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

Yingjie Cao*†, Xiaogang Zhu‡, Dean Sullivan⊥, Haowei Yang, Lei Xue*§, Xian Li¶,
Chenxiong Qian‖, Minrui Yan¶, Xiapu Luo†
*Sun Yat-sen University, †The Hong Kong Polytechnic University, ‖University of Hong Kong, China
‡Adelaide University, ¶Swinburne University of Technology, Australia
⊥University of New Hampshire, US

***Abstract*—Real-time operating systems (RTOS) often expose double-fetch vulnerabilities when the kernel reads the same user-space memory location multiple times without ensuring consistency between fetches. Conventional static analysis cannot inspect proprietary, commercial off-the-shelf (COTS) RTOS kernels, and dynamic heuristics, which rely on broad time-window thresholds, suffer from high false positive rates and heavy emulation overhead. To address these challenges, we present ISOLATOS, the first hardware-supported framework for detecting double-fetch bugs in COTS RTOS. By leveraging modern CPU kernel-isolation features, ISOLATOS enables kernel isolation so that cross-boundary accesses can be captured by triggering page faults. ISOLATOS then records page-fault metadata on each user-memory fetch. Finally, multiple fetches in the same system call are determined as a double-fetch bug, based on the lifecycle of system calls that ISOLATOS instruments into COTS RTOS. We evaluate ISOLATOS on three widely used RTOS, including QNX, VxWorks, and seL4, and demonstrate a 79.3× reduction in runtime overhead compared to state-of-the-art emulation-based detectors. ISOLATOS also detects double-fetch bugs with lower false positive rates than other tools. Our approach uncovers 43 previously unknown vulnerabilities in COTS RTOS (41 confirmed by vendors, 2 CVEs assigned). Additionally, we have demonstrated the real-world impact of our findings in automotive systems by exploiting them.**

I. INTRODUCTION

Real-time operating systems (RTOSes) dominate the cyber-physical system industry, powering applications in Internet of Things (IoT), aerospace, and power plants. With more than 2.2 billion embedded devices running systems like QNX [1] and VxWorks [2], their compact footprints, exceptional responsiveness, and deterministic performance make them ideal for applications demanding efficient and time-critical resource utilization. This trend is further driven by the expanding intelligent automotive landscape, which requires reliable real-time processing capabilities. BlackBerry QNX was embedded in more than 215 million vehicles worldwide by 2022, an increase of 20 million from the previous year [3], and more than 4,000 companies utilize WindRiver VxWorks [4].

As the deployment of RTOSes in safety-critical applications increases, the security of these systems become critical concerns. Limited research has explored the detection of double-fetch bugs in modern Commercial Off-the-Shelf (COTS) RTOSes despite their potential to be exploited. Double-fetch attacks occur when the kernel reads data from user-space and subsequently, in the same thread, uses the data without verifying whether it has changed [5]. During this interval between reference and use, a concurrently running user-space process may alter the data, resulting in inconsistencies of which the kernel is unaware. This directly violates the kernel's trust in its own validation logic. Such vulnerabilities can be exploited to trigger a range of security issues, including kernel crashes [6] and privilege escalation [7]. The implications of double-fetch bugs are particularly severe in safety-critical applications, where the integrity and reliability of the operating system is required to prevent catastrophic outcomes. In automotive or aerospace contexts, such exploits risk transforming software flaws into physical harm.

Detecting and mitigating double-fetch bugs is imperative given the critical role COTS RTOS plays in maintaining the safety and functionality of safety-critical devices. However, existing methods either cannot detect double-fetch bugs in COTS RTOS [8], [9], [10], [11] or incur high false positives and overhead [12]. Tools such as Coccinelle [11] and Deadline [10] require source code, especially user-space pointers in transfer functions, to detect double-fetch bugs. However, source code is not available due to the proprietary nature of COTS RTOS, which precludes the use of tools that require it. Bochspwn [12] detects double-fetch based on emulation and a simple time window for multiple fetches, resulting in both high overhead and false positives. Another factor inhibiting existing tools is that double-fetch bug detection requires that multiple fetches of user-space data occur in the same system call. However, the implementation of COTS RTOSes frequently violates this requirement due to the use of preemption and multi-CPU mechanisms. Thus, multiple fetches of user space data may be from different system calls. This consequently increases false positives when using Bochspwn.

These issues motivate us to investigate whether it is possible for *a tool to detect double-fetch bugs in COTS RTOS both*

§ The corresponding author.

Network and Distributed System Security (NDSS) Symposium 2026
23-27, February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230568
www.ndss-symposium.org

quickly and accurately? To answer this question, our key insight is that kernel isolation based on hardware features introduces low overhead and enables differentiation of double fetch sources. Therefore, in this paper, we propose ISOLATOS to detect double-fetch bugs in COTS RTOS by re-enabling kernel isolation. However, this is non-trivial for COTS RTOSes because we have to 1) identify context-switching operators without source code; 2) differentiate both preemptive and multi-cpu kernel thread fetches from those within the same thread; and 3) efficiently handle page faults caused by directly re-enabling kernel isolation.

In this paper, we propose ISOLATOS in order to achieve quick and accurate detection of double-fetch bugs. ISOLATOS first re-enables kernel memory isolation by configuring its control bit in the associated control register. Afterwards, when there is a page fault due to a user-land memory access from the kernel, system control flow is redirected into a custom fault handler. In the fault handler ISOLATOS records the running context relevant for double-fetch detection, including the program counter of the cross-boundary access, CPU information necessary to distinguish incorrect logs due to preemption, and target addresses of the reference. Afterwards, the fault handler in ISOLATOS retrieves the crash context saved in each cross-boundary operation to allow recovery. Afterwards, ISOLATOS distinguishes these memory access operations within a system call's lifecycle. Integrating these innovative solutions allows ISOLATOS not only to achieve efficient double-fetch bug detection, but also to maintain a significantly lower false positive rate than state-of-the-art (SoTA) tools, thereby preserving precision.

We evaluate ISOLATOS by applying it to several widely-used COTS RTOSes, including QNX, VxWorks, and seL4. These systems are known for their deployment in critical applications, making them ideal candidates for our study. The results of our evaluation indicate that ISOLATOS is highly effective in detecting double-fetch bugs within these environments. Compared to other existing tools, ISOLATOS demonstrated superior performance in terms of precision and false positives, successfully identifying double-fetch bugs that other tools missed or misidentified. Bochspwn [12], for instance, can generate a false positive rate of more than 87% while dealing with preemption. Coccinelle [11] and Deadline [10] can not detect double-fetch bugs in COTS RTOSes because their detection is based on the use of user-space pointers in transfer functions. Finally, the source code methods [8], [9], [10], [11] fail because often only a binary can be recovered in COTS RTOS. Our evaluation in Section VI confirms that ISOLATOS not only addresses the specific challenges of analyzing closed-source COTS RTOS binaries but also sets a new benchmark in double-fetch bug detection.

This paper makes the following four key contributions:

- ISOLATOS uses a novel hardware-based kernel isolation mechanism (SMAP/PAN) to efficiently and precisely detect double-fetch bugs. To the best of our knowledge, it is the first tool to use this approach.
- We develop and implement a prototype of our framework, named ISOLATOS, which not only establishes a low-overhead exception handler to recover crash context without disrupting normal execution but also recovers and instruments all kernel entry points to distinguish a system call's life-cycle during preemption. A comparison with SoTA tools reveals that ISOLATOS is 70× more efficient, while also mitigating false positives.
- Using ISOLATOS, we found the first double-fetch vulnerability in QNX procnto kernels 6.6, 7.0, and 8.0. It is the first local privilege escalation publicly disclosed in this system. This vulnerability has real-world impact by affecting production vehicles, including the most popular automotive manufacturers.
- We perform a comprehensive evaluation of ISOLATOS in widely used COTS RTOSes, including QNX and VxWorks. Our experiments uncover a total of 43 double-fetch bugs, with 41 previously unknown and 39 assigned CVEs. Furthermore, we apply ISOLATOS to the seL4 microkernel binary, detecting an additional double-fetch bug, thus demonstrating the effectiveness of our approach across various COTS RTOS platforms.

II. BACKGROUND

In this section, we present the necessary background knowledge on double-fetch bugs, preemption in COTS RTOS, and hardware-based kernel isolation to understand the design decisions and contributions of ISOLATOS.

A. Double-Fetch Bugs

Double-fetch bugs are a significant concern in RTOSes like VxWorks and QNX, which are designed for applications requiring precise timing and predictable behavior. These bugs arise when user-space data is fetched multiple times in kernel space without proper synchronization between accesses [13], [14]. As shown in Figure 1, after the initial reference to `ker_a1 = p->a` in the kernel, the write operation `p->a = user_a0` in user-space changes the contents of `a`. This, therefore, leads to an inconsistency in the kernel data when `ker_a2 = p->a` is referenced because `ker_a1` and `ker_a2` are now different. The requirements of RTOSes, such as rapid and efficient data transfer, increase the risk of inconsistencies if data changes between references. Any alteration by another task or interrupt can lead to critical system failures or unpredictable behavior, directly impacting the system's ability to meet its stringent real-time demands.

Although a considerable number of double fetch bugs are benign, *i.e.,* they cannot be exploited, those that can result in serious security issues to the system. Wang et al. [5] show that 46.2% of the exploitable double fetch bugs can cause privilege escalation issues (such as CVE-2008-2252 [15]), 39.5% can cause information leaks (such as CVE-2016-6130 [16]), 42.9% can cause bypass issues (such as CVE-2016-6236 [17]), and 11% can cause denial-of-service issues (such as CVE-2016-6156 [18]). Furthermore, in the latest PWN2OWN, a

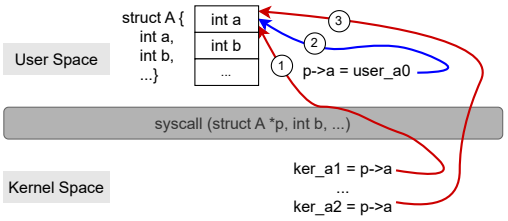


Fig. 1: A demonstrating double-fetch bug. In detail, between the two fetching operations ① and ③ (i.e., `ker_a1=p->a` and `ker_a2=p->a`) in kernel space, the actual value of `p->a` is changed in user-space due to operation ②, and thus a data inconsistency is triggered in kernel space.

