

西安電子科技大學
XIDIAN UNIVERSITY



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

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







Why Study Ordering Bugs ?

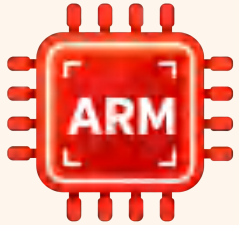
x86 / TSO



Stronger ordering guarantees

Looks correct on x86

ARM / RISC-V



Allows more load/ store reorderings

Exposed on weak-memory architectures

stale read / missed signal / memory-safety failure

CVE-2020-25603

Thread 1

```
d->bucket[port] = new_data;

d->valid_evtchns += 1;
```

Thread 2

```
if (port < d->valid_evtchns)

    return d->bucket[port];
```

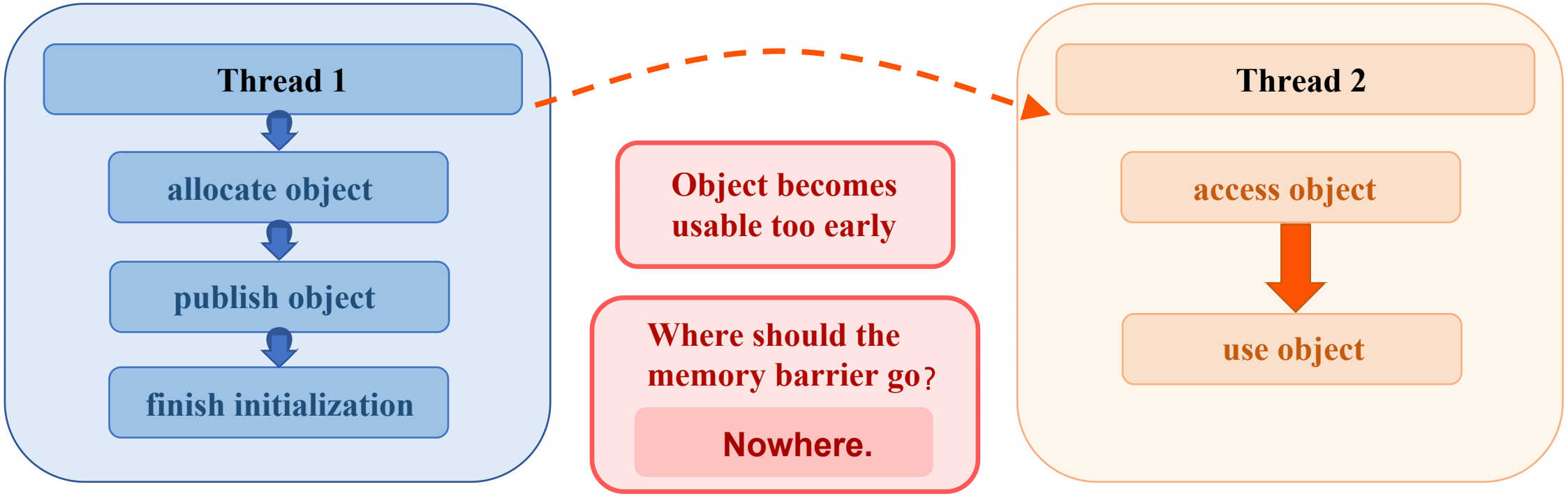
Under weak memory, valid_evtchns may become visible before bucket initialization.

missing memory barrier → **null / stale data** → **crash**

This is the ordering bug we already know how to reason about.

