2023 30th IEEE International Conference on Electronics, Circuits and Systems (ICECS) | 979-8-3503-2649-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICECS58634.2023.10382921

Hardware Mitigation and Verification For Rogue In-Flight Data Load Attacks

Nimish Mathure, Sudarshan K. Srinivasan, Kushal K. Ponugoti and Arun Govindankutty
Department of Electrical & Computer Engineering
North Dakota State University (Fargo, ND, USA)
{nimish.mathure, sudarshan.srinivasan, kushalkumar.ponugoti, arun.g}@ndsu.edu

***Abstract*—Rogue In-Flight Data Load (RIDL) is a microarchitecture security attack that exploits store-to-load forwarding in the line fill buffer. Several microarchitecture-level mitigations have been proposed for defense against RIDL. However, errors and/or Trojans in the implementations of these mitigations can be exploited to render the microarchitecture vulnerable to RIDL. We propose a formal verification methodology that can be used to guarantee that line fill buffer implementations are immune to RIDL attacks. We also propose a hardware-mitigation for RIDL, inspired by our verification approach, that allows the use of store-to-load forwarding. We have demonstrated the efficacy of our approach using several memory pipeline benchmarks.**

***Index Terms*—Formal Verification, Hardware Trojans, Microarchitecture security attacks, Rogue In-Flight Data Load security attack.**

I. Introduction

Discovery of Rogue In-flight Data Load (RIDL) [1], which is a Microarchitectural Data Sampling (MDS) attack on modern microprocessors has exposed a security chasm. RIDL utilizes a passive eavesdropping approach on in-flight data flowing through the Line Fill Buffer (LFB) to leak information across address spaces and privilege boundaries. A large number of fence instructions need to be incorporated for software mitigation, causing significant performance impact and making RIDL impractical to be stopped by using just software mitigations. Hardware mitigations have been proposed that can be employed in new microarchitecture designs to prevent RIDL. It is possible that the mitigations can be infected by bugs or hardware Trojans that can render them ineffective. Therefore, it is necessary to develop verification methods that can be used to detect bugs and Trojans in the mitigations and guarantee invulnerability to RIDL attacks.

Contributions. Our specific contributions are as follows.

- A formal verification approach that can be used to check if a given microarchitecture implementation is invulnerable to RIDL and is guaranteed to detect bugs and hardware Trojans that would make the implementation vulnerable to RIDL attacks.
- A hardware mitigation that can be employed against RIDL that allows Line Fill Buffers to perform secure speculation.
- A generic Trojan template that can be utilized to design hardware Trojans to circumvent mitigations and make the microarchitecture vulnerable to RIDL attacks.

The approach is evaluated on a memory pipeline structure that includes Load Buffer, Store Buffer, L1 Data Cache (L1D), L2 Cache and Line Fill Buffer (LFB). The LFB included meets the specifications described in the RIDL attack [1].

The rest of the paper is organized as follows. Related work is reviewed in Section II. Background on memory pipeline components and RIDL is given in Section III. The proposed verification methodology is described in Section IV. Proposed RIDL mitigation is given in Section V. Trojan templates and threat model are given in Section VI. Verification results and conclusions are given in Sections VII and VIII, respectively.

II. Related Work

Cheang et al. [2] propose a formal correctness framework for programs called secure speculation that detects security vulnerabilities in programs. Their approach is targeted at verifying programs and not the microarchitecture implementation, which is the target of our approach. Therefore, their method cannot be used to check the correctness of microarchitecture mitigations and also cannot be used to detect Trojans that circumvent mitigations. Unique Program Execution Checking (UPEC) [3] [4] [5], a formal verification approach, has been employed to detect microarchitecture security vulnerabilities that can lead to transient execution attacks such as Spectre and Meltdown. Their approach has not been demonstrated to detect vulnerabilities that can lead to RIDL or verify RIDL mitigations.

III. Background

A. Memory Pipeline

The attacker exploits the following microarchitectural components to mount the RIDL attack: Store Buffer, L1D cache, and Line fill buffer.

