A Defense Mechanism Against Transient Execution Attacks On SMT Processors

Xin Jin¹, Ningmei Yu¹

Abstract: Transient execution attack does not affect the state of processor microarchitecture, which breaks the traditional definition of correct execution. It not only brings great challenges to the industrial product security, but also opens up a new research direction for the academic community. This paper proposes a defense mechanism for SMT processors against launching transient execution attacks using shared cache. The main structure includes two parts, a security shadow label and a transient execution cache. In the face of the side channel attacks widely used by transient execution attack, our defense mechanism adds a security shadow label to the memory request from the thread with high security requirement, so that the shared cache can distinguish the cache requests from different security level threads. At the same time, based on the record of security shadow label, the transient execution cache is used to preserve the historical data, so as to realize the repair of the cache state and prevent the modification of the cache state by mis-speculated path from being exploited by attackers. Finally, the cache state is successfully guaranteed to be invisible to any attacker's cache operations. This design only needs one operation similar to the normal memory access, thus reducing the memory access pressure. Compared with the existing defense schemes, our scheme can effectively prevent Spectre attack, and the overhead of performance is only 3.9%. 

key words: Transient attacks, security defense mechanism, SMT processors, cache attacks, security shadow label, transient execution cache

Classification: XYZ (choose one from Table II)

1. Introduction

By exploiting speculation technology designed[4][3] to optimize the performance of modern processors, some unauthorized instructions outside the expected data path can also be executed. Attackers can use transient execution to access confidential data with leaving side channel information in the microarchitecture, and then intercepted by other users of shared resources.

If ( x < array1_size)
#mistrain branch predictor
   Y = array2[ array1[x] * 4096 ]
#using array1[x] as subindex to access array2
Fig. 1. Example code of Spectre Attack

Take Spectre attack[1] as an example to introduce the behavior characteristics of transient execution attack[2][23]. Figure 1 shows the sample code executed by a Spectre attack. During the attack, an attacker will first mis-train the CPU’s branch predictor to predict the wrong branch direction, thus causing the CPU to temporarily violate the original program semantics by executing code that would not have been executed. It is with this speculated execution window that attackers can read confidential information stored outside the address space of the program. For example, in order to bypass ‘if’ operation, the attacker trains the branch predictor, which calls the above code with valid input to train the branch predictor to make the expected ‘if’ true. When the ‘if’ validity fails, the attacker begins to enter the attack stage. Before the branch result is determined, the CPU will guess boundary check is true. Therefore, the attacker can input an ‘x’ value outside the array1 boundary to access the memory, where the ‘x’ value is determined by the attacker based on the target data address that he tries to steal.

When the transient execution is finished, the data read from array2 with the secret data as the retrieval address is still kept in the cache. In order to steal the secret data, the attacker starts to implement the side channel attack against the cache[24][25][26][27][29][30]. Because of Flush_Reload attack[11] can provide cache line granularity attacks, which has higher precision than other side channel attacks[12] against cache, so it is widely used in obtaining confidential data in transient execution attack. The cache-based side channel attacks mainly utilize the time characteristic of cache access response, where Prime_Probe attacks[12] take advantage of cache miss events, in contrast, Flush_Reload mainly uses the cache hit event in cache, that is to say, the data access to "hit" in cache can get fast response. Therefore, when the attacker shares the cache with the victim, the data stored in the cache can be inferred by monitoring the temporal information of memory access. For example, the attacker accesses the corresponding cache data one by one again based on the historical address of the first stage flushing, and counts the corresponding access time. If the victim has no access to the cache, the attacker’s access will be ‘missing’. If the array2 data obtained by the victim based on the array1[x] * 4096 address exactly remains in this area, the attacker will ‘hit’ during the process of accessing address by address, and the value of the address is equal to array1[x]
In modern data centers, in order to improve the utilization of processor resources, SMT processors [19] are widely used. Applications from different users share the physical resources of the same processor, which naturally provides an intrusion environment for side channel attacks. When the malicious attacker submits his Trojan to the cloud for running, the scheduler will mix the Trojan and some innocent victim in the same CPU. SMT can share more resources for the target processor in fine-grained. Some newly disclosed vulnerabilities show that, these attacks against SMT processors can break through the fortified defense line of Intel processors again [20] [28].