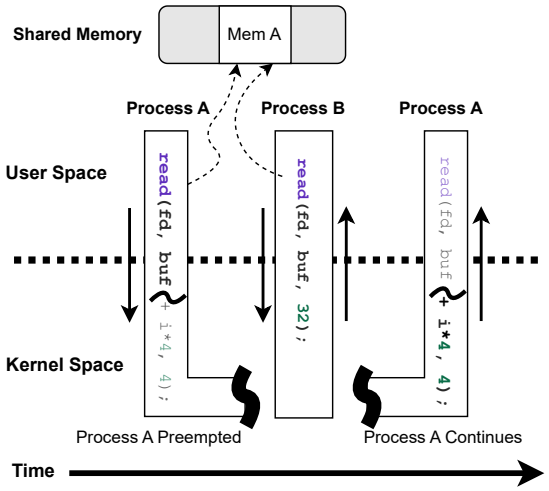


Fig. 2: Preemptions. Process B is created with a higher priority than Process A. The `read()` in Process A is preempted by Process B. The two `read`s in the processes A and B access the same file `fd` and the same buffer `Mem A`.

TOCTOU bug (an alias of a double-fetch bug) was successfully exploited on Tesla. Hackers won $100,000 for finding this vulnerability in a Tesla Model 3. In 2019 Black Hat USA, researchers released a kernel vulnerability for VxWorks against Boeing 787 to escalate privileges [19].

B. Preemption in Real-Time Operating Systems

Preemption is a fundamental scheduling mechanism in RTOSes that enables deterministic task execution and ensures that high-priority tasks meet their real-time deadlines. In addition, devices that commonly use RTOSes, such as routers and automotive gateways, typically consume significant I/O or safety-critical services, which means preemption is even more common. Unlike cooperative scheduling systems, preemptive scheduling allows higher priority tasks to interrupt and suspend lower priority tasks at any point during execution, ensuring that critical real-time tasks receive immediate processor attention. Modern COTS RTOS services like QNX and VxWorks implement sophisticated priority-based scheduling algorithms that support up to 256 distinct priority levels. Preemption operates at multiple levels: task-level preemption occurs when higher-priority user tasks interrupt lower-priority tasks, kernel-level preemption allows high-priority tasks to interrupt kernel execution including system call processing, and interrupt-driven preemption enables hardware interrupts to suspend any executing task for time-critical events.

The preemptive nature of RTOSes creates execution patterns in which multiple independent threads access the same memory locations during overlapping time periods. Consider the scenario shown in Figure 2 in which two processes execute a system call that accesses a shared memory address. `Process A` is preempted before performing its read to `Mem A` by a higher-priority `Process B`, which independently accesses the same location for its own system call. Both threads perform legitimate, independent operations within their own contexts.

RTOS environments frequently employ shared memory mechanisms for efficient inter-process communication, including message queues, semaphore objects, shared buffers, and memory-mapped I/O regions. Multiple kernel threads and user processes legitimately access these structures as part of normal system operation. Real-world scenarios exhibit complex multi-level preemption chains where hardware interrupts, high-priority tasks, and regular execution contexts may all access the same memory regions within short time periods. These temporal memory access characteristics, resulting from aggressive preemption policies and shared memory utilization, represent normal RTOS behavior essential for deterministic and predictable real-time performance rather than security.

C. Kernel Isolation

Kernel isolation mechanisms prevent unauthorized cross-boundary memory accesses by enforcing hardware-level access controls between user and kernel address spaces. These protections can detect and prevent double-fetch vulnerabilities by triggering exceptions when kernel code attempts to access user memory without proper authorization.

To mitigate kernel vulnerabilities, a series of mechanisms have been used, such as Supervisor Mode Access Prevention (SMAP) [20] and Privileged Access Never (PAN) [21], as shown in Table I. Table I shows that kernel isolation support varies significantly between CPU architectures, with modern Intel (Broadwell+) and ARM (v8.1+) processors providing hardware-based isolation through SMAP and PAN registers, while older architectures rely on software-based implementations or lack support entirely.

In addition to these, most general-purpose operating systems provide specific cross-context transfer functions (*e.g.,* `copy_from_user()` and `get_user()`) to avert illegal memory accesses during data exchange, with a considerable performance cost. However, due to real-time performance considerations, RTOS kernels directly dereference pointers without using cross-context transfer functions. In particular, the CPUs of embedded devices running RTOSes usually do not support these specific transfer functions [21]. The design of cross-context data transfer in COTS RTOSes makes them more vulnerable to double-fetch bugs than Unix-like systems. COTS RTOSes are conservative in using a synchronization

TABLE I: Kernel Isolation Supports from Architectures.

CPU		Design	Technique
ARM	v7 (32-bit)	SW	Page Domain
	v8.0 (64-bit)	SW	TTBR
	v8.1	HW	PAN Reg
X86	pre-Broadwell	N/A	N/A
	Broadwell+	HW	SMAP
S/390	Any	HW	Addr Space
PowerPC	Radix MMU	HW	KUAP
	Hash MMU	HW	KUAP
	MPC 8xx	HW	KUAP
MIPS	Any	N/A	N/A

SW - Pure Software Implementation; HW - Hardware Support.

mechanism such as mutex and spinlock because their use may violate real-time principles. High-priority memory accesses can be prevented by a held lock, which results in priority inversion [22] or deadlock [23]. Besides, RTOSes are usually implemented with C/C++, which has long been criticized for the security issues caused by the direct use of pointers in memory accesses. Due to the direct use of pointers in cross-context data transfer, developers of RTOS kernels may not be aware that they are accessing user data.

III. Challenges

In what follows, we elaborate on the relevant challenges that need to be overcome to efficiently and accurately detect double-fetch bugs in COTS RTOSes.

C1: Identification of Cross-boundary Memory Accesses. Double-fetch bugs occur between user space and kernel space, which requires detection tools to identify cross-boundary memory accesses. Prior static approaches rely on the rich semantics found in source code, such as transfer functions. However, COTS RTOSes are closed source, resulting in kernel and user pointers appearing identical at the binary level, making static analysis approaches inapplicable to COTS RTOS environments. Moreover, the SoTA dynamic method, Bochspwn, also relies on specific memory operation APIs (e.g. `memcpy`), which cannot cover all the memory operations that exist in a RTOS. For instance, in general purpose operating systems (GPOS), most cross-boundary operations should be wrapped with `memcpy()` or `copy_from_user()` functions. In contrast, a COTS RTOS can directly refer to a user pointer. As shown in Listing 1, existing tools can work on a GPOS that has explicit cross-boundary memory operations. In an RTOS, the system directly dereferences the user-pointers, which are unfortunately undetectable with existing tools. Besides, this Bochspwn is slow because of the application of Bochs emulation (13-18X slower) and instrumentation (2.2-2.65X slower), eventually leading to over 30X overhead compared to native virtualization (e.g. virtual box). Notably, native virtualization is fast, but common emulators (e.g QEMU, VMWare) are only supported by Linux, MacOS, and Windows. If we naively run a RTOS on them, it will be even slower in comparison.

```
// Explicit Transfer Functions
int GPOS_syscall_handler
    (struct user_request *user_ptr) {
    struct kernel_data kdata;
    // Explicit memory operation
    if (memcpy(&kdata, user_ptr,
        sizeof(kdata)) != 0) return -EFAULT;
    // Validate data
    if (kdata.size > MAX_SIZE) return -EINVAL;
    // Second fetch
    // explicit and detectable
    if (memcpy(&kdata, user_ptr,
        sizeof(kdata)) != 0) return -EFAULT;
    // Process data using kdata.size
    // potential double-fetch bug
    return process_data(&kdata);
}

// COTS RTOS - Direct Access to User Memory
// NO Explicit Transfer Function
// Invisible to API-based Detection
int rtos_syscall_handler
    (struct user_request *user_ptr) {
    // First fetch
    // Direct pointer dereference
    int size = user_ptr->size;
    // Validate data
    if (size > MAX_SIZE) return -EINVAL;
    // Second fetch
    int size2 = user_ptr->size;
    // Process data using size2
    // Vulnerability not detected
    return process_data_with_size(size2);
}
```

Listing 1: Comparison of cross-boundary memory access patterns in General Purpose OS vs. COTS RTOS. Linux uses explicit transfer functions that are detectable by existing tools, while COTS RTOS employs direct pointer dereferencing that bypasses API-based detection mechanisms.

Thus, *identifying cross-boundary memory access accurately and efficiently in COTS RTOS remains a challenging task.*

C2: Preemption and Multi-CPU Identification. COTS RTOSes implement time-critical features, which means that the kernel is fully preemptable, even within a system call. Imprecision arises in detecting double-fetch bugs in COTS RTOS mainly because of the potential multi-threaded execution caused by preemption and multi-CPU fetches. In RTOSes, by default, a lower priority thread can be preempted by another thread with a higher priority. Similarly, kernel threads from other CPUs can access shared memory. This results in imprecision for existing tools attempting to detect double-fetch bugs because they cannot differentiate whether fetches are from the same thread [12]. To differentiate fetches, the approach must collect information about current kernel threads. In addition, the approach must capture the status of each system call. However, because preemption is managed internally by the kernel, there is no explicit signal indicating when a higher-priority thread preempts a lower-priority thread while executing its system call. To capture preemption, one needs to modify system kernels and implement code for interactions between the emulator and preemption. However, the kernel binary is relocated during execution. For example,

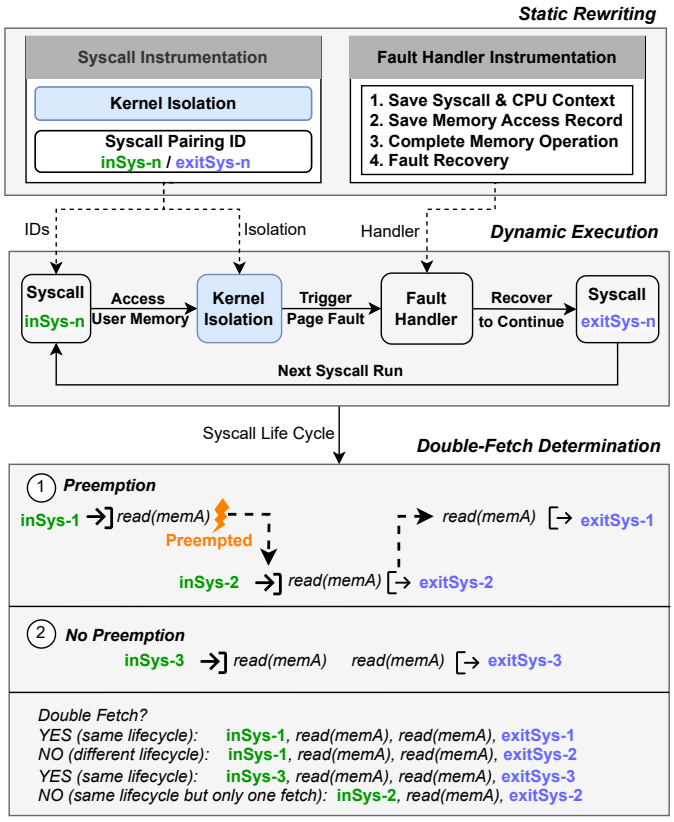


Fig. 3: Workflow of ISOLATOS. By static rewriting, we instrument code to enable kernel isolation and handle page faults, and further use life cycle to determine double-fetch.

a `mov` instruction referencing 0xc45e1 after dynamic loading would translate to "`48 89 3d e1 45 0c 00`". However, in the static hex binary, the actual hex value is "`48 89 3d 00 00 00 00`", indicating that the offset value is not fixed. Therefore, if we directly modify the binary, the last four bytes of this instruction will be overwritten when loaded with "`e1 45 0c 00`". Currently, there is no existing tool that can hook the binary kernel to adapt to the emulator. Therefore, *detecting double fetch bugs under preemption is a vital challenge.*

C3: Recovery From Page Fault. To address the aforementioned two challenges, our approach is to re-enable kernel isolation through which every user memory access under a system call will trigger a page fault. However, enabling kernel isolation directly in COTS RTOS will lead to numerous kernel faults, hindering system execution. Therefore, properly re-enabling kernel isolation without affecting the execution of the whole system remains challenging, leading to the third challenge. The fundamental challenge lies in the isolation mechanism itself: when a page fault occurs due to a kernel isolation violation, the exception handler executes in a separate, isolated memory region from the original kernel context. Recovering from this fault requires ISOLATOS to jump across different execution contexts while preserving the original program state and maintaining control over subsequent memory accesses.

