SPECBOX: A Label-Based Transparent Speculation Scheme Against Transient Execution Attacks

Bowen Tang, Chenggang Wu, Zhe Wang, Lichen Jia, Pen-Chung Yew, Yueqiang Cheng, Yinqian Zhang, Chenxi Wang, Guoqing Harry Xu

Abstract—Speculative execution techniques have been a cornerstone of modern processors to improve instruction-level parallelism. However, recent studies showed that this kind of techniques could be exploited by attackers to leak secret data via transient execution attacks, such as Spectre. Many defenses are proposed to address this problem, but they all face various challenges: (1) Tracking data flow in the instruction pipeline could comprehensively address this problem, but it could cause pipeline stalls and incur high performance overhead; (2) Making side effect of speculative execution imperceptible to attackers, but it often needs additional storage components and complicated data movement operations. In this paper, we propose a *label-based transparent speculation* scheme called SPECBOX. It dynamically partitions the cache system to isolate speculative data and non-speculative data, which can prevent transient execution from being observed by subsequent execution. Moreover, it uses thread ownership semaphores to prevent speculative data from being accessed across cores. In addition, SPECBOX also enhances the auxiliary components in the cache system against transient execution attacks, such as hardware prefetcher. Our security analysis shows that SPECBOX is secure and the performance evaluation shows that the performance overhead on SPEC CPU 2006 and PARSEC-3.0 benchmarks is small.

Index Terms—Transient Execution Attack, Cache Partition, Shared Cache Access Control

1 INTRODUCTION

Five decades of exponential growth in processor performance has led to today's ubiquitous computation infrastructure. At the heart of such rapid growth are many optimizations employed in today's processors. Among them, using speculative execution to alleviate pipeline stalls caused by control flow transfer and memory access is one of the most effective optimizations in modern processors. However, recent studies have shown that this kind of techniques can introduce security vulnerabilities and be exploited by attackers via *transient execution attacks (TEAs)* such as the well-known Spectre [1]. These vulnerabilities are widely present in billions of processors produced by mainstream manufacturers such as Intel, AMD and ARM.

Listing 1 is a proof-of-concept (PoC) code in the Spectre attack. First, the attacker steers the index value to be less than `SIZE`, thereby training the branch predictor to choose the fall-through branch in Line 5. After that, the attacker steers the index value to be greater than `SIZE`, which leads to an out-of-bound access to the secret during the speculative execution. Then, the attacker utilizes the secret to index the `dummy` array. This will load the `dummy[val*64]` item into the cache. Eventually, when the processor determines that it is a mis-prediction in Line 5, it will squash the instructions of Lines 6-7 and follow the correct path. In the current design, the processor will not clean up the side effects in the cache (i.e. the data brought in during the mis-speculative execution). The attacker can scan the `dummy` array in the cache, and measure the access time of each array element. According to the access latency, the attacker can decide which item has been loaded into the cache, and thereby infer the secret value.

```
1  #define SIZE 10
2  uint8 array[SIZE];
3  uint8 dummy[256*64];
4  uint8 victim(int index) {
5     if (index < SIZE) {
6        uint8 val = array[index];
7        return dummy[val*64];
8     } else return 0;
9  }
10 victim(8);// Step1: train the branch
11 flush_cache();// Step2: prepare cache layout
12 victim(100); // Step3: access the secret
13 timing_dummy(); // Step4: measure cache layout
```

Listing 1: A proof-of-concept (PoC) code of the Spectre attack.

To mitigate TEAs, several approaches have been proposed in recent years. The first category of approaches is to delay the use of the data until the instructions that produce the data are no longer speculative, i.e. the data is no longer "tainted" [2] and "safe" to use. For the example in Listing 1, the out-

- *Zhe Wang is the corresponding author.*
- *Bowen Tang, Chenggang Wu, Zhe Wang, Lichen Jia are with Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with University of Chinese Academy of Sciences, Beijing 100049, China. Email: {tangbowen, wucg, wangzhe12, jialichen}@ict.ac.cn*
- *Pen-Chuang Yew is with Computer Science & Engineering Department of the University of Minnesota-Twin Cities, MN 55455, USA. Email: yew@umn.edu*
- *Yueqiang Cheng is with the Head of Security Research at NIO USA. E-mail: strongerwill@gmail.com*
- *Yinqian Zhang is with Department of Computer Science and Engineering of the Southern University of Science and Technology, Shenzhen 518055, China. Email: yinqianz@acm.org*
- *Chenxi Wang and Guoqing Harry Xu are with Computer Science Department of University of California, Los Angeles, CA 90095-1596, USA. Email: {wangchenxi, harryxu}@cs.ucla.edu*

of-bound index value in Line 6 cannot be used to access the `dummy` array until the branch condition in Line 5 is resolved. The representative schemes are STT [2], SDO [3], NDA [4], ConditionalSpec [5], Delay-on-Miss [6] and SpecShield [7]. These defenses comprehensively prevent the execution of instructions that may cause information leakage. However, delaying data propagation can cause pipeline stalls, which may incur high performance overhead [2], [4], [6], [7].

The second category is to keep speculative execution non-blocking but make it "invisible" to the subsequent instructions if it fails. In Listing 1, the array elements of `dummy` loaded into the cache during the speculative execution will be removed and cannot be accessed in the `timing_dummy()`. From the perspective of microarchitecture designers, this approach is preferable because of its compatibility with other optimization mechanisms that are critical to processor performance [8]. The representative works are SafeSpec [9], InvisiSpec [10], CleanupSpec [11] and MuonTrap [8].

To achieve the invisible speculative execution, InvisiSpec, SafeSpec, and Munontrap choose to add an extra storage to keep the speculatively installed data. When the speculative instructions are committed, the speculatively installed data will be *re-installed* into the cache hierarchy from the memory system or the newly added storage to make it visible. In contrast, CleanupSpec allows the data to be installed in the cache hierarchy during the speculative execution, and the replaced data is stored into a newly added storage. The replaced data will be *re-installed* into the cache hierarchy to rollback the cache state only when speculation fails. Since most speculation will succeed, CleanupSpec has a higher performance in general [11]. But, the *re-install* operations are required no matter whether the speculation succeeds or fails. Such additional data movement on the cache hierarchy can reduce its benefit and degrade its performance.

In this paper, we re-exam the invisible speculative execution strategy along with the cache design, and propose a new transparent speculation scheme, called SpecBox. It modifies the cache to support the invisible speculative access efficiently. The speculative data and the non-speculative data are distinguished in the cache, so the extra storage and data movement are no longer needed. To achieve this, SpecBox divides each cache set into two domains. Each cache line in the set is distinguished by a 1-bit label to indicate which domain it is in. The *temporary domain* contains the speculative data and the *persistent domain* contains the non-speculative data. When the speculation fails, the speculatively installed cache lines in the temporary domain will be invalidated. When the speculation succeeds, SpecBox only needs to flip the bits to switch the corresponding cache lines from the temporary domain to the persistent domain. Thus, it totally avoids the data movement required in other schemes, and hence, improves the performance.

In addition to isolating the speculative data in the single-core, we also extend the *label-based method* to the multi-core environment and make the speculative data invisible across cores. *Physically* isolating hardware threads' speculative data from each other can also achieve this, but it limits the scalability and has low resource utilization. The core idea in SpecBox is *to dynamically mark the thread ownership of each shared cache line in the temporary domain, and emulate a behavior that is similar to accessing the thread-private cache*. To achieve that, SpecBox attaches each cache line with an N-bit label, with each bit bound to a hardware thread (HT). If the bit corresponding to a HT is set, it means this HT owns this cache line. When a HT accesses a cache line which is not owned by this HT in the temporary domain, SpecBox will simulate a latency equivalent to a miss and then sets the corresponding bit, even if it has been speculatively installed by other threads. When a HT evicts a cache line owned by other HTs in the temporary domain, SpecBox will just reset its corresponding bit. The set/reset actions are similar to the P/V semaphore operations in parallel programming, so we call the N-bit label *thread-owner semaphore (TOS)*.

To summarize, we make the following contributions:

1) We propose a new *label-based transparent speculation* scheme, called SpecBox, against transient execution attacks. It leverages a cache partitioning approach on speculative data, which eliminates the need for data movement during the switch between speculative and non-speculative execution.
2) We propose a thread-ownership semaphore to logically isolate shared data among threads to prevent side channels to be formed in the shared cache.
3) The security analysis shows that the proposed *label-based transparent speculation* scheme is secure. And the performance evaluation shows that the overhead of SpecBox is substantially lower than that of STT, InvisiSpec and CleanupSpec on SPEC CPU 2006 and PARSEC-3.0 benchmarks.

The rest of the paper is organized as follows. Section 2 reviews the transient execution attacks and existing defenses. Section 3 explains the threat model. Section 4 summaries the challenges SpecBox confronts. Section 5 details the design of SpecBox. Section 6 presents the security analysis of SpecBox. Section 7 provides the security and performance evaluation. Discussion, related works, and conclusion are provided in section 8, section 9, and section 10 respectively.

2 Background

2.1 Speculative Execution

Speculative execution techniques, such as branch prediction [12] and memory disambiguation [13], are commonly used on modern out-of-order processors to improve performance. To avoid pipeline stalls, the processor continues to speculatively execute instructions beyond a branch instruction along a predicted path before the branch condition is resolved. If the prediction fails, the mis-speculated instructions will be squashed. A *re-order buffer* (ROB) is used to maintain correctness after the out-of-order execution. When an instruction reaches the head of the ROB and has completed its execution, it updates the machine state and releases its held resources, this process is called *commit*. An instruction not yet committed is called an *in-flight instruction*.

