This variant further reduces the performance overhead of the EOP in the SPECWANDS-Spectre mode, because instructions within the same basic block have the same speculative degree, and thus no preemption and re-execution occurs.

A. Speculative Status Checker

Speculative status checker (SSC) plays the role to check whether an instruction is speculative or not, and compute its speculative degree if it is. Such information is needed in the NOP and EOP policies. SSC is associated with ROB, which provides a global program ordering and execution status of each HT. When instructions are inserted/removed in/from ROB, or ROB receives updated speculation information, SSC will be activated to check and update the speculation status of each ROB entry, and then notify the instruction scheduler. The speculation status is recorded in the following two fields in each ROB entry.

1) *Spec_Flag:* 1-bit tag indicating whether the instruction is speculative or not.
2) *Spec_Degree:* 7-bit tag specifying the *speculative degree* of the instruction.

In a simple implementation, SSC can scan from the header of ROB, setting the *Spec_Flag* of each instruction to nonspeculative and the *Spec_Degree* to zero, until it encounters an unresolved branch. The *Spec_Flag* of the subsequent instructions are all set to speculative, and the *Spec_Degree* will be incremented with the number of scanned unsolved branches. However, such an implementation may incur significant overhead and power consumption when ROB is large as in modern CPUs (more than 200). In SPECWANDS, we limit the width of each scan and adopt a progressive scanning strategy. In each round, SSC only scans a fixed number of instructions (usually the same as the issue width), and records the results in three intrinsic registers for next scan.

1) *Last_Pos:* 8-bit field pointing to the ending entry of this scan.
2) *Last_NS:* 8-bit field pointing to the entry of the last nonspeculative instruction in this scan.
3) *Spec_Degree_Counter:* 8-bit field accumulating the number of unresolved branches encountered.

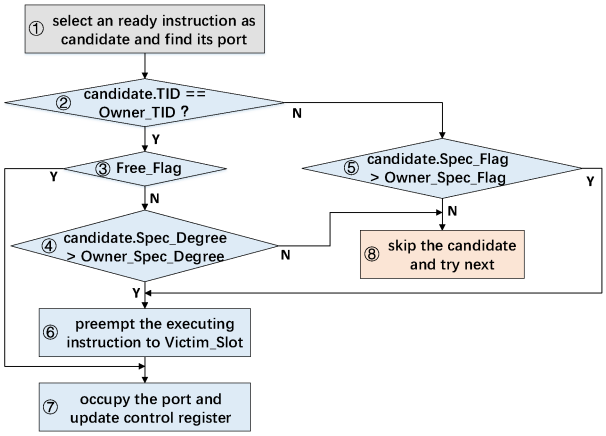


Fig. 10. Workflow of the instruction issuing logic in the SPECWANDS scheduler.

When a ROB squash occurs due to a mispeculation, SSC can quickly reset the *Last_Pos* to the position of the youngest unsquashed instruction, and recover the *Last_NS* and *Spec_Degree_Counter* from the *Spec_Flag* and *Spec_Degree* fields of that instruction.

B. Enhanced SMT Instruction Scheduler

Each reservation station (RS) entry needs to be tagged with two fields: *Spec_Flag* and *Spec_Degree*, which will be updated by SSC and used by the SMT instruction scheduler. The scheduler also needs to add a *Victim_Slot* for each issuing port to temporarily store each preempted instruction. When an instruction is issued, its opcode and operands are fed into the EU and also stored in the *Victim_Slot*. Once the scheduler decides to preempt the instruction, the port asserts the *kill* signal to the EU, which clears the internal state of the unpipelined unit and allows for accepting new operands in the next cycle. The *kill* signal simultaneously activates the port's *Victim_Slot* to reinsert the preempted instruction into the RS. In addition, each port needs an intrinsic control register to record the information for scheduling. The register includes the following four fields.

1) *Free_Flag:* 1-bit field indicating whether the port is free or not. Only when it is cleared (i.e., free), the following four fields can be valid.
2) *Owner_TID:* 1-bit field indicating which HT is occupying the port.
3) *Owner_Spec_Flag:* 1-bit field recording whether the occupying instruction is speculative.
4) *Owner_Spec_Degree:* 7-bit field recording the speculative degree of the occupying instruction.

Fig. 10 shows the workflow of the enhanced SMT instruction scheduler. *Step-1:* Select a ready instruction from RS as the candidate. *Step-2:* Determine whether the candidate's thread ID (TID) matches the port's *Owner_TID*. If they match, go to step-3; otherwise, go to step-7. *Step-3:* Check the port's *Free_Flag* to determine whether it is free or not. If it is free, go to step-7; otherwise, go to step-4. *Step-4:* If the candidate's TID matches the port's *Owner_TID*, but the port is not free, it means the port is being occupied by another instruction from

the same HT. In this case, according to the EOP policy, we can just compare their speculative degrees. If the candidate wins, go to step-6 for preemption; otherwise, go to step-8. *Step-5:* If the candidate does not match the port's *Owner_TID*, it means the port is occupied by another HT. According to the NOP policy, we need to check whether the candidate is nonspeculative and the owner instruction is speculative or not. If so, go to step-6; otherwise, go to step-8. *Step-6:* According to the NOP or EOP policy, preempt the occupying instruction and put it in the port's *Victim_Slot*. *Step-7:* Issue the candidate, update the port's control register. *Step-8:* When this step is reached, it means issuing of this candidate has failed. So, just skip it and schedule the next instruction.