To be specific, ISOLATOS deliberately enables memory exceptions and it is essential to trigger exceptions multiple times to detect double-fetch bugs. However, the instruction that triggers the exception will NOT be executed, instead the control flow will jump to the exception handler. If we want to maintain the integrity of program semantics, then we should complete its execution. A potential solution would be to execute the instruction with isolation enabled since it is a cross-boundary access. However, this fails to work because if we jump back to the instruction with isolation off, we will not be able to take back control of the kernel instructions. This would mean that we could not detect further "fetches" on subsequent instructions, which would prevent double-fetch detection entirely. Thus, *to recover execution from a kernel exception is a vital challenge.*

IV. Design of ISOLATOS

This section describes ISOLATOS, a tool designed for dynamic detection of double-fetch bugs that runs natively on hardware, instead of virtual machines, as explained in C1. ISOLATOS leverages hardware-based kernel isolation features to precisely identify cross-boundary memory accesses and distinguish legitimate concurrent operations from actual double-fetch vulnerabilities. The approach consists of four main components: (1) static kernel entry identification and instrumentation, (2) dynamic system call boundary tracking with isolation control, (3) comprehensive page fault handling and recovery, and (4) double-fetch pattern analysis. A high-level overview of this process is shown in Figure 3.

Kernel double-fetch bugs can only exist in OSes with user/kernel separation. Although some bare-metal RTOSes do not implement user/kernel separation, most industry-leading COTS RTOSes [24], such as Blackberry QNX, Wind River VxWorks, Green Hills INTEGRITY, Lynx OS, and DeO, use user/kernel separation. Therefore, in this paper, we design and implement ISOLATOS, focusing on COTS RTOSes. Throughout the paper, we assume that attackers can manipulate data shared between user- and kernel-space and aim to exploit double-fetch bugs from user-space. In the following sections, we explore each functional aspect of ISOLATOSin detail.

A. Static Kernel Entry Identification and Instrumentation

To address **C1**, ISOLATOS employs static analysis to identify entry points into the kernel and strategically instrument them with isolation control mechanisms. This approach enables comprehensive coverage of all system call invocations while maintaining low performance overhead.

Kernel Entry Point Discovery. Identifying the address at which the binaries are loaded is essential for instrumentation. Kernel handler entries can be ascertained by referencing hardware documentation and inspecting kernel binaries during initialization. System call handlers between different architectures are triggered by various methods, mainly split into fixed and dynamic methods. Fixed approaches use interrupts, directing control flow to fixed addresses. Dynamic approaches

utilize MSRs (Model-Specific Registers) to register handler addresses at the system boot stage. This can be analyzed using static methods.

For instance, in QNX 8.0 kernel, pertinent instructions are shown in Listing 2. Line 2 assigns ECX to a constant value 0xc0000082, representing the configuration for kernel entry. Line 5 assigns RAX the value syscall_trap_0 + pcpu * 0x20, which designates the entry position of the kernel. This value comprises two segments: syscall_trap_0, indicating the initial syscall trap, and pcpu, the CPU identifier that will perform the syscall. The factor 0x20 specifies the interval between consecutive syscall traps, such as syscall_trap_0 and syscall_trap_1, meaning that each CPU has an associated system call trap.

```
1  MOV     pcpu,R12D
2  MOV     ECX,0xc0000082
3  MOV     RAX,pcpu
4  SHL     RAX,0x5
5  ADD     RAX,qword ptr[syscall_trap_0]
6  MOV     RDX,RAX
7  WRMSR
```

Listing 2: Assembly in QNX 8.0 X86-64, which sets up WRMSR to syscall_trap_0 at kernel entry. Assigning ECX with value 0xc0000082 indicates kernel entry being setup.

We subsequently use this attribute to identify dynamic kernel entries. Initially, we identify system calls related to MSRs using hardware documentation. Then, we examine all WRMSR assembly instructions and their most recent assignments to ECX and EAX prior to these to locate the kernel handler's entry point. Following this, we amend values to adjust the kernel entry address during the initialization phase, allowing us to hook each system call.

Kernel Handler Instrumentation. Once kernel entry points are identified, ISOLATOS instruments the kernel handler entry with two critical operations that enable efficient and accurate double-fetch detection. 1) *Kernel Isolation Activation.* We enable kernel isolation by setting the appropriate isolation bits in the control registers. For x86-64 architectures, this involves setting the 21st bit (SMAP) in the CR4 register. For AArch64, isolation is enabled by setting the PAN register to 0x0b1. This configuration ensures that any subsequent cross-boundary memory access will trigger a page fault that can be intercepted and analyzed. 2) *System Call Pairing ID Registration.* We assign unique system call identifiers in separate memory and establish boundary markers (inSys-n and exitSys-n) to track a system call's lifecycle. This addresses **C2** by differentiating memory accesses from different execution contexts, including those from preempted threads or concurrent CPU executions. The unique system call identifier n is specified using unique values, e.g. an increment series, time stamps. When the current system call is preempted by another one, the next system call will be registered with a different marker. The unique system call marker is maintained throughout its entire lifecycle, *i.e.,* until it exits, at which point exitSys-n signal will be generated to end the record.

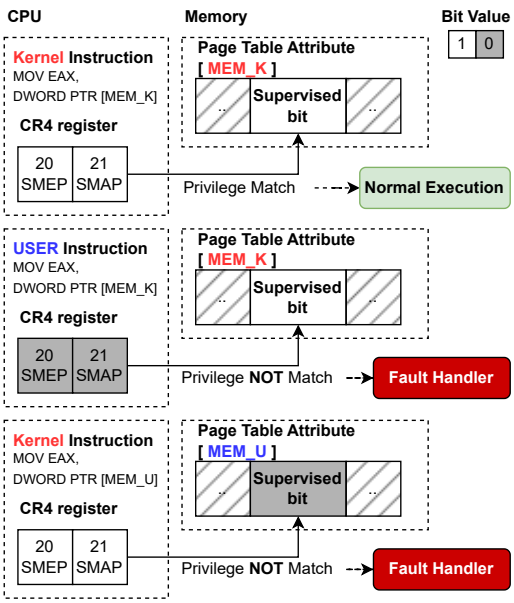


Fig. 4: To enable kernel isolation mechanism, we can setup register accordingly. For x86-64, the difference between CR4.SMAP and the supervised bit will cause page fault.

The subsequent task involves finding addresses for kernel instrumentation. A robust and comprehensive approach to modifying COTS RTOSes is to employ the kernel's built-in extension mechanism, typically referred to as its Board Support Package(BSP). After identifying both the method and target for hooking, we establish a function to hook that then configures kernel isolation and boundary tracking.

B. Enable Isolation and Page Fault Handler

During the execution of a system call, ISOLATOS leverages the previously established instrumentation to monitor cross-boundary memory accesses and maintain execution context information. This section addresses both **C2** and **C3**.

Execution after Isolation-Enabled. At each system call entry, the execution flow is transferred to the instrumented kernel handler where kernel isolation has been enabled. With isolation active, any attempt by kernel code to access user-space memory triggers a hardware-generated page fault. This mechanism provides hardware-level detection of cross-boundary memory accesses without relying on API interception or heuristic pattern matching.

To initiate kernel isolation, we start by identifying flag registers that manage user/kernel attribute settings by referring to the chip's programming guides. We then proceed to configure the memory attributes in linear memory to match the flags specified in these registers. It should be noted that kernel isolation was not activated even with the user/kernel attributes set, despite user/kernel bits in their page table entries set for the COTS RTOSes we evaluated. In order to fully enable isolation, we set the isolation attribute in the control registers after setting up the page flags for user/kernel memory. This process is illustrated in Figure 4.

The page fault handler is a critical component of ISOLATOS that must handle cross-boundary access violations while maintaining system stability and execution semantics. The handler is integrated as part of the static system call patcher, ensuring comprehensive coverage of all potential access scenarios.

Page Fault Context with Preemption Awareness. When a cross-boundary access triggers a page fault, ISOLATOS's handler records essential information for subsequent double-fetch analysis. This mechanism specifically addresses **C2** by maintaining per-system-call context through two key components. The *system call lifecycle* tracking records the target address of memory accesses along with the current system call pairing ID (`inSys-n`). This pairing ID is crucial for distinguishing system call lifecycles and preventing false positives caused by legitimate concurrent accesses from different execution contexts. In preemptive COTS RTOS environments, multiple threads may access the same user-space addresses, but only accesses within the same system call context constitute potential double-fetch vulnerabilities. The *execution context* capture involves recording the instruction pointer (IP) that raised the fault and the address being accessed. This information, combined with the system call pairing ID, enables accurate correlation of memory accesses even in the presence of thread preemption or multi-CPU execution.

Instruction Replay and Execution Recovery. To maintain the integrity of program semantics and system stability while addressing **C3**, ISOLATOS implements an instruction replay mechanism through three sequential operations. *Program execution integrity* is maintained by re-executing faulting memory operations with isolation temporarily disabled to complete interrupted execution. This ensures that the original program semantics are preserved to keep the system running. *Fault recovery* involves restoring execution context and continuing system call processing, where the handler leverages hardware debugging features to read instructions, analyze their functionality, and restore memory to its appropriate state. Finally, *isolation restoration* re-enables kernel isolation after the instruction successfully completes to continue monitoring subsequent memory accesses within the same system call.