1) Store Buffer: Store instructions have a high latency thereby resulting in stalling the processor pipeline. When a store operation is encountered, the data to be written is placed into the store buffer along with the address of the corresponding memory location. The store buffer keeps track of the pending writes and their associated memory addresses. The CPU can then proceed with executing subsequent instructions while the store buffer handles the eventual memory updates.

This work was supported by the National Science Foundation under Grant CNS-2117190.

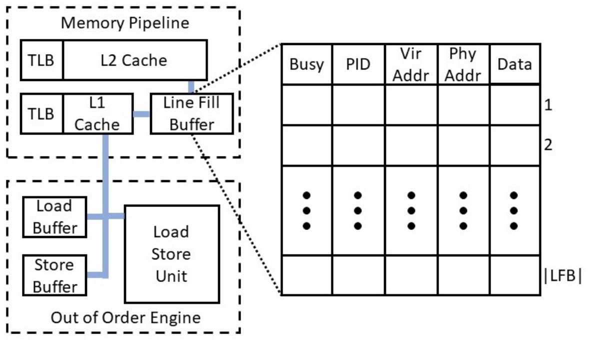

Fig. 1. Line Fill Buffer

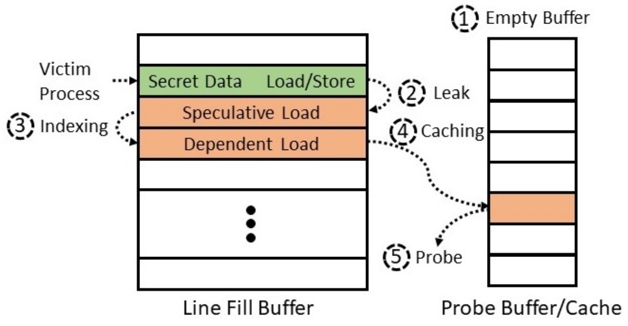

Fig. 2. Overview of RIDL attack

2) Line Fill Buffer: Line Fill Buffer (LFB) is used to improve memory throughput and is situated between the L1 cache and the L2 cache as shown in Figure 1. If there is an L1 cache miss, the memory access (both reads and writes) that caused the miss is allocated a line in the LFB and waits there until the line is retrieved from a lower-level cache or memory. This allows the L1 cache to be non-blocking and continue to process other memory requests. The LFB can also contain evicted data from L1 that needs to update lower-level cache or memory.

3) Store-To-Load Forwarding: The Load Store Unit (LSU) when processing a load instruction will scan the store buffer and the LFB for stores that have a matching address and will forward data if a match is found. This is known as store-to-load forwarding and was proposed in patent [6].

B. RIDL

The Rouge In-flight Data Load (RIDL) security attack consists of five stages as show in Figure 2. The attack process is required to run on the same core as the victim process. In the first stage, the attack process empties the L1 Data cache by flushing the lines using the *clflush* instruction. A buffer is also setup in memory. In the second stage, the attack process assumes that data from the victim process is available in the LFB. The attack process executes a load with the assumption that it will capture a value from the victim process through store-to-load forwarding in the LFB. In the third stage, the attack process executes a second load to the buffer, such that, the data from the first load is used as the index into the buffer for the second load. In the fourth stage, the data from the second load is cached. It is key to note here that the line number that is now filled will correspond to the data from the victim process. In the fifth stage, the attack process accesses each element of the buffer. Since the cache lines have been flushed in the first stage, the buffer entry corresponding to the second load will give a hit, and all other accesses to the buffer will be misses. Since misses take much longer to process, timing difference between a hit and miss is exploited to identify the index, which is the data forwarded from the victim process.

We make two notes here. First, store-to-load forwarding is key to launch a RIDL attack. Second, the data value obtained from the victim process may not be useful, as there is no guarantee that the victim process has data in the LFB. However, the above five stages when repeated a large number of times has been demonstrated to leak useful data. Therefore, while RIDL attacks are slow, they are still very potent and must be defended against using all measures possible.