VI. Security Analysis

A rigorous proof of the SNI principle involves formally modeling the workflow of the entire scheduler and enumerating all possible instruction sequences. This task demands substantial manual effort even with the help of advanced automated verification tools. As a result, we propose a workaround instead by aiming the proof target from an attacker's perspective. The core proof of the interthread SCA mitigation on a CPU with 2-context SMT has been achieved through the Owicki-Gries method [43], which is widely used to verify the correctness of concurrent systems. Due to page limit, we only provide a summary of the proof here. More details can be found in [44].

We abstract the attack process to two HTs: HT_{sender} and HT_{receiver}. They continuously call the function *acquire()* to obtain a shared issue port. The function *acquire()* follows the NOP and LOP policies, determining acquisition success according to the operation's speculative status (*op.status*), the port's last nonspeculative owner (*port.owner*) and the port's occupying speculative status (*port.status*). If *acquire()* returns `true` the HT must call *release()*, which evaluates whether the port is preempted by others and resets (*port.status*). Both *acquire()*/*release()* are atomic, but the interval between them is not. If either *acquire()* or *release()* fails, the perceived delay of HT_{receiver}'s operation ($op_{\text{receiver}}.delay$) will exceed zero.

Our target is to prove that the system satisfies the invariant property outlined in (1), where $\mathcal{I}$ and $\mathcal{T}$, respectively, denote the value of the secret (1 or 0) and the delay value (greater than 0 or 0). This property realizes the SNI principle, ensuring that any information attackers glean from the system, specifically the delay time of HT_{receiver}'s operation, remains independent of the secret data (assuming it has only 1-bit). Notably, the proof must be based on a critical assumption, which is all HT_{sender}'s operations dependent on the secret are speculative; otherwise, the attack is considered beyond the scope of the transient-execution attack

$$\mathcal{M} \models \Box(\mathcal{I}(\text{secret}) \perp\!\!\!\perp \mathcal{T}(op_{\text{receiver}}.\text{delay})). \tag{1}$$

The proof process unfolds in the following bottom-up steps. First, we conclude several primitive invariant properties of *acquire()*/*release()* as (2) shows. For *acquire()*, if *port.owner* equals *op.tid* or *port.status* is nonspeculative, *port.owner* will never be changed, and vice versa. As for *release()*, *port.status* will always be reset; and if *port.owner* equals *op.tid*, it will return true, and vice versa

$$\begin{aligned}\mathcal{P}_{\text{acq}} &\triangleq \Box(((\text{port.owner} = \text{op.tid}) \vee \text{port.status})) \\ &\quad \Leftrightarrow \otimes\text{port.owner} \\ \mathcal{P}_{\text{rel}} &\triangleq \Box(((\text{port.owner} = \text{op.tid}) \Leftrightarrow \text{res})) \\ &\quad \wedge \neg\text{port.status}.\end{aligned} \tag{2}$$

Next, we derive the compound invariant properties of HT_{sender} and HT_{receiver}. As depicted in (3), HT_{sender} can never alter *port.owner*, and *port.status* is always speculative. For HT_{receiver}, whether $op_{\text{receiver}}.delay$ is greater than 0 depends on the conjunction of *acquire()*'s and *release()*'s return value

$$\begin{aligned}\mathcal{P}_{\text{sender}} &\triangleq \Box(\otimes\text{port.owner} \wedge \neg\text{port.status}) \\ \mathcal{P}_{\text{reciver}} &\triangleq \Box\big(\big(res_{\text{acq}} \wedge res_{\text{rel}}\big) \Leftrightarrow \neg\mathcal{T}(op_{\text{receiver}}.\text{delay})\big).\end{aligned} \tag{3}$$

Finally, we reason about the critical invariant property concerning the isolation between HT_{sender} and HT_{receiver}, signifying that the property is always satisfied regardless of their interleaving. As shown in (4), $\mathcal{I}$(secret) remains independent of *port.owner* and $op_{\text{receiver}}.status$. By unifying all the aforementioned properties, we can derive the ultimate target, i.e., the independence between $\mathcal{I}$(secret) and op_{receiver}.delay

$$\begin{aligned}\mathcal{P}_{isolation} \triangleq \Box((&\mathcal{I}(\text{secret}) \perp\!\!\!\perp port.owner) \wedge \\ &(\mathcal{I}(\text{secret}) \perp\!\!\!\perp op_{\text{receiver}}.status)).\end{aligned} \tag{4}$$

TABLE II
Parameters Used in the Simulated Micro-Architecture for the Baseline Data. The Label Shown After Each Component Indicates Its Sharing Strategy Among Multiple HTs, Where "**S**" Means Total Sharing, "**P**" Means Fair Partition, and "**M**" Means Multiplexing With Yield Scheme

