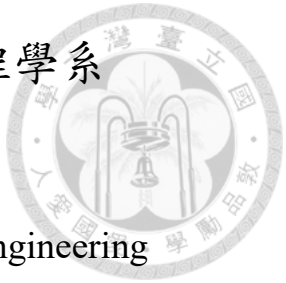


國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

透過基於時間間隔的監測器恢復內核模組錯誤

Kernel Modules Fault Recovery by a Time Interval-based Monitor

李宥霆

You-Ting Li

指導教授: 黎士瑋 博士

Advisor: Shih-Wei Li, Ph.D.

中華民國 113 年 9 月

September 2024

doi:10.6342/NTU202404466

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

透過基於時間間隔的監測器恢復內核模組錯誤

Kernel Modules Fault Recovery by a Time Interval-based Monitor

本論文係李宥霆君（學號R11922002）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國113年9月24日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 24 September 2024 have examined a Master's thesis entitled above presented by YOU-TING LI (student ID: R11922002) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

駱士瑋	陳郁方	蕭旭君
（指導教授Advisor）		

系主任/所長Director: 陳祝嵩

doi:10.6342/NTU202404466

doi:10.6342/NTU202404466

摘要

作業系統核心的隔離對於系統本身安全的保護有著極大的好處。截至 2023 年為止，Linux Kernel 的行數已超過三千六百萬行程式碼，龐大的代碼量對程式除錯的難度日益增加，系統核心工程師不經意的錯誤可能會導致核心毀損而 panic，如 null pointer deference、use after free。因此，我們在 Arm v8 的架構下提出了一套備份機制，首先會選定其中一種 system call，將其定義為我們的 compartment，並藉由分析該 compartment 的 call graph，找出與核心其他部份 share 的 global memory。針對這些記憶體內容，我們實作了一套 Time Interval Based 的 Monitor，他會紀錄並備份所有 process 對該記憶體位置的更改情況。當隔離區域程式碼遇到錯誤時，先透過 hypercall 進入 EL2 回復暫存器的狀態，再讓 monitor 回復共享記憶體的內容，最後讓 process 回到進入 system call 前的系統狀態，並返回錯誤代碼，以讓使用者有機會處理核心錯誤，藉此達到系統核心保護及恢復的目的。

關鍵字：系統安全、系統核心隔離、核心可用性

doi:10.6342/NTU202404466

doi:10.6342/NTU202404466

Abstract

The isolation of the operating system kernel is of great benefit to the security of the system itself. As of 2023, the Linux Kernel will have more than 36 million lines of code, and the huge amount of code will make it increasingly difficult to debug the program, and inadvertent mistakes made by the kernel engineers may cause the kernel to be corrupted and become panic, e.g., null pointer deference, use after free. Therefore, we propose a backup mechanism in the Arm v8 architecture, trying to prevent kernel crash after these mistakes causes kernel errors. Firstly, we will select one of the system calls, which is defined as our compartment, and analyze the control flow graph of the compartment to find out the global variables shared outside of the compartment. And then, we design a Time Interval Based Monitor, which can record and back up all value changes to the memory addresses. When the compartment encounters an error, the monitor can recover tracked memory locations, and allows the process to return to the original system state before entering the compartment, thus achieving the purpose of protecting the system kernel and

doi:10.6342/NTU202404466

maintaining kernel availability.

Keywords: System Security, System Kernel Isolation, Kernel Availability

doi:10.6342/NTU202404466

Contents

doi:10.6342/NTU202404466

doi:10.6342/NTU202404466

doi:10.6342/NTU202404466

doi:10.6342/NTU202404466

List of Figures

doi:10.6342/NTU202404466

doi:10.6342/NTU202404466

List of Tables

doi:10.6342/NTU202404466

doi:10.6342/NTU202404466

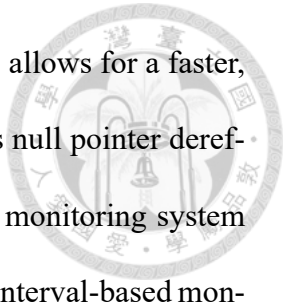
Chapter 1 Introduction

In modern operating systems, ensuring the availability of system kernels has become critical due to the increasing number of vulnerabilities and potential attack vectors. The Linux kernel, with its vast codebase, faces a constant influx of vulnerabilities, particularly those related to memory management, such as errors that trigger exceptions and cause the CPU to enter an exception handler. These vulnerabilities can lead to memory corruption, posing significant risks to system stability and availability. Without effective recovery mechanisms, these errors often result in kernel crashes and extended system downtime.

