USENIX Security Symposium 2025

The Cost of Performance: Breaking ThreadX with Kernel Object Masquerading Attacks

Xinhui Shao [†], *Zhen Ling* [†], *Yue Zhang* [‡], *Huaiyu Yan* [†], *Yumeng Wei* [†], *Lan Luo* [§], *Zixia Liu* [§], *Junzhou Luo* [†], *Xinwen Fu* [*]

[†] *Southeast University*

[‡] *Drexel University*

[§] *Anhui University of Technology*

[*] *University of Massachusetts Lowell*

東南大學
SOUTHEAST UNIVERSITY

安徽工业大学
ANHUI UNIVERSITY OF TECHNOLOGY

Real-time Operating System (RTOS) Security Protections

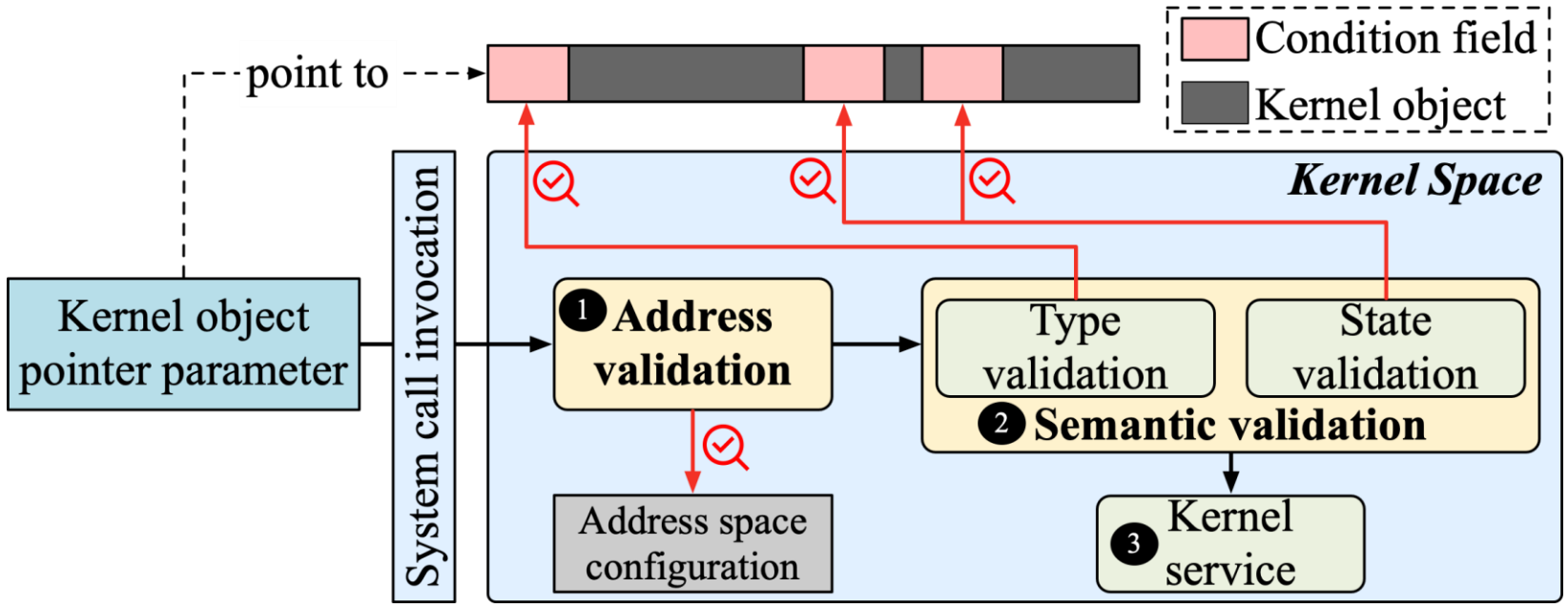
Among state-of-the-art RTOSs, only **Zephyr OS** and **ThreadX** provide comprehensive protection.

	Hardware		**Software**
	Privilege Separation	Memory Access Control	**Parameter Sanitization**
FreeRTOS	✓	✓	✗
LiteOS-M	✗	✓	✗
Mbed OS	✗	✓	✗
ThreadX	✓	✓	✓
Zephyr OS	✓	✓	✓

Parameter Sanitization in ThreadX

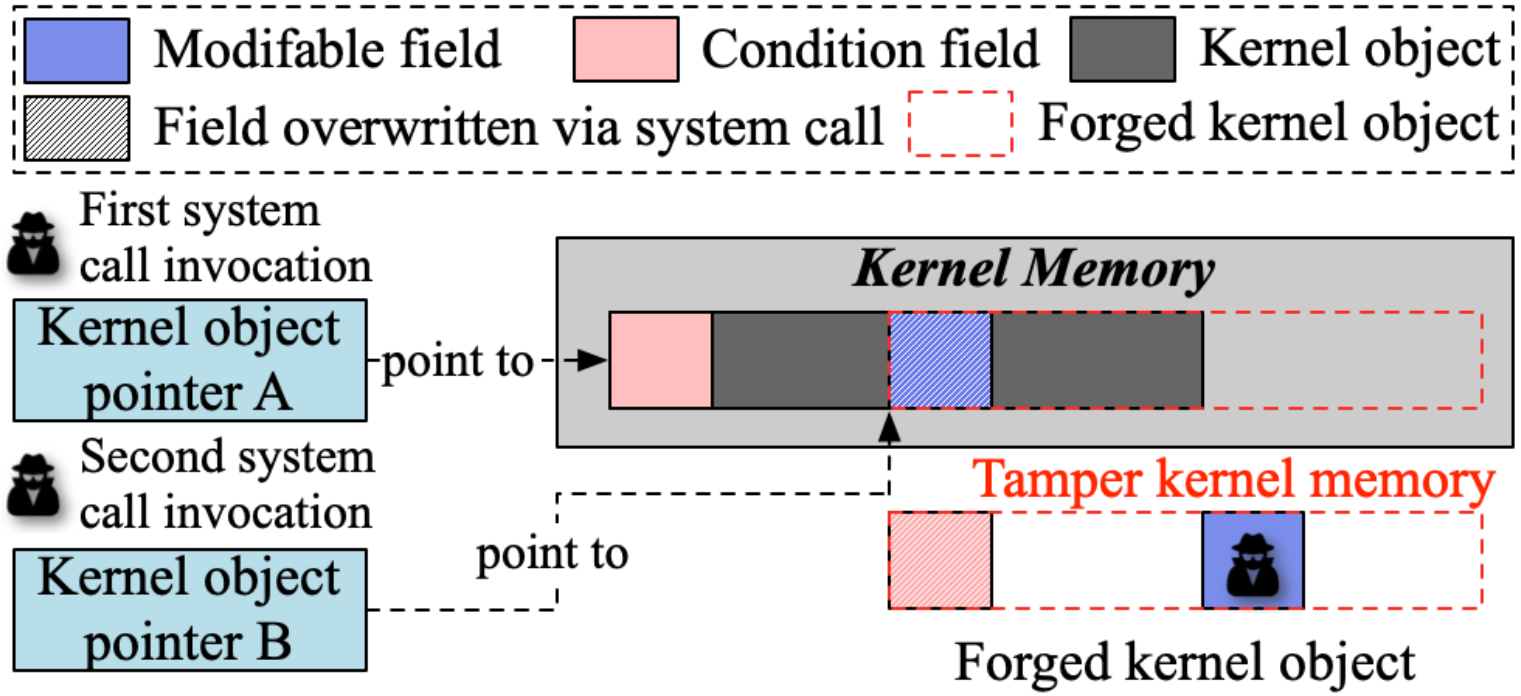
Unlike Zephyr, ThreadX applies performance optimization to parameter sanitization.