Motivation

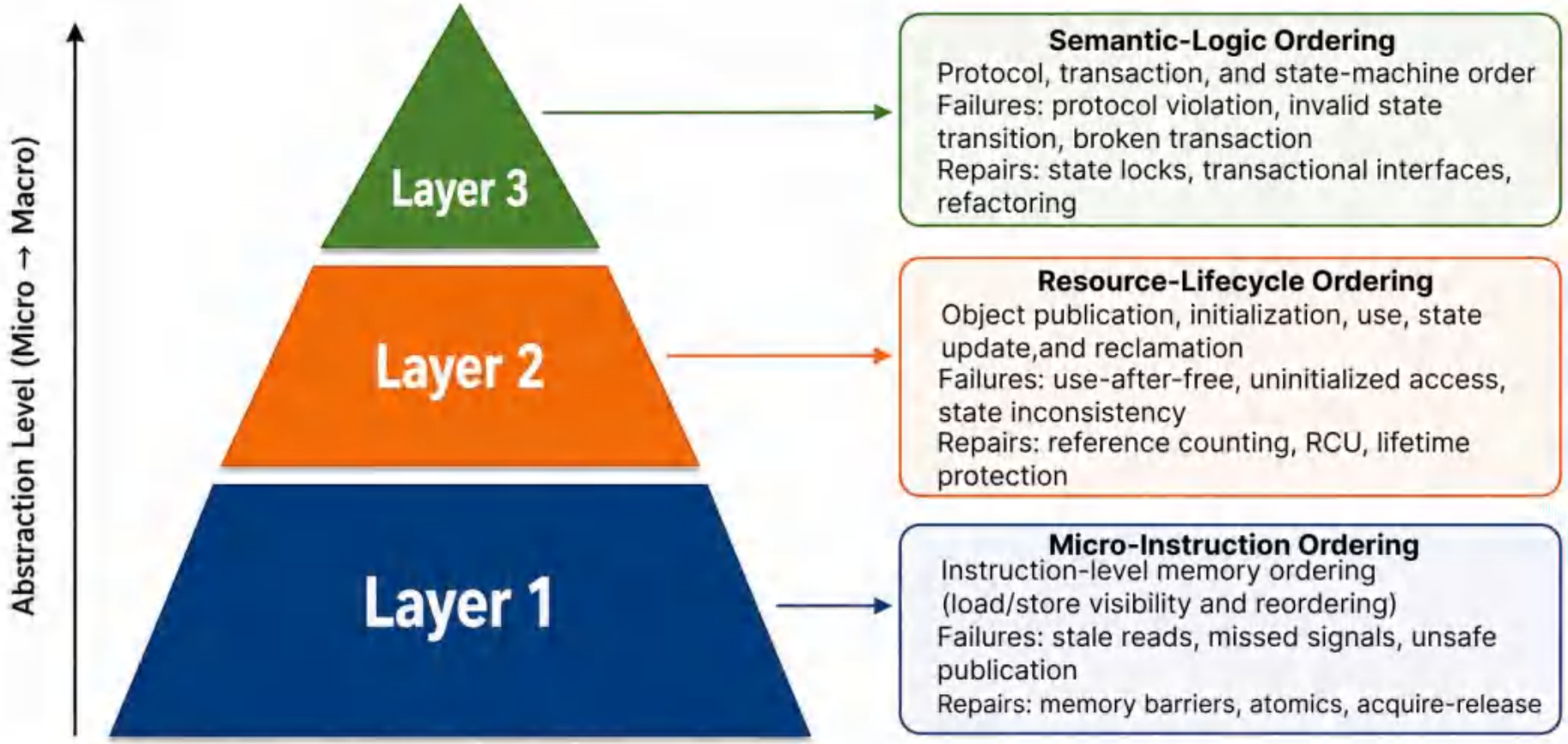
What if a Memory Barrier Is Not Enough ?



early publication → premature use → invalid / uninitialized access

The problem is not memory visibility.
The resource lifecycle is ordered incorrectly.

Multi-Granularity Ordering (MGO)



MGO classifies ordering bugs by the abstraction level of the violated ordering assumption—from memory operations, to resource lifecycle, to system semantics.