A widely adopted approach for handling such issues in system recovery is checkpointing, which captures the entire system state at specific intervals to enable recovery when a failure occurs. However, existing checkpointing solutions are generally coarse-grained, targeting recovery at the user level [5, 11, 18, 22] or virtual machine (VM) level [3, 12]. These solutions aim to restore the entire system / process to a previous state after an error, which may not be suitable for more frequent or fine-grained failures like system call errors in the kernel. For example, works such as QEMU snapshots [1, 14–16] offers recovery mechanisms for virtualized environments but do not provide the granularity necessary for efficient recovery at the kernel system call.

Our design offers a more fine-grained approach by focusing on recovering system

doi:10.6342/NTU202404466



calls within the kernel rather than entire systems or applications. This allows for a faster, more targeted recovery process, specifically addressing errors such as null pointer dereferences and use-after-free vulnerabilities that trigger exceptions. By monitoring system calls and saving memory and register states at critical points, our time-interval-based monitor allows for efficient recovery without needing to revert the entire kernel or application state.

When compared to other checkpointing approaches such as CRAK [22], which provides transparent checkpointing and restart capabilities at the application level, and Kckpt [5], which operates at the user-space level, our design is more fine-grained and specifically targets system call recovery within the kernel. CRAK focuses on process migration and provides checkpointing for networked applications, allowing processes to resume even after failure. However, it does not focus on kernel-level errors that trigger exceptions, which are common causes of system failures. Similarly, Kckpt is designed for user-space checkpointing in the UnixWare kernel and captures user-space information, but it does not extend to kernel space, limiting its applicability in protecting the kernel from memory corruption.

In contrast, our work extends these concepts by applying recovery mechanisms at the system call level within the kernel itself, providing an even finer granularity of protection. Unlike CRAK and Kckpt, which focus on capturing the state of user-level processes and user-space data, our approach targets memory corruption vulnerabilities in the kernel and ensures the availability and stability of the system without needing to revert the entire user-space or virtual machine state.

Furthermore, ARM TrustZone-based solutions [4, 19, 20] focus on providing recov-

doi:10.6342/NTU202404466

ery by isolating and protecting trusted execution environments (TEEs). These approaches, while effective, offer recovery mechanisms at a higher granularity, focusing on entire components or systems rather than individual system calls.

By addressing the granularity gap between existing coarse-grained checkpointing methods and the fine-grained system call recovery approach presented in this thesis, our work demonstrates an innovative approach to system call level protection. This makes our solution particularly effective in recovering from memory corruption issues, ensuring system availability without incurring the overhead of rolling back the entire system state.

In summary, this thesis makes the following contributions:

1. We propose a fine-grained recovery mechanism for mitigating memory corruption vulnerabilities at the system call level, focusing on errors that cause the CPU to enter the exception handler.

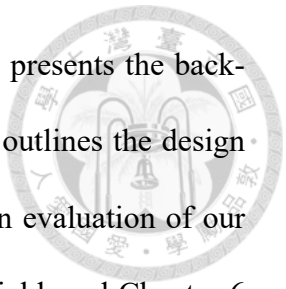
2. We introduce a time-interval-based monitoring system that enables efficient recovery from these errors while minimizing performance overhead.

3. We compare our fine-grained approach with existing checkpointing methods that target recovery at broader levels, such as VMs or applications, and demonstrate how our solution provides more targeted recovery.

4. We evaluate the performance of our prototype on the SeKVM architecture, demonstrating its effectiveness in maintaining system availability.

