THE HONG KONG POLYTECHNIC UNIVERSITY
香港理工大學

IsolatOS:

Detecting Double Fetch Bugs in COTS RTOS by Re-enabling Kernel Isolation

Cao et al.

Network and Distributed System Security Symposium 2026

February 2026

Opening Minds • Shaping the Future
啟迪思維 • 成就未來

Presentation Outline

Contents

Motivation

- Real-time Operating Systems (RTOS) dominate cyber-physical systems
  - IoT, aerospace, automotive, power plants
  - 2.2+ billion embedded devices
  - QNX in 215+ million vehicles (2022)
- Double-fetch vulnerabilities pose critical security risks
  - Kernel reads user-space memory multiple times
  - Data inconsistency between fetches
  - Can lead to privilege escalation, information leaks
- Existing detection methods fail for COTS RTOS
  - Static analysis requires source code
  - Dynamic methods have high overhead (30-80×)
  - Cannot handle preemption accurately

Research Question

Can we detect double-fetch bugs in COTS RTOS both quickly and accurately?

Key Insight

Hardware-based kernel isolation features (SMAP/PAN) can efficiently identify cross-boundary memory accesses with minimal overhead

- **Quickly**: 79.3× faster than emulation-based approaches
- **Accurately**: Lower false positive rates through lifecycle tracking

Contents

Double-Fetch Vulnerability

Vulnerability Pattern

```
// Kernel Space
ker_a1 = p->a;  // First fetch
if (ker_a1 > MAX)
    return ERROR;

// Time window for race condition

ker_a2 = p->a;  // Second fetch
process(ker_a2); // Use potentially
                 // modified value

```

Security Impact:

- 46.2% lead to privilege escalation
- 39.5% cause information leaks
- 42.9% enable security bypasses
- 11% result in denial-of-service

Real-world Example:

- PWN2OWN Tesla: $100,000 bounty
- VxWorks on Boeing 787

RTOS Characteristics

Preemption in RTOS:

- Priority-based scheduling
- 256 distinct priority levels
- Kernel fully preemptable
- Multiple threads access same memory

Memory Access Patterns:

- Direct pointer dereferencing
- No `copy_from_user()` wrapper
- Shared memory for IPC
- Limited synchronization (real-time constraints)

Challenge

Legitimate concurrent accesses appear identical to double-fetch bugs when using time-window detection

Hardware Kernel Isolation

Architecture	Support	Mechanism
Intel x86-64 (Broadwell+)	Hardware	SMAP (CR4 bit 21)
ARM v8.1+	Hardware	PAN Register
ARM v7/v8.0	Software	Page Domain/TTBR
PowerPC	Hardware	KUAP
MIPS	None	N/A

Key Observation

Modern CPUs provide hardware isolation between user/kernel memory, but COTS RTOS disable these features for performance

Contents

Challenge 1: Cross-boundary Identification

General Purpose OS:

```
// Explicit transfer function
if (copy_from_user(&kdata,
    user_ptr, sizeof(kdata)))
    return -EFAULT;
// Detectable by tools

```

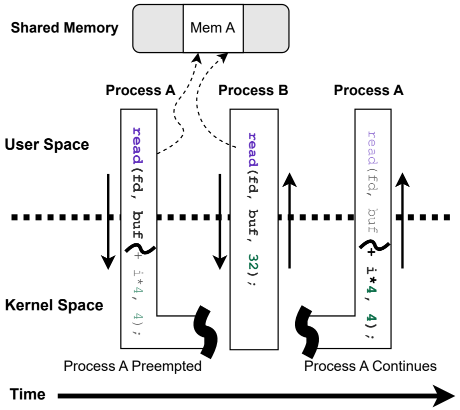
COTS RTOS:

```
// Direct dereference
int size = user_ptr->size;
// Invisible to API detection

```

- No source code available (proprietary)
- User/kernel pointers identical at binary level
- Emulation overhead: 30-80× slower

Challenge 2: Preemption & Multi-CPU



Preemption Complexity:

- Multiple threads access `memA` legitimately
- No explicit preemption signals in kernel
- Binary relocation: `0xc45e1` → dynamic offset

