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

Bowen Tang\textsuperscript{1,2}, Chenggang Wu\textsuperscript{1,2}, Zhe Wang\textsuperscript{1,2}, Lichen Jia\textsuperscript{1,2}, Pen-Chung Yew\textsuperscript{3}, Yueqiang Cheng\textsuperscript{4}, Yinqian Zhang\textsuperscript{5}, Chenxi Wang\textsuperscript{6}, Guoqing Harry Xu\textsuperscript{6}

\textsuperscript{1}State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, \textsuperscript{2}University of the Chinese Academy of Sciences, \textsuperscript{3}University of Minnesota at Twin Cities, \textsuperscript{4}NIO Security Research, \textsuperscript{5}Southern University of Science and Technology, \textsuperscript{6}University of California, Los Angeles

\{tangbowen, wucg, wangzhe12, jialichen\}@ict.ac.cn, \textsuperscript{3}yw@emn.edu, \textsuperscript{4}strongerwill@gmail.com, \textsuperscript{5}yinqianz@acm.org, \textsuperscript{6}\{wangchenxi, harryxu\}@cs.ucla.edu

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.

I. 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 [28]. 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 \texttt{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 \texttt{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 \texttt{dummy} array. This will load \texttt{dummy[val\times64]} 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 \texttt{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.

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” [73] and “safe” to use. For the example in Listing 1, the out-of-bound index value in Line 6 cannot be used to access the \texttt{dummy} array until the branch condition in Line 5 is resolved. The representative schemes are STT [73], SDO [72], NDA [67], ConditionalSpec [31], Delay-on-Miss [47] and SpecShield [4]. 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 [4], [47], [67], [73].

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 \texttt{dummy} loaded into the cache during the speculative execution will be removed and cannot be accessed in the \texttt{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 [1]. The representative works are SafeSpec [23], InvisiSpec [69], CleanupSpec [46] and MuonTrap [1].

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 [46]. 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.

II. BACKGROUND

A. Speculative Execution

Speculative execution techniques, such as branch prediction [70] and memory disambiguation [16], 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 reorder 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.

B. 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 [9]. 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 [13].

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 [68]. The information carrier in a persistent covert channel is usually the layout of a storage unit,
such as the cache [55], translation lookaside buffer (TLB) [57], and branch target buffer (BTB) [70], or a state change such as the on/off state of high bits in the AVX2 vector register [51]. 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) [54], execution unit port [2] and memory bus [10]. 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.

C. Transient Execution Attacks

Transient execution attacks (TEA) can be divided into two categories: the Spectre-type and the Meltdown-type attacks [9]. 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) [53], BTB [70] and return stack buffer (RSB) [14]. After that, the attacker will bypass the protection code to execute a wrongly speculative path, such as a branch check [27], data cleanup [48] and stack pointer switching [36], and then access the secret. Finally, the attacker will transmit the secret through a timing-based covert channel.

D. Existing Defenses Against TEAs

For TEAs, a common defense is to prevent sensitive data from being transmitted to covert channels. SpecShield [4] 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 [31] 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 [67] and STT [73] 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 [72] 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 [69] chooses to add a speculative buffer to keep the speculatively installed data. Similarly, SafeSpec [23] and MuonTrap [1] 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 [46] 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.

III. 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 [2], [6], it can be mitigated by turning off SMT, using security-sensitive thread scheduling [45], or using time-division multiple accessing (TDMA) on shared resources [58]. These methods are orthogonal to our approach.

Also, we do not consider physical covert channels, such as electromagnetic signals [38] and power consumption [24], [63]. 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.

IV. PROBLEM ANALYSIS

A. Encoding Methods of Cache Covert Channel

As mentioned in subsection II-B, 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.

• 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.

- **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.

**B. 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 an 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, and the other as the receiver. Using browser as an example, multiple threads of a malicious plug-ins can occupy different cores, or switching a cache line’s domain from the other 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.

**V. 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.

**A. Domain Partition**

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

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. 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. If the temporary domain is not full, it will install a new cache line in the temporary domain. 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. If the data is not in the temporary domain, it just ignores the request. 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. 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** and **Load B** 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, 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.