IV. Verification Methodology

Memory pipelines in modern microarchitectures do not prevent data from being speculatively forwarded between instructions from different threads/processes (we call cross-process forwarding), which is what is exploited by RIDL attacks. If cross-process forwarding is not allowed, then the resulting microarchitecture will be invulnerable to RIDL. Our formal methods approach to ascertain if a microarchitecture is vulnerable to RIDL or not is based on detecting such behavior. We propose an inductive invariant termed In-process Store-to-load Forwarding Isolation (ISFI) invariant, which if satisfied, guarantees that the microarchitecture will not allow cross-process store-to-load forwarding.

Inductive invariants are predicates that are valid in every reachable state of a design, encompassing the initial/reset states as well. The verification of inductive invariants involves a two-step process. First, the invariant is assessed for its validity in the reset states. Second, it is verified that if the invariant holds true in a particular state, it will continue to hold true in the subsequent states reachable from that state. This comprehensive examination by inductive invariants ensures their consistency throughout the design's states, ensuring correctness and reliability.

Definition 1. (Inductive Invariant) *inv() is an inductive invariant iff, for all $w, v \in \mathbb{M}$ and for all $w_0 \in \mathbb{M}_0$, the following holds: (1) inv(w_0); and (2) inv(w) $\rightarrow$ inv(v); where v=ma-step(w).*

In the given context, we consider $\mathbb{M}$ as the collection of all states within the microarchitecture, while $\mathbb{M}_0$ specifically denotes the set of initial or reset states. The term *ma-step(w)* refers to a microarchitecture step, representing the transition from one state to another. Additionally, *inv()* represents the invariant that is being considered.

We define the ISFI invariant next. The Load Store Unit (LSU) processes memory operations (load or store) and has an address field and a data field. Store-to-load forwarding is implemented in the LSU. If the LSU is processing a load, it scans the LFB for store instructions that have a matching address with the load. If the address matches, the data is forwarded to the load from the store. Since there is no check to see if the store is from the same process as the load, or from a different process, cross-process forwarding is possible.

Our verification approach employs two history variables in the LSU called LSU.Addr-PID and LSU.Data-PID, which record the process ID of the address field and the data field of the memory operation in the LSU, respectively. History variables are used only for verification purposes and do not impact the functionality of the Design Under Verification (DUV), here the memory pipeline. Instructions in the LFB are also augmented with a process ID, called Instr-PID (again a history variable), which indicates the process that the instructions in the LFB belong to. If any data is forwarded in the LSU, then the history variables are updated accordingly.

Invariant 1. (In-process Store-to-load Forwarding Isolation (ISFI) Invariant) *If LSU.Busy=1 and LSU.Done=1, then LSU.Addr-PID = LSU.Data-PID.*

The invariant states that if the LSU is busy (LSU.Busy=1) and done with execution of the memory operation (LSU.Done=1), then the process ID of the instruction address should match with the process ID of the data in the LSU.

V. RIDL Process ID Mitigation

Schaik et al. [1] propose disabling speculative store-to-load forwarding as a mitigation that can prevent RIDL and its variants. Since store-to-load forwarding is a widely used optimization with well-established performance benefits, we propose a mitigation that retains store-to-load forwarding while ensuring invulnerability to RIDL. The proposed mitigation approach utilizes process IDs of instructions. The overall idea is to allow store-to-load forwarding only between instructions that have the same process ID. The forwarding algorithm that implements this mitigation is shown in Algorithm 1, which we describe next.

Let LSU, SB, L1D and LFB represent the Load Store Unit, Store Buffer, Level 1 Data Cache and Line Fill Buffer, respectively. LSU.Busy indicates if the LSU is currently busy. LSU.OP represents the opcode of the instruction being processed in the LSU. LSU.Data and LSU.Addr are the data and address, respectively, of the load or store instruction in the LSU. LSU.PID represents the process ID of the instruction in the LSU.