The fine-grained resource sharing of SMT processor not only provides more potential attack targets for attackers, because the shared resources are all in the core, the attacker can obtain higher attack bandwidth. When Meltdown [14] attacks the third level cache, the reading bandwidth is 12.4KB/s, while when attacking the first level cache, the reading bandwidth increases to 582KB/s. Compared with the attacker and the victim accessing the shared resources alternately in time sharing, the attacker can use SMT to run with the victim at the same time, which can eliminate the interference 'noise' caused by context switching and reduce the bit error rate. Meltdown experiments show that the bit error rate can be reduced by more than 6 times by SMT based attacks. For Spectre attack, because the branch predictor is also shared, attackers can not only directly affect the prediction results of branch predictor, but also directly control the jump address to expand the attack range.

In this paper, a defense mechanism for transient execution attack using shared cache is proposed, which mainly includes security shadow label and transient execution cache. In the face of transient execution attack, where Flush_Reload is used, security shadow labels are attached to the memory requests from security requirement threads, so that the shared cache can distinguish the cache requests from different threads. On this basis, different from existing designs [21] [22] storing all the speculation memory access requests in another external buffers, our design allows the speculation memory access requests to be directly stored in the cache, thus reducing the cost of data movement. In order to prevent the modification of cache caused by mispredicted path from being detected and utilized by attackers, the transient execution cache structure is added to the existing processor cache path. When speculation starts, the transient execution cache is used to store the original cache evicted by the security thread, so as to prevent the attack by using cache hit. At the end of speculation, based on the record of security shadow label, the cache state is repaired by using the historical data preserved in the transient execution cache to eliminate the modification of cache state by mispredicted path, so as to prevent the attack by using cache miss. Finally, the cache state is successfully guaranteed to be invisible to any attacker’s cache operations.

2. Security Architecture Design

Our security architecture design mainly includes two parts, the security shadow label and the transient execution cache. The details of each part will be introduced below.

2.1 Security Shadow Label

In order to identify the thread requests from different security levels and achieve security isolation in the shared cache, we design a security shadow label for the thread with security requirement to ensure that the cache data is not used by malicious threads.

2.1.1 Structure of Security Shadow Label

The security shadow label structure needs two modifications to the memory access path of the existing processor: the cache request sender and the cache data storage side.

For the sender of cache request, this structure will attach an security identification tag (Sec_ID) containing security priority information to the memory request of each thread, and send it to the underlying cache together. As shown in Figure 3, the Sec_ID mainly includes three pieces of information. Among them, the highest bit indicates whether the current thread has security enabled, 10:8 bits indicate the security priority of the current thread, and the low 8 bits refer to the thread ID of the current thread.

For the cache data storage side, the existing cache tag is expanded, and the security shadow label structure (Sec_Tag) is used.
is added. The lower 13 bits are used to retain the Sec_ID information from the thread that needs to be protected, and evict_tag bits are used to hold the address of the evicted data, and the highest bit is used to invalidate the Sec_Tag. The specific uses of each structure are described in detail below.

2.1.2 Security Isolation

**Hit Isolation:** In the existing processor’s cache request implementation, as shown in Eq.1, as long as the tag address in the request address (tag + set + offset) matches any tag address selected from the cache through the set index, it can be considered as a cache hit.

\[ \exists i [(\text{Sec}[i].\text{Tag} == \text{TagA}) \Rightarrow \text{hit}] \quad (1) \]

With the help of Sec_Tag for shared cache, cache blocks belonging to safe thread can only be accessed by requests with the same Sec_ID. When judging the cache request, it is necessary to compare the security shadow label with the security identification tag of the memory request, as shown in Eq.2. Only when the security identifier tag matches, can it be considered as a hit.