- Validate kernel pointer address ranges
- Checks fields (denoted as condition fields) directly on the pointed object



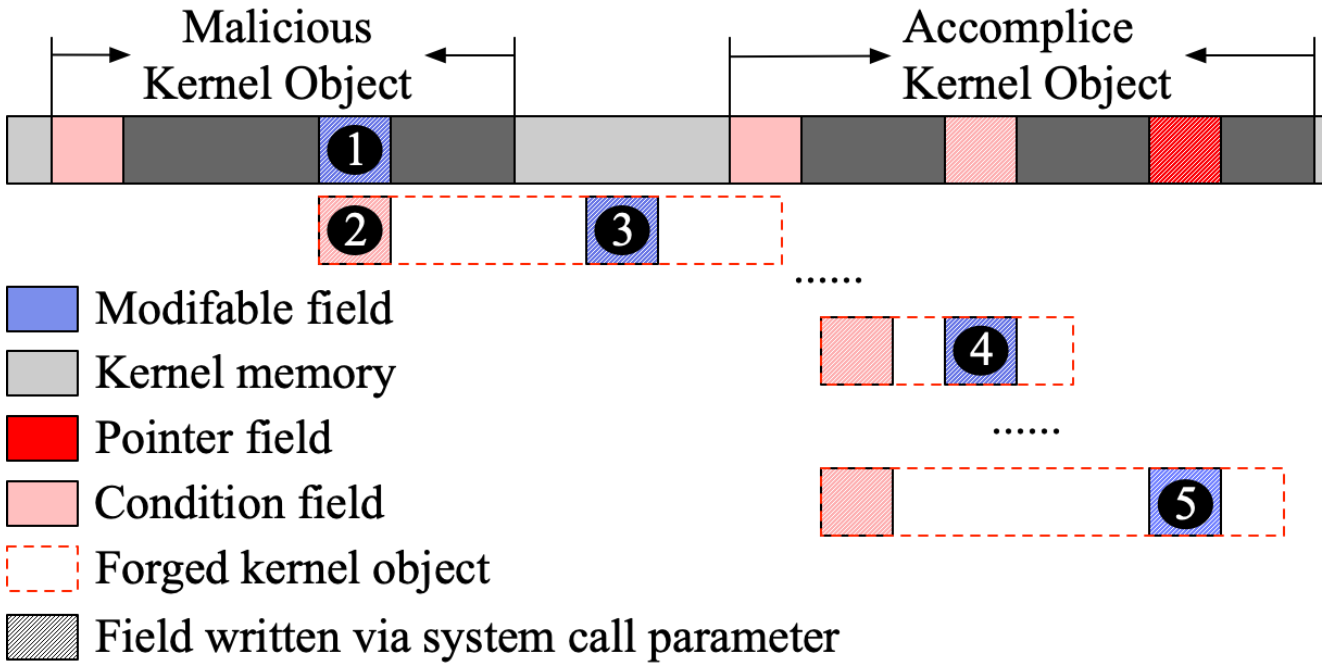
Vulnerability of ThreadX

Key Insight: Malicious user threads can **mimic fields for expected semantic validation** through modifying specific kernel object fields.