If the LSU is processing a load, it checks the SB (for a matching store), L1D (for a hit), and LFB (for a matching store). Pertinent fields of the SB, L1D, and LFB are described below, where C represents any of the aforementioned three components that the LSU checks. C[x].Busy indicates if x^{th} entry of the component is busy. C[x].Addr and C[x].Data represent the address and the data in the x^{th} entry of the component. C[x].PID represents the process ID of the instruction in the x^{th} entry of the component.

Algorithm 1: RIDL Process ID Mitigation

```
1  IF LSU.Busy=1 and LSU.OP=Load then
2      <SB_Hit, SB_Data> ← SB_chk();
3      <L1D_Hit, L1D_Data> ← L1D_chk();
4      <LFB_Hit, LFB_Data> ← LFB_chk();
5      IF SB_chk = True then
6          LSU.Data ← SB_Data
7      else if L1D_chk = True then
8          LSU.Data ← L1D_Data
9      else if LFB_chk = True then
10         LSU.Data ← LFB_Data
11 SB_chk(): SB_Hit ← False
12 for x=1, 1≤ x ≤ |SB|, x++ do
13     IF SB[x].Busy=1 and
14     SB[x].Addr=LSU.Addr and
15     SB[x].PID=LSU.PID then
16         SB_Hit ← True and
17         SB_Data ← SB[x].Data
18 LFB_chk(): LFB_Hit ← False
19 for x=1, 1≤ x ≤ |LFB|, x++ do
20     IF LFB[x].Busy=1 and
21     LFB[x].Op=Store and
22     LFB[x].Addr=LSU.Addr and
23     LFB[x].PID=LSU.PID then
24         LFB_Hit ← True and
25         LFB_Data ← LFB[x].Value
```

Functions SB_chk() and LFB_chk() are defined on lines 11 and 18, respectively. These functions correspond to the store-to-load forwarding logic for the Store Buffer and LFB that incorporates process ID mitigation. The functions return two values, a Boolean value indicating if there was a hit and Data from the component. In SB_chk(), each of the entries are checked sequentially (line 12) for a matching store to the load in the LSU. A match occurs if the entry is busy (line 13), the addresses match (line 14), and the process IDs match (line 15). If there is a match, then the function returns True for Hit and the Data from the store instruction with the match. The logic for SB_chk() is similar. L1D_chk() function represents the standard cache logic searching for a cache hit and is therefore not shown. The overall forwarding logic (starting in line 5) checks for matching stores in the SB, then checks L1D for a hit, and then the LFB for a matching store. The key idea here is that RIDL relies on data being forwarded from one process to another. The PID checks do not permit the aforementioned behavior and therefore RIDL attacks cannot be executed if this mitigation is in place.

VI. RIDL Process ID Mitigation Trojans

We identify hardware Trojans that can be used to circumvent the RIDL process ID mitigation. The threat model for these Trojans is similar to other hardware Trojans, i.e., spies or mercenaries who are part of the design cycle or supply chain can insert these Trojans in the RTL, netlist, or even during fabrication. The mitigation relies on checking if the process ID of the store instruction in the LFB/SB matches with the process ID of the load instruction in the LSU. The payload of the Trojans is to circumvent this check, thereby making the memory pipeline vulnerable to RIDL attacks. There are numerous ways that the check can be circumvented. One approach is to force the output wire corresponding to the check to a 1 value. Therefore, store-to-load forwarding will be enabled even if the process IDs do not match, allowing for RIDL attacks. The Trojans can be activated using input triggers, time-based triggers, functional triggers, or stealth triggers.

The two motivations for flagging these Trojans as part of this work is as follows. Firstly, even if the mitigation is in place to prevent RIDL attacks, it can be circumvented by unintentional implementation errors or Trojans. Therefore, it is essential to check the RIDL invulnerability invariant as part of the memory pipeline validation and verification process. Secondly, the Trojans can be inserted during fabrication. Therefore, post-fabrication testing methods must be incorporated to detect such Trojans.

