SpecLFB: Eliminating Cache Side Channels in Speculative Executions

相关信息

SpecLFB: Eliminating Cache Side Channels in Speculative Executions

Xiaoyu Cheng[1,2], Fei Tong[1,2,3,*], Hongyu Wang[4,5], Zhe Zhou[1,2], Fang Jiang[1,2], Yuxing Mao[4]

[1]*School of Cyber Science and Engineering, Southeast University, Nanjing, Jiangsu, China*
[2]*Jiangsu Province Engineering Research Center of Security for Ubiquitous Network, China*
[3]*Purple Mountain Laboratories, Nanjing, Jiangsu, China*
[4]*State Key Laboratory of Power Equipment Technology,*
School of Electrical Engineering, Chongqing University, China
[5]*Wiscom System Co., LTD, Nanjing, China*
[*]*Corresponding Author: ftong@seu.edu.cn*

https://www.usenix.org/conference/usenixsecurity24/presentation/cheng-xiaoyu

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

目录

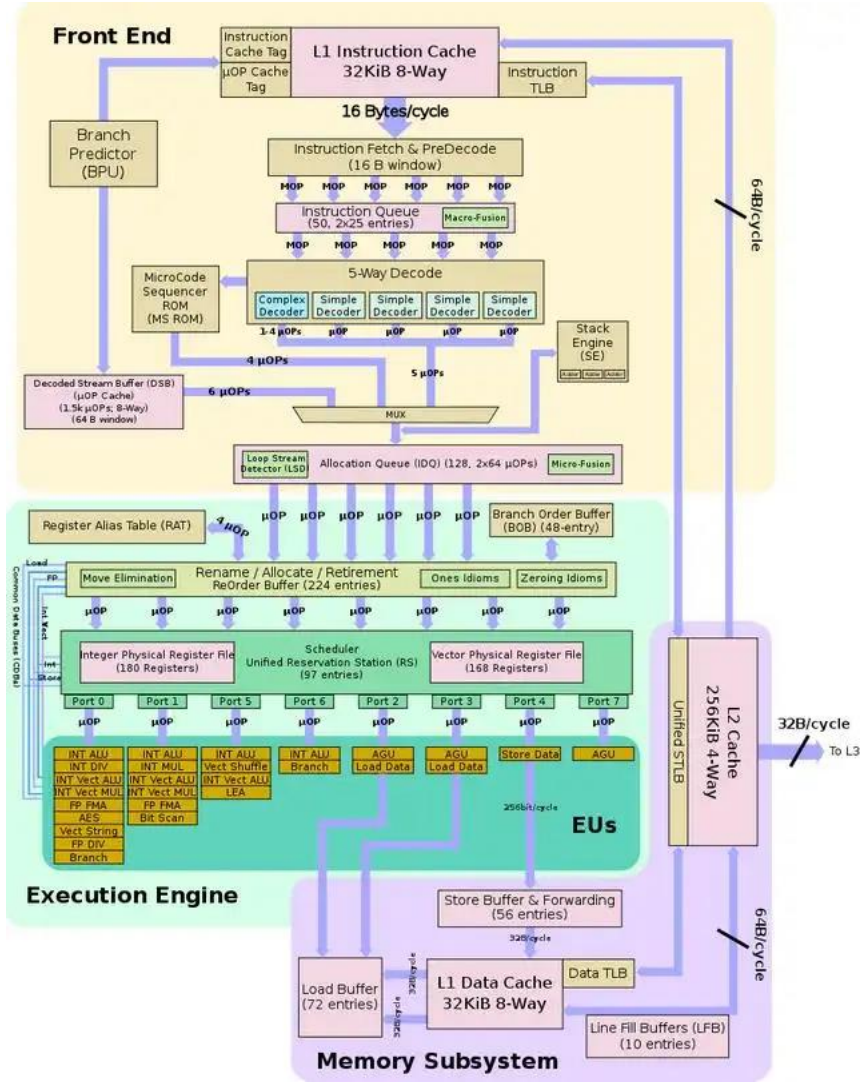
工作意义

SpecLFB

Defense against speculative cache side-channel attacks

It leverages the microarchitectural component Line-fill-buffer (LFB) and introduces a simple yet effective security check mechanism to LFB.

To prevent potentially unsafe speculative loads from being reloaded from lower-level cache into adjacent upper-level cache (e.g. from L2 cache into L1 data cache (L1D cache)), thereby preventing the establishment of cache side channels.