\[ \exists i [((\text{Sec}[i].\text{Sec_Tag} == \text{Sec_ID}) \& (\text{Sec}[i].\text{Tag} == \text{TagA})) \Rightarrow \text{hit}] \quad (2) \]

**Replacement Isolation:** Because cache replacement operations will also leak timing information, similar to the isolation of memory access requests mentioned above, based on the existing replacement algorithm, the replacement operation in this design will select the cache block with the same security identification label as the memory access request. The request with high security level can replace a low-level cache block, and the safe thread can replace the block of the non-safe thread. But the opposite is not true. For example, a malicious thread’s cache request cannot evict a safe thread’s cache block.

Each cache line has the corresponding metadata, which mainly includes the historical access information used to assist the cache replacement algorithm. When a cache access occurs, the metadata updates the history information based on the cache hit results, and indicates the cache blocks to be evicted for the cache replacement operation.

By adding a set of security mask registers (Security_Mask), we can achieve the isolation of metadata. The security mask register is used to record the storage state of a safe thread in a group of caches. Its bit width is the same as that of the set-association of cache, and each bit corresponds to a way. When the data of the secure thread is stored in one way of a set of caches, set corresponding bit of Security_Mask register to 1. For example, a 4-way cache requires 4-bit Security_Mask, and the initial value is 4'b0000. When the second way stores the safe thread data, its value will be updated to 4'b0010. A reverse operation of Security_mask can get the storage mask of unsafe thread. When the metadata needs to be updated, the Security_Mask and metadata are logically “and” operated first, then the metadata information can be updated.

2.2 Transient Execution Cache

By using the security shadow tag, the data in the shared cache can be protected from being detected by the attack thread. However, crafty attackers can still use the same thread to implement transient execution attacks and detect changes to the shared cache. In order to further eliminate the side channel, it is necessary to make the modified cache state invisible to the attackers that is to create the illusion that the data in the cache has not been modified for the attacker. This section describes how to use the transient execution cache structure to achieve the above goal. The cache structure after adding security shadow label and transient execution cache is shown in Figure 4.

The transient memory access is the memory instruction issued by the processor in the speculative execution. In Spectre attack, the attacker mainly uses the load instruction in the transient execution to send the secret data. In order to achieve security, it is necessary to clear the cache modification caused by load instruction in the mis-prediction path at the end of transient execution. Therefore, when the transient load instruction is issued, it needs to be labeled with the security label. Once the mis-prediction is confirmed, the cache state recovery operation is performed based on the annotation of the security label.

In order to mark the transient load instruction, our design makes the following modifications to the internal structure of the existing pipeline referring to the design of STT[16].

**Label start:** when a branch instruction appears in the decoding stage of the front-end of the pipeline, it indicates the beginning of the transient execution. When the predicted
result of the branch instruction is confirmed, it indicates the end of the transient execution. In order to identify the transient execution phase, a transient identification register (Trans_Visibility_Point register (TVP)) is added, which records the earliest branch instruction in the current pipeline but has not yet resolved. Because all instructions enter the reorder buffer in sequence, the ROB index number of the branch instruction in the ROB is selected and recorded in the TVP register, as shown in Figure 5. The larger the index number, the newer the corresponding instruction.

When dispatch the load instruction, the value of the TVP register is compared with the ROB index of the load instruction. If ROB index > TVP, it means that the load instruction enters the pipeline after the branch instruction, so it is a transient memory access and needs to be labeled. Therefore, it is necessary to expand the instruction by adding two fields: transient access index bit (Trans_Load_Index, TLI) and speculation status bit (SS), where TLI stores the ROB Index of the load instruction, SS setting indicates that the load is speculated execution. If the instruction is not speculative, its TLI field and SS bit are both 0.

Because the transient load instruction will have expanded information, it is necessary to expand the load queue at the back-end of the pipeline to store the TLI and SS fields corresponding to the load instruction, as shown in Figure 5. When a load instruction is issued, if its SS position is set, the security label is added to the memory access request.