As explained in **C3**, ISOLATOS must instrument the kernel while preventing it from crashing. Specifically, for each CPU, we can refer to its programming guide for details about the behavior of a page fault raised due to kernel isolation. However, in general, when such a page fault occurs, the program pointer will fall into a specific trap routine. Control registers will hold the page address that caused the fault and the error code in the flag registers. Consequently, we can filter cross-boundary memory accesses that fault with their error code and recover the instructions that caused the crash. By observation, this address is normally directly stored on the kernel stack. Thus, we can restore values from the stack to determine if it is a read operation accessing user memory.

C. Double-Fetch Pattern Analysis and Determination

To determine if there are any double-fetch bugs, ISOLATOS analyzes the recorded memory accesses to identify patterns

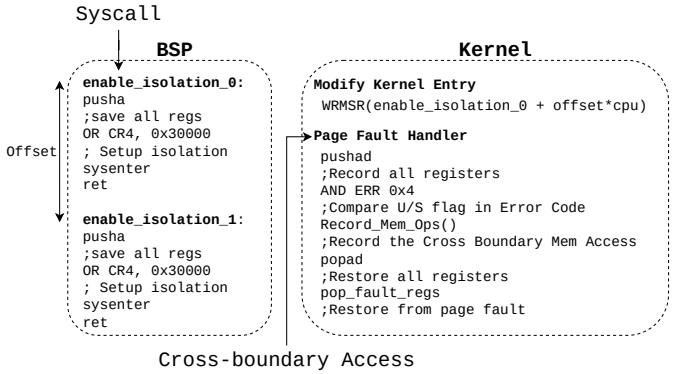


Fig. 5: Implementation of IsolatOS on Intel board. BSP provides a more friendly way to programming in C instead of writing assembly in binary. We modify the kernel entry by instrument the WRMSR operation. Then all system calls will be redirected into IsolatOS code. A user memory access from kernel will trigger a page fault with error number 0xe.

where the same memory address is accessed multiple times within a single system call boundary, *i.e.,* its lifecyle.

Access Correlation Analysis. Algorithm 1 in Appendix C shows the high-level procedure that ISOLATOS uses to correlate memory accesses within a single system call boundary. We record every cross-boundary access with their target address, instruction address, and associated system call pairing ID. The analysis then correlates accesses that share the same system call pairing ID, ensuring only memory accesses within the same execution context are considered for double-fetch detection. As shown in Figure 3, if multiple accesses to the same memory address are detected within the same system call boundary (same `inSys-n` identifier), the pattern is recorded as a potential double-fetch vulnerability. This approach can eliminates false positives caused by concurrent accesses from different system calls or rapid system call sequences. For instance, when syscall-1 (inSys-1) is preempted by syscall-2 (inSys-2), both of them read `memA`, and the read happens in a short time window. For Bochspwn, the two read operation will be regarded as a double-fetch. However, ISOLATOS can distinguish that the two read operations do not lead to a double fetch bug because the life cycle for each system call is marked with the pairing ID. Therefore, patterns such as {`inSys-1, read(memA), read(memA), exitSys-1`} will be identified as a double-fetch bug because the two fetches occur within the same system call. Executions that do not follow this pattern will not be identified as a double-fetch bug. For example, the pattern {`inSys-1, read(memA), read(memA), exitSys-2`} does not indicate a double-fetch bug because the two fetches occur within two system calls.

V. IMPLEMENTATION

To implement the design for COTS RTOSes, we utilize various features provided by our target systems, including QNX, VxWorks, and SeL4 independently. An example of this

process for QNX is given in Figure 5. We note, however, that the majority of ISOLATOS is target-independent as shown in Appendix D, Table VII. Only control of the user/kernel isolation bit, discovery of kernel entry, and page fault handling are specific to the architecture and require a one-time effort. However, the main body of ISOLATOS, is reusable across targets. We describe each step in more detail below.

A. System adaptation

To adapt to different operating systems, we need to figure out where to put the instrument code.

QNX. This OS requires licensing for use. Fortunately, for academic purposes, we can obtain an academic license through its official site. We consequently acquired QNX licenses for versions 6.6, 7.0, and 8.0 for our tests. QNX employs a micro-kernel architecture, meaning that the only way to integrate custom privileged instructions is by modifying the Board Support Package (BSP) [25]. To implement our design in QNX, we begin by incorporating assembly code into the BSP to activate kernel isolation. To extend the system, QNX offers a feature known as a "callout", which acts as a function pointer within the kernel binary. This can be used to introduce one additional system call under Ring 0. A callout is established in the startup program with a fixed memory address, enabling invocation from the user program. Subsequently, before executing each system call, we execute the code enabling isolation. To migrate the prototype from QNX 6.6 to QNX 7.0 and QNX 8.0, in total, spend us 8 hours, including debugging. BSPs have various features of boot loading, kernel initialization. So, we spend most time investigating where to insert our code.

For x86-64, we directly instrument the page fault exception number `0xe` defined by the CPU. To implement the *page fault handler*, we locate `trap0e` in the QNX kernel functions, then directly modify the instructions that do not have a cross reference to the relocation table. These instructions are modified to jump to the address right after the callout, which is also running in Ring 0. In this part, we read CR2, and `RSP-0x10` to get the page address that raises the exception.

VxWorks. In contrast to QNX, VxWorks is structured as a monolithic kernel, where the kernel driver code runs with privileges. Thus, we can implement a driver, rather than finding an entry from the BSP. Before invocation of the system call, we call the driver to activate kernel isolation. The remaining implementation is the same as QNX. We spend around 2 workdays to implement the driver. The first task is to identify the instruction we can replace to hook, it takes half a day. Then, to invoke driver, we need to use a different approach, to be specific, `ioctl`, this is a different implementation compared to QNX.

SeL4. SeL4 is an open-source operating system, allowing us to directly incorporate functionality into its source code. It is almost the same with QNX, but we spend hours to find the hook addresses of system calls' entries. In the end, it takes us less than 2 workdays to finish the implementation.

B. Architecture-Specific Code

Currently, ARM and Intel x86/x86-64 are the dominant chips that support kernel isolation. In our implementation, both are supported. A summarized table detailing the differences and similarities between x86-64 and ARM is in Appendix D, Table VII. Overall, except for the assembly code related to architecture, the rest can be reused.

Intel. implements kernel isolation through Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Protection (SMEP), which have been integrated into Linux since 2012. To activate these features, we begin by selecting a chip that supports them. In our implementation, we opt for the Intel 12th Generation i7, released in November 2021, which includes support for the aforementioned isolation features.

To enable SMAP and SMEP on this chip, we first identify Control Register 4 (CR4), which contains the flags for these features at the 20th and 21st bit positions. By setting these two bits to 1, SMAP and SMEP are activated. Given that the code will execute with `sysenter`, we initially save the execution context by pushing all registers onto the stack. We then proceed to modify the isolation bits. Following this, we restore the stack and return to the original system call entry to finalize the system call process. Subsequently, we replicate this code across 63 different instances for each CPU. Then, we register a page fault handler to log all SMAP and SMEP violations according to the fault flag, along with a recovery program that restores instruction pointers and stacks, allowing us to record memory accesses that violate boundaries.

ARM. implements isolation in a separate register named PAN. All registers are mapped in the kernel, and maintained by the variable `regs->pstate`. By setting the 22nd bit of this variable, PAN is then enabled. The RISC-like ARM ISA makes it easier to perform fault recovery because the length of each operation is fixed. To be specific, we can simply copy the access instruction from the program counter, then insert operations to disable PAN, access user memory, enable PAN, then jump to the next instruction.

C. Unified Fault Recovery.

Implementing fault recovery purely in software for x86-64 is not trivial due to the variable instruction length of the CISC architecture. Different instructions can span different byte lengths, making it difficult to consistently identify and recover from faults. Instead, we developed a unified approach that uses hardware debugging capabilities through the JTAG interface and GDB debugger. JTAG (Joint Test Action Group) is an industry standard for hardware debugging that provides direct access to a processor's internal state through dedicated debug pins on the chip. Unlike software-based debugging, JTAG enables non-intrusive access to the processor even when the operating system is compromised or unresponsive.

Our implementation connects GDB (GNU Debugger) to the target system's JTAG interface using specialized hardware adapters. This connection creates a debugging channel that operates independently from the target operating system. Hardware breakpoints differ fundamentally from software break-

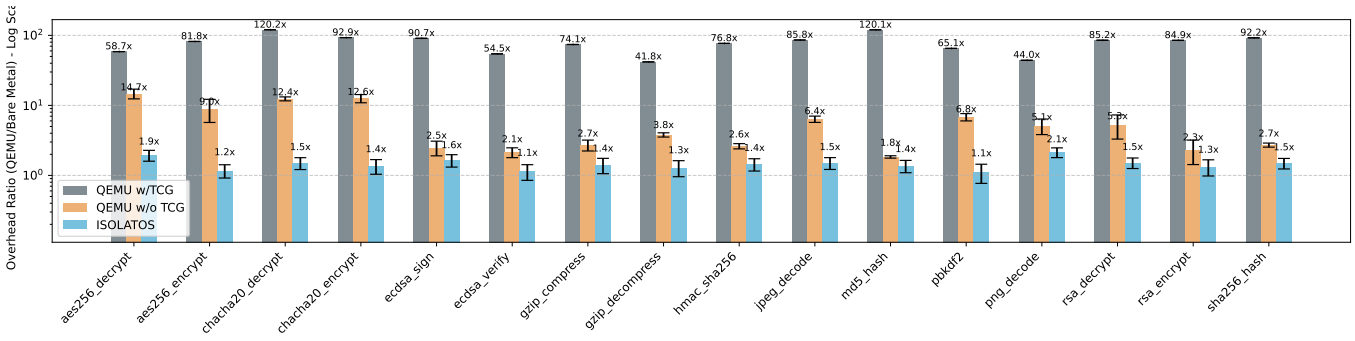


Fig. 6: Performance overhead of ISOLATOS on QNX 8.0 compared to QEMU-TCG and QEMU execution without TCG. The overhead is shown on a logarithmic scale, as TCG is running much slower than bare metal. On average, QEMU-TCG has a 79.3× overhead compared to bare metal native execution.

points. Rather than modifying code with trap instructions, hardware breakpoints use debug registers to pause execution at specific addresses without altering the executed code. The JTAG connection also provides direct memory inspection capabilities, allowing us to read and modify memory contents at any address without invoking system calls. This visibility enables us to analyze fault states that would normally crash the system. Similarly, direct access to CPU registers, including special-purpose registers that control isolation features such as CR4 on x86-64 or PSTATE on ARM, gives us precise control over the processor's execution environment.

