SPOILER-ALERT: Detecting Spoiler Attacks Using a Cuckoo Filter

Jinhua Cui
Hunan University
Changsha, China
jhcui@hnu.edu.cn

Yiyun Yin
Hunan University
Changsha, China
yiyun@hnu.edu.cn

Congcong Chen
Hunan University
Changsha, China
chencongcong@hnu.edu.cn

Jiliang Zhang*
Hunan University
Changsha, China
zhangjiliang@hnu.edu.cn

***Abstract*—Spoiler attacks leak physical address information, which is exploited to accelerate reverse engineering of virtual-to-physical address mapping, thus greatly boosting Rowhammer and cache attacks. However, existing approaches that detect data-leakage attacks no longer suit the requirements of identifying Spoiler. This paper proposes SPOILER-ALERT, the first hardware-level mechanism to detect the address-leakage Spoiler attacks in real time. It leverages a cuckoo filter module embedded into *Memory Order Buffer* component to screen buffer addresses on-the-fly. We further optimise the filtering algorithm to reduce false positives. We assess the effectiveness and performance based on prototype implementations, which achieve a detection rate of 99.99% and negligible performance loss. Finally, we discuss potential reactions of our detection mechanism after a Spoiler attack was discovered.**

***Index Terms*—detection mechanism, information leakage, cuckoo filter, transient execution attack**

I. INTRODUCTION

Microarchitectural vulnerabilities are unintended design flaws in hardware and could be exploited to compromise an operating system (OS) and steal confidential data. Worse yet, discovering such vulnerabilities is much more difficult, and the software-level patches may not work well on current CPU generations. Instead, CPU vendors may need to redesign hardware components. A typical example of such a microarchitecture design is transient execution, an optimization technique available on modern processors. By virtue of transient execution, application code can gain better performance and efficiency during computation. However, some well-known attacks that exploit this optimization technique, such as Meltdown [1] and Spectre [2], have caused breaches of data confidentiality in a benign OS and even a trusted execution environment (TEE).

Spoiler [3] as a special transient execution attack only targets the address information of a victim application while attacks like Meltdown directly leak secret data. In particular, Spoiler offers attackers a way to accelerate reverse engineering of virtual-to-physical address mapping that may finally lead to data breaches. Thanks to the way virtual-to-physical address mappings can be speeded up by 256 times, and the search of eviction sets can be improved by 4096 times. With efficient construction of such eviction sets, it has been proven that cache and Rowhammer [4] attacks can be boosted significantly.

Many hardware- and software-based approaches [5–15] have been proposed to mitigate transient execution attacks. However, most of the approaches are to defend against data-leakage attacks. Hence, they cannot work on Spoiler that only targets address information rather than secret data. Typically, defenses like NDA [15] and SSBD [14] can stop speculative load behaviors, but they slow down the performance seriously (e.g., with up to 125% performance loss by the NDA). Additionally, the aforementioned approaches are incapable of discovering the Spoiler attacks. Thus, it highlights the importance of designing an effective and low-overhead detection mechanism for Spoiler.

However, identifying malicious Spoiler behaviors is non-trivial as the attacker only leaks indirect information or intermediate state during program execution. In particular, Spoiler exploits the dependency resolution logic serving the speculative load, which hence makes users difficult to notice those subtle changes of microarchitectural state. In addition, constructing a high-accuracy detection system is challenging, because it needs to achieve low storage and performance overhead as well as few false positives and negatives.

In this paper, we propose a detection mechanism named SPOILER-ALERT that is able to effectively monitor all Spoiler's malicious behaviors. To the best of our knowledge, SPOILER-ALERT is the first approach for Spoiler detection. Our key observation is that Spoiler needs to fill the store buffer (SB) with n addresses that have the same offsets. With this feature, we utilize a cuckoo filter to record the address information, and warn users of an attack detected once the number of recorded addresses reaches a threshold. We further inspect the internals of the cuckoo filtering algorithm, and conduct optimizations to lower the false positives. We assess SPOILER-ALERT based on prototype implementation on gem5 platform. Experimental results demonstrate that SPOILER-ALERT can effectively detect Spoiler with a 99.9% detection rate and 0.24% performance overhead on average. Therefore, the proposed mechanism gains good capability of being able to know Spoiler attacks in real time to some extent while benign applications are not affected.