5. We outline potential future work to integrate our solution with other protection mechanisms, such as ARM’ s MTE and KASAN, to further enhance kernel security.

doi:10.6342/NTU202404466



The remainder of this thesis is organized as follows: Chapter 2 presents the background and related work in kernel recovery mechanisms. Chapter 3 outlines the design of our time-interval-based monitoring system. Chapter 4 provides an evaluation of our prototype implementation. Chapter 5 discusses related work in the field, and Chapter 6 concludes the thesis with key findings and contributions.

doi:10.6342/NTU202404466

Chapter 2 Background

2.1 Overview of the ARM Architecture

Our work is discussed based on ARM architecture. The ARM architecture is a family of Reduced Instruction Set Computing (RISC) architectures. It is widely used in a variety of applications, including embedded systems, mobile devices, and increasingly in servers and high-performance computing due to its power efficiency and performance characteristics. ARM processors use a simplified instruction set that enables higher performance and more efficient instruction execution compared to Complex Instruction Set Computing (CISC) architectures like x86. It provides both 32-bit (ARMv7) and 64-bit (ARMv8) architectures, supporting a wide range of applications from low-power embedded devices to high-performance computing.

2.1.1 Exception Level

The ARMv8 architecture introduces a hierarchical and structured exception handling mechanism categorized into four distinct exception levels (ELs). These levels provide different privileges and control over the system, enhancing security and separation between various execution environments. EL0 (User Level) is the least privileged level. It exe-

doi:10.6342/NTU202404466

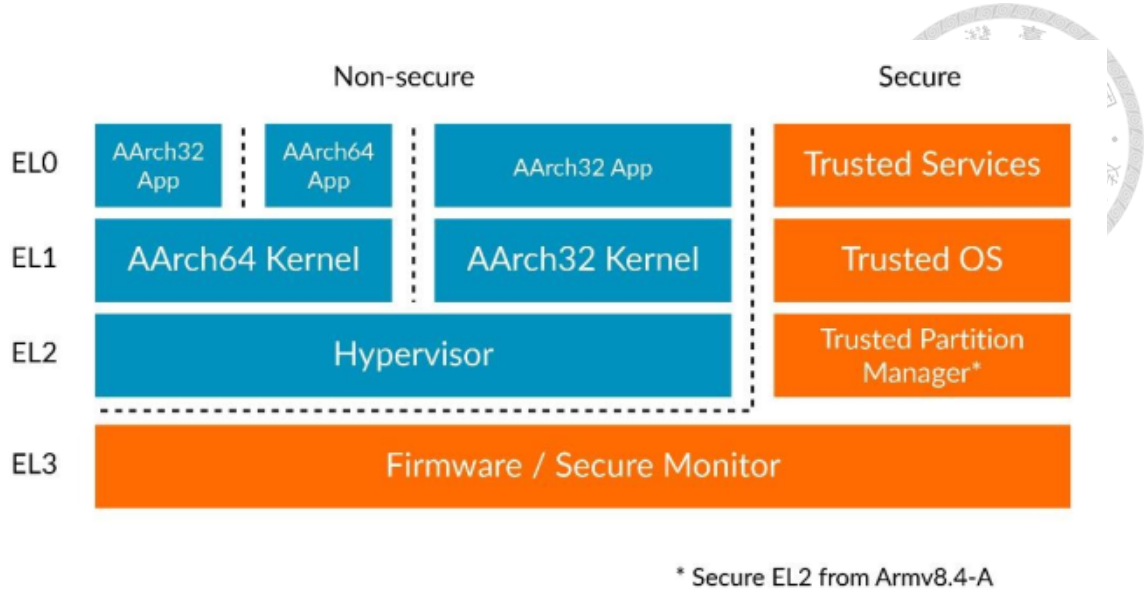


Figure 2.1: Arm Exception Level

cutes user applications and cannot access most system resources directly; EL1 (Kernel/ Operating System) executes operating system kernel. It manages resources and hardware directly, and can handle system calls from and exceptions from EL0/EL1; EL2 (Hypervisor) manages virtualization. It controls and monitors virtual machines running at EL1, and provides isolation between virtual machines; Last, EL3 (Secure Monitor) is the most privileged level. It executes secure monitor code and manages transitions between secure and non-secure states (Trusted Execution Environment). To advance to higher levels, you need to use exceptions (e.g., interruptions, page faults, etc.).

2.1.2 Key Registers

ARM architecture defines a variety of registers to manage its operations:

- General Purpose Registers. There are set of registers (e.g., R0-R15 in ARMv7, X0-X30 in ARMv8) for general computational purposes.
- Program Counter (PC). It holds the address of the next instruction to be executed.

doi:10.6342/NTU202404466

- Stack Pointer (SP). It points to the current stack location, separate for each exception level.
- Link Register (LR). It stores the return address for subroutine calls.
- Program Status Registers (CPSR/SPSR). CPSR (Current Program Status Register) holds the current state of the processor, while SPSR (Saved Program Status Register) saves the processor state when an exception is taken.
- Exception Link Registers (ELR_ELx). It holds the address to return to after handling an exception.
- Memory Management Registers (TTBR, TCR, etc.). They control memory translation and attributes for different memory regions.