Component	Parameter Value
Core Overview	8-issue, out-of-order, 2-context SMT, 2Ghz
Pipeline	Fetcher/Decoder/Register Renamer (**M**), 64-entry RS (**P**), 256 Int / 256 FP Physical Registers (**P**), 8 Issue Ports (**S**), 92-entry ROB (**P**), 32-entry LQ (**P**), 32-entry SQ (**P**)
BPU	Tournament branch predictor(**S**), 4096 BTB (**S**), 16 RSB (**S**)
Private L1-I Cache	32KB 64B line, 4-way, 1 cycle RT latency, 8 MSHRs (**S**)
Private L1-D Cache	64KB, 64B line, 8-way, 1 cycle RT latency, 8 MSHRs (**S**)
Shared L2 Cache	2MB bank, 64B line, 16-way, 8 cycles RT local latency, 16 cycles RT remote latency, 16 MSHRs (**S**)

VII. Evaluation

A. Experiment Setup and Methodology

We simulated a prototype of SpecWands on the Gem5 simulator [45] (version *fe187de9bd*) with the O3 CPU model. The parameter values of the main components are shown in Table II. We add issuing ports and configure the grouping of EUs based on the Intel Skylake microarchitecture. We first run SPEC CPU 2017 (rate) benchmarks with the *ref* input data. To cover various scheduling scenarios, we follow the methodology of previous works [18] by selecting the benchmark pair according to their types (e.g., integer or floating-point) and program characteristics (e.g., number of branches and L2 Cache misses) as Table III shows. To evaluate the impact on more realistic scenarios, we also run a popular embedding Javascript engine *Duktape* [46] (version 2.6)

TABLE III
SPEC 2017 BENCHMARKS ARE DIVIDED INTO FOUR CATEGORIES BASED ON THE NUMBER OF BRANCH INSTRUCTIONS AND L2 CACHE MISS RATE. "**I**" STANDS FOR INTEGER PROGRAMS AND "**F**" STANDS FOR FLOATING-POINT PROGRAMS. THE CATEGORIZATION REFERS TO PREVIOUS RESEARCH [49]

	Low L2 Cache Miss	**High L2 Cache Miss**
Low BrNum	**I**: x264, exchange2, perlbench **F**: named, lbm, cactuBSSN	**I**: xz, xalancbmk **F**: fotonik2d, bwaves
High BrNum	**I**: leela, deepsjeng **F**: povray, imagick, nab, parest	**I**: gcc, omnetpp, mcf **F**: blender, wrf, cam4, roms

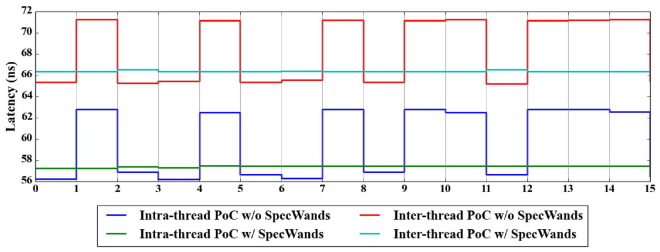


Fig. 11. Latency measured by the receiver of inter-/intra-thread SCA running on the native/SPECWANDS-hardened CPU. To improve legibility, we superimpose the results of interthread PoC above those of intrathread PoC, and shift them upward by a constant value.

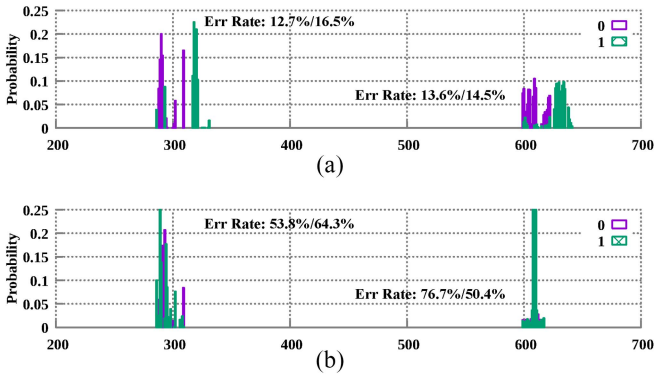


Fig. 12. Distribution of latency measured by the receiver code. Similar as Fig. 11, we shift the result of *SMoTher* attack to allow both results to be presented in a single figure. To obtain a more accurate error rate, we modify the default threshold of *SMoTher* and adopt the thresholding sampling algorithm of *SpectreRewind*. (a) Distribution of timing results (cycles) on native CPU. (b) Distribution of timing results (cycles) on SpecWands-hardened CPU.

with *Sunspider* [47] benchmarks as the input data. *SunSpider* includes the following eight categories of applications: 3-D modeling, data access, bit manipulation, encryption, complex control flow, mathematical libraries, regular expression processing, and data encoding. Like other prior work [20], we select one program from each category with the longest execution time, and adopt the tournament pairing scheme to evaluate the largest overhead for each program on different scheduling scenarios.