On modern processors, the side effects caused by the mis-speculated instructions such as the data brought into the cache memory during the speculative execution are not cleaned up. It will not affect the correctness of the program execution, but could impact the timing of subsequent instructions and can be used by attackers to create covert channels.

2.2 Timing-Based Covert Channel

The core logic of a transient execution attack can be divided into three steps: a) Accessing the secret through speculative execution; b) Encoding the secret through side effects that affect the subsequent execution timing; c) Decoding the secret through executing some operations and time their execution [14]. The first two steps are collectively referred to as the *sender*, and the third step is referred to as the *receiver*. The transmission channel between the sender and receiver is known as *timing-based covert channel* [15].

According to the length of the information carrier's life span, timing-based covert channels can be grouped into two types: the *persistent* and the *volatile* [16]. The information carrier in a *persistent covert channel* is usually the layout of a storage unit, such as the cache [17], [18], *translation lookaside buffer* (TLB) [19], and Paging Structure Cache [20], or a state change such as the on/off state of high bits in the AVX2 vector register [21]. In a *persistent covert channel*, the side effects encoded by the sender will exist for a relatively long period of time to allow the receiver to extract the secret value. In contrast, the information carrier of a *volatile covert channel* is usually a shared resource between threads, such as floating-point unit (FPU) [22], execution unit port [23] and memory bus [24]. The side effect exploited by the sender is to delay the execution time of the receiver by competing for such resources. Because resource competition is transient, the receiver must measure the timing while the sender is accessing the share resources. In this case, the sender and the receiver must be synchronized, which increases the difficulty for the attacker.

2.3 Transient Execution Attacks

Transient execution attacks (TEA) can be divided into two categories: the *Spectre-type* and the *Meltdown-type* attacks [14]. Attackers can directly access unauthorized memory locations or registers during the out-of-order execution in a Meltdown-type attack. Unlike the Spectre-type attacks, which primarily exploiting speculative execution, the Meltdown-type vulnerabilities are mostly due to unintended hardware bugs. They can thus be fixed in the hardware. For Spectre-type attacks, the attacker will first train the prediction units, such as the *pattern history table* (PHT), *branch target buffer* (BTB) and *return stack buffer* (RSB). After that, the attacker will bypass the protection code to execute a wrongly speculative path, such as a bound check [25], data cleanup [26] and stack pointer switching [27], and then access the secret. Finally, the attacker will transmit the secret through a timing-based covert channel.

2.4 Existing Defenses Against TEAs

For TEAs, a common defense is to prevent sensitive data from being transmitted to covert channels. SpecShield [7] checks whether there is an unresolved branch or an instruction that triggers an exception before a load instruction, and then determines whether the loaded data can be passed to subsequent instructions. ConditionalSpec [5] puts forward the concept and a detection method of "safe dependence", and proposes an S-pattern filtering strategy to improve the detection efficiency according to the characteristics of TEAs. NDA [4] and STT [2] use dataflow tracking similar to taint propagation, and track those instructions that may cause information leakage. Those instructions are forced to delay until their dependent instructions become safe. To reduce the overhead caused by the delay and the tracking analysis, SDO [3] adds a safe value prediction mechanism based on STT. But the performance overhead is still high due to the pipeline stalls that cannot be avoided.

Make speculative execution "invisible" by cleaning up all side effects after speculative execution could avoid the pipeline stalls, and prevent side effects from being used by attackers to transmit the secret data. InvisiSpec [10] chooses to add a *speculative buffer* to keep the speculatively installed data. Similarly, SafeSpec [9] and MuonTrap [8] use a *shadow cache* and a *non-inclusive L0 cache*, respectively. If speculation succeeds, the speculative installed data will be re-installed into the newly introduced storage; if speculation fails, the speculatively installed data in these storages will be cleaned up. Since most speculation will succeed, these methods will introduce non-trivial overhead. Hence, CleanupSpec [11] was proposed to allow the data be speculatively installed into the original cache, and only rollbacks the cache state via re-installing the replaced data when speculation fails.

Actually, the *re-install* operations are required no matter whether the speculation succeeds or fails. Such data movement triggered by the re-install operations will degrade performance. It motivates us to develop a scheme that can perform an "invisible" speculative execution inside the cache system without requiring any "data movement" during the switch between speculative and non-speculative execution.

3 Threat Model

In this paper, we mainly focus on the cache system that is vulnerable to the transient execution attacks. Here, we assume attackers have the following abilities:

- Ability to train the control flow prediction units and memory disambiguation units in order to exploit all Spectre- and Meltdown-like vulnerabilities.
- Ability to find/execute the gadgets and to access/encode secret data. The secret data here refers to various protected memory locations and registers.
- Ability to know the cache indexing method and its replacement strategy, which means the attacker can install or evict a cache line at any location in the cache.
- Ability to launch multiple threads located on a SMT core and/or different cores, and control their interleavings in order to transmit the stolen secret data via cache.

Our goal is to defeat TEAs that use *persistent covert channels*. For TEAs using *volatile covert channels*, such as port contention [23], [28], it can be mitigated by turning off SMT, using security-sensitive thread scheduling [29], or using time-division multiple accessing (TDMA) on shared resources [30]. These methods are orthogonal to our approach.

Also, we do not consider *physical* covert channels, such as electromagnetic signals [31] and power consumption [32], [33]. These channels are generally noisier, requiring longer time for attackers to observe the effects. At present, these types of covert channels cannot be used in transient execution attacks.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3144287, IEEE Transactions on Dependable and Secure Computing

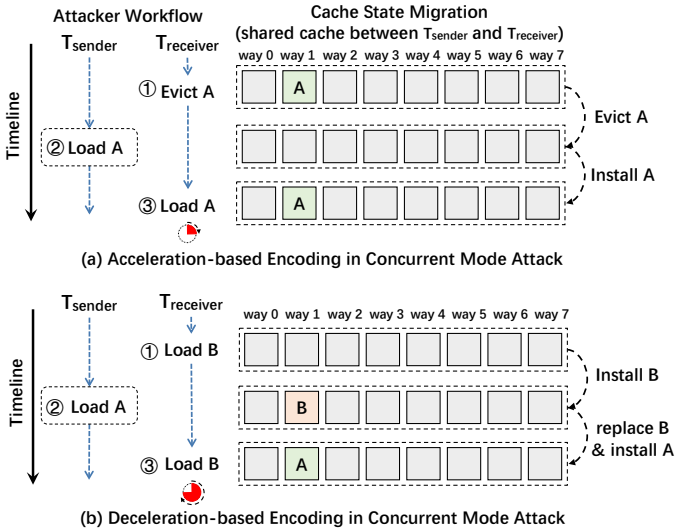


Fig. 1: The Encoding Methods in Concurrent Mode Attacks.

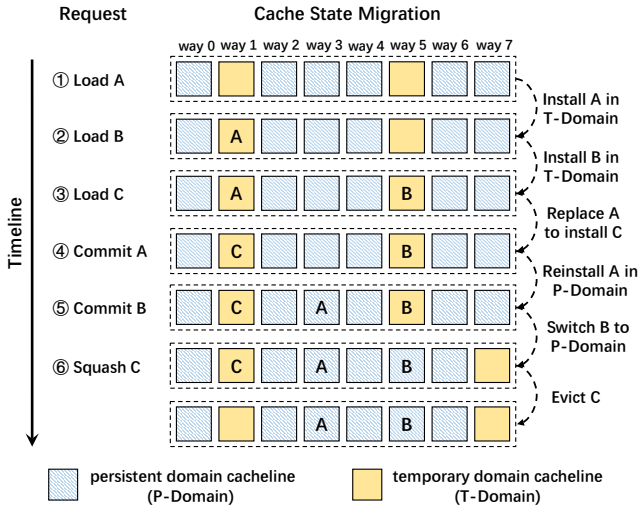


Fig. 2: An example of the state migration on one cache set.

4 Problem Analysis

4.1 Encoding Methods of Cache Covert Channel

As mentioned in subsection 2.2, the *sender* illegally accesses the secret and encodes it into the cache layout; the *receiver* infers the secret via timing access to the cache. The encoding schemes can have the following two types.

1) **Acceleration-based encoding.** The *sender* first evicts an item from the cache, then decides whether to re-install this item or not according to the speculative accessed secret value. Taken a 1-bit secret as an example, if the secret value is 1, the item will be re-installed back. Therefore, when the *receiver* accesses this item again, the timing will be short due to the cache hit. The *receiver* can reason about the secret value being 1. Otherwise, it is 0. Most Flush+Reload [18] attacks adopt such an encoding scheme.
2) **Deceleration-based encoding.** The *sender* first installs an item into the cache, then decides whether to replace this item with another item or not according to the secret value. Also taken a 1-bit secret as an example, if the secret value is 1, the new item will replace the original item. Therefore, when the *receiver* accesses the original item again, the timing will be long due to the cache miss. The *receiver* can conclude that the secret value is 1. Otherwise, it is 0. Most Prime+Probe [17] attacks adopt such an encoding scheme.