2.1.3 ARM TrustZone

ARM TrustZone is a security technology that creates a secure execution environment within ARM processors by dividing the system into two distinct worlds: the Secure World and the Non-Secure World. The Secure World handles sensitive operations and data, while the Non-Secure World runs regular applications and the operating system. TrustZone provides hardware-enforced isolation between these worlds, ensuring that critical data in the Secure World cannot be accessed by the Non-Secure World.

Key components of TrustZone include the Secure Monitor, which manages transitions between the two worlds, the TrustZone Address Space Controller (TZASC) for secure memory management, and the TrustZone Protection Controller (TZPC) for managing peripheral access. TrustZone supports secure boot processes, digital rights management

doi:10.6342/NTU202404466

(DRM), secure payment systems, and secure authentication and biometrics, enhancing security across a wide range of applications.

By isolating sensitive operations and data, TrustZone significantly reduces the attack surface, enhancing overall system security. Its flexibility and compatibility with existing ARM architectures make it a valuable security solution for devices ranging from smartphones to IoT devices. TrustZone ensures robust protection for sensitive operations, making it an essential technology for modern secure computing environments.

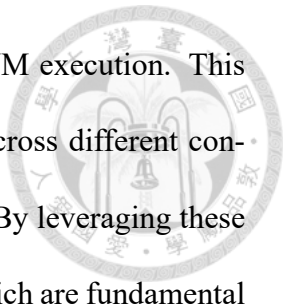
2.2 SeKVM

This thesis builds upon SeKVM, a formally verified architecture built on the Linux KVM hypervisor, enhances security by isolating memory address spaces and providing several critical register copying functions within the EL2 (Exception Level 2) privilege mode.

Memory isolation in SeKVM is achieved by mapping the hypervisor's critical components, such as the monitor's data and text sections, into an isolated memory address space that is separate from the general kernel and user space. This isolation is enforced by the hypervisor at EL2, which has higher privileges than the regular operating system running at EL1. By segregating memory in this manner, SeKVM ensures that sensitive data and execution contexts are protected from potential attacks originating from lower privilege levels, such as user space processes or even the kernel itself.

SeKVM also includes register copying functions within EL2, which are crucial for securely managing the state of the virtual machines (VMs). When a VM needs to be paused, resumed, or recovered from an error, these functions allow for the safe copying

doi:10.6342/NTU202404466



and restoration of the system registers, including those critical for VM execution. This mechanism is essential for maintaining the integrity of VM states across different contexts, such as during VM migration, snapshotting, or error recovery. By leveraging these features, SeKVM provides robust isolation and state management, which are fundamental for building secure multiprocessor environments.

2.3 LLVM Intermediate Representation (LLVM IR)

LLVM Intermediate Representation (LLVM IR) is a low-level programming language that serves as an intermediate step in the compilation process within the LLVM compiler framework. It is designed to be both human-readable and machine-independent, providing a common platform for optimizing code and supporting multiple target architectures. LLVM IR operates at a level of abstraction between high-level source code and machine code, making it suitable for various types of analysis and transformation during compilation.

One of the key advantages of LLVM IR is its flexibility in representing complex program structures, such as loops and function calls, while still being close enough to the hardware to allow for detailed optimization. This makes it an ideal choice for instrumenting code, as it allows for precise control over how operations are translated and executed on different hardware platforms. Additionally, the modularity of LLVM IR facilitates the development of custom optimization passes, enabling developers to insert, modify, or analyze specific instructions at this intermediate stage of compilation.

In the context of our project, we leverage LLVM IR to automate the instrumentation of our monitor API, allowing us to efficiently manage memory and process states within

doi:10.6342/NTU202404466

a compartmentalized environment.

2.4 Instrumentation Granularity for Monitor API

In our project, we explore different levels of instruction instrumentation granularity to automate the integration of our monitor API. The primary levels of granularity considered are *Function Level Granularity*, *Assembly Level Granularity*, and *LLVM IR Level Granularity*.