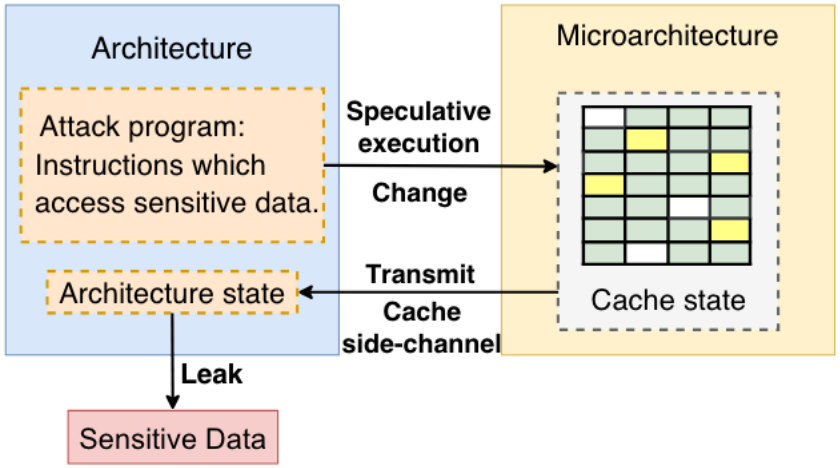


Prerequisite

The speculative cache side-channel attacks, which simultaneously exploit both speculative execution and cache side-channel vulnerabilities of processors.

Speculative execution attack exploits the side effects of the transient instructions which are mis-speculated and destined to be squashed.

A cache side channel attack can establish cache side channels, thus providing a covert channel for the speculative execution attacks.



Prerequisite

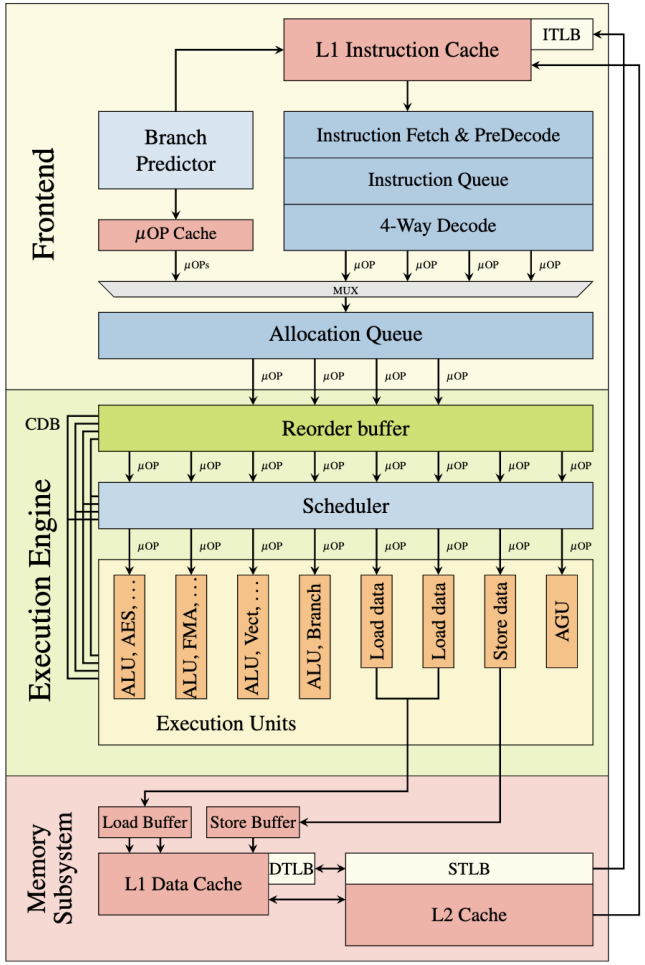
Transient Execution

In modern OoO processors, instructions issued in program order are decoded into micro-operations (μops), but the execution of μops may not follow the program order.

Since the execution suspension will reduce processor performance, the processor can predict the results of conditional branches and data dependencies through some components such as branch predictors, memory disambiguators, etc., and then execute along the speculative path.

Therefore, these instructions that are executed out of order and whose executions are incorrectly speculated may cause the microarchitectural state of the processor to change, even though their results are never committed to the architectural state due to processor rollback.

This is called transient execution, and these instructions are called transient instructions.