In order to make the cache modification state invisible to the attacker and avoid the secondary memory access caused by the use of extra buffer structure to retain the transient cache data, our design chooses to let the transient memory access modify the cache directly. By adding a transient execution cache structure, the original data evicted by the transient memory access is temporarily stored to make the attacker’s probe operation hit in the transient execution cache that create the illusion of unmodified cache state. The specific structural design is as follows.

As shown in Figure 5, on the basis of the cache structure described before, a transient execution cache (T_cache) is placed on the refill path of the corresponding cache, which adopts a fully associated cache structure, and its depth is consistent with the load queue. This is because that the depth of the load queue is the maximum number of transient memory access requests that can be issued during transient execution. Its working principle is as follows:

**Transient hit operation:** The transient load instruction with security label can directly access any cache block. The load instruction without security label can only access the cache block with non-security identifier.

**Transient replacement:** When a transient memory access request with a security label evict a non-secure cache data block, the security identification label information is stored in the security shadow label, which indicates that the cache block has been occupied by the security data. At the same time, the non-secure cache block is evicted to the transient execution cache, and the tag address of the non-secure cache block is also copied to Evict_Tag in the security shadow tag field for recovering the cache state.

When a non-secure load requests for an evicted cache block, a missing event occurs in the cache because it has been replaced by a safe thread cache block. In this case, compare the tag of the non-secure load and Evict_Tag, and a match event indicates that the cache block is evicted by the transient load, the transient execution cache can be searched to generate a hit. Therefore, the attacker will not be aware of the replacement operation executed before, and the modification of cache state caused by the transient memory access operation is invisible to the attacker.

Once the earliest branch instruction in the pipeline has been resolved, if the prediction is correct, the transient execution corresponding to the branch instruction will not be flushed, and its modification to the microarchitecture state will be retained. At this time, because this branch instruction has been resolved, it is necessary to refresh the value of TVP register and write the next branch instruction ROB_Index and broadcast the refreshed TVP value to the load queue. Compared with the TLI field of the transient load instruction waiting to be committed in the load queue, if TLI < TVP, it indicates that the load instruction depends on the branch instruction that has been resolved and belongs to the instruction of correct execution path. Therefore, the corresponding security verification bit (Security_Check bit, SC)
of the instruction is set. During commit stage, if the SC bit is 1, a security ID clear request (Security_ID_Clear, s_clear) is sent to the cache. When the corresponding cache block receives the request, the content of its security shadow label is cleared, so that it can be used by other threads. If the prediction is wrong, it means that all transient execution after the branch instruction need to be flushed. In order to clear the modification in the cache, the current TVP value needs to be broadcast to the load queue for item by item comparison, as shown in Figure 5. If TLI > TVP, it means that the load instruction depends on the branch instruction with the current mis-prediction, and sends invalid request to the cache according to the corresponding load address. When the corresponding cache block receives the request, the invalid bit of its cache block in security shadow label is set. Meanwhile, according to evict_Tag field address, send copy data request to transient execution cache, copy the previous evicted data to cache in order to eliminate the modification of cache caused by mis-prediction path.

3. Evaluation

3.1 Security Evaluation

Our security design is implemented on GEM5 simulator[13]. This section evaluates whether it has the ability to eliminate Spectre attacks. Using the attack code shown in Figure 1, choose to set the value of the secret data to the character ‘a’. Attackers exploit Flush_Reload attack to detect array array2, by accounting the delay of accessing its data, the address of access hit is the successful stolen. As shown in Figure 6, when the security protection is not turned on, the attacker access will miss on multiple consecutive addresses until the data index reaches 65, where a hit occurs and indicates that the victim program has read the character ‘a’. When the security protection is turned on, the attacker is always miss and does not hit the index 65. Therefore, this design effectively defends the Spectre attack.