In short, this paper has three major contributions:

- We propose a detection mechanism, SPOILER-ALERT, to successfully detect the microarchitectural Spoiler attacks with the assistance of a cuckoo filter module. We also summarize three key traits used for our detection approach after analyzing the internals of Spoiler.
- We design and implement the *first* Spoiler detection mechanism on the gem5 simulator, and assess the effectiveness

* Jiliang Zhang is the corresponding author.

978-3-9819263-7-8/DATE23/© 2023 EDAA

and performance of SPOILER-ALERT using applications from SPEC 2006.
- We discuss potential countermeasures to prevent Spoiler attacks discovered at runtime from two perspectives: adding noises, and flushing the SB.

II. THREAT MODEL AND DESIGN GOALS

We assume that Spoiler can be launched smoothly on a target system. This attack needs a precalculated number of store instructions to fill the store buffer (SB). These store instructions compete for resources and are blocked in hardware pipeline, so that the subsequent load instructions in Memory Order Buffer (MOB) can bypass them and be executed in advance. The attacker can infer the physical address information by means of the generation of wrong address dependency during the speculative execution. Therefore, we assume that the attacker and the victim are located on the same physical core. The malicious process runs on the target system with only user privilege, so it cannot execute privileged code and access virtual-to-physical address mappings (e.g., page mappings for a file). Furthermore, we assume that the target system does not deploy any security mechanisms to prevent speculative load execution (e.g., setting SSBD).

Our goal in this paper is to propose a practical detection approach to discover potential Spoiler attacks at runtime. This approach achieves: (1) high detection rates; (2) negligible runtime performance overhead and storage overhead; (3) zero modifications to the OS.

III. SPOILER ANALYSIS

A. Key traits of Spoiler Attacks

Spoiler [3] is a particular type of transient execution attack. It exploits advanced microarchitecture techniques available in modern processors, i.e., speculative load and the dependency resolution logic, to leak the mapping information of physical addresses. In fact, Spoiler only reveals the 12th to 19th significant bits of a physical address.

In Spoiler attacks, the Memory Order Buffer (MOB) is a key component to manage data access operations. Load instructions can be executed ahead of time and reordered by the MOB. Spoiler first makes the store buffer (SB) blocked through execution of plenty of store instructions. It then runs a speculative load instruction. During committing the load instruction, the MOB will sort all instructions and check the address dependency. When the check for an address dependency fails, the execution of the load instruction will be blocked, thus causing a high delay. Finally, the attacker can get a set of virtual addresses mapped to the same physical address (at least the lowest 20 bits are consistent). With the assistance of Spoiler, Rowhammer and cache side-channel attacks can be boosted significantly. For instance, Prime+Probe's construction of eviction sets can be accelerated by 4096 times.

Three key traits. According to the attack procedures of Spoiler, we observed three key traits summarized as follows:

T1. The attacker depends on large numbers of store instructions and one load instruction, and the offsets of these instruction addresses are identical.

T2. The store buffer is filled up with the store instructions.

T3. The store instructions on each sequential page are forwarded to the CPU iteratively.

The three traits stemmed from Spoiler are the basis for the detection mechanism. In the next section, we utilize them to achieve our goals.