Prerequisite

Spectre v1

The attacker mistrains the pattern history table, which determines whether the conditional branch instructions should take the branch or not.

```
if (x < array1_size)
    y = array2[array1[x] * L1_BLOCK_SZ_BYTES];
```

Spectre v2

The attacker mistrains the branch target buffer of indirect branch instructions with malicious target addresses corresponding to code snippets called gadgets, which can leak sensitive data,

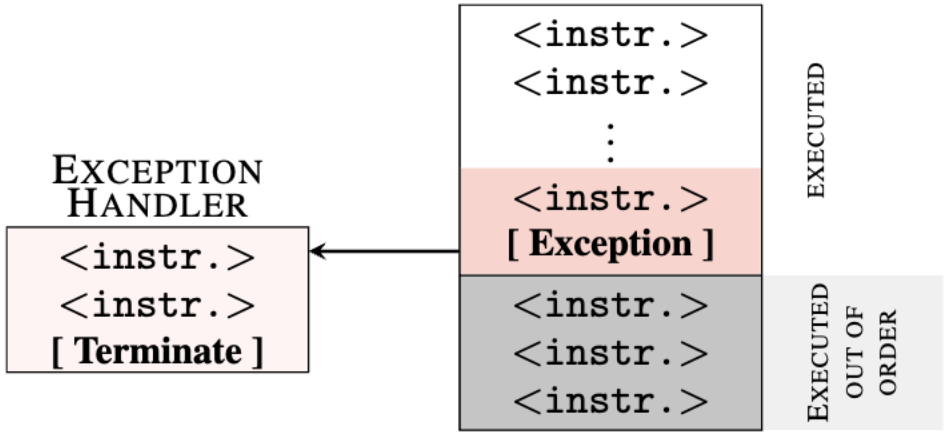
```
void victimFun(uint64_t x)
{
    uint64_t y = array2[array1[x] * L1_BLOCK_SZ_BYTES];
}
```

Prerequisite

Spectre v3 (Meltdown)

If an executed instruction causes an exception, diverting the control flow to an exception handler, the subsequent instruction must not be executed. Due to out-of-order execution, the subsequent instructions may already have been partially executed, but not retired.

However, architectural effects of the execution are discarded.




```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```

Prerequisite

Spectre v4

The attacker exploits the misprediction of the memory disambiguator (which is used to speculate which load instructions do not depend on any preceding store instructions) to trigger transient executions of unsafe load instructions, allowing access to sensitive data.

```
ptr = secret_ptr;
ptr = general_ptr;//store operation
x = *ptr;//load operation
y = array2[x];
```

Spectre v5 (SpectreRSB)

The attacker exploits the return stack buffer (RSB), which is a hardware stack used to track the return addresses of previous call instructions, to supply CPU with a malicious return address corresponding to a gadget (the same as in Spectre v2)

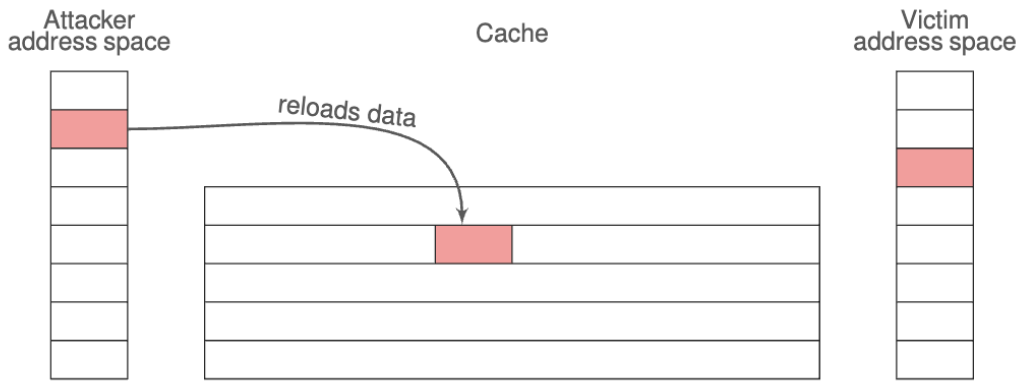
When encountering a ret instruction, the processor speculatively uses the top address of the RSB as the return address, as accessing the software stack is slower.