False Positive Generation:

$$\mathrm{FP} = \frac{\text{Concurrent Access}}{\text{Time Window}}$$

Technical Barrier:

- Must distinguish: `thread_A` vs `thread_B`
- Kernel binary modification prohibited
- Preemption occurs within system calls

Challenge 3: Fault Recovery

The Recovery Problem

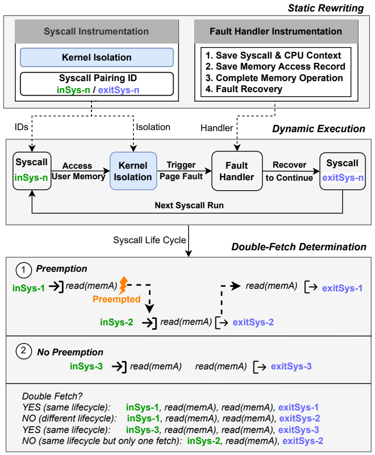
1. Page fault triggers on cross-boundary access
2. Exception handler runs in isolated memory region
3. Must complete faulting instruction
4. Must maintain control for subsequent detection
5. Cannot simply disable isolation and return

Paradox

If we disable isolation to execute the instruction, we lose control. If we keep isolation enabled, the instruction cannot complete.

Contents

System Overview



Static Rewriting
Syscall Instrumentation
Kernel Isolation
Syscall Pairing ID
inSys-n / exitSys-n
Fault Handler Instrumentation
1. Save Syscall & CPU Context
2. Save Memory Access Record
3. Complete Memory Operation
4. Fault Recovery
IDs
Isolation
Handler
Dynamic Execution
Syscall inSys-n
Access User Memory
Kernel Isolation
Trigger Page Fault
Fault Handler
Recover to Continue
Syscall exitSys-n
Next Syscall Run
Syscall Life Cycle
Double-Fetch Determination
1 Preemption
inSys-1 read(memA)
Preempted
read(memA) exitSys-1
inSys-2 read(memA) exitSys-2
2 No Preemption
inSys-3 read(memA) read(memA) exitSys-3
Double Fetch?
YES (same lifecycle): inSys-1, read(memA), read(memA), exitSys-1
NO (different lifecycle): inSys-1, read(memA), read(memA), exitSys-2
YES (same lifecycle): inSys-3, read(memA), read(memA), exitSys-3
NO (same lifecycle but only one fetch): inSys-2, read(memA), exitSys-2

Key Components:

1. **Static kernel entry identification**: Analyze `WRMSR/MSR` instructions to locate syscall handlers; instrument with isolation enable code
2. **Dynamic system call boundary tracking**: Assign unique pairing IDs (`inSys-n`, `exitSys-n`) to distinguish syscall lifecycles during preemption
3. **Page fault handling and recovery**: Capture cross-boundary accesses via hardware exceptions; execute faulting instruction with temporary isolation disable
4. **Double-fetch pattern analysis**: Correlate memory accesses within same syscall ID; $\mathcal{DF} = \{(addr, id) : count(addr, id) \geq 2\}$

Static Instrumentation

Kernel Entry Discovery:

```
MOV ECX, 0xc0000082  ; MSR addr
MOV RAX, syscall_trap_0
ADD RAX, pcpu * 0x20
WRMSR                ; Set entry

```

Analyze WRMSR instructions to find dynamic kernel entries

Instrumentation Actions:

1. Enable kernel isolation
   - x86-64: Set CR4 bit 21 (SMAP)
   - ARM: Set PAN register
2. Assign system call pairing ID
   - Unique ID per syscall
   - Track lifecycle (inSys-n, exitSys-n)

Page Fault Handler Design

Fault Context Recording

- Target address of memory access
- Instruction pointer that caused fault
- Current system call pairing ID
- CPU identifier for multi-core systems

Instruction Recovery Mechanism

1. Temporarily disable isolation
2. Execute faulting instruction
3. Re-enable isolation immediately
4. Continue to next instruction