B. Why is the Cuckoo Filter able to Identify Spoiler?

A filter is usually used to test whether an element belongs to a large data set. A prime example of such a filter is cuckoo filter [16], a space-efficient probabilistic data structure. In our scenario, to determine the Spoiler behaviors, plenty of CPU store instructions and their related data have to be monitored and recorded on-the-fly. Fortunately, a cuckoo filter allows defining our private variables (e.g., for the numbers of Spoiler's traits), so the storage overhead could be reduced greatly due to no involvement of another filter. On the other hand, the cuckoo filter supports dynamic deletion to existing elements. This feature can be used to optimize our detection mechanism to further reduce false positives. In contrast, the bloom filter [17] never embraces the aforementioned advantages, while other filter variants offering dynamic deletion and duplicate insertion also suffer from varying storage and performance overhead. Hence, we choose the cuckoo filter and retrofit it to achieve a high-accuracy and lower-overhead detection approach.

IV. SPOILER-ALERT

A. Overview

An attacker requires a large number of store instructions to launch Spoiler. In the process of such attacks, the store buffer (SB) is filled up, and a subsequent load will be blocked because of incorrect address dependency found. Our proposed approach, SPOILER-ALERT, attempts to determine whether the store in the SB matches the three attacking traits (see Section III).

We deploy the detection mechanism in the processor's Memory Order Buffer (MOB). An overview of microarchitecture design of SPOILER-ALERT is shown in Fig. 1. The CPU fetches instructions from L1 i-cache in order and then forwards them to the instruction queue. Once done, the instructions are decoded into micro operations during the decode stage. After an access instruction is executed, the results are submitted to L1 d-cache. Because the SB in the MOB can monitor each store instruction executed by the CPU, our detection mechanism can be embedded into it to track and record the address of each store. It further counts the store instructions with Spoiler's traits. If the value of the counter reaches a specified threshold, which means the system may be under the Spoiler attack.

B. Design Challenges and Solutions

Low overhead. Spoiler attacks rely on considerable store instructions with same offsets. SPOILER-ALERT has to take time to find such instructions and keep a record of them in an extra storage region. Naively, we can utilize two filters to accomplish our detection. The first records the fingerprint derived from the store instructions and the corresponding offsets, which in turn is used to match the fingerprint of instructions performed by Spoiler attacks. The other records the number of occurrences of

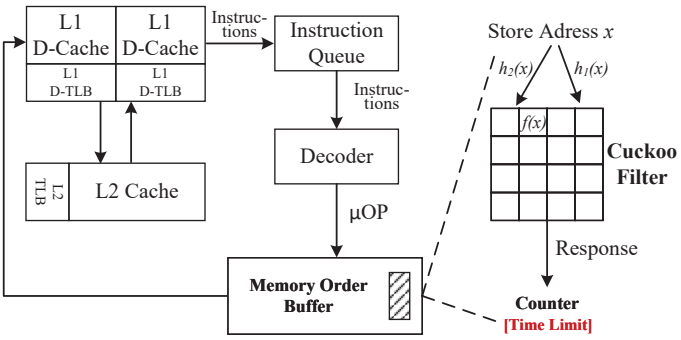

Fig. 1: Overview of SPOILER-ALERT

those matched fingerprints, which is used to count how many times the attack happens. However, this method surely incurs more performance and storage overhead.

Alternatively, we make use of a feature of the cuckoo filter, which allows acquiring two types of data (e.g., the fingerprint and the number) by only a single filter. Comparing with multiple filters, the unique filter in SPOILER-ALERT has less resource usage, which in turn brings significant decrease of the runtime cost and complexity.

False positives. The detection mechanism must be able to distinguish the attacker's store instructions and those of benign applications. The detection rate and false positives, thereby, can be guaranteed with little side effects to the normal programs.

The same-offset store instructions that are not controlled by attackers may also be accumulated over time, even in a benign program. As a result, the filter may gradually be filled up and the threshold is eventually hit, leading to false positives. In SPOILER-ALERT, we utilize two-step checks to assert whether the identified behaviors are genuinely from attackers. We first perform a coarse-grained detection where it compares the offsets for equivalence. Then it determines whether the deference of address of consecutive stores satisfies the attacker's traits. When both conditions above are true, it starts counting and checks if the counter hits the threshold pre-specified. If it is the case, the attacks were happening. Otherwise, it may be a false positive. Furthermore, we periodically reset the counters and clean the recorded data to reduce such false positives.