2.4.1 Function Level Granularity

At the function level, we utilize *error-handling functions* to restore the kernel state to a stable condition. In kernel programming, system developers typically implement error-handling paths within functions that may fail, such as those involving `kmalloc`. The advantage of using function-level granularity is that error-handling functions are generally straightforward to read and understand. However, a significant drawback is the need to customize each error-handling function to correspond to its respective original function, which complicates the automation of the instruction replacement process.

2.4.2 Assembly Level Granularity

In the paper Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing [21], the authors describe using a compiler to instrument assembly code during compile time, shifting memory addresses to the target address space and isolating compartment memory from the kernel address space. Assembly-level granularity facilitates automating the instruction replacement process. However, it presents challenges in

doi:10.6342/NTU202404466

distinguishing whether memory addresses are within the compartment.

2.4.3 LLVM IR Level Granularity

Using LLVM IR-level granularity offers several benefits. It simplifies the automation of the instruction instrumentation process by adding an LLVM optimization pass to instrument LLVM IR nodes. Additionally, it is easier to distinguish whether memory addresses are within the compartment, as the readability of the code is retained.

In our project, we ultimately selected LLVM IR-level granularity for instrumentation. This choice facilitates easier automation and provides a better distinction of memory addresses within compartments.

2.5 Control Flow Path

In operating systems, particularly in the Linux kernel, a control flow path refers to the sequence of execution that a process or thread follows during its runtime. This sequence includes the series of function calls, interrupts, and system calls made by a process while interacting with kernel services. The control flow path is essential in determining how the kernel handles various operations, such as memory management, I/O operations, and process scheduling.

When a process executes a system call, it transitions from user mode to kernel mode, where the kernel takes control to perform privileged operations on behalf of the process. The control path in the kernel begins when the process issues a system call, invoking a handler that manages the system request. This handler passes through various kernel

doi:10.6342/NTU202404466

subsystems, such as memory management or device drivers, depending on the nature of the request, and finally returns the result back to the process.

Understanding the control flow path is crucial in designing security mechanisms, as it helps identify critical points where errors or vulnerabilities that trigger exceptions and cause the CPU to enter an exception handler may occur. By monitoring and analyzing the control flow path, it is possible to instrument checks and recovery mechanisms that can prevent the kernel from crashing or being exploited.

doi:10.6342/NTU202404466

Chapter 3 Overview

The motivation for this work stems from the limitations of existing kernel recovery mechanisms, which are either too coarse-grained (e.g., checkpointing entire applications/ VMs) or only provide isolation without recovery. Our goal is to develop a fine-grained recovery system that operates at the system call level, enabling the kernel to recover from errors that trigger exceptions and cause the CPU to enter an exception handler like null pointer dereferences or use-after-free without rolling back the entire system state. This approach enhances system availability by minimizing downtime and ensuring that errors are contained within the kernel, rather than escalating to broader system failures.

Our time interval-based monitor isolates and tracks value changes for memory addresses within the compartment by instrumenting our provided monitor API and copying values to an isolated memory address space. A compartment is a set of functions for the kernel to execute to handle the system call. For instance, in Figure 3.2, function 1 and function 3 are included in our compartment, whereas function 2 is not.

Figure 3.1 illustrates the architecture of the time interval-based monitor. The green section represents our design components, including register/memory backup space, shared data space, and the monitor APIs. The register/memory backup space is a domain in isolated memory space that saves a copy of the register/memory state. The shared data space