```
void victimFun(uint64_t x)
{
    uint64_t y = array2[array1[x] * L1_BLOCK_SZ_BYTES];
}
```

Prerequisite

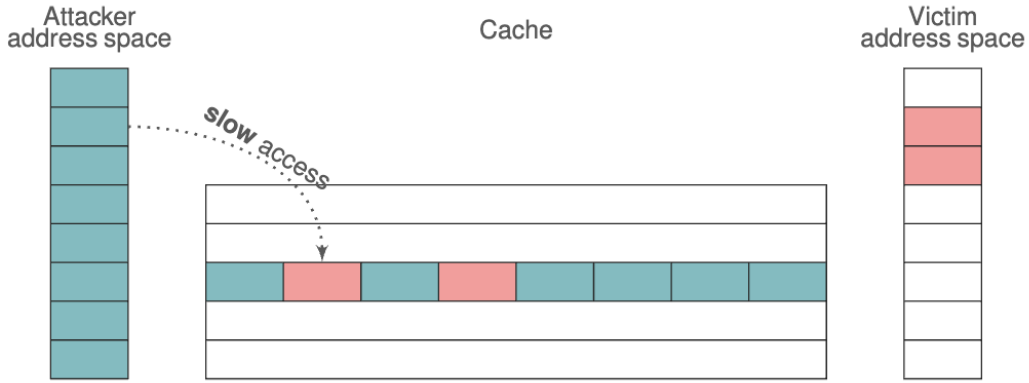
Cache Side-channel Attack

Flush-Based Attack(eg. Flush+Reload)



step 0: attacker maps shared library → shared memory, shared in cache
step 1: attacker flushes the shared line
step 2: victim loads data while performing encryption
step 3: attacker reloads data → fast access if the victim loaded the line

Conflict-Based Attack(eg. Prime+Probe)



step 0: attacker fills the cache (prime)
step 1: victim evicts cache lines while performing encryption
step 2: attacker probes data to determine if the set was accessed

目录

相关工作

Overall, most of the existing hardware defense solutions require additional data structures, data movement operations and/or complex logical calculations, which can cause excessive additional performance and hardware resource overhead.

Existing mechanisms for mitigating speculative cache sidechannel attacks can be classified into four types, including

1) limiting the execution of speculative instructions,
2) making the results of unsafe speculative executions invisible to the microarchitectural state,
3) delaying the executions of unsafe speculative instructions, and
4) reducing the accuracy of the covert channel.

相关工作

Limiting the execution of speculative instructions

Intel and AMD recommend using serialization instructions, e.g., lfence, in a branch. ARM introduces a full data synchronization barrier and instruction synchronization barrier that can be used to prevent speculation.

However, serializing every branch would be equivalent to disabling branch speculation entirely, severely degrading processor performance.

Reducing the accuracy of the covert channel

Many web browsers lower the accuracy of timers in JavaScript by adding jitter or even removing some timers.
Disruptive Prefetch and Prefender add noise to cache side channels by utilizing the content brought into the cache by data prefetchers.

But they can only achieve secure enhancement without fully defending against the attacks.

相关工作

Making the results of unsafe speculative executions invisible to the microarchitectural state