C. Cuckoo Filter

Cuckoo filters [16] offer the ability to dynamically add and remove elements to / from a hash table. The table is composed of a bucket array where each row is called a bucket. The bucket is used to store fingerprint information associated with the inserted element x. The fingerprint can be denoted as x_f, which is derived from a function of fingerprint f. Cuckoo filters provide a configurable length for the fingerprint x_f, for instance, which can be set to 2, 4, and 8 bits. Thus, the filters do not incur noticeable storage overhead. For each inserted element x, there are two hash functions used to derive the bucket indexes ($h_1(x)$ and $h_2(x)$) respectively, as shown in formula (1) and (2). Besides, the two functions expose an important property, i.e., one bucket index idx_2 can be derived by idx_1 and its hashed fingerprint, no matter if idx_2 is $h_1(x)$ or $h_2(x)$. The logic above can be expressed in the formula (3).

$$h_1(x) = hash(x) \tag{1}$$

$$h_2(x) = h_1(x) \oplus hash(x_f) \tag{2}$$

$$idx_2 = idx_1 \oplus hash(x_f) \tag{3}$$

During insertion of element x in the cuckoo filter, if a free slot is found in different buckets $h_1(x)$ and $h_2(x)$, the fingerprint of x will be placed in $h_1(x)$ first. If the two buckets are filled up, the cuckoo filter has to pick the spare bucket $h_2(x)$ and randomly remove the corresponding element (*e.g.*, a) from it. So the previous a can be substituted with x. With the formula (3), a can also obtain the spare bucket index. This is a process of relocation of the element a, which may be repeated many times until a slot is available.

If the same element is inserted $k*p$ times, the next insertion will fail, *i.e.*, the cuckoo filter can only insert the same element at most $k*p$ times. The k refers to the number of hash functions, and the p refers to the number of elements in one bucket.

D. Microarchitectural Implementation

Our detection mechanism identifies the store addresses with Spoiler's traits and reminds the users that the system was being under the risks. The mechanism is embedded into the store buffer (SB) and consists of four pieces: the cuckoo filter, the filtering algorithm, the counter, and controlling of the maximum time. These four parts are designed to implement SPOILER-ALERT according to the three traits. Fig. 2 describes the important microarchitecture details.

Adapting the cuckoo filter. We adapt the cuckoo filter to enable our detection approach. In the light of the **T1**, SPOILER-ALERT needs recording the number of store addresses that have same offsets to conduct the runtime detection. Normally, only one type of fingerprint, either the addresses of store instructions or the numbers of store addresses, can be held in a cuckoo filter. However, as discussed in Section IV-B, we utilize an advanced feature of cuckoo filters, which allows inserting same elements at most $k*p$ times without need for special hardware or software implementations. Thus, SPOILER-ALERT can make direct comparisons of the fingerprints of the store addresses and the corresponding numbers to determine malicious behaviors.

As shown in Fig. 2, SPOILER-ALERT takes the offsets of store addresses as the final input of the cuckoo filter, and stores their fingerprints into it. With the **T1** and the adapted cuckoo filter, whenever two bucket arrays are filled with the fingerprint of the same offset, the following new insertion with the same offset fails and returns $NotEnoughSpace$. This clearly reflects malicious Spoiler attacks being against the system (e.g., stealing secrets).