Layer 1: Micro-Instruction Ordering

```
// Thread 1: Successor
void mcs_lock(mcs_node_t *prev, mcs_node_t *me) {
    me->locked = 1;          // (A) Initialize node state
    prev->next = me;         // (B) Publish node pointer
}

// Thread 2: Predecessor
void mcs_unlock(mcs_node_t *me) {
    while (me->next == NULL) cpu_relax();
    if (me->next->locked == 1) {   // (C) Use
        published node
        // Logic proceeds assuming node is locked ...
    }
}
```

Figure 4: A Typical Unsafe Publication Pattern.

```
// Shared Variables
int pending_count = 0; // Work queue
int worker_idle = 1;   // Worker status token
// Thread 1: Sender
void signal_worker() {
    pending_count = 1;        // (A) Publish work
    if (worker_idle) {        // (B) Check worker state
        wake_up_worker();
    }
}
// Thread 2: Worker
void wait_for_job() {
    worker_idle = 0;                // (C) Mark worker busy
    if (pending_count == 0) { // (D) Check work
        sleep();
    }
}
```

Figure 5: A Typical Consistency Violation Pattern.

```
// Thread 1: Writer
void recycle_node(node_t *node) {
    node->key = NEW_KEY;          // (A) Update key
    node->value = NEW_VALUE;      // (B) Update value
}
// Thread 2: Reader
int lookup(node_t *node) {
    int v = node->value;          // (C) Read value
    if (node->key == OLD_KEY) { // (D) Validate key
        return v;
    }
}
```

Figure 6: A Typical Inconsistent View Pattern.

Layer 2: Resource-Lifecycle Ordering

```c
// Thread 1: Configuration
void configure_device() {
    nbd_genl_connect(..., .block_size = 4096);
    // Device configuration is still in progress
}
// Thread 2: User
void access_device() {
    int fd = open("/dev/nbd0", O_RDWR);
    ioctl(fd, NBD_DO_IT); // Use the device
}
```

Figure 7: A Typical Initialization-Use Race.

```c
// Thread 1: Initialization
void kvm_create_device() {
    struct kvm_device *dev = kzalloc(sizeof(*dev),
        ...);
    fd_install(fd, dev->file);  // (A) Publish fd
    dev->ops->init(dev);// (B) Continue initialization
}
// Thread 2: Teardown
void close_device(int fd) {
    close(fd);              // (C) Release the device
}
```

Figure 8: A Typical Teardown-Use Race.

```c
// Thread 1: State Mutator
void change_state(int fd) { // (A) Change object state
    connect(fd, &addr, sizeof(addr));
}
// Thread 2: Victim Operation
void trigger_bug(int fd) {  // (B) Use object state
    ioctl(fd, SIOCINQ, &val);
}
```

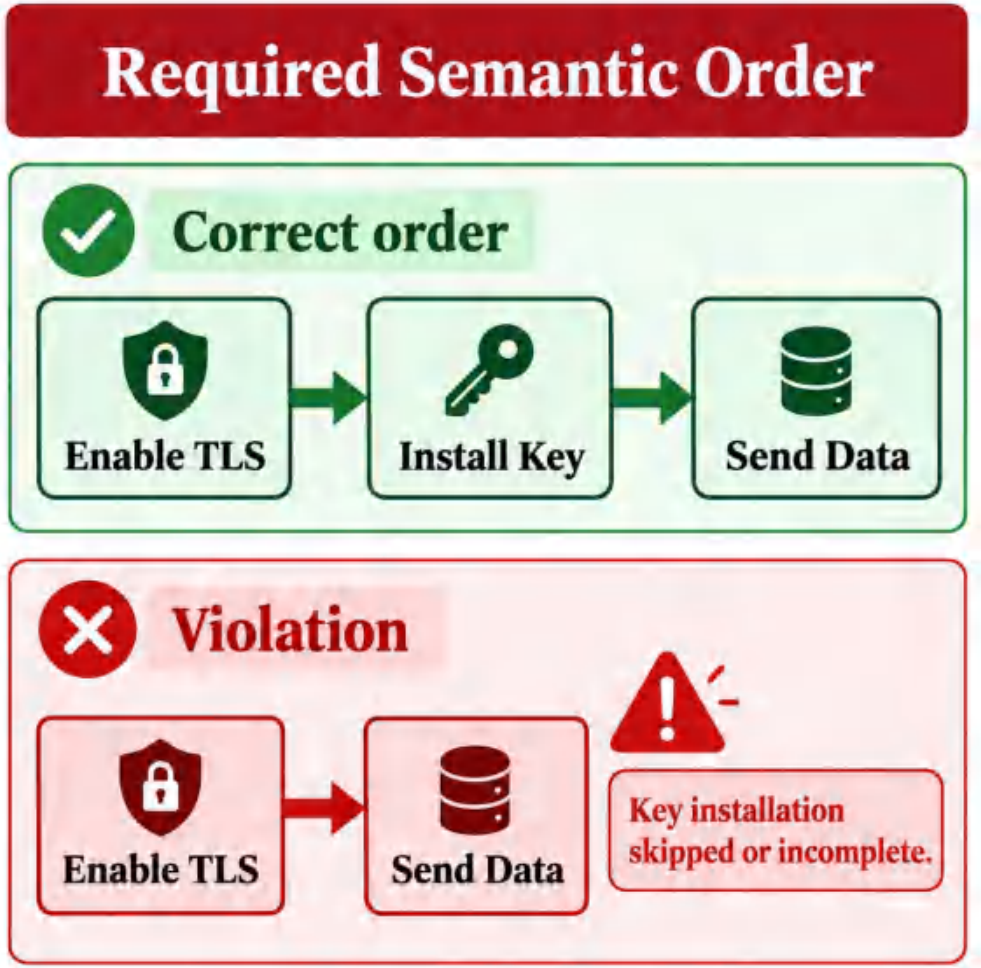
Figure 9: A Typical State Update Race.