Our fault recovery procedure begins when a kernel isolation fault occurs, such as a SMAP/SMEP violation on x86-64 or a PAN violation on ARM. At this point, our custom fault handler captures the fault address and relevant context information. Rather than attempting immediate recovery through potentially complex software mechanisms, the handler transfers control to a designated recovery routine where we have previously established a hardware breakpoint via the JTAG interface. When execution reaches this breakpoint, GDB automatically gains control of the processor. Our automated scripts then analyze the fault context, examining the specific instruction that triggered the fault, the memory addresses involved in the operation, and the current processor state, including all relevant register values. This analysis operates outside the context of the running system, providing a stable environment for fault characterization.

The actual recovery process involves temporarily deactivating the relevant isolation mechanism, for example, clearing the SMAP bit in the CR4 register on x86-64 systems. With isolation temporarily suspended, we can safely execute the faulting instruction in a controlled manner under debugger supervision. After the instruction completes successfully, we immediately re-enable the isolation mechanism to maintain the awareness of cross-boundary memory access. The debugger then instructs the processor to resume execution from the instruction following the faulting one.

TABLE II: Runtime Performance Analysis.

Benchmark Category	Bochspwn	QEMU-TCG	Relative Performance
Boot	2175	1778	18.3%
Memory-intensive	497	334	33.8%
CPU-intensive	476	386	23.2%

D. Hardware selection.

As shown in Table I, not all processors support hardware isolation. Thus, it is necessary to select the hardware to implement the design. As a result, we implement the x86-64 version on top of Intel I7-12700 processor, and the ARM version based on the Raspberry Pi 5. We did not include S/390 and PowerPC because there is no evaluation version provided by the COTS RTOS companies and we cannot afford the price to purchase them, which is over $100,000 USD.

VI. Evaluation

In this section, we evaluate the performance of ISOLATOS. Initially, we introduce four evaluation goals, followed by their answers along with the evaluation results.

A. QEMU-TCG vs Bochspwn in Overhead

Since Bochspwn cannot detect double-fetch bugs in COTS RTOSes, we use QEMU-TCG to run relevant experiments. QEMU-TCG collects execution logs similar to Bochspwn. We, therefore, argue that the overhead of QEMU-TCG is similar to Bochspwn. We evaluated the cold boot time of QEMU-TCG and Bochspwn. It takes QEMU-TCG 29 minutes and 38 seconds to boot up to the login page on Ubuntu desktop. Bochspwn is able to boot Ubuntu login page in 36 minutes and 15 seconds.

We also tested based on different tasks to evaluate the overhead similarity. We test the execution time of a set of similar tasks to evaluate the average time. Memory intensive task is to play a 10-second video. CPU-intensive task is compiling a hello-world program in command line. As the result shows in Table II that Bochspwn has overall 18.3%-33.8% overhead compared to QEMU-TCG.

TABLE III: Cross-boundary Operation Recording Similarities. Bochspwn and QEMU-TCG both work on Linux kernel to run the test cases with modified kernel code to enable ground truth. Log indicates the output log of function we write in the cross-boundary functions.

Test Case	Bochspwn/Log	QEMU-TCG/Log	Similarity Ratio
IPC Operations	265 / 283	372 / 372	93.6%
File I/O	921 / 993	1298 / 1300	92.9 %
Cryptographic	799 / 803	665 / 665	99.5 %
Summary	**Mean: 95.4% ± 4.1%**		

B. Similarity Between QEMU-TCG and BochsPwn

To establish QEMU-TCG as a methodologically valid substitute for Bochspwn in our comparative analysis, we conducted a systematic evaluation of memory operation recording capabilities and trace completeness. Our experimental framework employs a testing environment using Ubuntu Linux 32-bit, ensuring compatibility with both emulation platforms.

Memory Tracing Mechanism Comparison: Bochspwn operates on the Bochs x86 emulator platform, implementing memory access interception through direct emulator hooks. The tool records cross-boundary memory operations using time-window-based heuristics while maintaining comprehensive instruction-level execution traces. In contrast, QEMU-TCG utilizes QEMU's Tiny Code Generator intermediate representation, implementing dynamic binary translation with equivalent memory access instrumentation capabilities. Both tools provide functionally equivalent memory access pattern detection through full-system emulation with similar instrumentation overhead characteristics.

Test Case Design and Implementation: We systematically evaluated the completeness of recording memory operations through ten representative workload categories executed on Linux Kernel 5.17, compiled in 32-bit. The experimental test cases encompass: (1) inter-process communication (IPC) with shared memory and pipes creation and close; (2) file I/O operations with read and write current time; (3) cryptographic operations utilizing kernel Cryptographic operations to generate key files. We write a ground truth code in the test cases, which outputs all the cross-boundary accesses logs by rewriting the functions copy_from_user(), put_user(). We compare how many ground-truth cross-boundary accesses are recorded by Bochspwn and QEMU-TCG.

The main reason that Bochspwn can miss some records is because it uses "reps mov" as a pattern. It can miss other assembly transfers between kernel and user space. The experimental results, as shown in Table III, demonstrate that QEMU-TCG achieves near-identical memory operation recording completeness compared to Bochspwn, with a mean ratio of 95.4%. The correlation analysis demonstrates a strong linear relationship between the two instrumentation approaches. This empirical evidence establishes QEMU-TCG as a functionally equivalent substitute for Bochspwn in double-fetch detection.

Storage Requirements Analysis: Trace storage requirements demonstrate comparable characteristics between platforms. For 5-minute execution traces, Bochspwn generates 127.3 GB logs compared to QEMU-TCG's 132.1 GB, yielding a size ratio of 1.038×. Extended 30-minute traces produce 772.1 GB and 797.8 GB respectively, maintaining consistent 1.033× ratio. This storage overhead parity further validates functional equivalence.

Summary: Based on our comprehensive evaluation, QEMU-TCG serves as a statistically valid substitute for Bochspwn. The methodological justification encompasses four critical dimensions: (1) **Functional Equivalence** through equivalent memory access interception mechanisms via emulation-based instrumentation; (2) **Performance Parity** with demonstrated statistical equivalence in execution overhead within 50% variance; (3) **Trace Completeness** that achieves a 99.73% coverage ratio indicating near-identical memory operation recording. This methodological validation demonstrates QEMU-TCG as a good substitute for Bochspwn in our experimental framework.

C. Efficiency Improvement

We evaluated the detection efficiency of ISOLATOS compared to existing approaches. However, a direct comparison with Bochspwn, a well-known double-fetch bug detector, proves challenging because Bochspwn is built upon the Bochs emulator, which does not support QNX. Additionally, Bochspwn is currently limited to Windows and x86 Linux platforms. Cross-operating system comparisons would yield misleading results due to fundamental architectural differences. Therefore, to equitably assess the efficiency between memory-tracing based emulation and our kernel isolation-based method, we developed a memory-tracing technique following Bochspwn's architectural principles. We compared the performance differences between Bochspwn and QEMU-TCG implementation, which has similar features. More details about the similarity can be found in Section VI-B. We also compare the overhead between Bochspwn and QEMU-TCG in Section VI-A.

We constructed a comprehensive test suite to evaluate performance across diverse system interactions. We focus mainly on image processing, compression operations, and encryption to cover a wide range of system behaviors in the limited environment of a COTS RTOS. For image processing tests, we used a custom image processing library loader to perform various operations, including JPEG/PNG. Our compression benchmarks leverage GZIP operations on various data sets. We tested compression and decompression of random binary data (10KB, 10MB and 100MB files). We calculate the average execution time for each of them to measure the overhead. These tests were selected to evaluate both sequential and random memory access patterns typical in RTOS applications. For cryptographic operations, we implemented benchmarks using OpenSSL to evaluate symmetric encryption (AES-256, Chacha20), asymmetric cryptography (RSA-2048, ECDSA),

TABLE IV: Reported double-fetch bugs. The verified true positive cases are shown in brackets.

OS	QNX 6.6				QNX 7.0				QNX 8.0			
Tool Name	QEMU-TCG			ISOLATOS	QEMU-TCG			ISOLATOS	QEMU-TCG			ISOLATOS
Time Window (ms)	0.1	1	3	N/A	0.1	1	3	N/A	0.1	1	3	N/A
md5	5 (5)	12 (6)	109	9 (9)	3 (0)	9 (1)	85	9 (9)	4 (0)	11 (0)	85	1 (1)
sha256	2 (2)	18 (2)	95	4 (4)	4 (3)	14 (3)	75	4 (4)	6 (0)	16 (0)	61	1 (1)
RSA encrypt	2 (1)	23 (10)	31	12 (12)	0	19 (2)	27	12 (12)	18 (0)	26 (0)	55	0
RSA decrypt	3 (2)	26 (2)	34	3 (3)	9 (1)	21 (1)	30	3 (3)	11 (0)	24 (0)	32	0
ECDSA sign	0	17 (9)	42	7 (7)	1 (0)	15 (2)	16	6 (6)	14 (1)	16 (1)	17	1 (1)
ECDSA verify	2 (0)	20 (3)	20	8 (8)	1 (1)	17 (5)	18	7 (7)	18 (0)	19 (0)	19	0
MsgSend	5 (1)	23 (8)	37	9 (9)	4 (3)	19 (4)	33	5 (5)	12 (0)	34 (0)	39	0
ZIP Compression	0	28 (3)	219	4 (4)	0	24 (0)	158	3 (3)	9 (0)	26 (0)	38	0
PNG Decode	0	30 (1)	173	5 (5)	0	22 (3)	130	4 (4)	11 (0)	28 (0)	41	0

Note: For time window 3ms, we cannot prove all the true-positive cases due to the large amount of bugs.

TABLE V: Distribution of False Positive Root Causes.

False Positive Category	Count	Percentage
Temporal False Positives	1,746	80.6%
Preemption False Positives	405	18.7%
Uninitialized Memory False Positives	16	0.7%
Total	**2,167**	**100%**

and hash functions (SHA-256, HMAC-SHA256, MD5). We measured performance while encrypting and decrypting data blocks of various sizes (10KB, 10MB, 100MB) and signing/verifying operations with different key lengths. These cryptographic tests are particularly relevant for security-critical RTOSes where crypto operations frequently involve multiple memory accesses of user-provided data. Additionally, we created specialized test applications that deliberately exercised system call interfaces known to handle user-space pointers, as these represent the most likely vectors for double-fetch vulnerabilities. Each test application was executed on native QEMU emulation without QEMU's Tiny Code Generator (TCG), QEMU with TCG, and ISOLATOS to ensure consistent workload comparisons.

All tests were carried out on identical hardware configurations: an Intel i7-12700 processor with 128GB RAM and Samsung 970 PRO NVMe SSD storage. Each benchmark was repeated five times to minimize the impact of system variability, with results averaged across iterations. For ARM, we only have to a Raspberry Pi, upon which running QEMU is difficult. So we did not perform an overall test for ARM. We note, however, that testing a kernel vulnerability is a one-time effort, *i.e.,* where a vulnerability that exists on Intel normally will also exist on ARM. As shown in Figure 6, ISOLATOS exhibits an average execution overhead of only 45.7% compared to bare metal execution. QEMU-TCG, which is designed similar to Bochspwn for comparison purposes, has an overhead of 79.3×. In contrast, QEMU execution alone results in an average overhead of 5.58×.