B. Effectiveness Evaluation

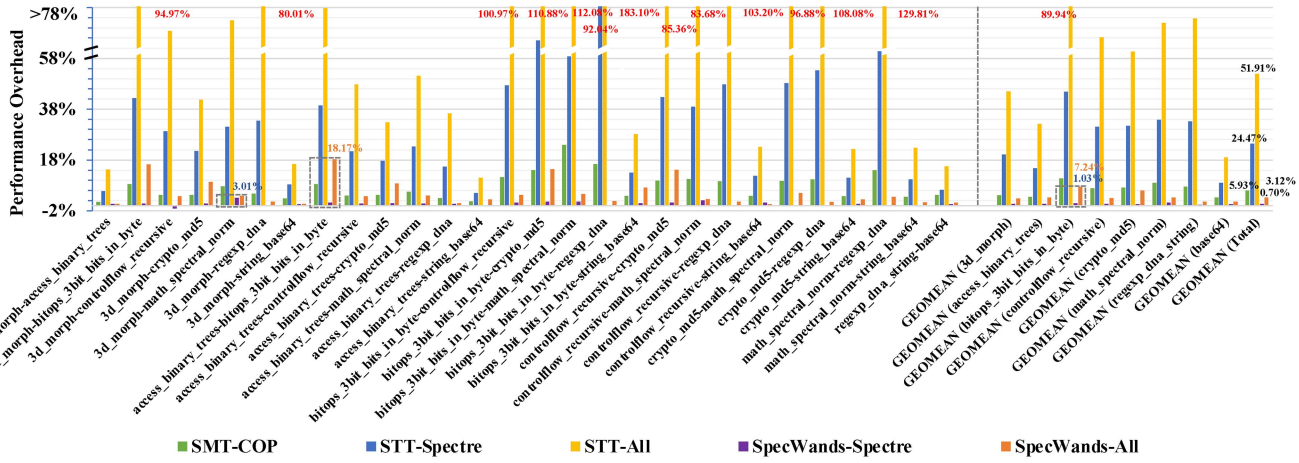
First, we construct two PoCs of inter-/intra-thread SCA exploiting Spectre-PHT as shown in Figs. 2 and 3. To maximize the window for contention, we choose the integer division unit as the covert channel, which takes the longest time (12 cycles) to complete the computation. In PoCs, each iteration can transmit 1 bit value, repeated 100 times to reduce statistical error. For the interthread PoC, considering that GEM5 does not support full-system simulation in the SMT mode, and the memory space of each HT is completely isolated, it is quite challenging to synchronize the HT_{sender} and HT_{receiver} in each iteration. Thus, we add an instruction in the ISA dedicated to synchronize HTs on SMT, whose function is similar to *pthread_barrier_wait()*. The simulation results of the SPECWANDS-hardened system compared to the native system are presented in Fig. 11. It shows that the attacker can accurately leak each bit in the native system, while unable to do so in the SPECWANDS-hardened system.

Second, we test SPECWANDS against two open-sourced attacks in the wild: one is *SMoTher* [14], an interthread SCA that exploits Spectre-BTB vulnerability [48]; the other is *SpectreRewind* [21], an intrathread SCA that exploits Meltdown vulnerability [12]. Because Meltdown vulnerability cannot be simulated on Gem5 microarchitecture that requires full system simulation, we rewrite the process of *SpectreRewind* to exploit Spectre-STL vulnerability [8]. Fig. 12 shows the distribution of the latency and the error rates for both attacks. We can see that, on the native CPU, the attacker has a lower error rate to distinguish whether the transmitted bit is 1 or 0, while on the SPECWANDS-hardened CPU, the error rate is already higher than 50%, which is equivalent to random guessing.

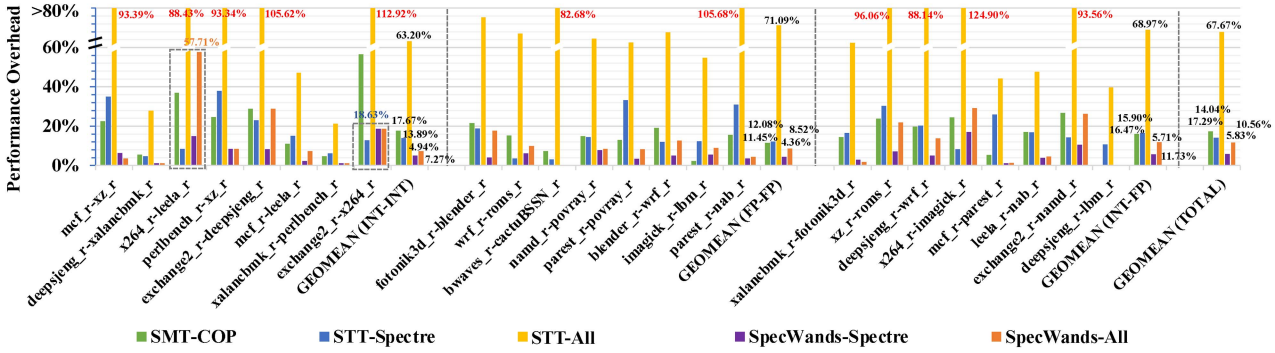
C. Performance Evaluation

We compare the performance of SPECWANDS with two other schemes, where SMT-COP [18] represents TDM scheme and STT [15] represents the speculative compression scheme. Since SMT-COP is not open-sourced, we reimplement its scheme on Gem5. To give a fair comparison, for SMT-COP, we do not implement other adaptive strategies that may sacrifice its security; For STT, we lift the protection for the persistent-channel components, such as cache/TLB and PHT/BTB/RSB. Similar to SPECWANDS, STT also has two defense modes, i.e., Spectre-Mode and All-Mode (called Futristic-Mode in their paper).