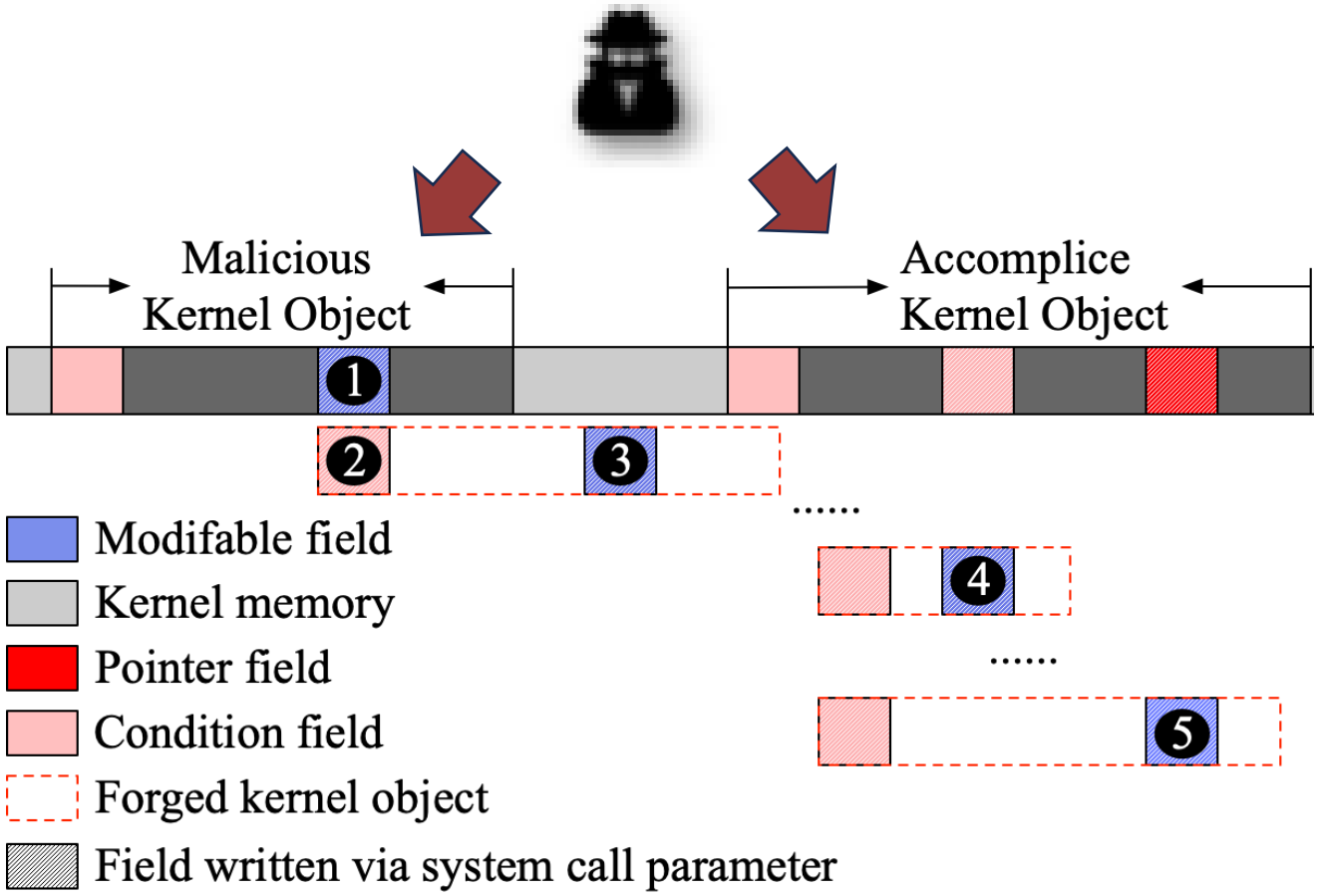
KOM Attack

We design **Kernel Object Masquerading (KOM)** attack–a confused deputy attack–through a **carefully selected sequence of system calls.**



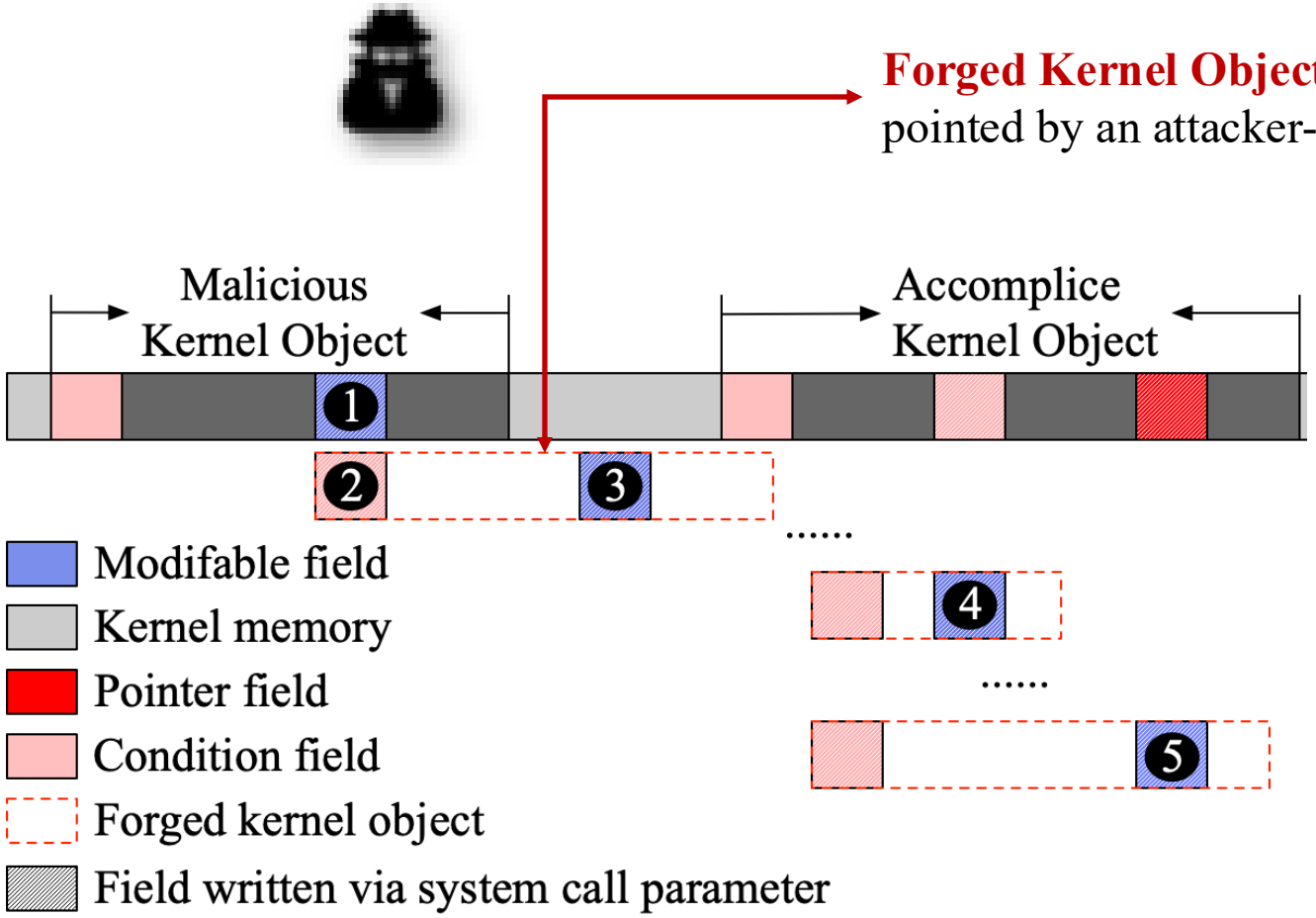
Enabling an attacker to achieve **privilege escalation** and **arbitrary memory access.**