Layer 3: Semantic-Logic Ordering

```c
// Thread 1: Protocol Upgrade
void upgrade_to_tls(int fd) {
    setsockopt(fd, SOL_TCP, TCP_ULP,
               "tls", sizeof("tls"));// (A) Enable TLS
    struct tls_crypto_info_aes_gcm_128 key_info = {
        ... };
    setsockopt(fd, SOL_TLS, TLS_TX,
       &key_info, sizeof(key_info));// (B) Install key
}

// Thread 2: Data Transmission
void send_data(int fd) {
    write(fd, "secret_data", 11);    // (C) Send data
}
```

Figure 10: A Typical Semantic Logic Violation.



Research Questions

RQ1 | What kinds of ordering-related concurrency bugs arise in real-world systems relevant to weak-memory and cross architecture reliability?

RQ2 | How do these bug classes differ in architectural exposure and repair strategies?

RQ3 | How well do existing concurrency analysis tools handle these bug classes in practice?

We study these questions through 58 real-world cases.

RQ1 — Distribution of Ordering Bugs

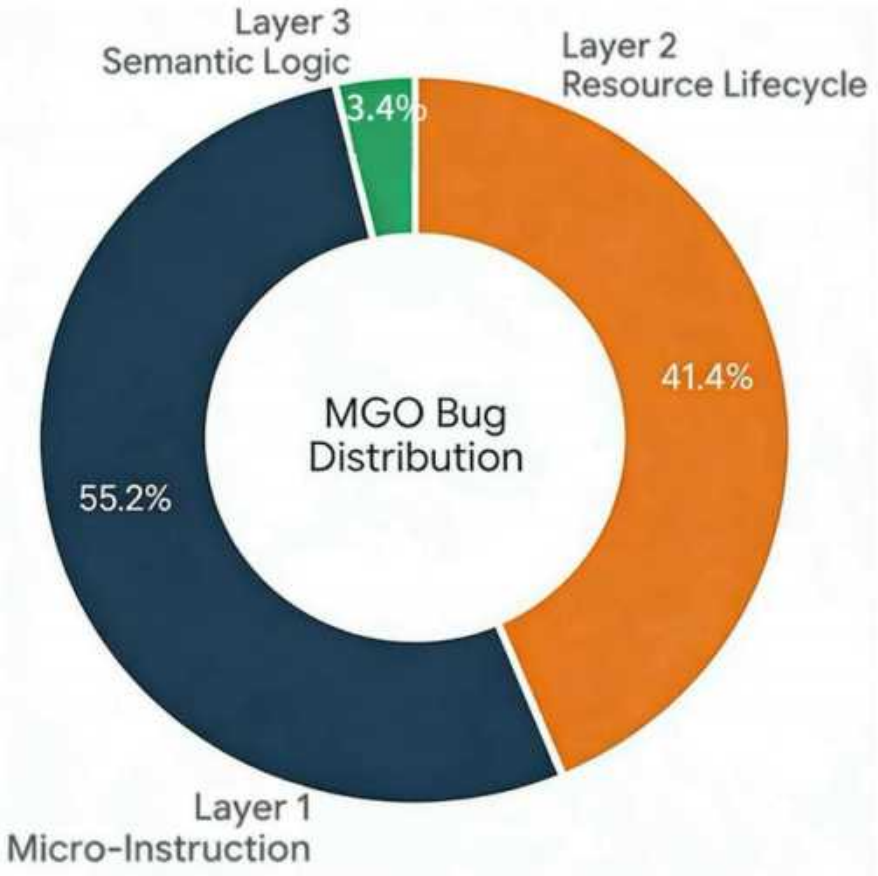


Figure 11: The Distribution of the 58 cases across MGO.

32/58	55.2%	Layer1
24/58	41.4%	Layer2
2/58	3.4%	Layer3

Answer to RQ1: Real-world ordering-related concurrency bugs span all three MGO layers; Layer 1 is the largest category, while Layer 2 also accounts for a substantial fraction of the cases.

RQ2 — Exposure & Repair

Table 2: Triggerability of the 58 cases across memory models and MGO layers.

MGO Layer & Pattern	Core Reordering Type	Count	More Observable Under	Less Observable Under
Layer 1: Unsafe Publication	Store-Store (S-S)	19	**WMM** (ARM64), **PSO**	**TSO** (x86, AMD64)
Layer 1: Consistency Violation	Store-Load (S-L)	5	**Across common models** (including TSO)	Not architecture-masked
Layer 1: Inconsistent View	Load-Load / Load-Store	8	**WMM** (ARM64, POWER)	**TSO**, **PSO**
Layer 2: Initialization-Use Races	Macro (Syscall/Interrupt)	14	**Higher-level interleavings, often WMM-amplified**	Often less exposed under TSO