Implementation: JTAG-GDB for architecture-independent recovery

Double-Fetch Detection Algorithm

```
Input: AccessLog with (addr, IP, syscall_id)
Output: Set of double-fetch bugs V
Group AccessLog by syscall_id;
foreach syscall group G do
    foreach unique addr in G do
        count ← accesses to addr;
        if count ≥ 2 then
            Add (addr, syscall_id) to V;
        end
    end
end
return V;
```

Pattern Detection:

Double-fetch detected: {`inSys-1, read(memA), read(memA), exitSys-1`}

Not a double-fetch: {`inSys-1, read(memA), inSys-2, read(memA), exitSys-2, exitSys-1`}

Lifecycle tracking eliminates false positives from preemption

Contents

System Adaptation

RTOS	**Architecture**	**Integration Method**
QNX 6.6/7.0/8.0	Microkernel	Board Support Package (BSP)
VxWorks	Monolithic	Kernel Driver
seL4	Microkernel	Source Code Modification

Hardware Platforms:

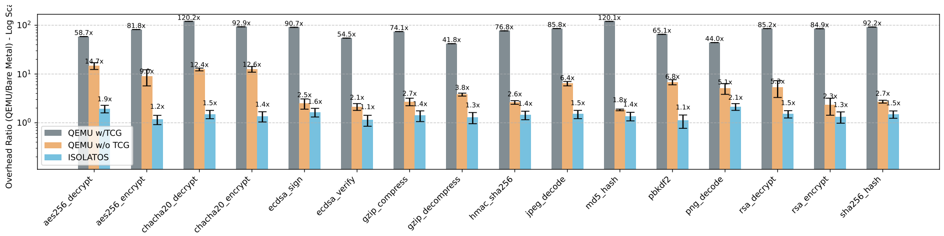
- Intel i7-12700 (x86-64 with SMAP/SMEP)
- Raspberry Pi 5 (ARM with PAN)

Development Effort:

- QNX: 8 hours for version migration
- VxWorks: 2 workdays for driver implementation
- seL4: 2 workdays for source integration

Contents

Performance Overhead



- **IsolatOS**: 45.7% average overhead
- **QEMU-TCG** (Bochspwn-like): 79.3× overhead
- **Improvement**: 173× faster than emulation

Precision Analysis

False Positive Rates:

- QEMU-TCG: 87.7% FP rate
- IsolatOS: Near-zero FP rate

False Positive Categories:

- Temporal (80.6%)
- Preemption (18.7%)
- Uninitialized memory (0.7%)

Test	**QEMU-TCG** (TP/Total)	**IsolatOS** (TP/Total)
MD5	5/12	9/9
SHA256	2/18	4/4
RSA	10/23	12/12
ECDSA	9/17	7/7
MsgSend	8/23	9/9

Real-world Vulnerabilities

RTOS	**Vulnerabilities**	**CVEs**	**Severity**
QNX 6.6/7.0	37	37	6 LPE, 31 DoS
QNX 8.0	2	Pending	Under analysis
VxWorks	3	2	Info leak, DoS
seL4	1	N/A	No impact
Total	**43**	**39**	

Case Study: 19-year-old QNX Vulnerability

- Local privilege escalation via arbitrary write
- Affects production vehicles from major manufacturers
- 76% exploitation success rate
- Critical time window: ~125 CPU cycles

Contents

Key Contributions

1. **Novel Approach:** First to leverage hardware kernel isolation (SMAP/PAN) for double-fetch detection
2. **Efficient Implementation:** 79.3× faster than emulation-based approaches with near-zero false positives
3. **Real Impact:** 43 vulnerabilities discovered (41 previously unknown, 39 CVEs assigned)
4. **Cross-platform:** Successfully applied to QNX, VxWorks, and seL4

Future Work

- **Extended Scope:**
  - Apply to TEE environments
  - Extend to hypervisor security
  - Support additional architectures
- **Automated Mitigation:**
  - Integrate with SafeFetch-like approaches
  - Automatic patch generation
  - Runtime protection mechanisms
- **Compiler-Level Detection:**
  - Detect compiler-introduced double-fetches
  - Static analysis integration

Thank You!

Questions?

IsolatOS: Detecting Double Fetch Bugs in COTS RTOS
by Re-enabling Kernel Isolation

NDSS 2026