4.2 Attack Modes

In most common attack scenarios, the above two encoding methods are performed in a *serialized* manner, that is, the receiver's decoding will not begin until the sender's encoding has been completed. However, in multi-thread applications, the encoding can be conducted in a *concurrent* manner. The attacker can launch two concurrent threads located on different cores or a SMT core, with one thread acting as the sender and the other as the receiver. Using browser as an example, multiple threads of a malicious plug-ins can occupy different cores. Their targets can be user's passwords or cookies, which are protected by the sandbox in the browser. The malicious plug-ins cannot access them directly, but can exploit the *Spectre-like* vulnerability to access them in a wrong-path speculative execution, and leverage the shared cache as a covert channel to transmit them.

Figure 1 shows how the two encoding methods can be implemented in a concurrent mode. For the acceleration-based method, the $T_{receiver}$ first evicts the item and then the T_{sender} speculatively accesses the secret and re-installs the item according to the 1-bit secret value (assuming the value is 1). Immediately afterwards, the receiver access the item again and measures the access time. At that time, the speculation of T_{sender} has not been committed yet, which means other defenses, such as *cleanup* or *rollback*, has not yet to be carried out. The $T_{receiver}$'s revisit will be hit. Similarly, for the deceleration-based method, after the T_{sender} has replaced the item primed by the speculative install of $T_{receiver}$, and before the process of T_{sender} is committed, the $T_{receiver}$ will experience a cache miss when the replaced item is accessed again.

5 Our Solution

To defeat the above attacks in both serial and concurrent modes and keep the speculative execution efficient, we proposed a *label-based transparent speculation* scheme, called SpecBox. In SpecBox, all speculative (i.e. *in-flight*) operations, such as loads/stores and instruction fetching that may affect the cache system, are regarded as unsafe until they reach the head of ROB and are committed. As mentioned earlier, the core ideas of SpecBox consists of two major components: domain partition and thread ownership semaphore. The details of them are provided in the following subsections subsection 5.1 and subsection 5.2. In addition, for other auxiliary components in the cache system such as coherence states, TLB and hardware prefetcher, SpecBox also secures them from being affected by the threatening speculative execution, which are presented in subsection 5.3.

5.1 Domain Partition

5.1.1 Partitioning Scheme

As shown in Figure 2, SpecBox partitions each cache set into two domains: (1) the *temporary domain* that contains the data

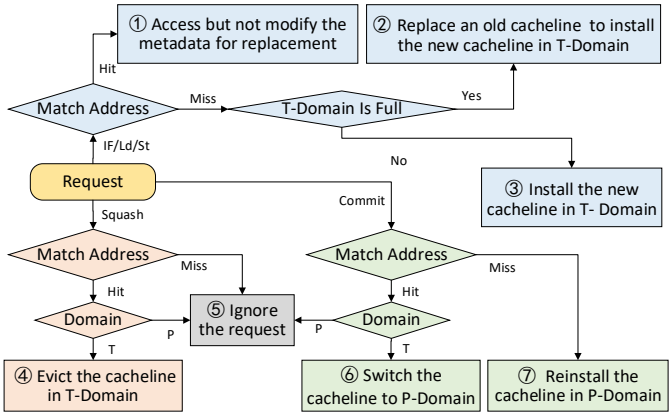


Fig. 3: Data access flow in temporary/persistent domains

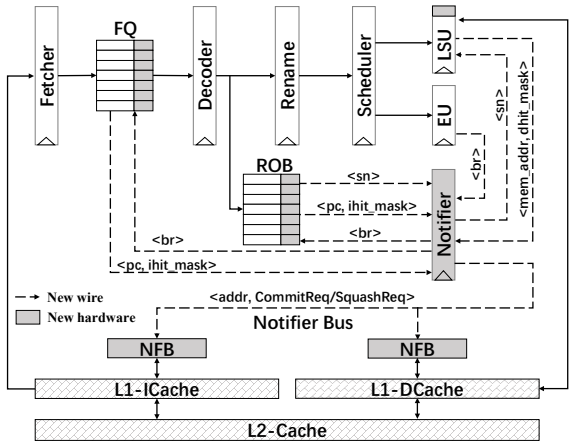


Fig. 4: The enhanced pipeline and cache system with the *notifier*.

installed by the *in-flight* operations, and (2) the *persistent domain* that contains the data installed by the *committed* operations. The partition is implemented by adding a 1-bit label, called T/P (Temporary/Persistent) flag, on each cache line's tag area. If the cache line's T/P flag is 0, it belongs to the temporary domain; otherwise, it belongs to the persistent domain. By modifying the T/P flag, SPECBOX can switch the domain the cache line belongs to. The domain needs not be contiguous in physical space but need to have a pre-determined capacity, so that it will not compete against each other. When installing a new cache line or switching a cache line's domain from the other domain, SPECBOX will check whether the current domain has reached its capacity. If it has, it will wake up the replacement module to select a cache line within the domain and replace it.

5.1.2 Access Control

Figure 3 shows the access flow on a cache with the proposed partitioning scheme. The access flow starts from the yellow box marked as "*Request*". ① If the request is an *in-flight* operation and is a cache "hit", regardless of whether it hits the temporary domain or the persistent domain, the operation can directly access the data without modifying the metadata for cache replacement. ② If it is a cache miss and the temporary domain is full, the cache will select one victim cache line from the temporary domain and replace it. ③ If the temporary domain is not full, it will install a new cache line in the temporary domain. ④ When an *in-flight* access operation is squashed, the cache will receive a squash request from CPU. It checks whether the data installed by the operation is still in the temporary domain and evicts the entry if so. ⑤ If the data installed is not in the temporary domain, it just ignores the request. ⑥ When an *in-flight* operation is committed, the cache will receive a commit request from CPU. It will check whether the data installed by the operation is in the temporary domain and if so, the cache converts the cache line to the persistent domain by setting the T/P flag. Meanwhile, to keep the capacity constant, the cache will evict one victim cache line from the persistent domain and switch it to the temporary domain. If the data installed by the operation is already in the persistent domain, it follows case ⑤ and ignores the request. ⑦ If the data is neither in the persistent domain nor in the temporary domain, the cache will re-install it in the persistent domain.

Figure 2 depicts an example of the state transition (i.e. domain switches) on one cache set during some request accesses. This set is 8-way associative, and the capacity ratio of the temporary and persistent domains is 2:6. Firstly, the cache receives two consecutive load requests, i.e., *Load A* (①) and *Load B* (②) in the figure. Since the A and B are missing in the cache, these requests will trigger cache miss handling events. After A and B are fetched from the next level cache or memory, they are installed into the temporary domain; Then, the cache receives the *Load C* request (③) and C is also missing. But, there is no more cache line in the temporary domain in the set, so the cache chooses to replace A with C in accordance with the replacement algorithm. After that, the *in-flight* operation to which the *Load A* request belongs is committed (④), the cache receives a *Commit A* request. But, A has been evicted from the set. The cache thus selects a victim cache line from the persistent domain and re-installs A. After that, the cache receives a *Commit B* request (⑤). It first selects a victim cache line from the persistent domain to evict, and switch it to the temporary domain, and then switch B to the persistent domain. Finally, the cache receives a *Squash C* request (⑥). It evicts the cache line that contains C.

5.1.3 Notify Framework

We added a notifier in the commit stage of the pipeline (shown in Figure 4). It notifies the cache system to switch domain or clean up the corresponding cache line when an *in-flight* operation is committed or squashed. The notifier performs different actions according to the types of operations.

For Load/Store. In order to know which level of the cache hierarchy has been accessed by a load/store instruction, SPECBOX adds some bits as *dhit_mask* in each entry of the *Load/Store Unit* (LSU). They record whether the access to the cache level has been hit or not. When the ROB commits or squashes a load/store instruction, it will signal the notifier with the *sequence_number* (sn) of ROB. Subsequently, the notifier acquires the LSQ with the *sn* for the *memory_address* and *dhit_mask*, and then generates the notification request.

For IFetch. Although the CPU does not track the instruction fetching process directly, we could still infer the process by analyzing the effects of instruction fetching. One portion of the instructions have been decoded and dispatched to the

ROB, and the other portion have not yet been decoded in the *Fetch Queue* (FQ). Similar to LSU, SPECBOX adds the *ihit_mask* bits in the entries of the FQ and ROB. When the *Execution Unit* (EU) resolves a branch and accepts/rejects the prediction of the branch, it will signal the notifier with the *resolved_branch* (br) to obtain the *PC_address* and *ihit_mask* of all the fetched instructions located in that branch from the FQ and ROB, and then generate the notification requests.

In order not to disturb the instruction fetching and load/store operations, the notification requests will be first sent to the L1 I-cache and L1 D-cache through the *Notifier Bus*. Then the L1 cache controllers filters these requests and forward the requests to the lower-level cache through the interconnect. Moreover, in our study, we find that there is a significant amount of notification requests for the same cache line caused by contiguous cache accesses (mostly due to instruction fetching). Therefore, SPECBOX adds a *Notification Fill Buffer* (NFB) in *Notifier Bus* with 16 entries (i.e. the size of the cache line) to merge the redundant requests to the same cache line.

5.2 Thread Ownership Semaphore (TOS)

Inter-thread isolation by allocating a private memory region to each thread could make the speculation invisible across cores, but it will lower the resource utilization and limit the scalability. In this paper, we propose a label-based solution called *thread ownership semaphore* (TOS). It is based two insights: 1) it's not necessary to keep it invisible for the data installed by the *committed* operations; 2) The data installed by the *in-flight* operations of one thread need not be forever invisible to other threads. If another thread has also accessed the data during the *in-flight* operations, it should be safe to share the data between the two threads. So, we conclude that the problem of maintaining invisibility between the threads is equivalent to the problem of tracking the ownership of each thread and emulate its "private" *temporary domain*.