Fig. 13(a) and (b) shows the performance overhead of the three defenses for Duktape Javascript engine and SPEC CPU 2017 (rate), respectively. From the figure, we can see that the overall performance overhead of SPECWANDS-Spectre/-All are 0.70%/3.12% and 5.83%/10.56%, which are much lower than 24.47%/51.91% and 14.04%/67.47% of STT, and also much lower than 5.93% and 17.29% of SMT-COP. For each benchmark pair, SPECWANDS significantly outperforms STT in both defense modes. And this advantage also shows in comparison between SPECWANDS and SMT-COP, except for a few cases such as *povray-calculix* and *x264_r-leela_r* in SPEC 2017. To facilitate more detail analysis, we record other statistics such as the latency of operand-ready instructions waiting for issuing, the busy rate of issuing port, etc., as shown in Table IV.



(a)



(b)

Fig. 13. Performance overhead of SMT-COP, STT-Spectre, STT-All, SPECWANDS-Spectre, and SPECWANDS-All on Duktape Javascript engine and SPEC CPU 2017 (rate) benchmarks. The bars taller than 80% are truncated, whose maximum values are marked in red. (a) Performance Overhead on DukTape Javascript Engine with Sunspider Benchmarks. (b) Performance Overhead on SPEC CPU 2017 (rate).

TABLE IV
MEAN STATISTIC OF RUNNING DUKTAPE/SPEC ON EACH SYSTEM. "**1**": SMT-COP, "**2**": STT-SPECTRE, "**3**": STT-ALL, "**4**": SPECWANDS-SPECTRE, "**5**": SPECWANDS-ALL, "**-SP**": SPECTRE-MODE. "**LW**": LATENCY OF OPERAND-READY INSTRUCTION WAITING FOR ISSUING; "**LT**": LATENCY OF ISSUE-TRYING, WHICH EQUALS THE LATENCY BETWEEN THE MOMENT OF INSTRUCTION BEING ISSUED FIRST TIME AND THE MOMENT BEING ISSUED SUCCESSFULLY (LAST TIME); "**LX**": LATENCY OF INSTRUCTION EXECUTION. "**BR**": BUSY RATE OF ISSUING PORTS; "**UR**": UTILIZATION RATE OF ISSUING WIDTH. "**FR**": FULL-EVENT OCCURRENCE RATE OF RS

	LW (c)	LT (c)	LX (c)	BR (%)	UR (%)	FR (%)
1	10.8/14.8	0.6/0.8	2.6/3.9	16.1/24.1	42.9/51.4	10.5/20.5
2	13.0/19.1	0.2/0.3	2.5/3.7	19.6/22.8	85.7/91.4	21.2/18.4
3	22.3/33.3	0.2/0.2	2.5/3.5	18.8/19.9	90.5/94.6	31.6/37.7
4	7.58/11.5	0.2/0.5	2.5/3.8	20.5/29.1	86.3/82.7	6.23/14.2
5	10.5/13.2	0.4/0.4	2.5/3.6	19.4/22.6	74.6/77.8	9.81/18.2

Versus STT: The main overhead of STT comes from the need to wait for the dependant instructions to become non-speculative, which may cause the pipeline to stall. Table IV indeed shows the system hardened by STT has a much larger issuing waiting latency than the other two (almost by 20–30 cycles). The trend is even more pronounced when the program has a larger branch resolution time or a dependency chain with higher L2 Cache miss, such as the pairs containing *control_flow_recursive*, *math_spectral_norm*, *crypto_md5* in Duktape engine, and *perlbench_r*, *mcf_r*, *parest_r* in SPEC 2017. But, once the instructions are allowed to be issued in STT, HTs are free to compete for the resources as in an unprotected system. So, its utilization of issuing bandwidth is the most efficient among the three.

Versus SMT-COP: The main overhead of SMT-COP is the time waiting for an HT's own time slice, which often leads to longer issuing waiting latency and much lower issuing port utilization (below 50%) as Table IV shows. For memory intensive programs, such as *bitops_3bit_bits_in_byt*, *exp_dna_string* in Duktape engine, and *x264_r*, *wrf_r*, and *exchange2_r* in SPEC 2017, such low utilization rates become more serious. The advantage of SMT-COP is its simplicity to implement, but its scalability is the worst among three schemes. The length of the time slice depends on the longest completion time of any unpipelined EU, and the waiting time is proportionate to the number of HTs supported on the CPU.

Additionally, a common factor that contributes to the performance overhead of STT and SMT-COP is their long issue delay, which frequently stalls the pipeline and drags down the overall performance. In contrast, the main overhead of SPECWANDS is the latency induced by the LOP policy, as well as the preemption/re-execution overhead induced by

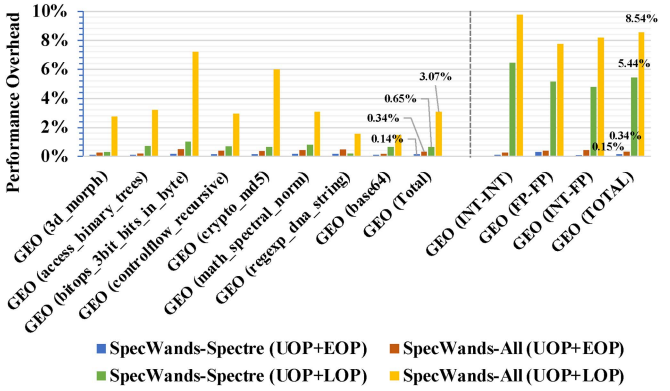


Fig. 14. Performance overhead of SPECWANDS using different combinations of policies on Duktape engine and SPEC 2017.