KOM Attack



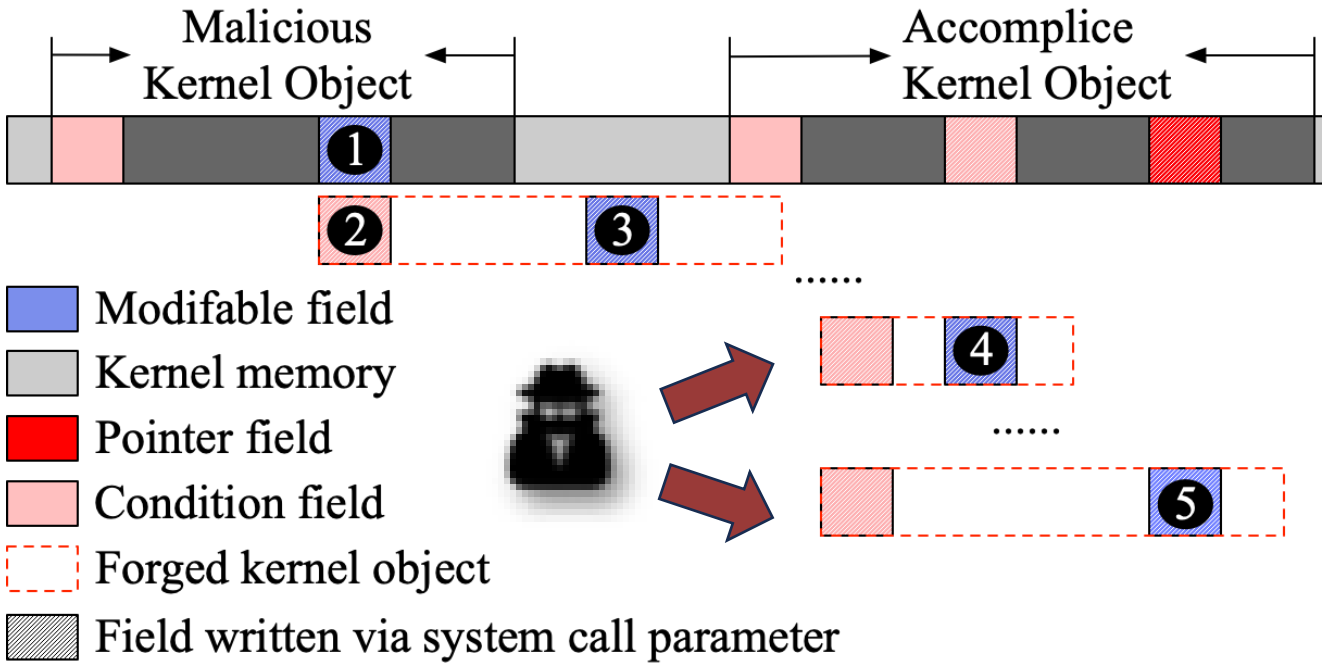
1. Creating **malicious kernel object** and **accomplice object**.