D. Precision and False Positive Analysis

We evaluate the precision of ISOLATOS compared to traditional memory-tracing approaches. Precision is a critical metric for vulnerability detection tools, as high false positive rates can significantly diminish their practical utility by overwhelming analysts with spurious reports. We launch test cases with identical configurations and execution parameters, analyzing the records that are identified as double-fetch bugs by each approach.

1) Experimental Design and Ground Truth Establishment: We define precision as the ratio of true positive detections (actual double-fetch vulnerabilities) to the total number of reported cases (true positives + false positives). To ensure a fair comparison, we executed identical test workloads across both ISOLATOS and QEMU-TCG, then applied three different time window thresholds to identify potential double fetches. In the final stage, we manually verified each reported vulnerability through code inspection and dynamic analysis.

The bugs reported by QEMU-TCG and ISOLATOS are shown in Table IV. Although QEMU-TCG reports more bugs than ISOLATOS, fewer true positives are identified by QEMU-TCG. There is one case (ECDSA sign) in QNX 6.6 QEMU-TCG that identifies more true positive cases than ISOLATOS. When we analyzed the root cause, we found that the OS happens to invoke a timer interrupt that also includes double-fetch bugs. ISOLATOS failed to register these because the timer interrupt was not invoked during its execution. Overall, a significant number of false positives are present in QEMU-TCG, with a 87.7% false positive rate under 0.1 and 1 ms time window. Note that we did not report the number of true positives in the 3 ms column because QEMU-TCG reported too many potential double-fetch bugs to manually verify within a reasonable time frame. Instead, we only report the number of false positives as we can verify those automatically. Further analysis of false positives/true positives can be found in Appendix A.

2) False Positive Taxonomies: Verifying each false-positive instance case-by-case would require significant manual effort. To identify the false-positive cases in QEMU-TCG, we classify false positives into 3 different categories. As shown in Table V, we have identified 2,167 false-positive cases, and classified them into these 3 categories. Meanwhile, there are also 170 cases where we could not determine whether it was a false-positive or true-positive.

Temporal False Positives (80.6%): To automatically identify these cases, we instrument pairing ID in the kernel entry of QEMU-TCG. Fixed time windows fail to respect system call boundaries. RTOS execution times vary dramatically (50-10,000+ cycles), causing unrelated memory accesses to be incorrectly correlated. Especially under high IPC related tasks,

TABLE VI: Vulnerabilities found by ISOLATOS.

ID	Syscall	Vulnerable variable	Impact	Severity	Status
QNX 6.6 & 7.0					
#1	ker_channel_create	kap→chid ;	DoS	Medium	CVE-2021-32025 (Fixed)
#2	ker_clock_time	*new* and *old* in ClockTime	DoS	Medium	CVE-2021-32025 (Fixed)
#3	ker_clock_adjust	*new* and *old* in ClockTime	DoS	Medium	CVE-2021-32025 (Fixed)
#4	ker_clock_period	*new* and *old* in ClockTime	DoS	Medium	CVE-2021-32025 (Fixed)
#5	ker_connect_attach	kap→pid / kap→chid	DoS	Medium	CVE-2021-32025 (Fixed)
#6	ker_connect_detach	kap→pid / kap→chid;	DoS	Medium	CVE-2021-32025 (Fixed)
#7	ker_connect_client_info	kap→pid / kap→chid	DoS	Medium	CVE-2021-32025 (Fixed)
#8	ker_connect_flags	kap→coid / kap→mask	DoS	Medium	CVE-2021-32025 (Fixed)
#9	ker_channel_create_attrs	kap→coid	DoS	Medium	CVE-2021-32025 (Fixed)
#10	ker_msg_sendv	kap→rparts / kap→rmsg	DoS	Medium	CVE-2021-32025 (Fixed)
#11	ker_msg_sendpulse	kap→coid / kap→priority	DoS	Medium	CVE-2021-32025 (Fixed)
#12	ker_msg_receivev	kap→rparts / kap→rmsg	DoS	Medium	CVE-2021-32025 (Fixed)
#13	ker_interrupt_attach	kap→flags	DoS	Medium	CVE-2021-32025 (Fixed)
#14	ker_interrupt_mask	kap→id	DoS	Medium	CVE-2021-32025 (Fixed)
#15	ker_interrupt_unmask	kap→id	DoS	Medium	CVE-2021-32025 (Fixed)
#16	ker_interrupt_query	kap→id / kap→type	DoS	Medium	CVE-2021-32025 (Fixed)
#17	ker_msg_current	kap→rcvid	DoS	Medium	CVE-2021-32025 (Fixed)
#18	ker_msg_readv	kap→rparts / kap→rmsg	DoS	Medium	CVE-2021-32025 (Fixed)
#19	ker_msg_writev	kap→rparts / kap→rmsg	LPE	High	CVE-2021-32025 (Fixed)
#20	ker_net_cred	kap→coid	DoS	Medium	CVE-2021-32025 (Fixed)
#21	ker_net_vtid	kap→info	DoS	Medium	CVE-2021-32025 (Fixed)
#22	ker_sched_set	kap→pid / kap→tid / kap→param	LPE	High	CVE-2021-32025 (Fixed)
#23	ker_signal_kill	kap→value	DoS	Medium	CVE-2021-32025 (Fixed)
#24	ker_sched_get	kap→pid / kap→tid / kap→param	LPE	High	CVE-2021-32025 (Fixed)
#25	ker_signal_fault	kap→sigcode	DoS	Medium	CVE-2021-32025 (Fixed)
#26	ker_signal_procmask	kap→sig	DoS	Medium	CVE-2021-32025 (Fixed)
#27	ker_signal_suspend	kap→sig_blocked	DoS	Medium	CVE-2021-32025 (Fixed)
#28	ker_signal_waitinfo	kap→sig_wait	DoS	Medium	CVE-2021-32025 (Fixed)
#29	ker_signal_return	kap→s→timeout_flags	DoS	Medium	CVE-2021-32025 (Fixed)
#30	ker_sync_create	kap→sync→__flags	DoS	Medium	CVE-2021-32025 (Fixed)
#31	ker_thread_create	kap→sync→__owner	LPE	High	CVE-2021-32025 (Fixed)
#32	ker_thread_destroy	kap→sync→__owner	LPE	High	CVE-2021-32025 (Fixed)
#33	ker_timer_create	kap→event	DoS	Medium	CVE-2021-32025 (Fixed)
#34	ker_timer_settime	kap→itime→interval_nsec	LPE	High	CVE-2021-32025 (Fixed)
#35	ker_timer_info	kap→param→_32 / kap→pid	DoS	Medium	CVE-2021-32025 (Fixed)
#36	ker_timer_alarm	kap→itime / kap→otime	DoS	Medium	CVE-2021-32025 (Fixed)
#37	ker_timer_timeout	kap→otime	DoS	Medium	CVE-2021-32025 (Fixed)
VxWorks 7					
#38	sysctl	VxWorks	Info leak	Low	CVE-2022-143544(Fixed)
#39	ioctl	VxWorks	Unknown	None	Confirmed & Ignored
#40	IPnet stack	VxWorks TCP/IP stack	Memory Corrupton RCE	High	Known issue
seL4					
#41	handleInvocation	seL4	/	None	Confirmed & Ignored
QNX 8					
#42	ker_msg_replyv	kap→smsg	Unknown	None	Confirmed & Ignored
#43	ker_msg_writev	kap→smsg	Unknown	None	Confirmed & Ignored

because there can be multiple processes or rapid kernel-user switches in some simple system calls.

Preemption False Positives (18.7%): High-priority thread preemption creates legitimate multi-access patterns that appear as vulnerabilities to time-window detectors. We observed 455 such cases in cryptographic operations where interrupt handlers access the same user buffers as preempted system calls. To automatically identify these cases, we also instrument pairing ID in the kernel entry of QEMU-TCG.

Uninitialized Memory (0.7%): There are memory accesses towards uninitialized addresses, which were not used any where else. This false-positive case has been considered into Bochspwn design as well. We also follow its design and spray the stack to detect this false positive case.

E. Discovery of Real-world Vulnerabilities

1) Real-world Vulnerabilities: Table VI presents a comprehensive overview of the vulnerabilities discovered by ISOLATOS. We discovered a total of 43 vulnerabilities across multiple commercial and open-source COTS RTOSes, demonstrating the effectiveness of our approach in identifying real-world security issues. The detail of ethical research is described in Appendix B.

QNX Neutrino RTOS. Most of our findings (37 vulnerabilities) were identified in QNX Neutrino RTOS versions 6.6 and 7.0, specifically in their system call handlers. These vulnerabilities predominantly involve double-fetch issues in pointer handling, where the kernel reads user-provided data more than once, creating race conditions. Most of these vulnerabilities (31) can lead to Denial of Service (DoS), while 6 can be exploited for Local Privilege Escalation (LPE). The 37 QNX vulnerabilities were assigned CVE-2021-32025 and have been fixed by the vendor in subsequent updates. There are 2 recently identified in QNX 8, whose impact are still under investigation.

VxWorks. In VxWorks, we identified 3 distinct vulnerabilities. The vulnerability in the IPnet TCP/IP stack (#40) was previously known but rediscovered by our tool. Additionally, we found an information leak through the `sysctl` interface (CVE-2022-143544) and an unspecified issue in the `ioctl` handler that has been confirmed by the vendor.

seL4 Microkernel. For the seL4 microkernel, we identified one vulnerability in the `handleInvocation` function. While this issue has been confirmed by the seL4 developers, it was assessed to have no direct security impact. After discussing with the project owner of seL4, we confirm that the bug exists in its debugging component, which is not an open service worth exploiting. Thus, this is a double-fetch bug, but it has no security impact.

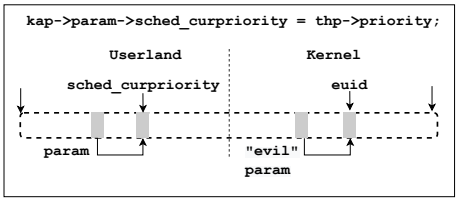
Severity Distribution. Of the 43 vulnerabilities, 5 were classified as high severity, 34 as medium severity, 1 as low severity, and the impact of the other 3 are still unknown. The high proportion of severe vulnerabilities highlights the critical importance of thorough testing of COTS RTOS kernels, especially in their system call interfaces. 1 of them was known before ISOLATOS's report.