Filtering algorithm. SPOILER-ALERT further filters the store addresses to avoid any potential false positives where the addresses without attack traits may occur many times. Note that two consecutive store operations executed by the Spoiler attacker should have different addresses and reside on sequential pages (e.g., 0x1000, 0x2000), as pointed out in

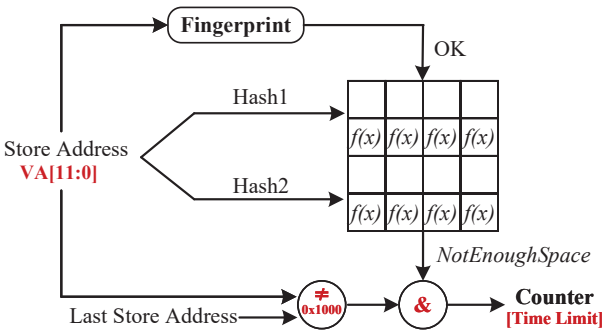

Fig. 2: The detection mechanism.

the **T3**. Therefore, SPOILER-ALERT only matches and counts the desirable store addresses. After the two procedures by the cuckoo filter and the filtering algorithm, a store address is considered as meeting the Spoiler's traits. In this case, we increase the counter.

In addition, the Spoiler attacker may perform out-of-order accesses. For instance, she could access pages 1, 3, 2, and 4 in this order to bypass the detection. However, since SPOILER-ALERT not only asserts whether the access is from sequential pages (e.g., the difference is equal to 0x1000), but also judges whether the page difference is a multiple of a single page (e.g., 0x1000). Thus, simply shuffling of page accesses is incapable of circumventing SPOILER-ALERT's detection.

Threshold. The counter for counting the attacking traits needs to set up a threshold to identify the Spoiler attack at its earliest time. According to the **T2**, the malicious store instructions will fill up the SB, so we can take the SB size as the threshold. Once the counter reaches the pre-specified threshold, SPOILER-ALERT warns the users of being under Spoiler hazards.

Maximum time. In malicious or benign executions, the counter value may sometimes exceed the threshold. To this end, SPOILER-ALERT pre-sets a maximum time for both the cuckoo filter and the counter. When the maximum is capped, the filter cleans up the recorded data and restarts a new round of detection. Meanwhile, the counter is reset.

Caveats. The original Spoiler exploits the aliasing effect available in modern processors, in which the load instructions are checked with the 12-19 bits of the store buffer entries. However, such sophisticated mechanisms are not present in the gem5 simulator. Instead, we could simulate the Spoiler attacks in a straightforward way. For instance, we obtain the physical addresses of each store and the load, and check if the 0–19 bits of both addresses conflicts. If so, it means a high delay occurs.

However, note that Spoiler attacks not only rely on the dependency resolution logic, but also contain three key traits summarized in Section III. In particular, the former are mostly independent of our detection mechanism while the latter is the key of detecting malicious behaviors. Although it is hard to reproduce Spoiler perfectly on the gem5, the key traits stemmed from Spoiler are strictly consistent with those in the simulator and hardware implementation. Hence, SPOILER-ALERT is able to identify such attacks accurately and completely.

V. EVALUATION

A. System Settings

We evaluate SPOILER-ALERT on the gem5 simulator [18] that is an open modular platform for computer system architecture research. The gem5 is equipped with a 4-core out-of-order processor with x86 instruction set architecture (ISA). Each core contains private L1, L2 cache. Table I shows the specific CPU configuration we used in the experiment. The detection code is mainly placed in the SB component. Our detection mechanism is able to identify the malicious attacks accurately and effectively. We assess SPOILER-ALERT from aspects of the effectiveness and impact on benign applications as well as runtime overhead and storage overhead.

We run 15 individual applications of SPEC CPU 2006 in SPOILER-ALERT to observe the impact (e.g., false positive). We first perform 100 million instructions (to warm up) before starting the workloads. We simulate execution of one billion instructions and count the corresponding execution time. The baseline in our experiments is the result of native run on the gem5 with SPOILER-ALERT disabled.

TABLE I: The parameters of the gem5 simulator.