KOM Attack



2. Creating initial **forged kernel object**.

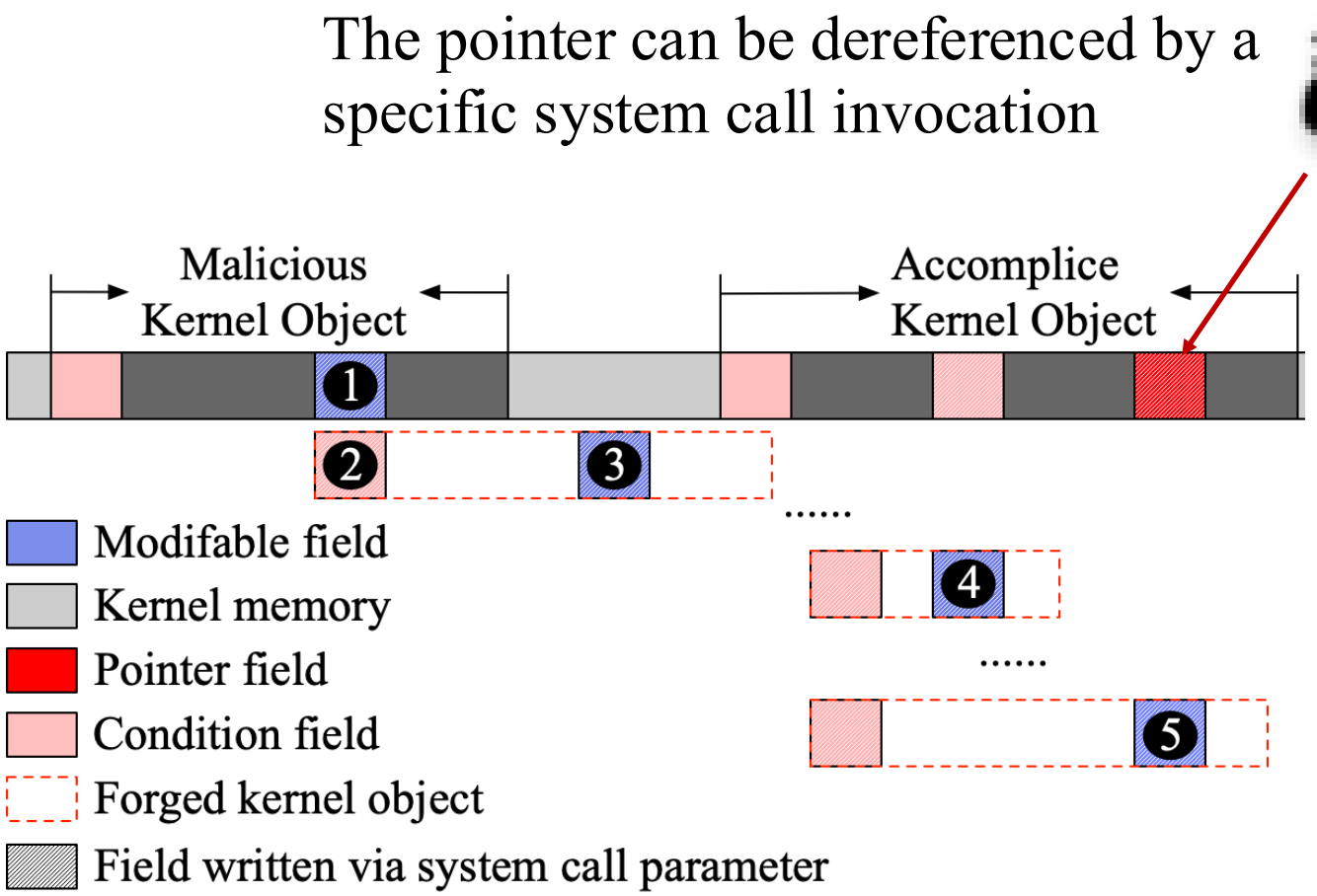
KOM Attack



3. Creating sequences of forged kernel objects to overwrite fields of the accomplice kernel object.

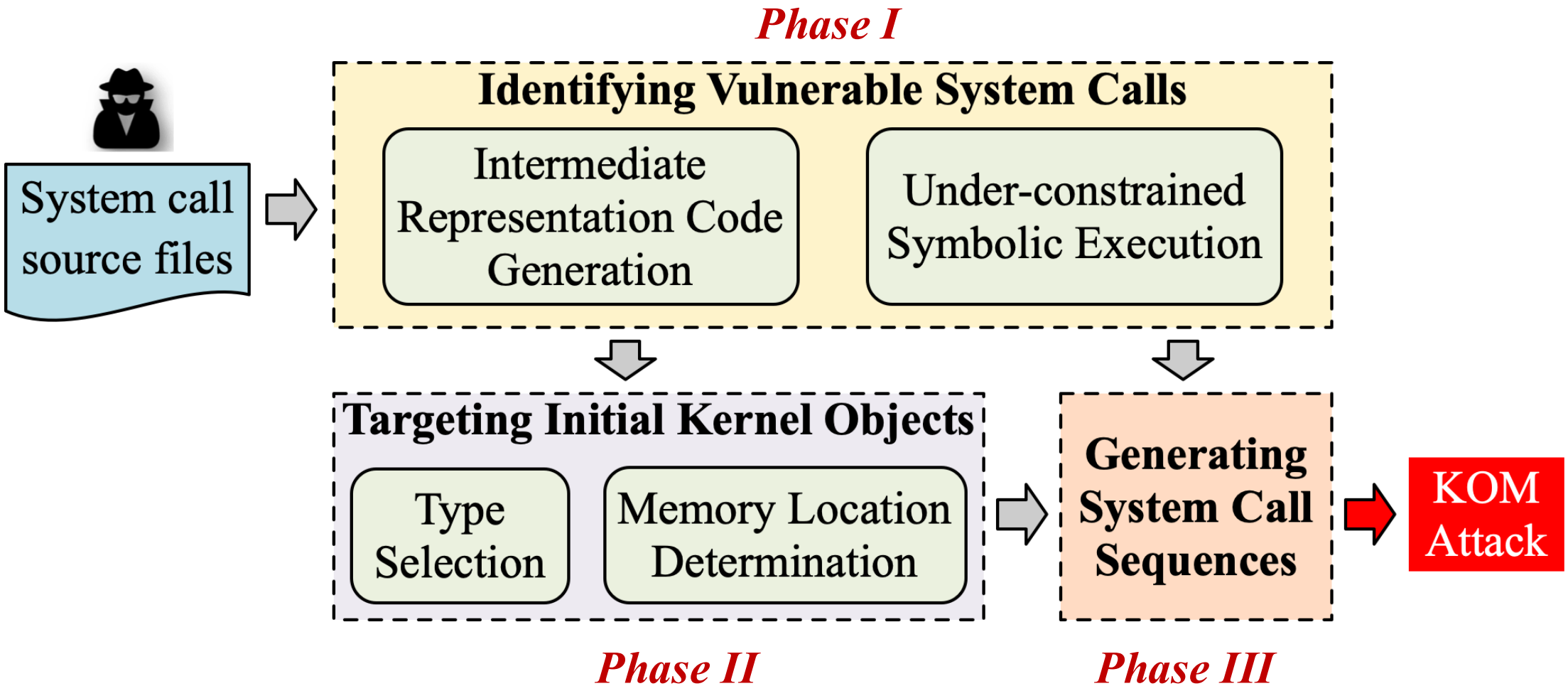
KOM Attack

The pointer can be dereferenced by a specific system call invocation

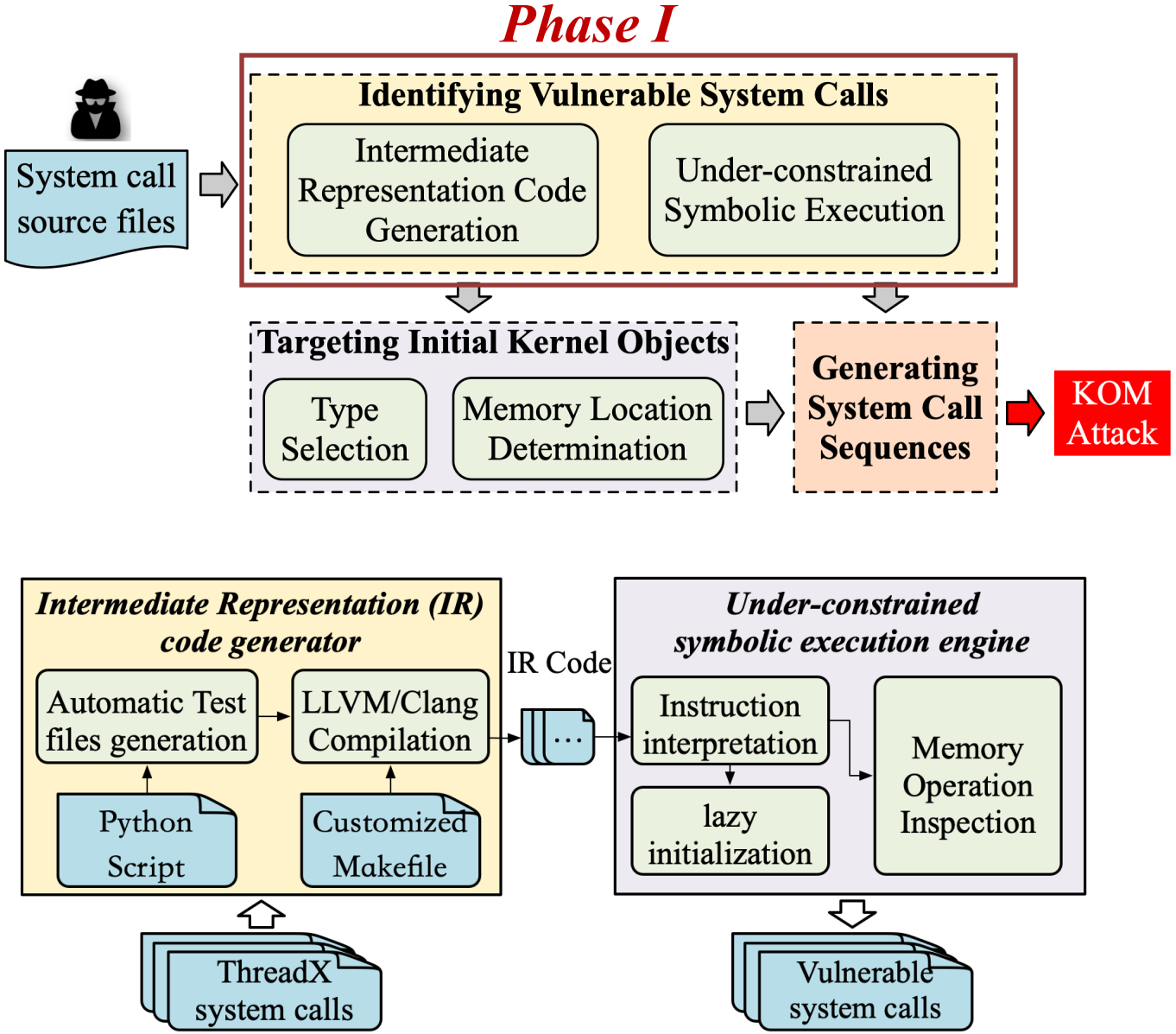


4. Dereferencing pointer of accomplice kernel object.