Layer 2: Teardown-Use Races	Macro (Syscall/Interrupt)	6	**Higher-level interleavings, often WMM-amplified**	Often less exposed under TSO
Layer 2: State Update Races	Macro (Syscall/Interrupt)	4	**Higher-level interleavings, sometimes WMM-amplified**	Often less exposed under TSO
Layer 3: Semantic Logic	Semantic/event ordering	2	**Protocol/event interleavings, sometimes WMM-amplified**	Not architecture-masked

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.

Answer to RQ2: Different MGO layers exhibit different architectural exposure and require different repair strategies; repairs should match the abstraction level of the violated ordering assumption.

RQ3: Tool effectiveness

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%)

Key Observations

- **Layer 1 performs best.**
  GenMC achieves the highest detection rate: 78.1%.
- **Layer 2 remains challenging.**
  GenMC and Nidhugg detect only 16.7% of the cases.
- **Layer 3 is not detected.**
  None of the four evaluated tools detect the Layer 3 cases.

Answer to RQ3: Existing tools are most effective on Layer 1, their effectiveness drops sharply on Layer 2, and none of the evaluated tools detect the Layer 3 cases.

Conclusion

Key Findings

- Ordering is multi-granular. Real-world ordering bugs span instruction, resource-lifecycle, and semantic levels.

- Exposure and repair differ by layer. Different bug classes behave differently across architectures, and repairs should match the violated abstraction level.

- Current tools mainly cover Layer 1. Detection effectiveness drops sharply for lifecycle bugs and remains very limited for semantic-ordering bugs.

Implications

- Go beyond load/store reasoning. Ordering analysis should consider higher-level system behavior.

- Model resource lifecycles. Tools should track publication, initialization, use, update, and reclamation.

- Support semantic-order constraints. Protocol and state-machine ordering should be explicitly modeled.



西安電子科技大學
XIDIAN UNIVERSITY

Thanks for listening !