Component	Parameter
CPU	4cores, 2.6GHz
Consistency strategy	MESI
Instruction queue	8-width, 64-entry
Load buffer/store buffer	8-width, 56-entry
Re-order buffer	8-width, 192-entry
L1 instruction/data cache	32KB/48KB, 4-way, 64B cacheline, 2 cycles
L2 cache	512KB, 16-way, 64B cacheline, 20 cycles
DRAM	4096MB DDR3_1600_8x8

B. Detection Results

We implement Spoiler attacks on the modified gem5 simulator, and observe the malicious behaviors when SPOILER-ALERT is disabled and enabled, respectively. The result shows that SPOILER-ALERT is able to successfully detect Spoiler attacks before the conflict comes up. Once detected, SPOILER-ALERT sends a warning message to the user and can easily deploy countermeasures discussed in Section VI, which is able to protect physical address information from leakage. By conducting real detections to the original Spoiler attacks [19] (simulated) and CPU-intensive (SPEC 2006) benchmark, SPOILER-ALERT achieves a detection rate of 99.99% without any false positives.

C. Counter Resetting

Theoretically, both the cuckoo filter and the counter will reach the given threshold after going through a long time of execution. Such a case finally causes a false positive. For this reason, we reset the counter and erase the recording data in the cuckoo filter periodically not to exceed the threshold.

To determine the time to reset, we run the Spoiler attack and SPEC CPU 2006 benchmark, and measure the average and the maximum time of hitting the threshold of each application workload, respectively. The results are shown in Table II. We

can clearly see that the average time of reaching the threshold shows huge time differences between Spoiler and individual benign applications. The differences in the maximum time (column 3) are further enlarged. So, in our experiment, we choose the value of 0.0115 seconds as the time interval to reset the cuckoo filter and the counter. The interval chosen is slightly greater than the Spoiler's maximum to ensure the accuracy of the detection and reduce false positives.

TABLE II: The average and maximum time to reach the threshold.

Application	Average time (s)	Maximum time (s)
Spoiler	0.0048	0.0113
bzip2	3142.1946	7537.1262
gcc	16662.7933	109609.9273
mcf	19063.6319	20778.4408
perlbench	-	-
milc	-	-
leslie3d	-	-
soplex	-	-
povray	-	-
hmmer	-	-
lbm	-	-
sjeng	-	-
specrand_f	-	-
specrand_i	-	-
xalancbmk	-	-
gobmk	-	-

D. Impact on Benign Applications

SPOILER-ALERT must have a low false positive in any detection task. We run 15 benign applications in SPEC CPU 2006 along with SPOILER-ALERT to test the potential impact. Experimental results show that our detection mechanism does not cause any false positives for these applications. As mentioned in Section V-C, a pre-specified time interval is used to reset the cuckoo filter and the counter. This operation gets rid of some false positives.

E. Overhead Evaluation

Performance overhead. We utilize SPEC CPU 2006 benchmark to evaluate the runtime performance overhead of SPOILER-ALERT. The average overhead incurred is 0.24% compared with the baseline where the detection is disabled. As shown in Fig. 3, the maximum overhead is 0.55% while the minimum is only 0.03%.

Storage overhead. The storage overhead mainly stems from the implemented cuckoo filter module that is embedded into the store buffer (SB). We allocate $4 \times 4096 = 16384$ entries, in each of which a 12-bit fingerprint is placed. In total, our SPOILER-ALERT requires an additional 24 KB of storage space (4.6% compared with the L2 cache size).

VI. DISCUSSION

A. Countermeasures against the Detected Attacks

Adding store/load noises. Spoiler needs to continuously execute a certain number of store instructions and a subsequent load instruction in a moving time window. To ensure the accuracy of the obtained physical address information, the attacker must make the number of store instructions in each window sufficient to fill the SB, that is, assuring that the store instructions are owned by the attacker process only.