Automated Framework for KOM Attacks

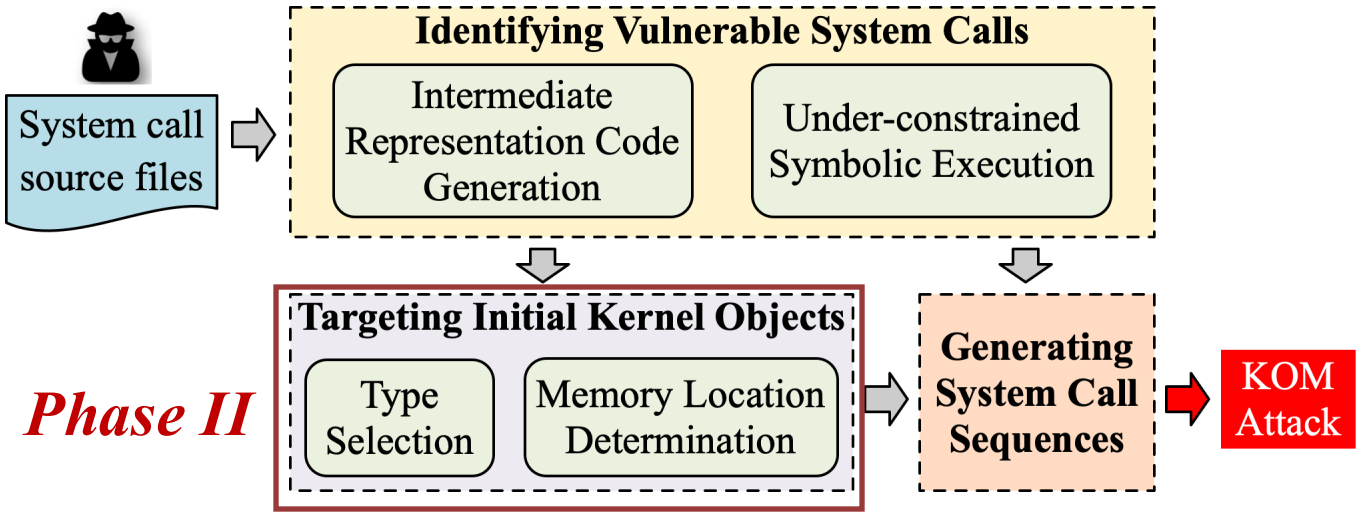


Automated Framework for KOM Attacks



- **IR Construction:** Convert ThreadX kernel into LLVM IR to enable symbolic execution.
- **System call Execution:** Symbolically run execution on all system calls.
- **Vulnerability Identification:** Record system calls that can modify kernel-object fields, marking them as candidates for KOM exploitation.