InvisiSpec adds an SB to store the speculative data leading to security issues. If the speculation is incorrect, the data stored in SB becomes invalid. Otherwise, the speculative data will be re-installed into cache.
Similarly, SafeSpec stores speculated data in a fully associative shadow structure of the load/store queue.

CleanupSpec allows the data to be installed into the cache during the speculative execution, and the replaced data is stored into a newly added data structure. The replaced data will be re-installed into the cache hierarchy to roll back the cache state only when speculation fails.

Overall, not only they require additional data structures to hide the cache lines fetched by the speculative loads but also the data movement caused by the re-installing operation still degrades processor performance.

相关工作

Delaying the executions of unsafe speculative instructions

Both NDA and STT use data flow tracking similar to taint propagation to track instructions that may cause information leakage. These instructions are forced to be delayed until their related instructions become safe.
SSE-RV, based on STT, uses existing ROB pointers in SonicBOOM to track speculated instructions, simplifying the tracking analysis logic.

However, it still requires a large taint file as a shadow structure to implement tainting, and unnecessary instructions may still be incorrectly identified as unsafe instructions, which increases the delay of execution.

目录

设计实现

When the data targeted by a memory access instruction is missed in the upper-level caches, the miss handling logic first checks MSHR to see if there is a pending request that matches the current one.
If so, the request is merged into the same entry. Otherwise, a new MSHR entry and LFB entry will be reserved for this data request.

When the requested cache line is fetched from the lower-level caches or memory, it is placed into LFB instead of directly being written into the upper-level caches.
When the cache eviction is complete, LFB is flushed into the cache arrays.

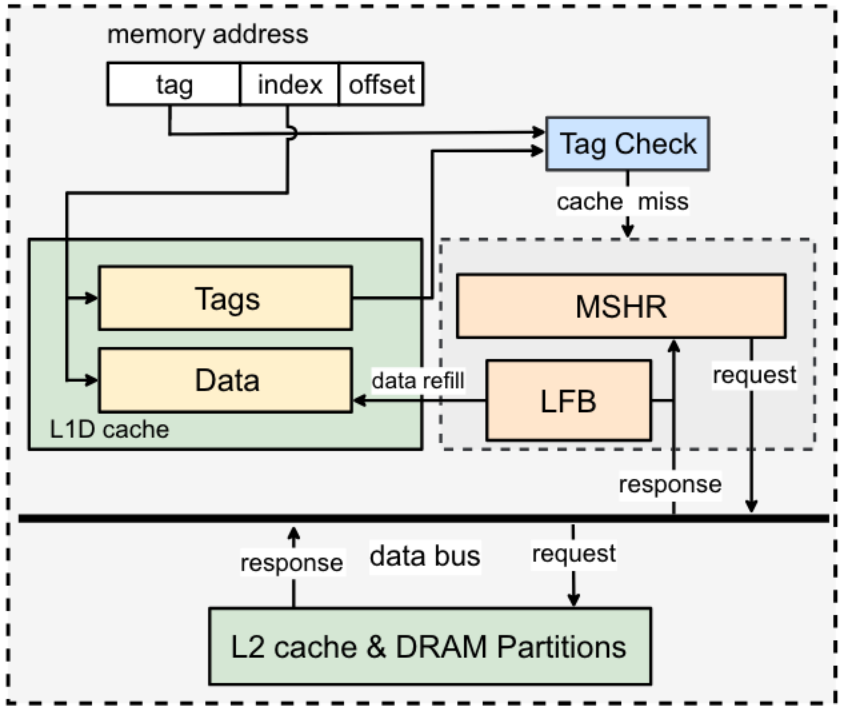
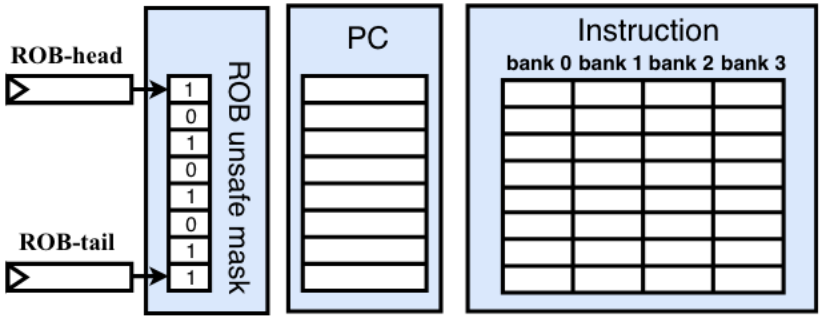


Figure 2: The working principle of LFB in L1D cache.

设计实现