doi:10.6342/NTU202404466

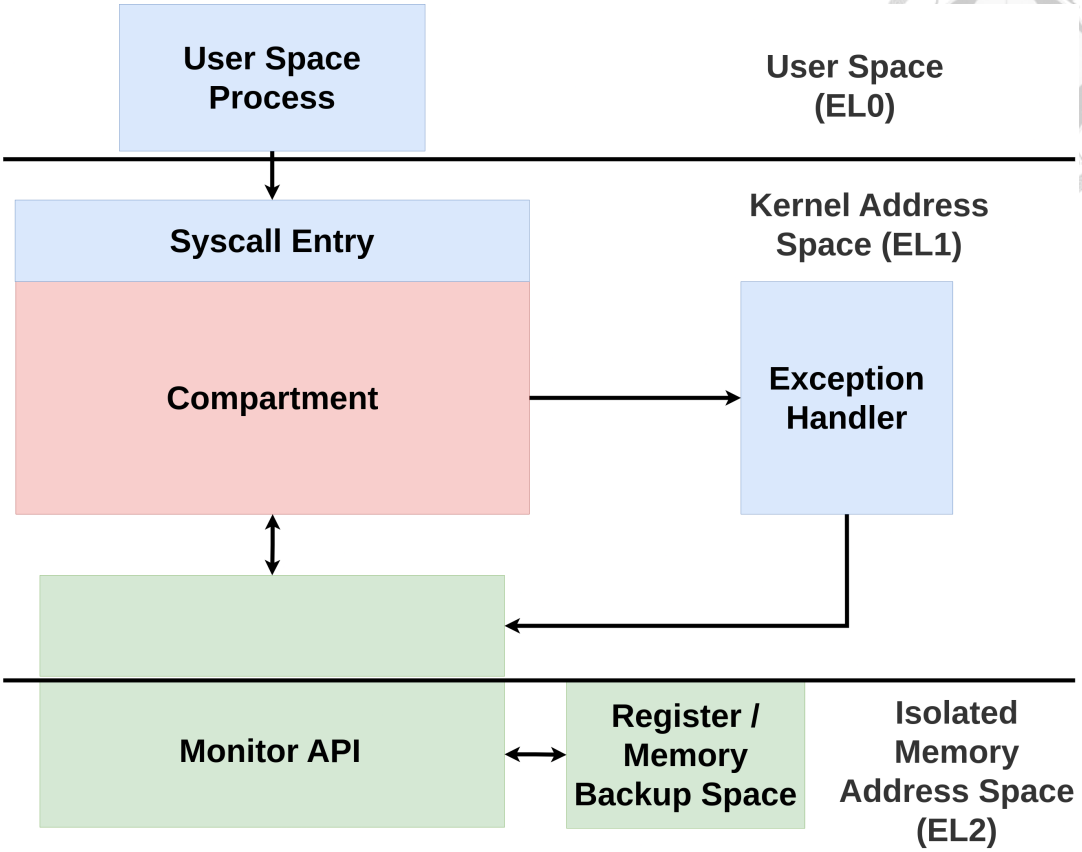


Figure 3.1: Architecture

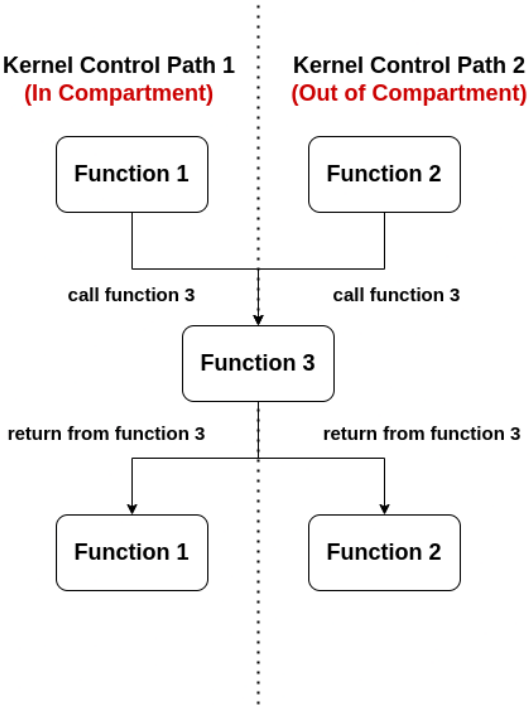


Figure 3.2: Compartment definition

doi:10.6342/NTU202404466

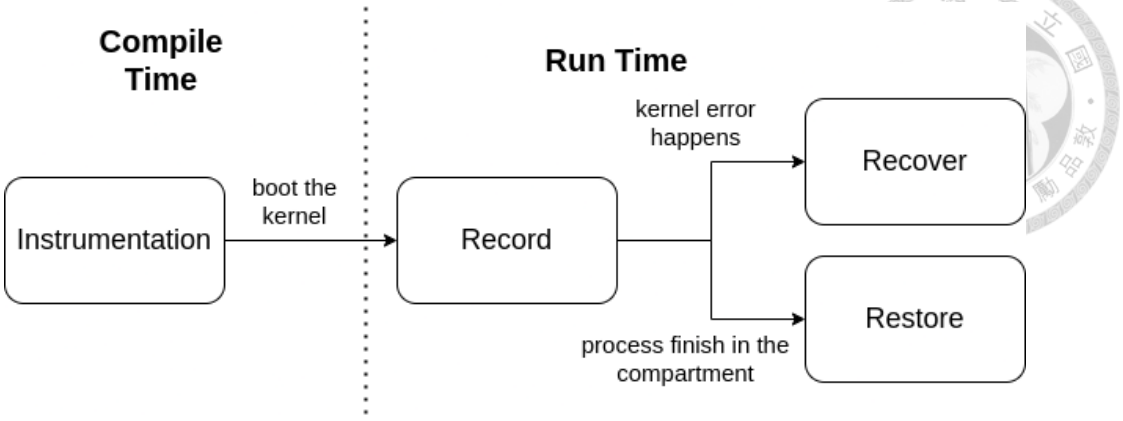
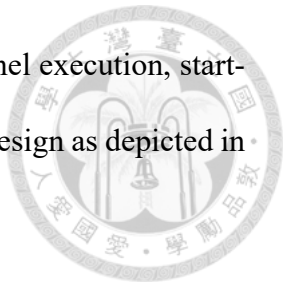


Figure 3.3: Compartment Workflow

is accessible by both the kernel and the monitor. The monitor APIs are entry points in the kernel address space for the compartment to access the register/memory backup space. Within the kernel address space, monitor APIs will call certain kernel APIs not supported in isolated memory address space (EL2), including kernel synchronization primitives APIs like `mutex_lock`/`mutex_unlock`, `spin_lock`/`spin_unlock`, and kernel memory management APIs like `kmalloc`/`kfree`. These APIs facilitate the functionality of copying memory state/register state to the register/memory backup space and putting states back to kernel address space when error occurs. The red section represents the compartment, which might contain buggy kernel code. For the red and blue sections, we instrument our monitor API.

When traversing a control path within the compartment, various types of data are accessed, including global shared data, global locks, and local data. Global shared data includes memory allocated before entering the compartment, such as function parameters and global variables. Global locks are synchronization primitives used in the compartment, including `mutex`, `spin_lock`, `mmap_lock`, and `rcu_lock`. Local data refers to memory allocated within the compartment, including function stacks and dynamically allocated memory by kernel memory allocation APIs [7].

doi:10.6342/NTU202404466



To summarize our design workflow from instrumentation to kernel execution, starting from the original Linux kernel, there are three main stages in the design as depicted in Figure 3.3:

1. Instrumentation Phase: Instrument the original load/store memory instructions on data that need to be backed up to our provided monitor APIs.