5.2.1 Access Control

TOS is an N-bit label added to each cache line's tag area. Each bit of TOS is bounded to a hardware thread (HT), a HT owns a cache line only if the corresponding bit is set. The ownership means the cache line can be accessed by the *in-flight* operations of the HT. When a HT accesses the cache line it disowns, the SPECBOX will emulate a latency equivalent to a cache miss and then set its corresponding TOS bit. On the other hand, if a HT evicts a cache line owned by other HTs, the SPECBOX will simply reset the TOS bit instead of evicting the cache line. In summary, for each cache line in the temporary domain, SPECBOX performs different actions according to the different situations:

- A thread can directly access a cache line it owns.
- When a thread accesses a cache line that it does not own, SPECBOX needs to check whether it is being owned by other threads or not: a) if it is *not* owned by any other thread, which means it is invalid or unused, then the thread can install the requested data in it; b) if it is being owned by another thread, then a cache miss will be emulated by SPECBOX to hide its true access latency. In either case, after the above process, the cache line will be marked as being owned by this thread.

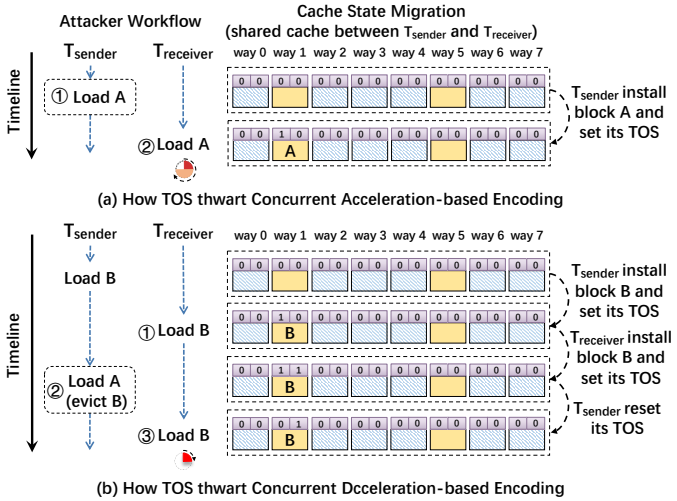


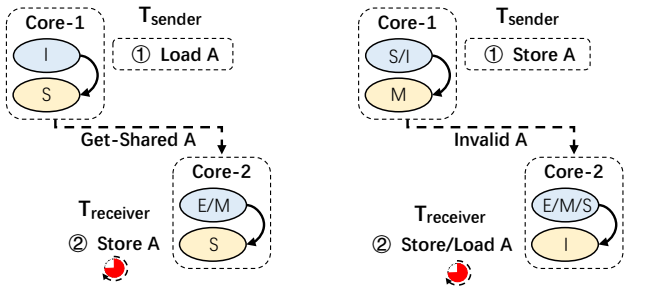
Fig. 5: The workflow of the TOS scheme under two encoding attacks.

- When a cache line needs to be replaced or squashed in a thread's *in-flight* operation, SPECBOX marks the thread as no longer owning this cache line, and checks whether other threads own this cache line or not. If they do, SPECBOX does nothing; otherwise, SPECBOX replaces or squashes this cache line.
- When a cache line needs to be committed in a thread's *in-flight* operation, this cache line is directly switched to the persistent domain.

Figure 5 shows how TOS mechanism can thwart the two encoding methods in the concurrent mode. For *acceleration-based encoding*, a *Load A* operation of T_{sender} installs cache line A in the *temporary* domain and set its TOS bit. Then, the $T_{receiver}$ also executes a *Load A* operation. Because $T_{receiver}$ is not owning the cache line A, it will experience a latency emulated by the SPECBOX, and regard it as a cache miss. For *deceleration-based encoding* as shown in Figure 5, the T_{sender} first installs the cache line A in the temporary domain by a *Load B* operation. Subsequently, the $T_{receiver}$ performs another *Load B* operation. Once the $T_{receiver}$ completes, the TOS of the two threads are both set. And then the T_{sender} performs a *Load A* operation, which tries to replace the cache line B. At that time, SPECBOX just resets the TOS of T_{sender} instead of evicting the cache line, and suspends the *Load A* request until it is committed or the $T_{receiver}$ resets the TOS also. Therefore, $T_{receiver}$ cannot detect the deceleration effect caused by T_{sender}.

5.2.2 Implementation of TOS on different systems

The TOS mechanism can also be easily implemented by adding labels in tag area and modifying the access controller of the cache. (1) For a hierarchical storage in a system that does not support SMT, the only component with concurrent issues is shared cache (i.e. LLC). The TOS can be efficiently implemented with the cooperation of cache-coherence directory and the T/P flag. When the T/P flag is P, the cache line can be accessed as usual because the TOS mechanism only targets speculative entries. When the T/P flag is T, the cache controller will process the accessing requests according to the above workflow using the ownership information recorded in the directory. (2) For a hierarchy storage in a



(a) Speculative Load Attack (b) Speculative Store Attack

Fig. 6: The coherence-state covert channel attacks.

system that supports SMT, the private L1/L2 caches are also vulnerable. We need to add N bits to each cache line for the TOS, where N is the number of SMT threads. Each bit indicates the ownership of the corresponding thread. And for LLC, we also need to extend the directory and allow it to distinguish different SMT threads on each core. Therefore, for N SMT threads per core on an M-core system, the LLC needs to add $(N-1)\cdot M$ bits for each entry.

5.3 Other Enhanced Components

In addition, SPECBOX also blocks other potential attack surfaces from the auxiliary components in the cache system during the speculative execution, such as coherence states, cache management instructions and hardware prefetcher.

Coherence States. The above-mentioned attacks use the cache layout as a medium for encoding. However, on a multi-core processor, cache coherence states may also be an attack surface. With the help of the *Single-Writer-Multi-Reader* (SWMR) characteristic in the *MESI* protocol, the attacker can complete *deceleration-based encoding* attacks via the following two approaches [34] as Figure 6 shows: (a) The T_{sender} executes a *Load A* in an *in-flight* operation on core-1. If the state of the cache line A on core-1 is *Invalid*, the cache controller will send *Get-Shared* requests to other cores, which can cause the copy of the cache line A on core-2 changing from *Exclusive/Modified* to the *Shared* state. After that, if the $T_{receiver}$ stores the cache line A on core-2, it will experience a longer latency because of the need to get an *Exclusive* state again; (b) In some highly-optimized processors such as Intel P6 and MIPS R10000, the LSU will preload the requested cache line and issue a *Get-Exclusive* request for the *in-flight* store instruction. On such processors, if the T_{sender} speculatively executes the store operation on core-1 and find that the state of the private cache line is not in the *Exclusive/Modified* state, the cache controller will send an *invalidation* request to core-2. Then, the $T_{receiver}$'s subsequent access will experience a cache miss.

Cache Management Instructions. In addition to speculative load/store instructions, an attacker can leverage some instructions for cache management during speculative execution to encode the secret, such as *prefetch*, *clflush* and *INVD*. For these rare but potentially-vulnerable instructions, SPECBOX stalls their execution, if they are issued during the speculative execution, and waits until they are committed to keep them from being used in TEAs.

Hardware Prefetcher. SPECBOX also supports *hardware prefetching* because it is critical to the program performance.

TABLE 1: Defense principles against various abstract attack scenarios. "A" and "B" represent thread A and other threads that share the cache with thread A, respectively; "t" represents the temporary domain; "p" represents the persistent domain; "evict" means a thread evicts a cache line; "install" means a thread installs a cache line; "access" means a thread accesses a cache line. *Assuming the secret is a 1-bit value, the expected measured result for an attacker can be either "fast" or "slow".*

Line	Mode	$S_{prepare}$	S_{send}	$S_{receive}$	Defense Principles
1	Serialized	$A^{evict}_{t/p}$	$A^{install}_{t}$	$A_{t/p}$ (fast)	The cache line installed in S_{send} will be cleaned up before $S_{receive}$.
2	Serialized	$A^{install}_{p}$	A^{evict}_{t}	$A_{t/p}$ (slow)	The cache line in the persistent domain cannot be evicted (replaced) in S_{send}.
3	Serialized	$A^{install}_{t}$	A^{evict}_{t}	$A_{t/p}$ (slow)	The cache line installed into the temporary domain by an *in-flight* operation is evicted later. It will be reinstalled into the persistent domain when that *in-flight* operation needs to be committed.
4	Concurrent	$B^{evict}_{t/p}$	$A^{install}_{t}$	$B_{t/p}$ (fast)	When thread B firstly accesses the cache line installed into the temporary domain by thread A in $S_{receive}$, the cache miss will be raised.
5	Concurrent	$B^{install}_{t}$	A^{evict}_{t}	$B_{t/p}$ (slow)	The cache line installed into the temporary domain by thread B cannot be evicted by thread A in S_{send}.
6	Concurrent	$B^{install}_{p}$	A^{evict}_{t}	$B_{t/p}$ (slow)	The cache line in the persistent domain cannot be evicted by thread A in S_{send}.

To ensure the security, SPECBOX defers the speculative accesses to train the hardware prefetcher until they are committed. And the prefetched data will be directly installed into the *persistent* domain.