After LFB obtains the missing data requested by the L1D cache, it judges whether the data can pass the security check mechanism according to the ROB unsafe mask bit corresponding to its related instruction. Only a mask bit of 0 can it pass the check and continue refilling the data into the L1D cache.



$$r_i = b_{i1} \,||\, b_{i2} \,||\, \ldots \,||\, b_{ij} \,||\, \ldots \,||\, b_{i(N-1)} \,.$$

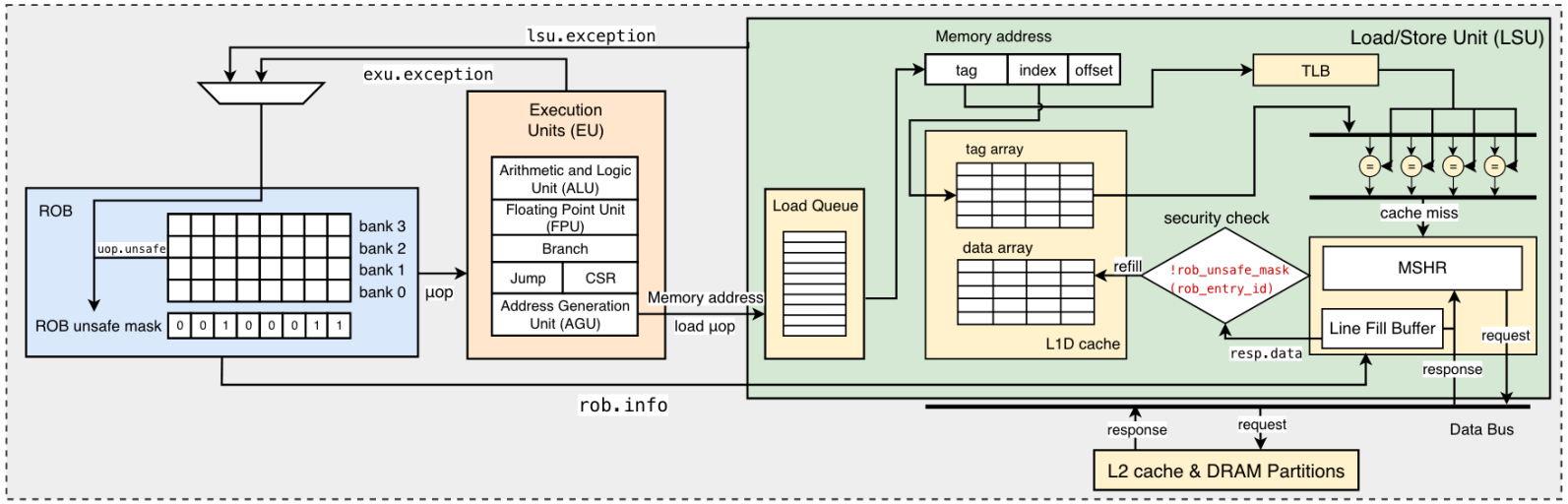


Figure 7: Design overview of SpecLFB.

设计实现

Speculation sources

- **Control flow prediction**. It predicts the execution path that a program will follow through the branch prediction unit (BPU), which may generate speculative loads in unresolved control flows.

- **Address speculation**. It predicts the data dependencies between loads and stores when the physical address is not fully available, which may generate speculative loads in an unresolved memory access order.

- **Value prediction (VP).** It predicts the result of a μop based on the execution history allowing dependent instructions to continue their execution without waiting for the result to become available, which may generate speculative loads in an unresolved value.

- **Other exceptions**. For the instructions that can cause other exceptions, such as unauthorized data access, arithmetic operations with overflow, etc., they will not be squashed until the processor can check for the exceptions.

设计实现

Parameter unsafe is a boolean variable. If an instruction meets one of the following three conditions:

- 1) it uses the load queue,
- 2) it uses the store queue (excluding a fence instruction that isolates the previous and subsequent store operations to ensure sequential execution), or
- 3) it is a branch/jump instruction,

then the unsafe parameters of the instruction's μops are set to true.

设计实现

Rules for updating ROB unsafe mask

- For the instructions in unresolved control flow, according to the information returned by BPU, SpecLFB traverses ROB in each cycle to update bi j depending on whether the instructions are still under the control flow.

- For the instructions in unresolved memory order, SpecLFB tracks data dependencies through the load queue (LDQ) and store queue (STQ) in the LSU. When placing a memory access instruction into the LDQ/SDQ, SpecLFB compares it with the instructions in every entry of the LDQ/STQ for checking data dependencies. If a data dependency is not found with an older store/load instruction, the bi j of the instruction should be updated to 0.