3.2 Performance Evaluation

In order to evaluate the impact of the security mechanism on the performance, this experiment uses the SPECCPU 2006 benchmark[17] as the workload to evaluate the execution time after the security guarantee is turned on. As shown in Table I, compared with the method of using software lfence instruction[18] and the design of customized buffer[21][22], this design has the least impact on performance, and only increases the execution time by 3.9% on average.

### Table I. Performance Comparison

| Related Designs   | Performance Overhead |
|-------------------|----------------------|
| Lfence            | 74%                  |
| InvisiSpec        | 15%                  |
| CleanupSpec       | 5.1%                 |
| Our Design        | 3.9%                 |

3.3 Hardware Overhead

The hardware overhead mainly includes three parts: load queue, security tag and transient execution cache.

a) The load queue needs to be expanded for each item to store the identification information of the transient load instruction, including the transient access index bit (Trans_Load_Index, TLI), whose bit width is equal to \( \log_2 \text{ROB} \). In this evaluation, the ROB depth is 224, so TLI is 8 bits; 1 bit prediction status bit (SS); 1 bit security verification bit (Security_Check bit, SC). Therefore, a total of 72 * (8 + 1 + 1) = 720 bits is required, where 72 is the load queue depth.

b) Security shadow label mainly includes two parts, security identification tag (Security_ID, Sec_ID) and tag. The security identification tag includes: 8-bit thread ID, 3-bit priority, 1-bit enable bit, 12 bits in total. The bit width of the original data label depends on the size of the cache and the correlation degree. The size of the first level cache is different from that of the second level cache. In order to protect all cache data blocks, the security shadow label is added to each cache in this design, and the label information of every evicted cache line is retained in the security shadow label.

c) Transient execution cache adopts a fully associative cache, and the size of the cache block is the same as that of the original cache, and the whole cache block is retained. The depth is the depth of load queue, which is 72 items in this design.

MCPAT[15] is used to evaluate the area and power consumption of hardware overhead. Based on the 22nm process, the area of Intel Haswell processor is 11.6106mm\(^2\) and the power consumption is 30.83w. The evaluation results of the three components with the largest area and power consumption are shown in Table II.

### Table II. Hardware Overhead

|                | L1 Sec_Tag | TCache | L2 Sec_Tag |
|----------------|------------|--------|------------|
| Area(mm\(^2\)) | 0.0041     | 0.09   | 0.0287     |
| Energy(nJ/req) | 0.0008     | 0.023  | 0.002      |
| Power(mw)      | 1.47       | 10.015 | 8.08       |

4. Conclusion

This defense mechanism first designs a security identification tag for the transient execution of memory instructions,
so that the shared cache can distinguish cache requests from different threads, and realize the security isolation of cache operations. On this basis, in order to prevent the modification of cache caused by mis-trained prediction path from being detected and utilized by attackers, a transient execution cache structure is added to the existing processor cache path to store the cache block evicted by the safe thread during the prediction execution. When the mis-prediction occurs, because the evicted data are kept in the transient execution cache, malicious requests will hit in the transient execution cache. At the same time, based on the record of security shadow label, the data modification caused by the wrong cache. At the same time, based on the record of security shadow label, the data modification caused by the wrong cache, malicious requests will hit in the transient execution cache. The combination of the two ensures that the cache state is invisible to any attacker. This design only needs one operation similar to the normal memory access operation, thus reducing the memory access pressure. The performance and security evaluation results show that the proposed scheme can effectively prevent Spectre attacks, and the overhead of performance is the least, only 3.9%, compared with the existing defense schemes.

**References**

[1] Kocher, Paul, et al. "Spectre attacks: Exploiting speculative execution." 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019.

[2] Szefer, Jakub. "Survey of microarchitectural side and covert channels, attacks, and defenses." Journal of Hardware and Systems Security 3.3 (2019): 219-234.

[3] Tomasulo, Robert M. "An efficient algorithm for exploiting multiple arithmetic units." IBM Journal of research and Development 11.1 (1967): 25-33.