TLB and Paging Structure Cache. TLB misses and Paging Structure Cache misses may occur during the process of address translation for speculative memory accesses. Attackers can thus treat them as covert channels in TEAs. Considering the low frequency of TLB miss occurrence, SPECBOX simply delays the execution of the *in-flight* operation triggering TLB miss until it is committed.

6 SECURITY ANALYSIS

After analyzing various existing TEAs, such as Spectre-PHT/BTB/RSB/STL [25], [26], [35], [36], [37], Lazy FP [22], MDS [38], [39], [40], LVI [41] and CacheOut [42], we abstract the transmission process of persistent convert channel as a three-stage model: $S_{prepare} \rightsquigarrow S_{send} \rightsquigarrow S_{receive}$. This is inspired by the previous work [43], [44] that is used to analyze the defense against the traditional side channel attacks. $S_{prepare}$ represents the stage of preparing the covert channel component. S_{send} represents the stage of accessing secret through speculative execution and encoding the secret into the covert channel. $S_{receive}$ represents the stage of extracting and decoding the information from the covert channel.

Based on the above model, Table 1 enumerates all possible actions of each stage (i.e. evict or install operation performs on any domain) and the expected states of $S_{receive}$ (i.e. slow or fast of execution time), and shows how SPECBOX can block them. In SPECBOX, we assume that the attacker knows all protection schemes and can manipulate any item in any domain during the $S_{prepare}$. However, because the speculative execution of S_{send} is illegal and will eventually be squashed, the attacker can ONLY manipulate the data in temporary domain. For serialized mode attacks, the reliable timing method of $S_{receive}$ decide that it MUST begin after $S_{prepare}$ and S_{send} complete (i.e. be committed in ROB). But for concurrent mode attacks, the order constraint of

TABLE 2: Parameters of simulated micro-architecture.

Component	Parameter Value
Core	8-issue, out-of-order, 2Ghz
Pipeline	64-entry IQ, 192-entry ROB, 32-entry LQ, 32-entry SQ, 256 Int / 256 FP registers
BPU	Tournament branch predictor, 4096 BTB
Private L1-I Cache	32KB, 64B line, 4-way, 1 cycle RT latency, 4 MSHRs
Private L1-D Cache	64KB, 64B line, 8-way, 1 cycle RT latency, 4 MSHRs
Shared L2 Cache	2MB bank, 64B line, 16-way, 8 cycles RT local latency, 16 cycles RT remote latency, 16 MSHRs
Coherence Protocol	inclusive, Directory-based MESI protocol
Network	4×2 mesh, 128b link width, 1 cycle latency per hop
DRAM	RT latency: 50 ns after L2

S_{send} and $S_{prepare}$ can be relaxed. In the followings, we will elaborate the process in each attack mode and SPECBOX's response via a representative example.

Serialized-Mode Attacks. Take the third attack (Line 3 in Table 1) as an example. In the $S_{prepare}$ stage, thread A installs a cache line into the temporary domain through an *in-flight* operation. In the S_{send} stage, thread A evicts this cache line via the replacement. In the $S_{receive}$ stage, thread A times the access to this cache line. With SPECBOX, when the *in-flight* operation in $S_{prepare}$ is committed, if the corresponding cache line is not present, SPECBOX will reinstall it. So the slowdown of the access time cannot be measured in $S_{receive}$.

Concurrent-Mode Attacks. Take the sixth attack (Line 6 in the table) as an example. In the $S_{prepare}$ stage, thread B installs a cache line. In the S_{send} stage, thread A performs an *in-flight* operation to evict this cache line. In the $S_{receive}$ stage, thread B times the access to this cache line. With SPECBOX, the cache line will not be evicted in S_{send} because thread B owns this cache line. The access to this cache line by thread B in $S_{receive}$ will hit the cache, and the slowdown of the access time cannot be measured.

As mentioned before, besides normal data, attackers can also exploit some metadata as the covert channel, such as replacement trace bits or coherence state. Since SPECBOX prevents the unsafe speculative modification of such metadata like other works, and carefully extends the cache without introducing new attack surface. SPECBOX can also be used to defend against the metadata attacks, such as Speculative Interference [45].

7 EVALUATION

7.1 Experimental Setup

We implemented SPECBOX based on O3 CPU and Ruby cache system using Gem5 [46]. The parameter settings of each component are shown in Table 2, which are consistent with other studies. We evaluated the performance using 27 benchmarks from SPEC CPU 2006 and 12 benchmarks from PARSEC-3.0. *dealII* and *tonto* are not included because they cannot be simulated correctly by the version *fe187de9bd* of Gem5 we used w/o SPECBOX. For SPEC benchmarks, we use the *ref* input set and skip the first 10 billion instructions in the fast-forward mode, and then perform cycle-level simulation on the next 1 billion instructions. For PARSEC benchmarks, under the full system with 8 cores, we use the *simmedium* input set and simulate the instructions in the region of interest (ROI) at the cycle level.

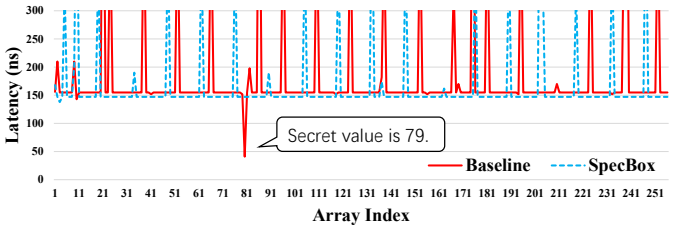


Fig. 7: Access latency of the auxiliary array using SPECBOX.

7.2 Defense Against Spectre PoC Code

To evaluate the mitigation effectiveness of SPECBOX, we chose the Spectre-PHT attack [25] as shown in Listing 1 that uses data cache as the covert channel. In the attack, we used a mis-predicted overflow access to obtain the secret whose value is 79, and then use the secret to index a 256-item auxiliary array. Finally, we measured the access latency of each array element 100 times. The access time is shown in Figure 7. We can see that when an access hits, the latency is below 50 cycles. In contrast, the latency exceeds 150 cycles if it misses. On the baseline processor without any protection, the attacker can clearly distinguish the access time difference of item 79 from other items in the array. However, using SPECBOX, the item is evicted from the data cache when the transient instruction is squashed. Therefore the attacker can no longer distinguish the difference in access time between item 79 and others.

7.3 Determine Domain Capacity

For SPECBOX, the capacity ratio of the two domains is crucial to the performance because it determines the maximum number of the speculative and non-speculative cache lines.

Two factors need to be considered when determining the domain capacity. The first is that the temporary domain should be sufficiently large to contain the data installed by the *in-flight* operations in any speculative window. These data will be frequently referenced within the current window and a period of time after being committed. Figure 8 shows the number of speculative cache lines in each cache sets sampled in the original cache system (i.e. w/o partitioning). We can see that, reserving 2 ways in L1-DCache and L1-ICache and 3 ways in the unified L2-Cache shared by multi-cores, are sufficient for most single- and multi-threaded programs.

The second is that the temporary domain should not be too large to deprive the capacity of the persistent domain. It is because the persistent domain contains all of the non-speculative and committed data in the cache. Its capacity is more sensitive to the programs that have more access patterns with long reuse distances, such as *mcf* and *GemsFDTD* in SPEC. Therefore, we counted the number of accesses to different ways in each cache set, and calculate the cumulative distribution in original (non-partitioning) cache system. The access order is decided by the replacement policy (e.g. LRU in SPECBOX). As the Figure 9 shows, for most programs, over 90% of accesses hit the first three ways in L1-DCache and L2-Cache, and the first way in L1-ICache. It thus will not impact the cache utilization much if we take away the last

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3144287, IEEE Transactions on Dependable and Secure Computing

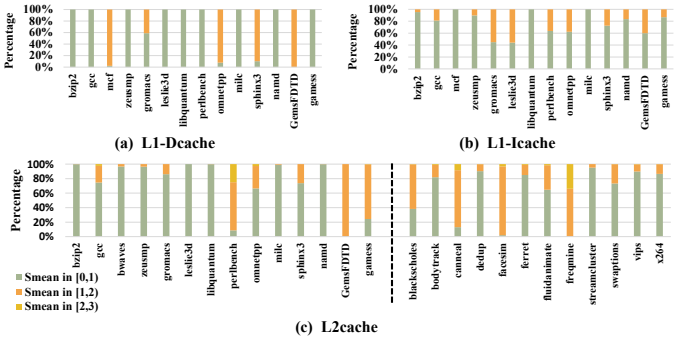


Fig. 8: We sampled a number of speculative cache lines in each cache set per Kilo instructions, and then calculate the mean of them as the representative value for each cache set (denoted as S_{mean}). The figure shows the percentage of the number of the cache sets with different S_{mean}. We found no cache set's S_{mean} is larger then 3. We selected some benchmarks with distinguished distributions as examples, and the remaining benchmarks are similar to these examples.