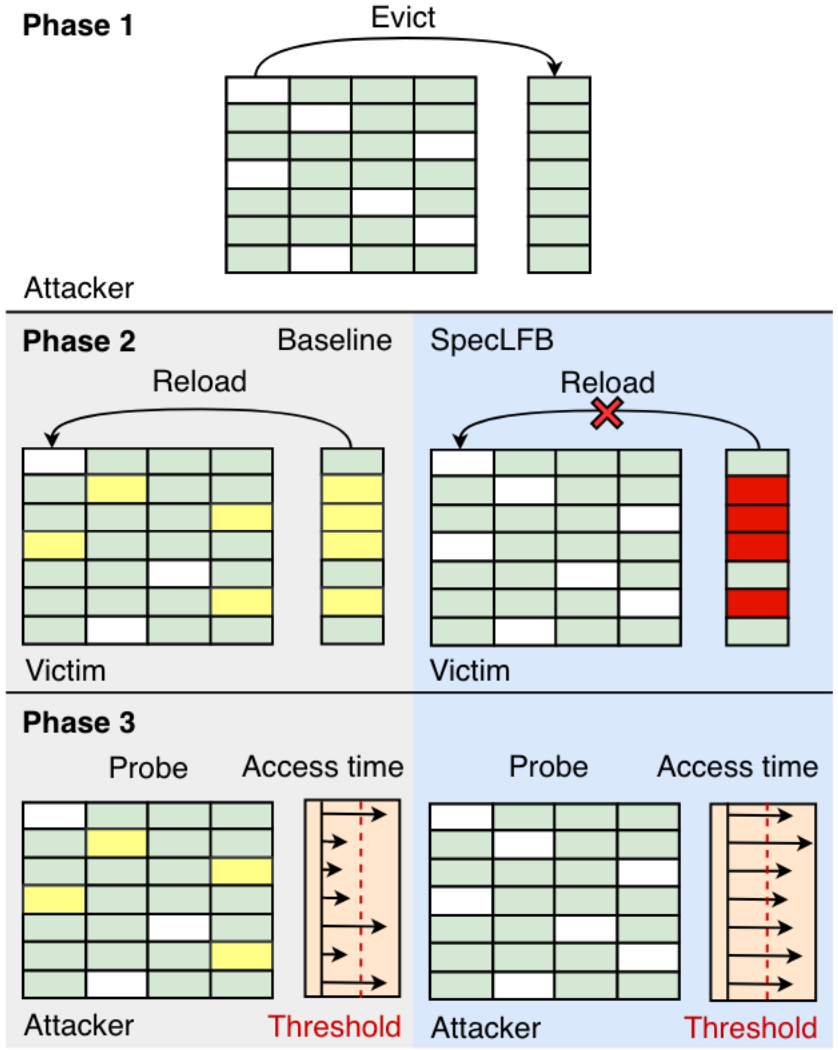
- For the instructions related to other exceptions, once the ROB receives exception information from backend units such as execution units and LSU, the bi j of the instruction and younger instructions cannot be updated to 0 until the exception has been handled.

设计实现

SpecLFB Security Analysis

In phase one, the attacker first evicts the specified cache lines from the cache.
In phase two, the victim runs the program under the conditions created by the attacker.
In phase three, the attacker probes the time it takes to execute a read at the memory address corresponding to the cache line evicted in phase one.

In the processors that adopt the SpecLFB scheme, cache lines represented by the red lattices will be regarded as data loaded by MUSLs. During the transient execution period (i.e., waiting for the processor to verify speculation), these cache lines cannot pass through the security check to be refilled into the cache.



目录

实验评估

Experimental Setup

Table 1: Processor configurations.

Processor	SonicBOOM Configurations	Gem5 Configurations
Baseline	Original SonicBOOM	Original O3 CPU
SSE-RV	SonicBOOM enhanced with SSE-RV	\
STT	\	O3 CPU enhanced with STT
SpecLFB	SonicBOOM enhanced with SpecLFB	O3 CPU enhanced with SpecLFB

SpecLFB is applied to both levels of the cache in the O3 CPU model in Gem5.

Compare the performance of SpecLFB to the state-of-the-art defense, STT, which was also implemented in Gem5.

Chipyard v1.8.0 generated RTL for each of the three SonicBOOM-based processors shown in Table 1

Simulations through Verilator v4.210 &&
A hardware prototype based on Xilinx EK-KC-705 FPGA platform (with a clock frequency of 50MHz) burned with the above three SonicBOOM cores.

Table 2: Gem5 and SonicBOOM-FPGA parameters.

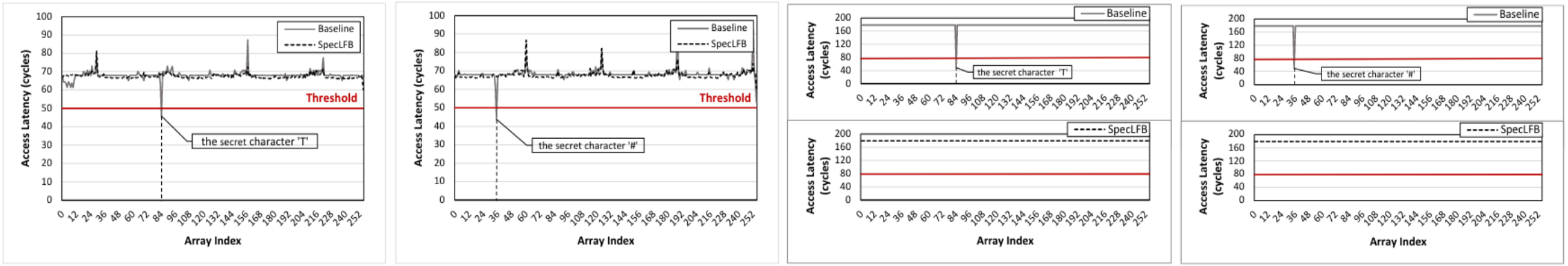
Parameter	SonicBOOM-FPGA	Gem5
ISA	RV64	X86-64
Frequency	FPGA@50MHz	simulate@2GHz
Processor type	2-decode 4-issue MediumBoom O3CPU	8-decode 8-issue DerivO3CPU
ROB/LDQ/STQ	64/16/16 entries	192/32/32 entries
L1I Cache	16KB, 4-way, 64B line	32KB, 8-way, 64B line, 4 MSHRs
L1D Cache	16KB, 4-way, 64B line, 2 MSHRs	32KB, 8-way, 64B line, 4 MSHRs