2) Exploitation of a 19-year-old vulnerability: We exploit a real-world double-fetch vulnerability that poses substantial risks for COTS RTOSes, especially with an increasing number of them incorporating privilege management systems. For instance, we identify a vulnerability in a recent model of a renowned German vehicle utilizing QNX. Exploiting this vulnerability can break down the vehicle's system. However, in consideration of the confidentiality associated with the German car manufacturer, we highlight a different vulnerability (#24 in Table VI). This vulnerability in QNX can result in arbitrary read and write operations.

As depicted in Figure 7(a), the dereference `kap->param->sched_curpriority` at line 15 represents a sequence of two pointers. Here, `kap` points to data originating from user space. In scenarios involving double-fetch bugs, the second retrieval of `kap->param` is not constrained by the condition in line 10. Hence, a user thread has the freedom to assign any value to `kap->param`. This implies that the value `thp->priority` at line 15 can write to any specified memory address. As illustrated in Figure 7(b), such indiscriminate writing can pave the way for privilege escalation. An adversarial thread can manipulate the kernel into overwriting its *euid* with the value 0, simply by assigning the address of `sched_curpriority` to the *euid*'s address. Consequently, this thread gains root privileges, achieving a privilege escalation.

```
1  undefined4 ker_sched_get(int act,kerargs_sched_get *kap)
2  {
3      ...
4    //first fetch
5      psVar1 = kap->param;
6      ...
7      psVar2 = (sched_param_alltype *)
8         act->process->boundry_addr;
9    // check if kap->param in boundary
10   if (psVar2 < psVar1) {
11    return 0xe;
12     }
13     ...
14   //second fetch
15   kap->param->sched_curpriority = thp->priority;
16     ...
17 }
```

(a) Double-fetch in ker_sched_get function



(b) Arbitrary write exploitation in the code snippet

Fig. 7: Dereferencing a user pointer affected by double-fetch bug will cause arbitrary write.

The exploitation process requires precise timing to succeed. We developed an exploitation methodology that employs multiple worker threads to increase the probability of a successful race condition. Our timing analysis revealed that the critical time window between the first check and the second fetch is approximately 125 CPU cycles on modern QNX systems, requiring high-precision thread synchronization. Using thread prioritization and processor affinity settings, we achieved a reliable exploitation success rate of 76% after an average of 12 attempts. We then implemented a proof-of-concept that demonstrates the LPE attack.

VII. Related Work

None of the existing solutions are designed to detect double-fetch in COTS RTOSes. The previous tools can be categorized into two groups, *i.e.,* static and dynamic methods.

Static Analysis Tools. Coccinelle [11] and DEADLINE [10] are two of the static analysis tools. Static analysis tools perform static analysis on the intermediate representation (IR) compiled from source code and thus support multiple architectures. For example, Coccinelle [11] analyzes the IR code to detect double fetch bugs based on the code patterns that involve transfer functions. However, it is not accurate enough because simply adopting specific code patterns produces high false-positive and false-negative rates. For instance, Coccinelle cannot identify any patterns from the code in Figure 7 because

all the fetches in the code snippet directly dereference user pointers. Although DEADLINE [10] achieves high accuracy by using formal representation to precisely model a double-fetch bug as two accesses towards overlapped memory, it still relies on the identification of transfer functions and does not work well on non-Unix-like systems. Moreover, static analysis tools usually depend on source code and decompilation, hence they cannot be applied to binary code directly [26], [27]. One possible solution is to lift the binary code to IR first and then run these tools on the IR code [28], [29], [30]. However, the lifting process does not recover more semantic information than the binary code [31]. Thus, the limited semantic information in the lifted IR would decrease the effectiveness of static analysis.

Dynamic Tracing Tools. The dynamic tracing tool, Bochspwn [12], is developed based on full-system emulation. Bochspwn inspects the system's memory while executing and reports a double-fetch bug when detecting two memory reads accessing the same user-space address within a short time window. Although the dynamic tracing-based technique does not require source code, it usually generates a tremendous number of false positives. Moreover, Bochspwn is implemented on top of the Bochs emulator [32], which only supports IA-32/64(x86/64) architecture; thus, it does not support other architectures, such as ARM and MIPS, which are commonly adopted to run COTS RTOSes. DECAF [9] combines state-of-the-art cache side attacks with kernel-fuzzing to automatically identify double fetch bugs. However, the method relies on specific CPU design features to implement a side-channel method to detect them. If it needs to be migrated to a COTS RTOS with a different architecture, significant manual effort will be required.

VIII. Discussion

In this section, we discuss the limitations and scalability aspects of ISOLATOS. Furthermore, we outline potential avenues for enhancement and identify emerging research objectives to guide our future endeavors.

Limitation of ISOLATOS. One evident limitation of ISOLATOS lies in its dependency on documentation to invoke system calls. Not all system calls come with clear instructions on their usage. For instance, in VxWorks, we are unable to locate any documentation elucidating the application of $semGiveHardSc$ and $randNumGenCtlSc$. Additionally, there is an inherent need for manual intervention when formulating code to invoke these system calls. While tools like syzgen [33] are making strides in automating the generation of system calls, they are not entirely exempt from manual touch points. Moreover, ISOLATOS can be improved by substituting the JTAG-GDB fault recovery with an assembly code.

Future Work. Double-fetch bugs become inconsequential in systems that lack a distinct privilege demarcation across memory regions, such as in multi-threaded user programs or bare-metal systems. The manifestation of double-fetch bugs primarily occurs in systems that support memory isolation with a privilege model, coupled with multi-threading capabilities. As a result, platforms like TEE [34], [35] and hypervisors [36] are susceptible to double-fetch bugs, making them detectable by ISOLATOS. Furthermore, it is essential to pinpoint the boundaries of the privileged memory region and intercede at the interfaces responsible for data transfer between regions with varied privilege levels. Also, double-fetches can be introduced by the compiler [37], so it is also worth investigating whether COTS RTOSes are also affected. An intriguing avenue for future exploration is the mitigation of double-fetch bugs. For example, SafeFetch [38] has implemented an approach to mitigate that on Linux, which is promising for COTS RTOS.

IX. Conclusion

In this paper, we presented ISOLATOS, a novel hardware-assisted approach for quickly and accurately detecting double-fetch vulnerabilities in RTOS. ISOLATOS leverages hardware-based kernel isolation features to efficiently identify cross-boundary memory accesses indicative of double-fetch vulnerabilities. It then instruments a fault handler to recover from faults and record information. To determine a double-fetch, ISOLATOS introduces a lifecycle system to check if multiple fetches are from the same system call. Our evaluation across three major platforms, including QNX, VxWorks, and seL4, demonstrates the efficiency and accuracy of our approach. ISOLATOS successfully identified 43 unique vulnerabilities.

Acknowledgment

We thank the anonymous reviewers for their constructive comments. This work was partly supported by the Hong Kong RGC Project (PolyU15227825) and HK RGC Grant for Theme-based Research Scheme Project T43-513/23-N.

References

[1] B. Limited, "Blackberry qnx software is now embedded in over 195 million vehicles," https://www.prnewswire.com/news-releases/blackberry-qnx-software-is-now-embedded-in-over-195-million-vehicles-301315834.html, 2021, [Online; accessed October-2022].

[2] D. Z. Ben Seri, Gregory Vishnepolsky, "Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS," ARMIS.URGENT/11, Tech. Rep., 2019.

[3] "Blackberry software is now embedded in over 215 million vehicles," https://www.prnewswire.com/news-releases/blackberry-software-is-now-embedded-in-over-215-million-vehicles-301572840.html, 2022.

[4] "Companies using windriver vxworks," https://enlyft.com/tech/products/windriver-vxworks, 2022.

[5] P. Wang, K. Lu, G. Li, and X. Zhou, "A survey of the double-fetch vulnerabilities," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 6, p. e4345, 2018.

[6] "[patch-resend] backports: Fix double fetch in hlist_for_each_entry*_rcu," https://www.spinics.net/lists/backports/msg03072.html, 2014, Accessed: 2022-July.

[7] "Red hat bugzilla – bug 166248," https://bugzilla.redhat.com/show_bug.cgi?id=166248, 2005, Accessed: 2022-July.

[8] P. Wang, K. Lu, G. Li, and X. Zhou, "Dftracker: detecting double-fetch bugs by multi-taint parallel tracking," *Frontiers of Computer Science*, vol. 13, no. 2, pp. 247–263, 2019.

[9] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 587–600.