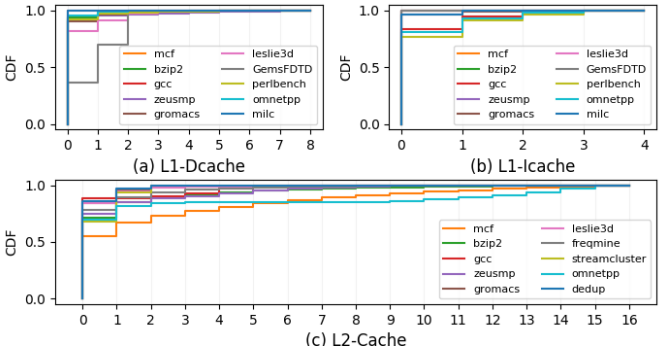


Fig. 9: We counted the distribution of hits in all cache sets and calculated the arithmetic mean as the representative distribution for each benchmark. The horizontal axis in the figure represents the cache lines in one cache set, which are arranged in LRU order from left to right, and the vertical axis is the cumulative distribution of hit counts. We selected some benchmarks with distinguished distributions as examples, and the remaining benchmarks are similar to these examples

2 or 3 ways for the temporary domain. This observation is consistent with other approaches that adopt way-partitioning schemes [47].

7.4 Performance Overhead

We compared SPECBOX with other open-sourced hardware-based defenses for TEAs. The first is STT [2]. It represents the scheme that delays the instructions that are dependent on the transient instructions. The second is InvisiSpec [10]. It represents the scheme that cleans up the side effect of transient instructions. The third is CleanSpec [11]. It represents the scheme that rollbacks the cache layout when speculation fails. In order to have a fair comparison with SPECBOX, for both InvisiSpec and STT, we choose the *futuristic model* that can resist all Meltdown-type and Spectre-type attacks; for CleanupSpec, we choose the rollback strategy for L1-DCache and L2-Cache. Because InvisiSpec and CleanupSpec are currently not used in the instruction cache, we implemented a version of SPECBOX only on data cache and not on instruction cache (i.e. SPECBOX-NI) to compare with other schemes.

In Figure 10 and Figure 11, we can see that the overall performance overheads of SPECBOX for SPEC and Parsec are 2.17% and 5.61%, respectively, which are much lower than 20.97% and 32.27% with STT, 21.29% and 11.62% with InvisiSpec, and also better than CleanupSpec (4.77% for SPEC). The largest overheads of SPECBOX are 8.96% for *GemsFDTD* in SPEC and 14.96% for *debup* in PARSEC, which are also better than InvisiSpec and STT. When we disable the protection on the instruction cache (i.e. L1-ICache and the corresponding regions in L2-Cache), the performance overheads of SPECBOX are reduced to 1.76% and 3.85%, respectively.

Versus STT. The main overhead of STT comes from pipeline stalls caused by the delayed execution. Therefore, for benchmarks with more dependent instructions that can cause pipeline stalls such as *lbm, ommnetpp* and *gcc* in SPEC, and *swaptions, ferret* and *bodytrack* in PARSEC, the performance of STT is inferior to InvisiSpec, CleanupSpec and SPECBOX. On the other hand, STT has the advantage that, if an instruction such as the load instruction whose dependent instructions have become safe, STT can simply lift the protection and allow it to modify the states for replacement and coherence. It has much less overhead for those programs with less dependence but with a large amount of cache accesses, such as *sphinix* and *GemsFDTD* in SPEC and *streamcluster* in PARSEC.

Versus InvisiSpec. The main cost of InvisiSpec comes from reload operations when a speculative instruction is committed. The main intention of the reload operation is to prevent subsequent replacement of the committed data in Speculative Buffer (SB) from being exploited by attackers. Another intention is that the SB is private to each core and, hence, will not receive invalidation requests from other cores, which may lead to the violation of the memory consistency model. It needs to reload the data from the cache hierarchy when the *in-flight* load instruction is committed, and validate the current value with the used value during the speculative execution. If the validation fails, it must squash and re-execute the instructions following that load instruction. From the Figure 10, we can see that for memory-intensive applications, such as *leslie3d, GemsFDTD and bip2*, the performance differences between InvisiSpec and other schemes are particularly distinct. The situation is more serious in some PARSEC benchmarks such as *canneal* and *facesim*. It is because more invalidation requests are generated in those benchmarks due to cache coherence transactions in a multi-core system, hence, the validation failures and re-execution will occur more frequently.

Versus CleanupSpec. CleanupSpec addresses the problem in InvisiSpec by allowing the replacement of a speculative instruction and restoring the layout if it is squashed. This scheme works well for most of the programs, such as *libquantum* and *omnetpp* in SPEC benchmarks. But for some programs with higher misprediction rates, such as *astar*, *bzip2* and *gobmk*, it will incur a certain amount of performance overhead (10%-20%), which is larger than that in InvisiSpec and SPECBOX. And for some cases with low misprediction rates but need to process subsequent squashed instructions, like *calculix* and *sphnix3*, the stall from rollback operations will also degrade the performance.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3144287, IEEE Transactions on Dependable and Secure Computing

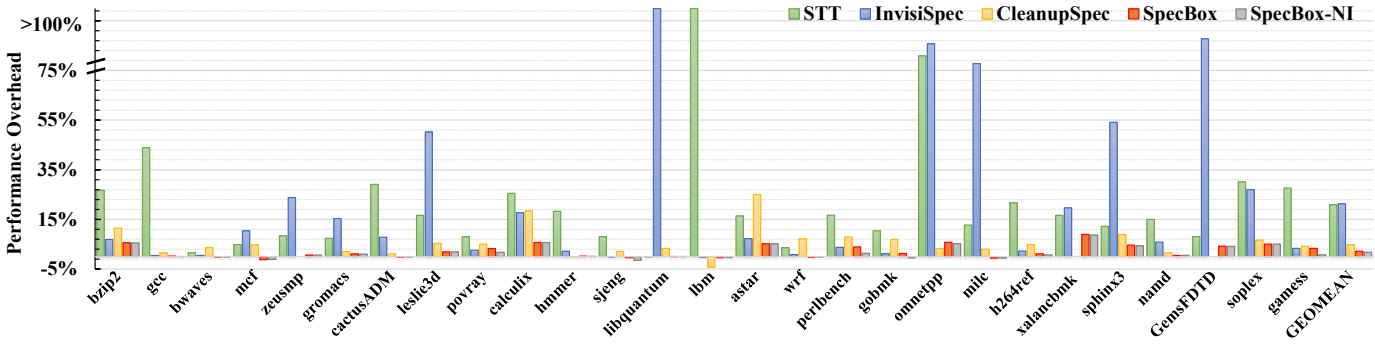


Fig. 10: The performance overhead of SPECBOX, SPECBOX with no-icache protected (denoted as SPECBOX-NI), CLeanupSpec [11], InvisiSpec [10] and STT [2] on SPEC CPU 2006 benchmarks. For CleanupSpec, three benchmarks (i.e. *zeusmp, xalancbmk, and GemsFDTD*) hang and failed in our simulation.

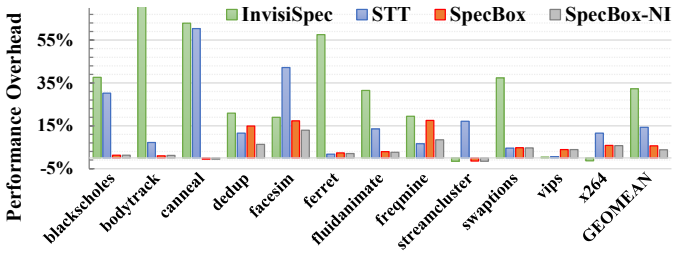


Fig. 11: The performance overhead of InvisiSpec, STT, SpecBox and SpecBox-NI on PARSEC 3.0 benchmarks. CleanupSpec currently does not support simulating multi-threaded applications in Gem5.

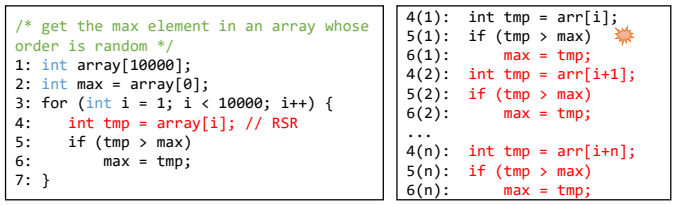


Fig. 12: A simplified code snippet with a RSR access pattern.

7.5 Main Cost Analysis

7.5.1 Squashes due to wrong-path execution

The essence of caching is to exploit the space and temporary locality in programs to overcome the memory wall. The unchanged size of cache line guarantee the spatial locality not be broken by SPECBOX. The first change brought by SPECBOX is eliminating the temporary locality of the data installed by the wrong path execution, which is also used in InvisiSpec and CleanupSpec. Previous studies have shown that wrong-path execution has the following two side effects [48].

Prefetching Effect. The data loaded during wrong-path reference will be re-referenced by the future correct-path execution. Figure 12 gives an simplified example of this scenario. The code in the example tries to scan an array and find its maximum element. In an out-of-order execution, the statements in Lines 4-6 from several iterations of the loop will be executed in the same speculative window. However, the array elements are un-ordered, the branch prediction on Line 5 will fail frequently, so the subsequent 4(2), 4(3), . . . , 4(n) instructions will be squashed. In SPECBOX, we call such a sequence *Reference-Squash-Rereference* (RSR). Figure 13 shows the incremental ratio of cache misses from the committed memory instructions in each cache level without limiting the domain capacity, and only perform cleanup operations for the *temporary* domain. From the results, We can see that RSR access patterns exist in most workloads. Especially, when they occur frequently in L1-DCache, it will cause a certain degree of performance overhead as shown in *xalancbmk* and *soplex* in SPEC.