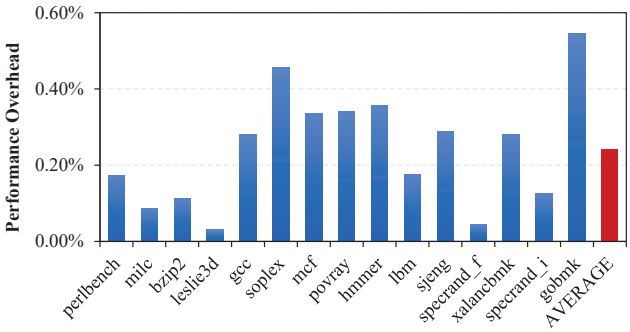

Fig. 3: The performance overhead of SPOILER-ALERT.

Therefore, we can prevent the attacker from obtaining the correct physical address information by destroying the attacker's store instructions in the moving window, or causing much high latency. Specifically, after a Spoiler attack is detected, we can execute some additional store instructions to fill the SB. Assuming that the attacker's load instruction collides with the additional store instructions, the attacker could only get incorrect physical address information. Besides, we can add speculative load instructions to generate much high latency, making it difficult for the attacker to identify the one generated by the attacker's load.

Flushing the SB. Spoiler needs store instructions to be wholly resident in the SB. If we change the condition, i.e., flushing some filled store instructions, the attacker cannot infer the correct physical address information from the timing observation. Therefore, after finding Spoiler attacks, we can clean the SB or selectively replace some entries to break this fixed moving window, thus invalidating Spoiler.

B. Limitation

There perhaps exists a corner case that SPOILER-ALERT may fail to identify Spoiler attacks in the first place when the counter is maliciously reset in the middle of an attack. This means that an attack could be reported with delays. However, our detection approach never misses out a real attack (false negative). Since the attacker has to continue accessing the following pages to achieve its goals, the threshold will eventually be capped in the subsequent time windows. Thus, SPOILER-ALERT is capable of discovering the attack behaviors during runtime.

VII. RELATED WORK

Transient-execution defenses. Multiple defenses have shown that transient execution attacks can be alleviated by (1) preventing transient instructions from accessing secret data; (2) protecting the microarchitecture state during transient execution; (3) disrupting the leakage channel [20]. OO7 [5] detects the vulnerable code locations through static analysis and then inserts a small number of $fences$ to force code instructions to be executed in sequence. SafeSpec [8] and InvisiSpec [9] separate instructions' speculative state and committable state. The data generated by the speculative instruction is stored

in a temporary structure and will be transferred to the cache when the speculation is checked correctly. In addition, adding noises into attacker's side channels used for data transmission is possible. For example, it can randomize the cache access time [12] or disable the read time stamp counter (RDTSC). Thus, it is difficult for attackers to acquire the real data.

Comparing with the data-leakage defenses above, there are only a few defense strategies that can alleviate address-leakage attacks, i.e., Spoiler attacks studied in this work. Intel releases SSBD microcode update [14] to resist Spectre V4 [13]. It inserts a $fence$ between the store and load instructions to disable speculative execution. Since Spoiler relies on such optimization (i.e., speculative load), SSBD is effective against Spoiler. However, enabling the SSBD protection must configure the BIOS manually. Further, it could bring a serious performance impact. NDA [15] can defend against all speculative execution attacks, but it incurs up to 125% performance overhead.

Detection methods. [21] designs a fine-grained non-intrusive detection system to identify Spectre-type attacks. Importantly, it tracks the cyclic interference, which is shared by all known cache side-channel attacks based on competition. SpecTaint [22] performs dynamic taint analysis to capture data flow patterns on speculative execution paths. It finally deploys a semantic-based detector to discover available Spectre gadgets. Notably, there exists two hardware performance counters (HPCs): Cycle_Activity:Stalls_Ldm_Pending and Ld_Blocks_Partial:Address_Alias. Both have a strong positive/negative correlation with Spoiler to some extent. Although it is possible to detect Spoiler by monitoring the two counters, not all of the CPUs have support for the two HPCs.