[10] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 661–678.
[11] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How Double-Fetch situations turn into Double-Fetch vulnerabilities: A study of double fetches in the linux kernel," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1–16.
[12] M. Jurczyk and G. Coldwind, "Identifying and exploiting windows kernel race conditions via memory access patterns," 2013.
[13] V. Duta, M. J. Aloserij, and C. Giuffrida, "{SafeFetch}: Practical {Double-Fetch} protection with {Kernel-Fetch} caching," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1207–1224.
[14] A. Bhattacharyya, U. Tesic, and M. Payer, "Midas: Systematic kernel tocttou protection," in *31st USENIX Security Symposium (USENIX Security 22)*, no. CONF, 2022.
[15] M. Corporation, "Cve-2008-2252: Windows kernel memory corruption vulnerability," https://nvd.nist.gov/vuln/detail/CVE-2008-2252, 2008, Accessed: 2024-November.
[16] NIST, "Cve-2016-6130: Race condition in the sclp_ctl_ioctl_sccb function in linux kernel," https://nvd.nist.gov/vuln/detail/CVE-2016-6130, 2016, race condition in the sclp_ctl_ioctl_sccb function in drivers/s390/char/sclp_ctl.c in the Linux kernel before 4.6 allows local users to obtain sensitive information from kernel memory by changing a certain length value, aka a "double fetch" vulnerability. Accessed: 2024-November.
[17] ——, "Cve-2016-6236: Denial of service vulnerability in dropbox lepton 1.0," https://nvd.nist.gov/vuln/detail/CVE-2016-6236, 2017, the setup_imginfo_jpg function in lepton/jpgcoder.cc in Dropbox lepton 1.0 allows remote attackers to cause a denial of service (out-of-bounds read) via a crafted jpeg file. NVD Published Date: 02/02/2017. Accessed: 2024-November.
[18] ——, "Cve-2016-6156: Race condition in the ec_device_ioctl_xcmd function in linux kernel," https://nvd.nist.gov/vuln/detail/CVE-2016-6156, 2016, race condition in the ec_device_ioctl_xcmd function in drivers/platform/chrome/cros_ec_dev.c in the Linux kernel before 4.7 allows local users to cause a denial of service (out-of-bounds array access) by changing a certain size value, aka a "double fetch" vulnerability. Published: 08/06/2016. Accessed: 2024-November.
[19] "Arm-IDA-And-Cross-Check-Reversing-The-787-Core-Network," 2019. [Online]. Available: https://i.blackhat.com/USA-19/Wednesday/us-19-Santamarta-Arm-IDA-And-Cross-Check-Reversing-The-787-Core-Network.pdf
[20] Intel, "Intel® Supervisor Mode Access Protection (SMAP)," https://cdrdv2.intel.com/v1/dl/getcontent/633935, 2021, [Online; accessed September-2022].
[21] "Learn the architecture - aarch64 memory model," https://developer.arm.com/documentation/102376/0100/Permissions-attributes, 2022, Accessed: 2022-July.
[22] qnx, "Priority inheritance," [Online; accessed October-2022]. [Online]. Available: http://www.qnx.com/developers/docs/7.0.0/index.html#com.qnx.doc.neutrino.getting_started/topic/s1_msg_prio_interitance.html
[23] "Deadlock," [Online; accessed October-2022]. [Online]. Available: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html
[24] "Compare the Top Real-Time Operating Systems (RTOS) of 2022," https://sourceforge.net/software/real-time-operating-systems-rtos/?sort=rating_count, [Online; accessed Nov-2022].
[25] J. T. Taylor and W. T. Taylor, "Board support package," in *The Embedded Project Cookbook: A Step-by-Step Guide for Microcontroller Projects*. Springer, 2024, pp. 213–224.
[26] I. U. Haq and J. Caballero, "A survey of binary code similarity," *Acm computing surveys (csur)*, vol. 54, no. 3, pp. 1–38, 2021.
[27] J. Park, S. Lee, J. Hong, and S. Ryu, "Static analysis of jni programs via binary decompilation," *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3089–3105, 2023.
[28] Z. Liu, Y. Yuan, S. Wang, and Y. Bao, "Sok: Demystifying binary lifters through the lens of downstream applications," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1100–1119.
[29] Q. Zhan, X. Hu, X. Xia, and S. Li, "React: Ir-level patch presence test for binary," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 381–392.
[30] T. Toor, "Decompilation of binaries into llvm ir for automated analysis," Master's thesis, University of Waterloo, 2022.
[31] A. Spanier and W. Mahoney, "Static analysis using intermediate representations: A literature review." Academic Conferences and publishing limited, 2023.
[32] "THE CROSS PLATFORM IA-32 EMULATOR," https://bochs.sourceforge.io/, [Online; accessed September-2022].
[33] B. Lenard, J. Wagner, A. Rasin, and J. Grier, "Sysgen: System state corpus generator," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES '20. New York, NY, USA: Association for Computing Machinery, 2020.
[34] T. Cloosters, M. Rodler, and L. Davi, "TeeRex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 841–858. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters
[35] S. Han and J. Jang, "Mytee: Own the trusted execution environment on embedded devices," 01 2023.
[36] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing." New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3460120.3484811
[37] J. Xu, L. Di Bartolomeo, F. Toffalini, B. Mao, and M. Payer, "Warpattack: bypassing cfi through compiler-introduced double-fetches," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1271–1288.
[38] V. Duta, M. J. Aloserij, and C. Giuffrida, "SafeFetch: Practical Double-Fetch protection with Kernel-Fetch caching," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1207–1224. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/duta

APPENDIX

A. Further Analysis of False Positives

In Figure 8 we show that among all the page faults ISOLATOS caught, around 30% of them trigger SMAP, indicating user-memory access. In addition, double-fetch bugs only represent 7.6% on average. This demonstrates that user memory accesses in the system are quite common, which is a critical reason why Bochspwn has such high false positives.

We also noticed that the most common cases that cause false positives are the "small" system calls, which have fewer instructions. As shown in Table IV, the larger the time window, the more false-positives can be found. This is because in a preset, fixed time window one can predetermine the instructions that are going to be executed. While the number of instructions varies for different system calls. We calculated that there are 417229 instructions executed per second using the execution logs, which mean in 1 ms there are roughly 417 instructions executed. When we randomly picked 10 system calls, we found that 9 of them have more than 417 instructions.

The kernel isolation trigger provides valuable insights into the kernel's interaction with user memory. We observed consistent patterns in which certain types of system calls, particularly those that handle data transfer between user and kernel space, showed higher trigger rates. I/O-related system calls, including ChannelDestroy(), MsgSendv(), exhibited SMAP trigger rates up to 45%, while memory management calls showed rates around 25%.

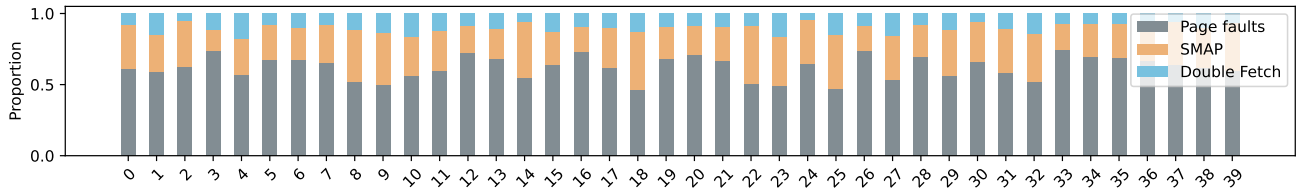


Fig. 8: The proportion of Double Fetch, triggering SMAP among all page faults. The left column for each OS represents the percentage of page faults that trigger SMAP protections (approximately 30% on average), while the right column represents the proportion of actual double-fetch bugs (7.6% on average) among all monitored memory accesses. This significant difference explains the high false positive rates in previous detection approaches.

Algorithm 1: Double-Fetch Detection via Kernel Isolation.

Input : Kernel Binary K, System Call Test Suite T
Output: Set of detected double-fetch vulnerabilities V

```
// Static Kernel Entry Identification
Function IdentifyKernelEntry(K):
    Initialize entry points set E ← ∅;
    foreach instruction inst in K do
        if inst sets kernel entry register then
            Extract entry address from inst;
            Add entry address to E;
    return E;

// Syscall Handler Instrumentation
Function InstrumentHandler(E):
    foreach entry e in E do
        EnableKernelIsolation();
        InsSysIdGenerator(e);
        ExitSysIdGenerator();

// Page Fault Handler with System Call Context
Function PageFaultHandler():
    if page fault == Error_isolation then
        addr_target ← fault address;
        addr_inst ← instruction pointer;
        id_syscall ← current system call ID;
        // Record access for analysis
        AccessMonitor(addr_target, addr_inst, id_syscall);
        // Recovery mechanism
        DisableKernelIsolation();
        ExecuteFaultInstruction();
        EnableKernelIsolation();
        JumpToNextInstruction();

// Access Monitoring with System Call Tracking
// id_syscall indicates pairing ID
Global AccessLog ← ∅;
Function AccessMonitor(addr_target, addr_inst, id_syscall):
    Record access: (addr_target, addr_inst, id_syscall) → AccessLog;

// Double-Fetch Pattern Analysis
Function AnalyzeDoubleFetchPattern(AccessLog):
    V ← ∅;
    Group AccessLog by id_syscall;
    foreach system call group G do
        foreach unique addr_target in G do
            count ← number of accesses to addr_target;
            if count ≥ 2 then
                Add (addr_target, G.id_syscall) to V;
    return V;

// Main Detection Process
E ← IdentifyKernelEntry(K);
InstrumentHandler(E);
Install PageFaultHandler();
EnableKernelIsolation();
// Execute test suite
foreach test t in T do
    Execute t;
// Analyze collected data
V ← AnalyzeDoubleFetchPattern(AccessLog);
return V;
```

B. Ethical Security Research

All the vulnerabilities are reported to the OS companies, and most of them are fixed. Those that did not introduce an impact were ignored by their security team. For Blackberry, considering the complexity of automotive and energy industry, we negotiated that the publicly disclosure time will be 150 days after the first report email sent. They notified us that they had pushes patches to all their customers. VxWorks also released the patch in 90 days. At the time of the paper submission, there are still two vulnerabilities reported but not yet fixed. We did not disclose any details of their exploitation in this paper to prevent malicious use.

C. Detecting Double-Fetch Bugs

To determine if there are any double-fetch bugs, we check if there are any two fetches accessing the same memory address between the life cycle of current system call. As shown in Algorithm 1, we first perform kernel entry identification, which iterates over the binary looking for kernel entry identifiers before inserting a handler that establishes isolation and system call ID registration necessary for boundary tracking, and then install a page fault handler responsible for recording the faulting address its instruction pointer and system call ID before calling `AccessMonitor`. In `AccessMonitor`, we record every cross-boundary access, with its targeted access address and its instruction address. To implement *Access Monitor*, we initialize a global variable `V` as a list of structures with two members `addr_i` and `addr_p` that we then check to determine if a double-fetch vulnerability exists. We use `AnalyzeDoubleFetchPattern` to determine if there are multiple accesses, whereupon they will be recorded to a double-fetch vulnerabilities set V.

D. Architecture-Specific Implementation

To implement the design for COTS RTOSes, we utilize various features provided by our target systems, including QNX, VxWorks, and SeL4 independently. An example of this process for QNX is given in Figure 5. We note, however, that the majority of ISOLATOS is target-independent as shown in Table VII. Only control of the user/kernel isolation bit, discovery of kernel entry, and page fault handling are specific to the architecture and require a one-time effort. However, the main body of ISOLATOS, is reusable across targets.

TABLE VII: Implementation Comparison between ARM and Intel x86-64 Architectures.

Component	Intel x86-64	ARM (v8.1+)
Architecture-Specific Differences		
Isolation Control	• SMAP: CR4 register bit 21	• PAN: PSTATE register bit 22
Kernel Entry Discovery	• Dynamic: WRMSR instruction • Entry stored in: MSR register `0xC0000082`	• Fixed: Exception vector table • SVC handler at fixed address
Page Fault Handling	• Exception vector: #PF (0x0E) • Fault address: CR2 register • Error code: On stack • Handler: `trap0e` function	• Exception vector: Data Abort • Fault address: FAR_EL1 register • Syndrome: ESR_EL1 register • Handler: `el1_da` function
Architecture-Independent Components		
System Call Pairing	• Unique ID generation using timestamp or counter • Stored in kernel memory region accessible to fault handler • Maintained throughout system call lifecycle • Cleared on system call exit	
Fault Recovery	• JTAG-based debugging interface for both architectures • GDB connection through hardware debug adapter • Hardware breakpoints at recovery points • Automated script execution for state restoration	
Access Recording	• Global data structure: `AccessLog` • Record format: `(target_addr, inst_addr, syscall_pairing_id)` • Lock-free implementation for multi-CPU support • Fixed-size circular buffer to prevent overflow	
Instruction Completion	• Temporary isolation disable before execution • Single-step execution of faulting instruction • Immediate re-enable of isolation • Return to next instruction address	