TABLE I
RIDL Invulnerability Invariant Verification Results

Benchmark	Invariant Violated?	Time (Secs)
MP-No-Mit	Yes	0.01
MP-Mit	No	0.01
MP-Mit-Bug	Yes	0.01
MP-Mit-Trojan-1	Yes	0.01
MP-Mit-Trojan-2	Yes	0.01
MP-Mit-Trojan-3	Yes	0.01

VII. Experimental Results

Table 1 gives the results for the RIDL invulnerability invariant verification. The benchmarks are named as follows. Benchmark *MP* is the memory pipeline RTL without any mitigations. The memory pipeline configuration includes a 4-entry store buffer, a 4-entry load buffer, and a 4-entry LFB. The suffixes *-No-Mit* and *-Mit* incorporate no mitigations against RIDL and the proposed process ID mitigation described in Section IV, respectively. Benchmark with suffix *-Bug* has an implementation bug, where one of the process ID check wires is stuck at 1. Benchmarks with suffix *-Trojan-n* incorporate RIDL process ID Trojans as described in Section V. *-Trojan-1* uses a rare value of one of the cache lines as the trigger. *-Trojan-2* uses a rare value of one of the entries in the LSU as the trigger. *-Trojan-3* uses a combination of a rare value of a timer and a cache line as the trigger. Trojans 1 and 2 are functional Triggers. Trojan 3 is a stealth trigger that is a combination of a functional trigger and a time-based trigger.

The invariant verification was performed automatically using the z3 Satisfiability Modulo Theories (SMT) solver version 4.8.7 [7]. Verification experiments were performed on a Linux 64-bit operating system running on an Intel(R) Core(TM) i9 - 12900K CPU @ 3.2 GHz with 32 GB RAM. The invariant accurately classified all benchmarks and produced a counterexample for benchmarks with bugs and Trojans. Verification times are also very efficient, thereby demonstrating the efficiency and scalability of the verification approach.

VIII. Conclusion

The proposed Process ID mitigation allows for store-to-load forwarding to be used. The proposed verification methodology is very efficient. Together, the mitigation and the verification approach can be used to defend against RIDL attacks while enabling store-to-load forwarding. Hardware Trojans designed to circumvent the mitigation will be detected by the verification approach. Testing methods will need to be developed to detect Trojans inserted during fabrication as the formal verification method is not applicable post-fabrication. For future work, we propose to extend this verification methodology to other microarchitecture data sampling attacks that rely on store-to-load forwarding, such as zombie load and fallout.

References

[1] S. van Schaik et al., "RIDL: Rogue In-Flight Data Load," 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 88-105, doi: 10.1109/SP.2019.00087.

[2] Cheang, K., Rasmussen, C., Seshia, S., Subramanyan, P.: 'A formal approach to secure speculation', IEEE 32nd Comp. Sec. Foundations Symp. (CSF), Hoboken, NJ, USA, June 2019, pp. 288–28815

[3] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel and W. Kunz, "A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors," 2020 57th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 2020, pp. 1-6, doi: 10.1109/DAC18072.2020.9218572.

[4] J. Müller, M. R. Fadiheh, A. L. D. Antón, T. Eisenbarth, D. Stoffel and W. Kunz, "A Formal Approach to Confidentiality Verification in SoCs at the Register Transfer Level," 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 2021, pp. 991-996, doi: 10.1109/DAC18074.2021.9586248.

[5] M. R. Fadiheh et al., "An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors," in IEEE Transactions on Computers, vol. 72, no. 1, pp. 222-235, 1 Jan. 2023, doi: 10.1109/TC.2022.3152666.

[6] Dec 20, 2012- A. M. D. (n.d.). Store-to-load forwarding. Justia. Retrieved March 27, 2023, from https://patents.justia.com/patent/20140181482

[7] Moura, L.d., Bjørner, N.: 'Z3: An Efficient SMT Solver', Int. Conf. on Tools and Algo. for the Const. and Analysis of Sys. (TACAS), Springer, Budapest, Hungary, March-April 2008, pp. 337–340