Our proposed detection mechanism that focuses on analyzing instruction addresses is able to accurately detect the address-leakage Spoiler attacks. The resulting SPOILER-ALERT only introduces less than 1% performance overhead and low storage overhead, which makes it more practical for deployment. In addition, SPOILER-ALERT is a pure hardware design without modification to the OS/software.

VIII. CONCLUSION

In this work, we highlight the need for accurately identifying the Spoiler attacks, one popular type of transient execution attacks. We have proposed a hardware-based detection mechanism named SPOILER-ALERT, which introduces a cuckoo filter module in the Memory Order Buffer to detect malicious Spoiler's behaviors dynamically. Our prototype implementation demonstrates the effectiveness of SPOILER-ALERT, which gains a high detection rate and only negligible performance overhead. Moving forward, we hope this work initiates serious consideration for the deployment of both address-leakage defense and detection mechanisms in future processor design.

IX. ACKNOWLEDGMENTS

We thank the anonymous DATE reviewers for their constructive suggestions. This work is supported by the National Natural Science Foundation of China under Grant No. 62122023, U20A20202 and 61874042, the Science and Technology Innovation Program of Hunan Province under Grant No. 2021RC4019, and the Natural Science Foundation of Fujian Province under Grant No. 2021J01544.

REFERENCES

[1] P. K. et al., "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2019, pp. 1–19.
[2] V. Kiriansky and C. A. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *CoRR*, vol. abs/1807.03757, 2018.
[3] S. I. et al., "SPOILER: speculative load hazards boost rowhammer and cache attacks," in *USENIX Security Symposium*, 2019, pp. 621–637.
[4] Y. K. et al., "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA*, vol. 42, 2014, pp. 361–372.
[5] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via program analysis," *IEEE Trans. Software Eng.*, vol. 47, no. 11, pp. 2504–2519, 2021.
[6] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "Specfuzz: Bringing spectre-type vulnerabilities to the surface," in *USENIX Security Symposium*, 2020, pp. 1481–1498.
[7] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Speccfi: Mitigating spectre attacks using cfi informed speculation," in *IEEE S&P*, 2020, pp. 39–53.
[8] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. B. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *DAC*, 2019, pp. 1–6.
[9] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *MICRO*, 2018, pp. 428–441.
[10] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *ISCA*, 2020, pp. 132–144.
[11] J. Fustos, F. Farshchi, and H. Yun, "Spectreguard: An efficient data-centric defense mechanism against spectre attacks," in *DAC*, 2019, pp. 1–6.
[12] D. Trilla, C. Hernández, J. Abella, and F. J. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems," in *DAC*, 2018, pp. 98:1–98:6.
[13] J. Horn, "Speculative execution, variant 4: speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.
[14] L. Culbertson, "Addressing new research for side-channel analysis," https://newsroom.intel.com/editorials/addressing-new-research-for-side-channel-analysis/, 2018.
[15] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: preventing speculative execution attacks at their source," in *MICRO*, 2019, pp. 572–586.
[16] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *CoNEXT*, 2014, pp. 75–88.
[17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[18] N. L. B. et al., "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.
[19] S. I. et al., "Spoiler," https://github.com/saadislamm/SPOILER, 2019.
[20] W. Xiao-Hui, H. Ye-Ping, M. Heng-Tai, Z. Qi-Ming, and L. Shao-Feng, "Microarchitectural transient execution attacks and defense methods," *Journal of Software*, vol. 31, no. 2, pp. 544–563, 2020.
[21] A. Harris, S. Wei, P. Sahu, P. Kumar, T. M. Austin, and M. Tiwari, "Cyclone: Detecting contention-based cache information leaks through cyclic interference," in *MICRO*, 2019, pp. 57–72.
[22] Z. Q. et al., "Spectaint: Speculative taint analysis for discovering spectre gadgets," in *NDSS*, 2021, pp. 21–25.