Automated Framework for KOM Attacks

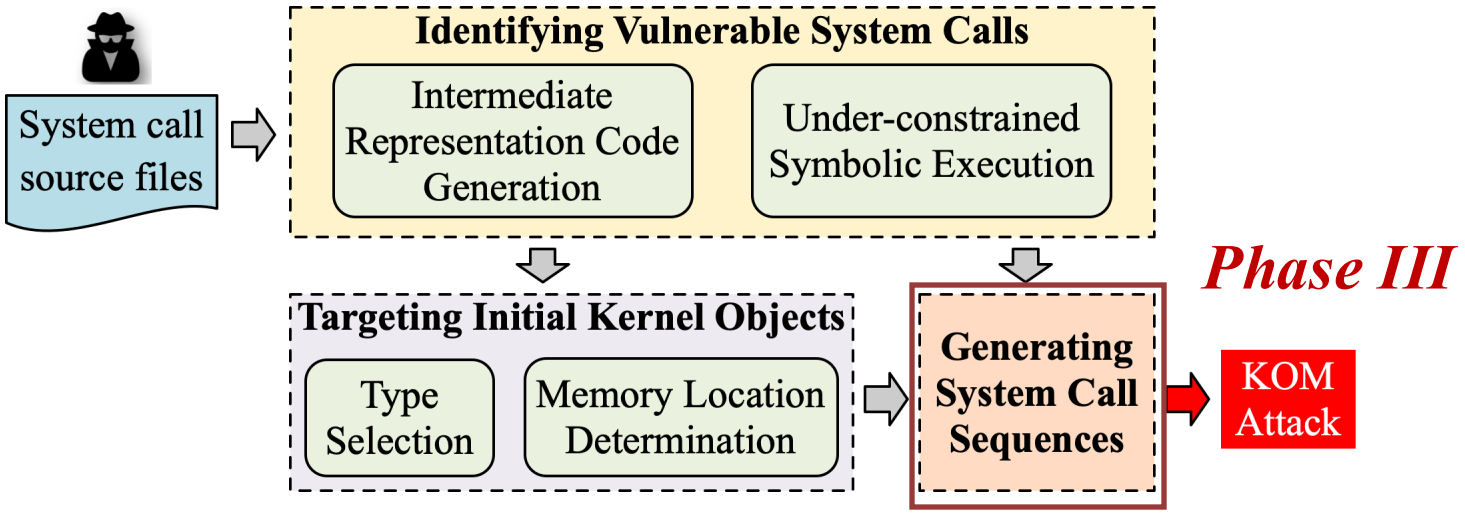


Select **candidate malicious and accomplice objects** by type and memory location.

Type Selection: choose kernel-object types whose fields can be modified and whose pointer dereferences are **fully controllable**

Memory Location Determination: create objects to read allocator returns, or reverse-engineer memory to find object pointers.

Automated Framework for KOM Attacks



We use DFS over vulnerable system calls to find feasible sequences.

- **System call Emulation:** iterate vulnerable system calls.
- **Field Emulation:** enumerate modifiable fields for each system call.

Experimental Results

RQ1 What system calls can be used to perform KOM attacks

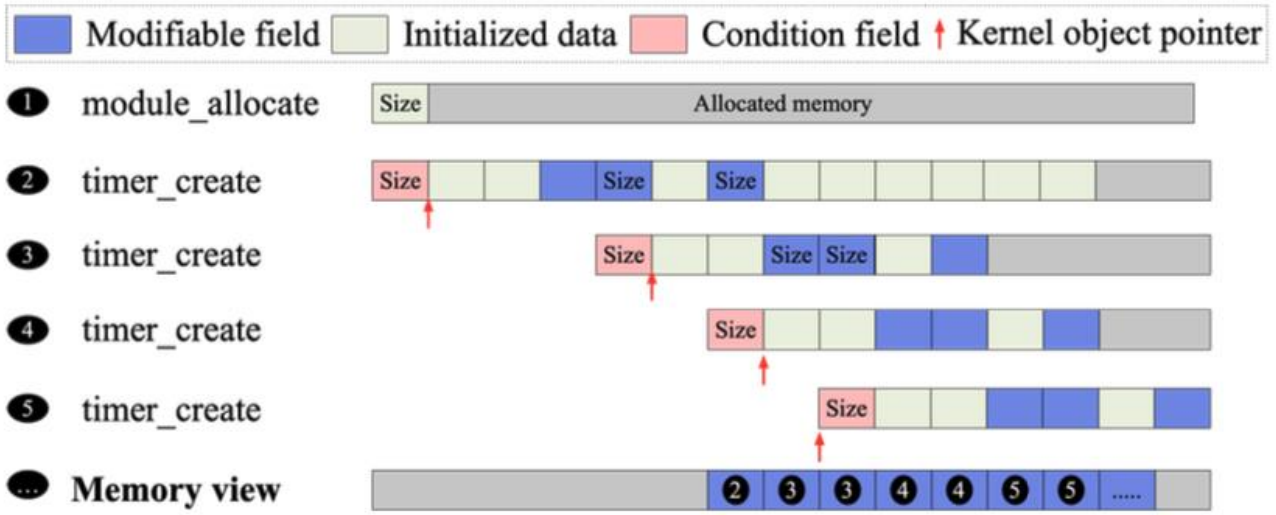
Identification Result						Performance Metrics		
System Call	$\#M_1$	$\#M_2$	$\#M_3$	$\#C_{max}/C_{min}$	Status	Time	#Path	#Ins
block_allocate	3	0	0	2/2	N	23	236	14882
block_pool_create	4	4	2	1/1	N	9	88	5224
block_pool_delete	0	0	0	N/A	N	47	567	26626
block_pool_info_get	0	0	0	N/A	N	12	236	8384
block_pool_prioritize	0	0	0	N/A	N	6	57	3145
block_release	0	0	0	N/A	N	1	8	296
byte_allocate	0	0	0	N/A	N	9	60	4532
byte_pool_create	6	6	2	1/1	N	5	52	3215
byte_pool_delete	0	0	0	N/A	N	49	567	26626
byte_pool_info_get	0	0	0	N/A	N	11	236	8384
byte_pool_prioritize	0	0	0	N/A	N	6	57	3145
byte_release	0	0	0	N/A	N	34	209	11836
event_flags_create	1	1	0	1/1	N	5	34	2546
event_flags_delete	0	0	0	N/A	N	49	567	26626
event_flags_get	3	1	1	6/4	N	124	957	68965
event_flags_info_get	0	0	0	N/A	N	7	134	4697
event_flags_set	3	0	0	27/1	N	403	6679	239501
event_flags_set_notify	2	0	0	1/1	N	2	14	729
mutex_create	1	1	0	1/1	N	8	58	4736
mutex_delete	3	0	0	4/3	N	17,987	158975	9276099
mutex_get	3	0	0	6/3	N	540	5125	326769
mutex_info_get	0	0	0	N/A	N	11	236	8384
mutex_prioritize	0	0	0	N/A	N	6	57	3145
mutex_put	5	0	0	7/3	N	33,648	289827	16714889
queue_create	8	8	2	1/1	N	7	64	4426
queue_delete	0	0	0	N/A	N	48	567	26584
queue_flush	3	0	0	2/2	N	45	570	23542
queue_front_send	3	0	0	4/3	N	473	11190	341484
queue_info_get	0	0	0	N/A	N	13	236	8384
queue_prioritize	0	0	0	N/A	N	7	57	3145
queue_receive	5	0	0	7/3	N	484	5379	294694
queue_send	5	0	0	5/3	N	500	11295	344838
queue_send_notify	2	0	0	1/1	N	2	14	729
semaphore_ceiling_put	3	0	0	3/3	N	62	1493	44416
semaphore_create	2	2	1	1/1	N	4	34	2720
semaphore_delete	0	0	0	N/A	N	48	567	26584
semaphore_get	2	0	0	3/2	N	21	214	13489
semaphore_info_get	0	0	0	N/A	N	6	134	4667
semaphore_prioritize	0	0	0	N/A	N	6	57	3145
semaphore_put	2	0	0	3/3	N	58	1487	42228
semaphore_put_notify	2	0	0	1/1	N	2	14	729
thread_create	15	12	7	1/1	N	461	4018	279885
thread_delete	0	0	0	N/A	N	3	26	2439
thread_entry_exit_notify	0	0	0	N/A	N	2	17	797
thread_info_get	0	0	0	N/A	N	41	824	30026
thread_preemption_change	2	2	2	5/3	N	4	80	3273
thread_priority_change	6	4	0	7/3	A	N/A	N/A	N/A
thread_relinquish	0	0	0	N/A	N	1	10	386
thread_reset	1	0	0	2/2	N	1	7	386
thread_resume	0	0	0	N/A	N	32	344	22128
thread_suspend	0	0	0	N/A	N	5	62	3597
thread_terminate	0	0	0	N/A	N	58	461	32214
thread_time_slice_change	2	2	2	1/1	N	1	23	840
thread_wait_abort	0	0	0	N/A	N	46	512	29787
timer_activate	1	0	0	6/6	N	2	28	1365
timer_change	2	2	2	2/2	N	1	20	806
timer_create	7	5	3	2/1	N	44	425	27154
timer_deactivate	1	0	0	6/4	N	8	107	5103
timer_delete	0	0	0	N/A	N	5	26	2949
timer_info_get	0	0	0	N/A	N	46	902	33275