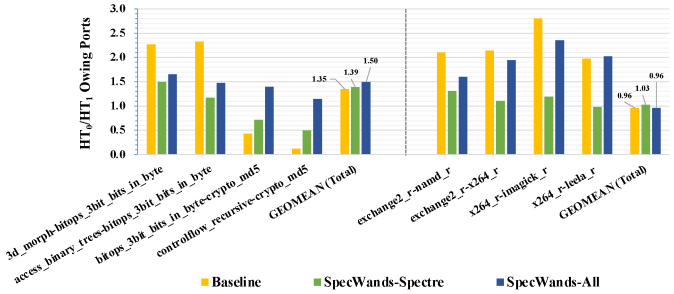


Fig. 15. Ratios of issue ports owned by two HTs (HT_0/HT_1). Under the LOP policy, the number of issue ports owned by each HT is sampled per 10 000 cycles, and then their arithmetic average is calculated.

TABLE V
OVERHEAD OF DYNAMIC POWER CONSUMPTION RUNNING DUKTAPE/SPEC ON EACH SYSTEM. BECAUSE THE RUNNING TIME AND EXECUTED INSTRUCTIONS OF EACH BENCHMARKS PAIRS DIFFER UNDER DIFFERENT SYSTEMS, WE CALCULATE THE AVERAGE POWER CONSUMPTION PER NANOSECOND

	EUs (%)	RS (%)	ROB (%)
SMT-COP	1.99/2.50	2.01/4.32	1.85/2.78
STT-Sp	1.14/2.89	7.96/9.12	10.34/12.03
STT-All	2.09/2.99	9.97/12.92	14.63/17.21
SPECWANDS-Sp	1.90/5.01	2.86/6.75	6.16/10.09
SPECWANDS-All	2.63/6.59	4.17/9.26	6.81/10.92

the NOP and EOP policies. Nevertheless, its overall overhead remains relatively low. The highest overheads observed in SPECWANDS-Spectre/-All are 18.63%/57.71%, respectively, which occur in the *exchange2_r-x264_r* and *x264_r-leela_r* workloads of SPEC 2017. These overheads are still far below the highest overheads observed in STT (37.81% and 124.90%) and is comparable to the highest overhead in SMT-COP (56.59%). Moreover, as illustrated in Fig. 13(b) for more realistic applications, the largest overheads of SPECWANDS-Spectre/-All in different scenarios are approximately 1%/7%, with the worst case scenarios not exceeding 3%/15%. These values are far more acceptable to developers compared to the extreme cases observed in the other two defenses.

D. More Detailed Analysis on Performance Overhead

First, we analyze the impact of different scheduling policies on the overall performance. Based on their defense scopes, we separate the policies into two groups: 1) enabling NOP and EOP policies only for intrathread SCAs and 2) enabling NOP and LOP policies only for interthread SCAs. The results are shown in Fig. 14. The performance overhead of the NOP+EOP is around 0.1%–0.3%. This is because most of the issuing port contention comes from speculative instructions. Thus, the preemption caused by NOP occurs only very infrequently. And since the native scheduler tends to issue older ones when it confronts multiple ready instructions within an HT, the preemption caused by EOP also occurs infrequently. In contrast, the overhead of the NOP+LOP is relatively high at around 3%-8%, which basically constitutes most of the overall overhead. It shows that, although we try to exploit the temporal locality of the issuing port using LOP, it still cannot satisfy the bandwidth demand of all HTs.

Next, we analyze the benchmark pairs that have a high performance overhead under the NOP+LOP policies. We sample the number of issuing ports occupied by each HT and calculate the ratio of that number for HT_0 and HT_1 under native FCFS policy (as baseline) and SPECWANDS, respectively. The results are shown in Fig. 15. We find that, for those program pairs that have a high overhead under LOP, one of them must be a dominant program that has a higher occupancy rate under the native policy, such as *3bit_bits_in_byte*, *exchange2_r*, and *x264_r*. When using the LOP policy, the nondominant HT can have a larger share of the issuing port, which can slow down the dominant HT as a result. For this situation, one workaround is to referring more nonspeculative access history of each HT for resource allocation instead of direct ownership inheriting. Such revised policy may give more opportunities to the HT that exactly needs the resource more eagerly. We will further evaluate more improvement solutions in future works.

E. Power Consumption Evaluation

We applied McPAT [50] (version 1.3) to model the power consumption of SMT-COP, STT, and SPECWANDS. The results are shown in Table V. We only modify the scheduler instead of introducing new RAM components, thus the incurred hardware cost and static power consumption (such as gate leakage, subthreshold leakage) are negligible. Here, we only measure dynamic power when running Duktape engine and SPEC 2017. To ensure a more accurate measurement, we patch the code of SSC and *Enhanced Scheduler* in SPECWANDS, as well as the code of *Data Flow Tracking* and *Tainting/Untainting* in STT, so that these actions can be reflected in the statistics of relevant pipeline components, i.e., EUs, RS, and ROB.

