ASE 2026 · MUNICH

Understanding Real-World Ordering-Related Concurrency Bugs under Weak-Memory and Cross-Architecture Settings

XIDIAN UNIVERSITY
Xi'an, China

Shaohao Wang, Cheng Wen, Yuandao Cai, Shengchao Qin, Jie Su, Mengda He, Xiaoxue Ma, Cong Tian

Xidian University | Hong Kong University | Teesside University

PAPER DETAILS

Introduction and Memory Models

1. Why study ordering-related bugs?

Modern systems software runs across x86, ARM and RISC-V, whose memory models provide different ordering guarantees. Hardware and compilers may reorder operations or delay their visibility to other threads.

Small litmus tests capture low-level anomalies, but production failures also depend on object lifetimes and protocol state. We study this gap through 58 manually confirmed ordering-related concurrency cases.

Table 1: Reorderings Permitted by Different Memory Models

Memory Model	Store-Load	Store-Store	Load-Load	Load-Store
SC	×	×	×	×
TSO (x86)	✓	×	×	×
PSO (SPARC)	✓	✓	×	×
WMM (ARM/RISC-V)	✓	✓	✓	✓

Motivating Real-World Failure

2. Publication order in CVE-2020-25603

The writer initializes a bucket, then updates a readiness boundary. The reader may see that boundary before the data and access stale or null values. A memory barrier restores data-before-publication ordering.

init: port=0, d->bucket[port]=null, d->valid_evtchns=0	
Thread1: allocate_bucket()	**Thread2:** access_bucket()
1. d->bucket[port]= new_data	3. if(port < d->valid_evtchns)
2. d->valid_evtchns+=1	4. return d->bucket[port]

Fig. 1. Simplified publication-ordering violation in CVE-2020-25603.

Multi-Granularity Ordering (MGO)

3. Classify the violated ordering assumption

MGO identifies the **primary violated ordering assumption**: L1 concerns memory visibility; L2 concerns initialization, use and reclamation; L3 concerns protocol or state-machine order. Repairs support attribution, but do not define the layer.

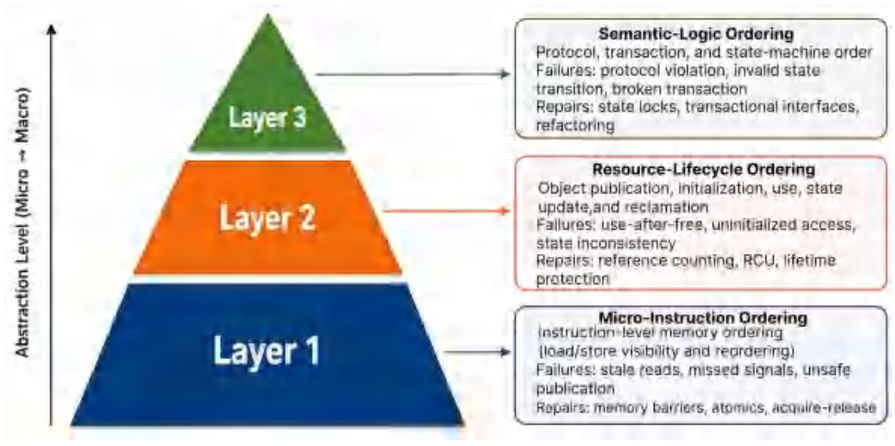


Fig. 3. The Multi-Granularity Ordering (MGO) taxonomy.

Dataset and Main Distribution

4. RQ1: What kinds of failures occur?

We curate 58 manually confirmed cases from public reports, patches, mailing lists and CVEs. Sources include kernels, databases, networking, middleware, runtimes and benchmark-style reproductions.

MGO layer	Cases	Share	Common failure
L1: Micro-instruction	32	55.2%	Stale reads; missed signals
L2: Resource lifecycle	24	41.4%	Use-after-free; invalid object state
L3: Semantic logic	2	3.4%	Protocol / state-order violations

Lifecycle and semantic bugs account for 44.8%. Layer 2 is almost as large as Layer 1; Layer 3 has only two cases.

Tool Evaluation and Results

5. RQ3: How well do existing tools work?

We evaluate CBMC, GenMC, Nidhugg and IRhunter using original reproducers or minimized benchmarks that preserve each documented ordering condition. A case is detected when a reported violation corresponds to the known bug.

Table 5: Tool Detection Results Across MGO Layers.

Benchmark	Total Cases	Detected Bugs (Detection Rate %)			
		CBMC	GenMC	Nidhugg	IRhunter
Layer 1	32	14 (43.8%)	25 (78.1%)	18 (56.3%)	18 (56.3%)
Layer 2	24	0 (0.0%)	4 (16.7%)	4 (16.7%)	0 (0.0%)
Layer 3	2	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)

GenMC detects 78.1% of Layer 1 cases. The best Layer 2 detection rate is only **16.7%**, and none of the four tools detects either evaluated Layer 3 case. Instruction-level reasoning therefore covers only part of the failure space.

Practical Implications

6. Toward cross-architecture reliability

Reduce the exploration space. Slicing-guided reduction can isolate shared variables, synchronization operations and lifecycle events before costly interleaving exploration.

Model object validity. Track allocation, initialization, publication, use, state updates and reclamation, so low-level executions can be linked to lifecycle failures.

Specify semantic order. Protocol and state-machine constraints are needed to detect incomplete transitions, such as sending data before a TLS upgrade is complete.

Repair Strategies Across MGO Layers

7. RQ2: Match the repair to the root cause

The three layers require different correctness mechanisms. Low-level ordering enforcement, lifetime protection and semantic atomicity address different assumptions; choosing the wrong level can leave the root cause unresolved or over-synchronize a hot path.

Table 3: Repair strategies across MGO layers.

Layer	Core Issue	Typical Repair Strategy
Layer 1 (Micro-instruction)	Instruction-level memory reordering or visibility delay causes stale reads, missed signals, or unsafe publication.	Use memory barriers, atomic instructions, or acquire-release operations to constrain load/store visibility and ordering.
Layer 2 (Resource lifecycle)	Incorrect ordering among object publication, initialization, use, state update, or reclamation leads to use-after-free, uninitialized access, or inconsistent state.	Use reference counting, RCU grace periods, lifecycle locks, deferred publication, or other lifetime-protection mechanisms.
Layer 3 (Semantic logic)	Multi-step protocol, transaction, or state-machine transitions are interrupted by concurrent operations, violating logical atomicity.	Enforce semantic atomicity using state locks, transactional interfaces, or protocol/state-machine refactoring.

Repair mismatch: a barrier alone cannot guarantee object lifetime; per-field locks may expose incomplete transitions. L2/L3 can fail under sequential consistency.

Conclusion. Real-world ordering failures extend beyond isolated memory reorderings. Cross-architecture reliability requires **memory visibility, object-lifecycle validity and semantic atomicity** to be analyzed together.

41st IEEE/ACM International Conference on Automated Software Engineering · October 12–16, 2026

DOI: 10.1145/3832783.3834494