[4] Hennessy, John L., and David A. Patterson. Computer architecture: a quantitative approach. Elsevier, 2011.

[5] Kocher, Paul, et al. "Spectre attacks: Exploiting speculative execution." 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019.

[6] Yan, Mengjia, et al. "Invisispec: Making speculative execution invisible in the cache hierarchy." 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2018.

[7] Saileshwar, Gururaj, and Moinuddin K. Qureshi. "Cleanup-Spec: An" Undo" Approach to Safe Speculation." Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. 2019.

[8] Khasawneh, Khaled N., et al. "Safespec: Banishing the spectre of a meltdown with leakage-free speculation." 2019 56th ACM/IEEE Design Automation Conference (DAC). IEEE, 2019.

[9] Ainsworth, Sam, and Timothy M. Jones. "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state." 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020.

[10] Kiriansky, Vladimir, et al. "DAWG: A defense against cache timing attacks in speculative execution processors." 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2018.

[11] Yarom, Yuval, and Katrina Falkner. "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack." 23rd USENIX Security Symposium (USENIX Security 14). 2014.

[12] Tromer, Eran, Dag Arne Osvik, and Adi Shamir. "Efficient cache attacks on AES, and countermeasures." Journal of Cryptology 23.1 (2010): 37-71.

[13] Binkert, Nathan, et al. "The gem5 simulator." ACM SIGARCH computer architecture news 39.2 (2011): 1-7.

[14] Lipp, Moritz, et al. "Meltdown." arXiv preprint arXiv:1801.01207 (2018).

[15] Thoziyoor, Shyamkumar, et al. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, 2008.

[16] Yu, Jiyoung, et al. "Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data." Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. 2019.

[17] Henning, John L. "SPEC CPU2006 benchmark descriptions." ACM SIGARCH Computer Architecture News 34.4 (2006): 1-17.

[18] INTEL. Q2 2018 Speculative Execution Side Channel Update, May 2018.

[19] Margaritov, Artemiy, et al. "Stretch: Balancing qos and throughput for colocated server workloads on snt cores." 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2019.

[20] Schwarz, Michael, et al. "ZombieLoad: Cross-privilege-boundary data sampling." Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019.

[21] Yan, Mengjia, et al. "Invisispec: Making speculative execution invisible in the cache hierarchy." 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2018.

[22] Saileshwar, Gururaj, and Moinuddin K. Qureshi. "Cleanup-Spec: An" Undo" Approach to Safe Speculation." Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. 2019.

[23] Canella, C, Van Bulck, J, Schwarz, M, et al. A systematic evaluation of transient execution attacks and defenses[C]//28th USENIX Security Symposium (USENIX Security 19). 2019: 249-266.

[24] Spreitzer, R, Gérard B Towards more practical time-driven cache attacks[C]//IFIP International Workshop on Information Security Theory and Practice. Springer, Berlin, Heidelberg, 2014: 24-39.

[25] Acııçmez, O, Brumley, B B, Grabher, P New results on instruction cache attacks[C]//International Workshop on Cryptographic Hardware and Embedded Systems. Springer, Berlin, Heidelberg, 2010: 110-124.

[26] He, Z, Lee, R B. How secure is your cache against side-channel attacks?[C]//Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. 2017: 341-353.

[27] Yarom, Y, Genkin, D, Heninger, N. CacheBleed: a timing attack on OpenSSL constant-time RSA[J]. Journal of Cryptographic Engineering, 2017, 7(2): 99-112.

[28] Götzfried, J, Eckert, M, Schindel, S, et al. Cache attacks on Intel SGX[C]//Proceedings of the 10th European Workshop on Systems Security. 2017: 1-6.

[29] Ostvik, D A, Shamir, A, Tromer, E. Cache attacks and countermeasures: the case of AES[C]//Cryptographers’ track at the RSA conference. Springer, Berlin, Heidelberg, 2006: 1-20.

[30] Kocher, P C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems[C]//Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 1996: 104-113.