- **31 / 60 system calls** are capable of modifying kernel-object fields
- **Two classes:** 17 system calls set fixed values; 13 allow arbitrary field modifications
- **Security impact:** both classes are exploitable — enabling data corruption, kernel crashes, or other KOM effects

ThreadX exposing a broad attack surface for KOM that could result in data corruption or kernel crashes.

Experimental Results

RQ2 Can our attack be effective in various attack environments?



Realize arbitrary write by chaining vulnerable system calls on timer kernel object

System call chain: module_allocate → timer_create (create malicious timer) → write size into two modifiable fields → call timer_create twice to instantiate forged timers at higher addresses → repeat to advance writes upward.

Experimental Results

RQ3 How is the exploitability of each type of kernel object in KOM attack?

Type	Malicious Kernel Object	$\#M_3$	Accomplice Kernel Object	Fully Controllable Pointer Dereferencing
Block	✔	2	✔	✔
Byte	✔	2	✔	✔
Event	✔	1	✔	✔
Mutex	✗	0	✔	✔
Queue	✔	2	✔	✔
Semaphore	✔	1	✔	✔
Thread	✔	7	✔	✔
Timer	✔	3	✔	✔

```
1 UINT  queue_send(TX_QUEUE *queue_ptr, VOID *source_ptr, ULONG wait_option)
2 {
3     // We omitted code of address validation for clarity
4     suspended_count =  queue_ptr -> tx_queue_suspended_count;
5     if (queue_ptr -> tx_queue_available_storage != TX_NO_MESSAGES)
6     {                                                                  Semantic
7         if (suspended_count == TX_NO_SUSPENSIONS)                      Validation
8         {
9             // Code omitted
10            source =       TX_VOID_TO_ULONG_POINTER_CONVERT(source_ptr);
11            destination =  queue_ptr -> tx_queue_write;
12            size =         queue_ptr -> tx_queue_message_size;  Privleged Pointer
13            TX_QUEUE_MESSAGE_COPY(source, destination, size)    Dereferencing
14        }
15    // ...
16    }
17    return(status);
18}
```

Example code of fully controllable pointer dereferencing

- **Malicious types:** All listed object types except **Mutex** are usable
- **Accomplice types:** Any object type can act as an accomplice and enable **fully controllable pointer dereferencing**

Many kernel-object types can serve as malicious or accomplice objects, enabling scalable forged-object creation

Experimental Results

RQ3 How efficient is our symbolic execution?

Host Platform: Ubuntu 20.04 server, Intel Xeon E5-2620 v2, 64 GB RAM.

Engine: KLEE-based symbolic-execution implementation.

- **Average Analysis Time**: Most system calls were analyzed within **one minute**
- **Longest Analysis Time**: The mutex_put system call required almost **9 hours** to explore ~289,000 paths
- **Unfinished Case**: One system call did not terminate normally, because of a constraint-solving issue

Our RTOS-optimized symbolic execution is both efficient and effective, achieving high coverage within practical resource budgets.

Experimental Results

RQ5 What are the implications of our KOM attacks

KOM Attack on Different Platforms					
Hardware Platform			Attacks		
Board	Vendor	Core	*MPU	*Read	*Write
NUCLEO-U575ZI-Q	STM	cortex-m33	✔	✔	✔
b-l475e-iot01a	STM	cortex-m4	✔	✔	✔
olimex-stm32-h405	Olimex	cortex-m4	✔	✔	✔
netduinoplus2	Wilderness Labs	cortex-m4	✔	✔	✔

ThreadX-powed Platforms: STM development board, OLIMEX board, and Wilderness Labs board (in addition to QEMU).

Implemented Attacks: Successfully mounted KOM primitives that produced arbitrary memory access, and were capable of disabling the MPU.

Our experiments confirm that KOM is practical on real hardware, undermining core security guarantees and posing serious risks to RTOS systems.

Thanks you

Contact: xinhuishao@seu.edu.cn