3) **Notifier**: 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 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.

B. 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.

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.
- 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 Tsender installs cache line A in the temporary domain and set its TOS bit. Then, the Treceiver also executes a Load A operation. Because Treceiver 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 Tsender first installs the cache line A in the temporary domain by a Load B operation. Subsequently, the Treceiver performs another Load B operation. Once the Treceiver completes, the TOS of the two threads are both set. And then the Tsender performs a Load A operation, which tries to replace the cache line B. At that time, SPECBOX just resets the TOS of Tsender instead of evicting the cache line, and suspends the Load A request until it is committed or the Treceiver resets the TOS also. Therefore, Treceiver cannot detect the deceleration effect caused by Tsender.

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 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.

C. 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.

Fig. 5: The workflow of the TOS scheme under two encoding attacks.
(a) Speculative Load Attack (b) Speculative Store Attack

Fig. 6: The coherence-state attacks.

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 [59] as Figure 6 shows: (a) The $T_{sender}$ speculatively executes the load operation, which may cause the state of other cores’ copies changing from Exclusive state to Shared state, and then the $T_{receiver}$ will slow down if it exclusively accesses these copies again. (b) The $T_{sender}$ speculatively executes the store operation which may invalids other cores’ copies so that the $T_{receiver}$’s subsequent access will be missing. SPECBOX leverage the scheme used in CleanupSpec [46] to address such a problem. It will delay the potentially malicious speculative state transitions, i.e. delaying the completion of the speculative load/store operations that may cause the above state changes in the MESI protocol. According to the evaluation of CleanupSpec [46] on some multi-thread workloads, the memory operations that may trigger the above two transactions are not frequent (about 2.4%), so it will only introduce a small performance overhead.

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. 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 Cache. TLB misses and Paging 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.

| Line Mode | $S_{prepare}$ | $S_{send}$ | $S_{receive}$ | Defense Principles |
|-----------|---------------|-------------|---------------|--------------------|
| Serialized | $A_{evict}^{t/p}$ | $A_{install}^{t/p}$ | $A_{func}^{t/p}$ (fast) | The cache line installed in $S_{send}$ will be cleaned up before $S_{receive}$. |
| Concurrent | $B_{install}^{t/p}$ | $B_{install}^{t/p}$ | $A_{func}^{t/p}$ (slow) | 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. |
| Concurrent | $B_{install}^{t/p}$ | $A_{install}^{t/p}$ | $B_{func}^{t/p}$ (slow) | The cache line installed into the temporary domain by thread B cannot be evicted by thread A in $S_{send}$. |

VI. SECURITY ANALYSIS

After analyzing various existing TEAs, such as Spectre-PHT/BBT/RSB/STL [27], [29], [30], [37], [48], Lazy FP [54], MDS [8], [50], [61], LVI [60] and CacheOut [62], we abstract the transmission process of persistent covert channel as a three-stage model: $S_{prepare}→S_{send}→S_{receive}$. $S_{prepare}$ 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 I 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 $S_{send}$ 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 I) 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 \( S_{\text{receive}} \).

**Concurrent-Mode Attacks.** Take the sixth attack (Line 6 in the table) as an example. In the \( S_{\text{prepare}} \) stage, thread B installs a cache line. In the \( S_{\text{send}} \) stage, thread A performs an in-flight operation to evict this cache line. In the \( S_{\text{receive}} \) stage, thread B times the access to this cache line. With SPECBox, the cache line will not be evicted in \( S_{\text{send}} \) because thread B owns this cache line. The access to this cache line by thread B in \( S_{\text{receive}} \) 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 [5].

**TABLE II:** Parameters of simulated micro-architecture.

| 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    |
| PBU                  | 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              | 4x2 mesh, 128b link width, 1 cycle latency per hop |
| DRAM                 | RT latency: 50 ns after L2                |

**VII. EVALUATION**

**A. Experimental Setup**

We implemented and simulated SPECBox based on an out-of-order (O3) processor and Ruby cache system using Gem5 [7]. The parameter settings of each component are shown in Table II, which are consistent with the settings in other studies. We evaluated the performance using 27 benchmarks from SPEC CPU 2006 and 12 benchmarks from PARSEC-3.0. The reason some benchmarks such as dealII and tonto are not included is 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.