L2 Cache	512KB, 16-way, 64B line	2MB, 16-way, 64B line, 20 MSHRs

实验评估

SpecLFB Security Evaluation

The length of the secret string randomly generated exceeds 100 characters at each run.

In the processors without any defense mechanism, the success rate of leaking the secret value is 100%. With SpecLFB, the success rates in Chipyard and Gem5 drop to below 0.01% and to 0, respectively.



(a) Spectre v1 in Chipyard (b) Spectre v4 in Chipyard (c) Spectre v1 in Gem5 (d) Spectre v4 in Gem5

Figure 11: Access latency measured in the Spectre attacks through cache-based side channel.

实验评估

Performance Evaluation

Run SPEC2017 in the FPGA-based hardware prototype and Gem5 to evaluate SpecLFB.

In the FPGA-based hardware prototype, performance overhead for a processor running a workload is defined as the workload's execution time normalized to that of the Baseline,
while in Gem5, it is defined as the IPC (Instructions per Clock Cycle) of the workload normalized to that of the Baseline.

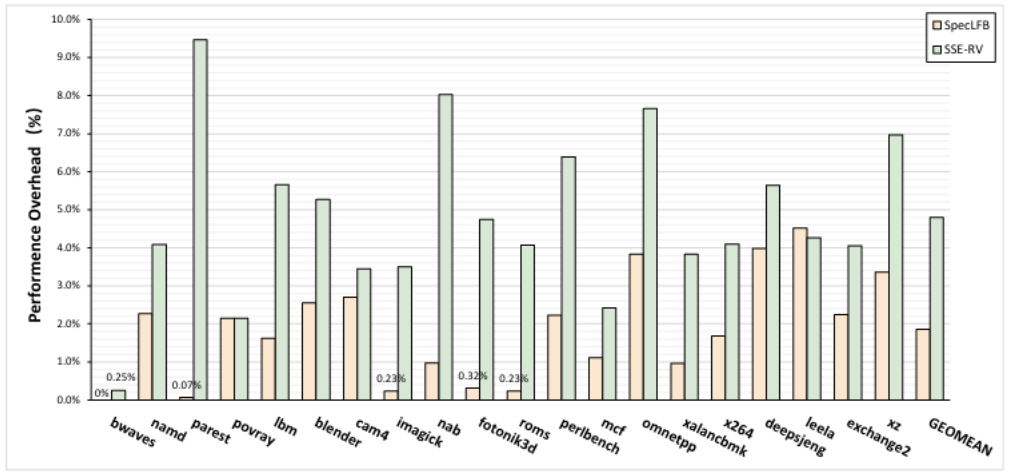


Figure 12: The performance overhead comparison between SpecLFB and SSE-RV [39] both running the selected workloads contained in SPEC2017.

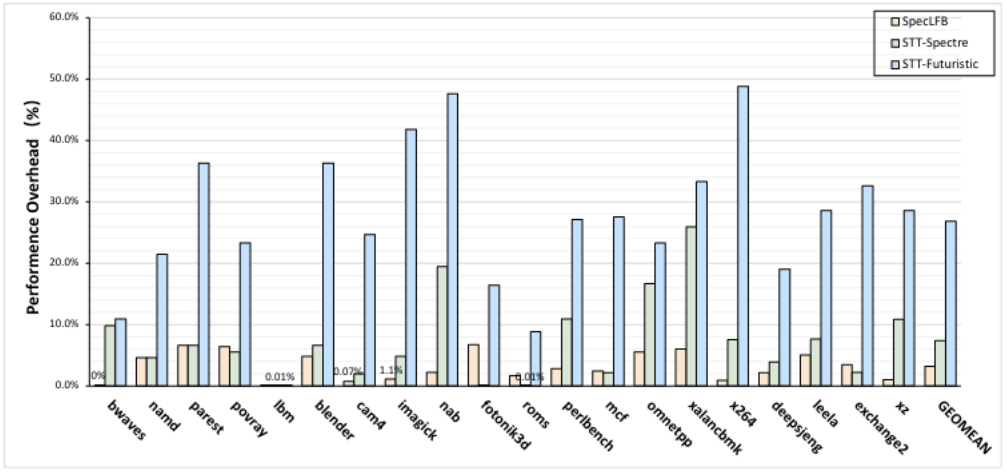


Figure 13: The performance overhead comparison between SpecLFB and STT [57] both running the selected programs contained in SPEC2017.

实验评估

Performance Evaluation

FPGA resources utilization

SpecLFB only adds less than 1% of resources compared to the baseline, and consumes fewer resources than SSE-RV, especially in terms of LUTs (LookUp-Tables) and FFs (Flip-Flops).

ProSpeCT also implements its defense hardware prototype on the open-source OoO RISC-V processor Proteus and deploys it in FPGA for resource evaluation.
While the size of BOOM and Proteus are quite different, SpecLFB has advantages in terms of both the absolute number and relative proportion of additional resource overhead.

Table 4: FPGA resources utilization.

Scheme	Core	Device	LUTs	FFs
Baseline	SonicBOOM	Xc7a325T	169,463	93,994
SSE-RV	SonicBOOM	Xc7a325T	172,538 (+1.81%)	94,567 (+0.61%)
SpecLFB	SonicBOOM	Xc7a325T	170,765 (+0.77%)	94,283 (+0.31%)
Baseline	Proteus-O3	Xc7a35T	16,847	11,913
ProSpeCT	Proteus-O3	Xc7a35T	19,728 (+17.1%)	12,600 (+5.8%)

谢谢！