Pollution Effect. The data loaded by wrong-path accesses will never be re-referenced but will replace other useful data. Furthermore, these wrong-path accesses will trigger the hardware prefetcher, which may pollute the cache indirectly. This situation rarely occurs in general out-of-order processors, and usually only occurs in more aggressive optimizations, such as run-ahead execution [49]. However, in SPECBOX, the domain isolation scheme and secure prefetcher scheme alleviate the pollution effect caused by the wrong-path accesses.

7.5.2 TOS for multi-core systems

In multi-core systems with SPECBOX, one important source of overheads comes from the emulated latency needed to support TOS scheme when different cores perform speculative accesses to a shared cache line simultaneously. The latency is not added to all simultaneous accesses but only to the first access from a core. Figure 14 shows the number of such accesses to L2-Cache in systems with different number of cores. We can see that as the number of cores increases, the number of such accesses gradually increases. And in cases such as *canneal* and *freqmine*, TOS scheme does have some impact there.

In addition, as the number of cores increases, due to the limited temporary domain capacity, the competition among multiple cores for the same cache set will increase. This is because TOS does not allow the cache line in the speculative state to be unilaterally evicted by a single owner. Figure 15 shows the performance of most PARSEC benchmarks gradually decreases as the number of cores increases. But for some benchmarks, such as *bodytrack, ferret* and *vips*, when the number of cores exceeds 8, there is a significant drop in performance.

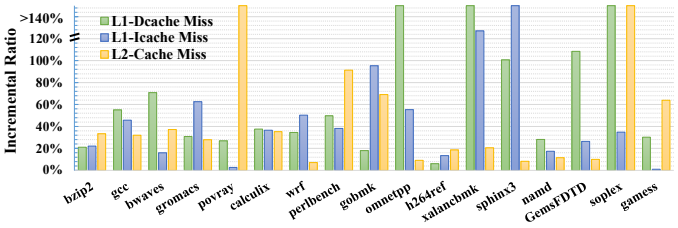


Fig. 13: The incremental ratio of cache misses from the committed memory operations in the original cache system and those in SPECBOX w/o domain capacity limitation.

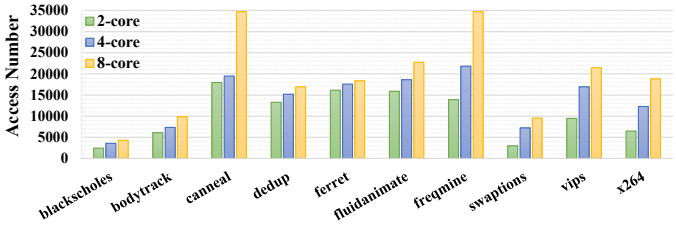


Fig. 14: The number of simultaneous speculative accesses to L2-Cache in PARSEC on different number cores. Some cores access the cache line for the first time.

TABLE 3: The hardware and power overhead of SPECBOX.

Metric	L1-ICache	L1-DCache	L2-Cache (2/4/8-core)
Area	0.18%	0.38%	0.61% / 0.61% / 0.92%
Access time	0.15%	0.01%	3.03% / 3.03% / 3.31%
Dynamic read energy	0.77%	0.63%	0.05% / 0.05% / 0.10%
Dynamic write energy	4.72%	2.21%	0.01% / 0.01% / 0.14%
Leakage power	1.52%	0.93%	2.09% / 2.09% / 2.47%

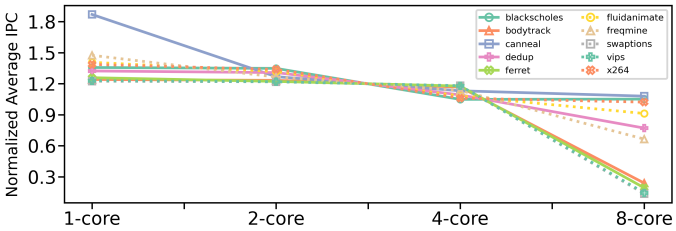


Fig. 15: The normalized IPC of the multi-core system with SPECBOX. (*facesim* and *dedup* crashed or hang, thus not included)

7.6 Hardware Cost and Power Consumption

In order to implement the *domain partitioning mechanism*, SPECBOX adds 1 bit to each cache line in the tag array. To implement the *thread ownership semaphores (TOS)* for L1-ICache, L1-DCache on a core with two physical SMT threads, SPECBOX adds additional 2 bits for each cache line in the tag array. For the shared L2 cache, the SPECBOX adds 2, 4, 8 bits, respectively, to implement TOS on 2-core, 4-core, and 8-core systems. We used CACTI-6.5 [50] to evaluate the hardware cost and the power consumption for the additional storage requirements. The results are shown in Table 3. As we can see, the SPECBOX introduces modest hardware and power consumption overhead.

8 DISCUSSION

Domain Capacity Configuration. As shown in our evaluation (see 7.3), a pre-determined capacity ratio can satisfy the requirement of most programs with today's cache sizes and associativity. However, for some performance-sensitive applications, dynamic adjustment of the domain capacity can provide more flexibility. SPECBOX uses a set of privilege registers *domain_cap* for capacity re-configuration at each level cache. The registers can only be controlled by special serialized instructions to avoid being controlled by illegal speculative execution. When the capacity of the temporary domain becomes zero, it means that the system gives up all security protection. Thus, the cache will treat all access as non-speculative accesses, and the notifier will suspend squash/commit requests to the cache.

Extending on other components. In addition to the cache system, SPECBOX can also be extended to other storage components and thread-shared resources that can be exploited as persistent covert channel in TEAs. For example, Intel processors will turn off the high bit of an execution unit if the AVX2 instruction is not executed for a long time, and turn it back on when it is executed. The turn-on operation will take a lot of time, resulting in a difference in execution time, which can be exploited by attackers [21]. In this case, we can also add SPECBOX's TOS scheme to AVX2 units. When a thread turns on the high bit of the AVX2 unit in an *in-flight* operation, the corresponding bit in the TOS is set to 1. At this time, if another thread also needs to perform the AVX2 instruction, and its corresponding TOS bit is 0, it will wait for a latency that is equivalent to the duration of turning it on. When a thread has not used the AVX2 instruction for a while, the CPU will first set the corresponding bit in the TOS to 0, and wait until all of the N bits in the TOS are all 0, then it can turn off the high bit.

9 RELATED WORK

In addition to the *delaying* and *invisible speculation* approaches mentioned in subsection 2.4, there are other defenses achieved through other software or hardware means.

Preventing Speculative Execution on Sensitive Code. The most direct way to prevent TEAs is to insert Fence or SSBB instructions in the security-sensitive code. Moreover, Spectre-type attacks can be prevented by preventing the control-flow prediction unit from being trained by attackers. IBRS/STIBP/IBPB [51] and Retpoline [52] prevent branch prediction unit of different permissions and different threads from interfering each other. CSF [53] modifies the decode stage in pipeline to automatically inject fence instructions before branch instructions. These defenses has a simple defense principle but may cause serious performance degradation compared with SPECBOX.

Preventing Illegal Speculative Accesses. Another type of mitigation is to prevent unauthorized instructions from obtaining the secret's value during speculative execution. KPTI [54] separates the page table entries and TLB entries in user-space from those in kernel-space. Chrome and Webkit browsers [55] prevent cross-site transient access through index masking and pointer poisoning. OISA [56] ensures that the accesses to sensitive data must use special instructions from a customized instruction subset, and these instructions cannot be executed out of order. ConTExT [57] marks the protected memory pages and registers to prevent their data from being obtained in the out-of-order execution state.

Comparing with these defenses, SPECBOX can protect all the secret data from Spectre attacks instead of the special data assigned by software.

Partition for Software Domains. Partitioning based on processes or other software domains have been used to block existing side-channel attacks. Some prior work [58], [59], [60], [61] uses cache set partitioning to enhance security via page coloring on physical-page allocation, which may lead to possible high memory overhead. Catalyst [47] adopts Intel's CAT [62], a hardware-supported way partitioning scheme on the last-level cache, to protect the Xen hypervisor with a low memory overhead. SecDCP [63] dynamically adjusts domain sizes according to the number of incurred cache misses to improve the performance of CAT partitioning. DAWG [64] improves cache way partitioning to enhance the isolation for hits, misses and metadata across the domains. It also provides an optimized software solution for secure domain time-multiplexing, cache synonyms avoidance and efficient cross-domain data transfer. However, compare with SPECBOX, all these defenses require programmers to annotate the secrets for protection and cannot thwart the transient-execution attacks (TEAs) within the domain.

Randomization and Other Approaches. Randomization and other noise-injection schemes are another effective approach to defend against side-channel attacks. For example, CEASER [65] and RPcache [66] can randomize the mapping of the addresses, cache sets or TLB sets, which prevent attackers from preparing the data layouts in the target cache set. RFillCache [67] selects random victims during cache-line replacement to hide the side effects of cache replacement policies. Secure-TLB [43] adopts a similar randomization replacement strategy to defeat TLBleed attacks [19]. FTM [68] targets cross-core Flush+Reload attacks by delaying the first access to the last-level cache. All these defenses can only be applied to a limited set of covert channels. However SPECBOX is a more general defense approach for all types of persistent covert channel in TEA.