As we can see from Table V, SPECWANDS has a slightly higher power overhead on EUs compared to SMT-COP and STT, which mainly comes from the preemption and the re-execution required in the NOP and EOP policies. STT also consumes more power than SPECWANDS on RS and ROB accesses because both *Taint* and *Untaint* operations in STT require frequent accesses to these two components.

VIII. Conclusion

In this article, we propose a priority-based scheduler, called SpecWands, to defend against TEAs that exploit inter-/intra-thread contention on a system component as a covert channel. SpecWands contains three scheduling policies: 1) *NOP:* it allows nonspeculative operations to preempt speculative operations at any time; 2) *LOP:* it allocates the resource to the speculative operations which belong to the thread occupying the resource most recently; and 3) *EOP:* it gives the earlier speculative operations in a thread higher and preemptive priority over the later speculative operation within the same thread. These three policies batch multiple continuous speculative operations into a group, which can exclusively occupy the resource for a certain period of time without any delay. The performance evaluation shows that SpecWands has a significant performance advantage over other state-of-the-art approaches, such as speculative compression and TDM.

References

[1] W.-M. Hu, "Lattice scheduling and covert channels," in *Proc. IEEE Comput. Soc. Symp. Res. Security Privacy*, 1992, pp. 52–61.

[2] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *Proc. IEEE Symp. Security Privacy*, 2015, pp. 623–639.

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

[4] C. Canella et al., "A systematic evaluation of transient execution attacks and defenses," in *Proc. 28th USENIX Security Symp.*, 2019, pp. 249–266.

[5] J. Van Bulck et al., "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proc. 27th USENIX Security Symp.*, 2018, pp. 991–1008.

[6] S. Van Schaik et al., "RIDL: Rogue in-flight data load," in *Proc. IEEE Symp. Security Privacy (SP)*, 2019, pp. 88–105.

[7] J. Van Bulck et al., "LVI: Hijacking transient execution through microarchitectural load value injection," in *Proc. IEEE Symp. Security Privacy (SP)*, 2020, pp. 54–72.

[8] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-leak forwarding: Leaking data on meltdown-resistant CPUs (updated and extended version)," 2019, *arXiv:1905.05725*.

[9] A. Luţaş and D. Luţaş, "Bypassing KPTI using the speculative behavior of the SWAPGS instruction," in *Proc. BlackHat Europe Conf.*, 2019, pp. 1–20. [Online]. Available: https://i.blackhat.com/eu-19/thursday/eu-19-Lutas-bypassing-KPTI-using-the-speculative-Behavior-of-the-SWAPGS-instruction-wp.pdf

[10] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read arbitrary memory over network," in *Proc. Eur. Symp. Res. Comput. Security*, 2019, pp. 279–299.

[11] M. Behnia et al., "Speculative interference attacks: Breaking invisible speculation schemes," 2020, *arXiv:2007.11818*.

[12] M. Lipp et al., "Meltdown: Reading kernel memory from user space," in *Proc. 27th USENIX Security Symp.*, 2018, pp. 973–990.

[13] P. Kocher et al., "Spectre attacks: Exploiting speculative execution," in *Proc. 40th IEEE Symp. Security Privacy (S P)*, 2019, pp. 1–19.

[14] A. Bhattacharyya et al., "SMoTherSpectre: Exploiting speculative execution through port contention," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2019, pp. 785–800.

[15] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 954–968.

[16] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 572–586.

[17] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," in *Proc. 28th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2019, pp. 151–164.

[18] D. Townley and D. Ponomarev, "SMT-COP: Defeating side-channel attacks on execution units in SMT processors," in *Proc. 28th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2019, pp. 43–54.

[19] U. Nezir, B. Lus, and G. Kucuk, "Improved resource scheduling for lightweight SMT-COP," in *Proc. 6th Int. Conf. Comput. Sci. Eng. (UBMK)*, 2021, pp. 575–580.

[20] M. Taram, X. Ren, A. Venkat, and D. Tullsen, "SecSMT: Securing SMT processors against contention-based covert channels," in *Proc. USENIX Security Symp.*, 2022, pp. 3165–3182.

[21] J. Fustos, M. Bechtel, and H. Yun, "SpectreRewind: Leaking secrets to past instructions," in *Proc. 4th ACM Workshop Attacks Solutions Hardw. Security*, 2020, pp. 117–126.

[22] T. Rokicki, C. Maurice, and M. Schwarz, "CPU port contention without SMT," in *Proc. Eur. Symp. Res. Comput. Security*, 2022, pp. 209–228.

[23] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *Proc. IEEE Symp. Security Privacy (SP)*, 2020, pp. 1–19.

[24] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 392–403.

[25] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptogr. Eng.*, vol. 8, no. 1, pp. 1–27, 2018.

[26] W. Xiong and J. Szefer, "Survey of transient execution attacks," 2020, *arXiv:2005.13435*.

[27] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2018, pp. 104–117.

[28] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2018, pp. 974–987.

[29] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 494–505.

[30] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *Proc. 28th USENIX Security Symp. (USENIX Security)*, 2019, pp. 675–692.

[31] J. Cho, T. Kim, S. Kim, M. Im, T. Kim, and Y. Shin, "Real-time detection for cache side channel attack using performance counter monitor," *Appl. Sci.*, vol. 10, no. 3, p. 984, 2020.