**B. Defense Against Spectre PoC Code**

To evaluate the mitigation effectiveness of SPECBox, we chose the Spectre-PHT attack [27] 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.

**Fig. 7:** Access latency of the auxiliary array using SPECBox.

**C. 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 set 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.

**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 \( S_{\text{mean}} \)). The figure shows the percentage of the number of the cache sets with different \( S_{\text{mean}} \). We found no cache set’s \( S_{\text{mean}} \) is larger than 3. We selected some benchmarks with distinguished distributions as examples, and the remaining benchmarks are similar to these examples.
Fig. 9: The performance overhead of SpecBox, SpecBox with no-icache protected (denoted as SpecBox-NI), CleanupSpec [46], InvisiSpec [69] and STT [73] on SPEC CPU 2006 benchmarks. For CleanupSpec, three benchmarks (i.e. zensmp, xalancbmk, and GemsFDTD) hang and failed in our simulation.

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 10 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 2 or 3 ways for the temporary domain. This observation is consistent with other approaches that adopt way-partitioning schemes [34].

D. Performance Overhead

We compared SpecBox with other open-sourced hardware-based defenses for TEAs. The first is STT [73]. It represents the scheme that delays the instructions that are dependent on the transient instructions. The second is InvisiSpec [69]. It represents the scheme that cleans up the side effect of transient instructions. The third is CleanSpec [46]. 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 9 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 debug 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.

SpecBox 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, omnetpp 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 sphinx and GemsFDTD in SPEC and streamcluster in PARSEC.

**SpecBox 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 9, 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.

**SpecBox 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, bcip2 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 calcullix and sphinx3, the stall from rollback operations will also degrade the performance.

E. Main Cost Analysis

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 installe 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 [40].

**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 alxanchbm and soplex in SPEC.

![Figure 12](image_url)  
**Fig. 12:** A simplified code snippet with a RSR access pattern.

![Figure 13](image_url)  
**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.

**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 [41]. However, in SpecBox, the domain isolation scheme and secure prefetcher scheme alleviate the pollution effect caused by the wrong-path accesses.

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 (note: facsim and dedup crashed or hang, thus not included). But for some benchmarks, such as bodytrack, ferret and vips, when the number of cores exceeds 8, there is a significant drop in performance.

F. 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 [39] to evaluate the hardware cost and the power consumption for the additional storage requirements. The results are shown in Table III. As we can see, the SpecBox introduces modest hardware and power consumption overhead.

| 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% |

VIII. DISCUSSION

Domain Capacity Configuration. As shown in our evaluation (see VII-C), 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 that can be exploited as persistent covert channel in TEAs. To reduce power consumption, 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 [51]. In this case, we can also add TOS 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.

Defense against TSA. To defend against serial-mode attacks, SpecBox prevents the receiver from detecting any side effect after the sender exits its speculative window. However, assume that the receiver and sender are active in the same speculative window and the length of the speculative window can be measured, the attacker may still be able to gather side effects of the sender execution. SpectreRewind [12] has shown that such attacks are feasible in practice, which leverages contention on an execution unit as a covert channel. However, its variant that leverages contention on the cache (which is the focus of SpecBox) has yet to be shown feasible. Such potential vulnerability also exists in other existing defenses against cache covert channels. SafeSpec [23] discusses this vulnerability (called Transient Speculative Attack (TSA)), and proposes two possible solutions. One solution is to change the speculative storage to fully associative, and increase its capacity to avoid contention. The other solution is to partition the speculative storage to accommodate sender and receiver branches separately. Both solutions will incur some hardware overhead. The second solution can be extended in SpecBox, which will be our future work.

IX. RELATED WORK

In addition to the delaying and invisible speculation approaches mentioned in subsection II-D, 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 [19] or SSBB [3] instructions in the security-sensitive code. Moreover, Spectre-type attacks can be prevented by preventing the