10 CONCLUSION

This paper presented a *label-based transparent speculation* scheme, called SPECBOX, to defend against transient execution attacks via cache system. It partitions each cache set into a *temporary* and a *persistent* domain, by attaching a 1-bit label to each item and isolate the side effect of *in-flight* and *committed* operations. An *in-flight* operation can only affect the temporary domain, and the affected items will be switched to the persistent domain when the operation needs to be committed. To avoid the change of the temporary domain being observed by other synchronous executing threads, SPECBOX introduces *thread ownership semaphores*, which dynamically marks the thread ownership of each (shared) item in the temporary domain and emulates a thread-private storage. Analysis and extensive experiments have shown that SPECBOX is not only secure, but also practical and efficient.

ACKNOWLEDGMENTS

This research was supported by the National Natural Science Foundation of China (NSFC) under grant 61902374 and U1736208. Pen-Chung Yew is supported by the NSF under the grant CNS-1514444. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] 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 *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[3] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 707—720.

[4] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 572–586.

[5] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 264–276.

[6] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 723–735.

[7] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 151–164.

[8] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 132–144.

[9] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[10] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.

[11] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An" undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 73–86.

[12] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2, pp. 124–134, 1992.

[13] A. S. Huang, G. Slavenburg, and J. P. Shen, "Speculative disambiguation: A compilation technique for dynamic memory disambiguation," *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2, pp. 200–210, 1994.

[14] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium*, 2019, pp. 249–266.

[15] 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.

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

[17] Y. A. Younis, K. Kifayat, Q. Shi, and B. Askwith, "A new prime and probe cache side-channel attack for cloud computing," in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, 2015, pp. 1718–1724.

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

[19] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks," in *27th USENIX Security Symposium*, 2018, pp. 955–972.

[20] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "Aslr on the line: Practical cache attacks on the mmu." in *NDSS*, vol. 17, 2017, p. 26.

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

[22] J. Stecklina and T. Prescher, "Lazyfp: Leaking fpu register state using microarchitectural side-channels," *arXiv preprint arXiv:1806.07480*, 2018.

[23] 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)*, 2019, pp. 870–887.

[24] C. Cardenas and R. V. Boppana, "Detection and mitigation of performance attacks in multi-tenant cloud computing," in *1st International IBM Cloud Academy Conference, Research Triangle Park, NC, US*, 2012, p. 48.

[25] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[26] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-leak forwarding: Leaking data on meltdown-resistant cpus," *arXiv preprint arXiv:1905.05725*, 2019.

[27] A. LUȚAȘ and D. LUȚAȘ, "Bypassing kpti using the speculative behavior of the swapgs instruction," 2019.

[28] 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.

[29] A. Russo and A. Sabelfeld, "Securing interaction between threads and the scheduler," in *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, 2006, pp. 13–pp.

[30] 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)*, 2019, pp. 43–54.

[31] Z. Martinasek, V. Zeman, and K. Trasy, "Simple electromagnetic analysis in cryptography," *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*, vol. 1, no. 1, pp. 13–19, 2012.

[32] A. Velinov, A. Mileva, and D. Stojanov, "Power consumption analysis of the new covert channels in coap," *International Journal On Advances in Security*, vol. 12, no. 1 & 2, pp. 42–52, 2019.

[33] S. K. Khatamifard, L. Wang, A. Das, S. Kose, and U. R. Karpuzcu, "Powert channels: A novel class of covert communicationexploiting power management vulnerabilities," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 291–303.

[34] C. Trippel, D. Lustig, and M. Martonosi, "Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," *arXiv preprint arXiv:1802.03802*, 2018.

[35] 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.

[36] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[37] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies WOOT 18*, 2018.

[38] 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.

[39] 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)*, 2019, pp. 88–105.

[40] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 769–784.

[41] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[42] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," *cacheoutattack. com*, p. 16, 2020.

[43] S. Deng, W. Xiong, and J. Szefer, "Secure tlbs," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 346–359.

[44] ——, "Analysis of secure caches using a three-step model for timing-based attacks," *Journal of Hardware and Systems Security*, vol. 3, no. 4, pp. 397–425, 2019.

[45] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison *et al.*, "Speculative interference attacks: Breaking invisible speculation schemes," *arXiv preprint arXiv:2007.11818*, 2020.

[46] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[47] 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)*, 2016, pp. 406–418.

[48] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt, "Understanding the effects of wrong-path memory references on processor performance," in *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, 2004, pp. 56–64.

[49] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 2003, pp. 129–140.

[50] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 3–14.

[51] Intel, "Intel analysis of speculative execution side channels, https://www.intel.com/content/architecture-and-technology/intel-analysis-of-speculative-execution-side-channels-paper.html."

[52] M. F. A. Kadir, J. K. Wong, F. Ab Wahab, A. F. A. A. Bharun, M. A. Mohamed, and A. H. Zakaria, "Retpoline technique for mitigating spectre attack," in *2019 6th International Conference on Electrical and Electronics Engineering (ICEEE)*, 2019, pp. 96–101.

[53] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 395–410.

[54] Linux, "The current state of kernel page-table isolation, https://lwn.net/articles/741878/."

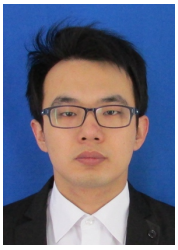
[55] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *28th USENIX Security Symposium*, 2019, pp. 1661–1678.

[56] J. Yu, L. Hsiung, M. El'Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high performance computing," in *The Network and Distributed System Security Symposium (NDSS)*, 2019.

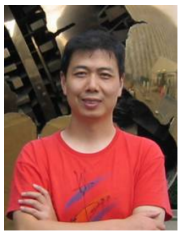
[57] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "Context: A generic approach for mitigating spectre." in *NDSS*, 2020.

[58] N. Waldin, M. Le Muzic, M. Waldner, E. Gröller, D. Goodsell, A. Ludovic, and I. Viola, "Chameleon: dynamic color mapping for

multi-scale structural biology models," in *Eurographics Workshop on Visual Computing for Biomedicine*, vol. 2016, 2016.
[59] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud," in *21th USENIX Security Symposium*, 2012, pp. 189–204.
[60] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008, pp. 367–378.
[61] T. Sherwood, B. Calder, and J. Emer, "Reducing cache misses using hardware and software page placement," in *Proceedings of the 13th international conference on Supercomputing*, 1999, pp. 155–164.
[62] Intel, "Introduction to cache allocation technology, https://software.intel.com/content/develop/articles/introduction-to-cache-allocation-technology.html."
[63] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.
[64] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 974–987.
[65] 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)*, 2018, pp. 775–787.
[66] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," pp. 494–505, 2007.
[67] F. Liu and R. B. Lee, "Random fill cache architecture," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 203–215.
[68] K. Ramkrishnan, S. McCamant, P. C. Yew, and A. Zhai, "First time miss: Low overhead mitigation for shared memory cache side channels," in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–11.



Bowen Tang is currently working toward the PhD degree in the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include computer architecture, compiler optimization and software security.

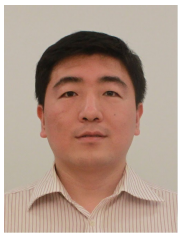


Chenggang Wu is a professor at Institute of Computing Technology, Chinese Academy of Sciences. His research was supported by National Science Foundation of China (NSF), the National High Technology Research and Development Program of China, and the National Science and Technology Major Project of China. His research interests include the dynamic compilation, including binary translation, dynamic optimization, bug detection on concurrent program, and software security.

Zhe Wang is currently an assistant professor at Institute of Computing Technology, Chinese Academy of Sciences. His research interests are in dynamic binary translation, multi-threaded program record-and-replay, operating systems, system virtualization, and memory corruption attacks and defenses.

Lichen Jia is currently working toward the PhD degree in the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include computer architecture and software security.

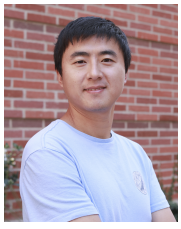
Pen-Chung Yew has been a professor in the Department of Computer Science and Engineering, University of Minnesota since 1994, and was the head of the department and the holder of the William-Norris Land-Grant chair professor between 2000 and 2005. He has also served on the organizing and program committees of many major conferences. His current research interests include system virtualization, compilers and architectural issues related multi-core/many-core systems. He is a IEEE fellow.



Yueqiang Cheng is a Director of Head of Security Research at NIO. He was senior staff security scientist of Baidu Research between 2017 to 2021. His research revolves around building secure systems and software, and also includes SGX security, virtualization security, rowhammer security, side-channel security, and autonomous driving security.

Yinqian Zhang is an professor of Department of Computer Science and Engineering Southern University of Science and Technology (SUSTech). Before joining SUSTech in 2021, he was an associate professor at Department of Computer Science and Engineering of The Ohio State University. His research interest is computer system security, with particular emphasis on cloud computing security, OS security and side-channel security.

Chenxi Wang is a postdoc in the Computer Science Department of University of California, Los Angeles. He has received the Ph.D. degrees in Institute of Computing Technology, Chinese Academy of Science in 2018. His research interests include computer system, especially for building hard core system, managed runtime and big data systems for emerging hardwares, such as non-volatile memory and disaggregated cluster.



Guoqing Harry Xu is an Associate Professor in the Computer Science Department of University of California, Los Angeles. He has led the development of a series of optimizing compiler and runtime frameworks in Microsoft Research and IBM Waston Research Center, which was accepted by many top conferences. His research interests include computer systems, ranging from programming languages and compilers, to runtime/operating/distributed systems and computer architecture.