[32] J. Depoix and P. Altmeyer, "Detecting spectre attacks by identifying cache side-channel attacks using machine learning," in *Proc. Workshop Adv. Microkernel Oper. Syst.*, 2018, p. 75.

[33] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InviSispec: Making speculative execution invisible in the cache hierarchy," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2018, pp. 428–441.

[34] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An 'undo' approach to safe speculation," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 73–86.

[35] S. Ainsworth and T. M. Jones, "MuonTrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2020, pp. 132–144.

[36] B. Tang et al., "SpecBox: A label-based transparent speculation scheme against transient execution attacks," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 1, pp. 827–840, Jan./Feb. 2023.

[37] Y. Zhang, Z. Zhu, and D. Meng, "DDM: A demand-based dynamic mitigation for SMT transient channels," in *Proc. IEEE Int. Conf. Parallel Distrib. Process. Appl. Big Data Cloud Comput. Sustain. Comput. Commun. Soc. Comput. Netw. (ISPA/BDCloud/SocialCom/SustainCom)*, 2019, pp. 614–621.

[38] X. Wu et al., "Partial-SMT: Core-scheduling protection against SMT contention-based attacks," in *Proc. IEEE 19th Int. Conf. Trust Security Privacy Comput. Commun. (TrustCom)*, 2020, pp. 378–385.

[39] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Trans. Comput. Syst.*, vol. 28, no. 4, pp. 1–45, 2010.

[40] M. Escouteloup, R. Lashermes, J. Fournier, and J.-L. Lanet, "Under the dome: Preventing hardware timing information leakage," in *Proc. Int. Conf. Smart Card Res. Adv. Appl.*, 2021, pp. 233–253.

[41] "Speculative load disordering / CVE-2021-33149." Intel. 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-load-disordering.html

[42] "Rogue system register read / CVE-2018-3640 / Intel-SA-00115." Intel. 2018. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/rogue-system-register-read.html
[43] S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Commun. ACM*, vol. 19, no. 5, pp. 279–285, 1976.
[44] B. Tang et al., "SPECWANDS: An efficient priority-based scheduler against speculation contention attacks," 2023, *arXiv:2302.00947*.
[45] N. Binkert et al., "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
[46] "Duktape Javascript engine." 2020. [Online]. Available: https://duktape.org
[47] "Sunspider Javascript benchmarks (1.0)." Webkit. 2020. [Online]. Available: https://webkit.org/perf/sunspider/sunspider.html
[48] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Proc. Cryptograph. Track RSA Conf.*, 2007, pp. 225–242.
[49] A. Limaye and T. Adegbija, "A workload characterization of the SPEC CPU2017 benchmark suite," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2018, pp. 149–158.
[50] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 469–480.

Bowen Tang is currently pursuing the Ph.D. degree with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

His research interests include system security, bug detection, and virtualization.

Chenggang Wu received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences (CAS), Beijing, China.

He is a Professor with the Institute of Computing Technology, CAS. His research interests include the dynamic compilation, virtualization, bug detection on concurrent program, and system security.

Prof. Wu has served on the program committees of many major conferences.

Pen-Chung Yew (Life Fellow, IEEE) received the Ph.D. degree from University of Illinois at Urbana–Champaign, Champaign, IL, USA.

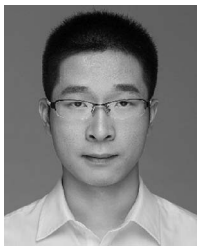
He is a Professor with the CSE Department, University of Minnesota-Twin Cities, Minneapolis, MN, USA, where he was the Head of the Department and the Holder of the William-Norris Land-Grant Chair Professor from 2000 to 2005. His current research interests include system virtualization, compilers, and architectural issues-related multicore/many-core systems.

Yinqian Zhang received the Ph.D. degree from the University of North Carolina at Chapel Hill, Chapel Hill, NC, USA.

He is a Professor with the CSE Department, Southern University of Science and Technology (SUSTech), Shenzhen, China. Before joining SUSTech in 2021, he was an Associate Professor with the CSE Department, Ohio State University, Columbus, OH, USA. His research interest is computer system security, with particular emphasis on cloud computing security, OS security, and side-channel security.

Mengyao Xie received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2022.

Since then, she has been working with the Institute of Computing Technology, Chinese Academy of Sciences. Her research interests include system security and virtualization.



Yuanming Lai received the M.S. degree from the Huazhong University of Science and Technology, Wuhan, China, in 2016.

He is currently with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include system security and machine learning.

Yan Kang received the M.S. degree from the Beijing University of Aeronautics and Astronautics, Beijing, China, in 2017.

She is currently working with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing. Her research interests include software and system security.

Wei Wang received the M.S. degree from Capital Normal University, Beijing, China, in 2021.

He is currently working with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include software security, adversarial attack, and robustness.

Qiang Wei is a Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. His main research interests are network and information system security, including software vulnerability analysis and cloud computing security.

Zhe Wang received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences (CAS), Beijing, China

He is an Associate Professor with the Institute of Computing Technology, CAS. His research interests are in dynamic binary translation, multithreaded program record-and-replay, operating systems, system virtualization, and memory corruption attacks and defenses.