2. Recording Phase: Processes save a copy of variable values in the compartment to the memory backup space during runtime before an error occurs. In this stage, we aims to preserve the current state of critical variables in this phase, enabling recovery in case of errors.

3. Recovery / Restoration Phase: In the recovery phase, the kernel register state saved before entering the compartment path is copied from the memory backup space to the kernel address space by the process after an error occurs. In the restoration phase, the memory state is copied from the memory backup space to the kernel address space to restore the system and continue kernel execution. This phase aims to restore the kernel to its previous stable state, minimizing the impact of errors that trigger exceptions and cause the CPU to enter an exception handler.

doi:10.6342/NTU202404466

Chapter 4 Threat Model

In this work, we assume that an attacker can exploit vulnerabilities within a compartment of the kernel, specifically targeting errors that trigger exceptions and cause the CPU to enter an exception handler, including overflow to access unmapped memory or access memory with incorrect permission, use-after-free that cause crashes, divide-by-zero, stack overflow and NULL pointer dereferences. Our Trusted Computing Base (TCB) includes the parts of the kernel outside the compartment, the monitor, and the register/memory backup space. We assume that the memory regions used by these components, depicted as grey regions in Figure 4.1, remain secure and unmodified by attackers.

Attackers can not hijack the kernel's control flow by manipulating function pointers or employing code-reuse attacks such as **Return-Oriented Programming (ROP)** and **Jump-Oriented Programming (JOP)** attacks in our work. However, we assume there might contain vulnerabilities that will not break the original kernel control flow in the compartment.

We also assume that programmers will hold the respective locks when accessing global variables. We expect programmers will replace the global shared variable referenced in every function with our monitor API. If there are instances where the global variable is not tagged, we will be unable to track and record the changes, thus preventing

doi:10.6342/NTU202404466

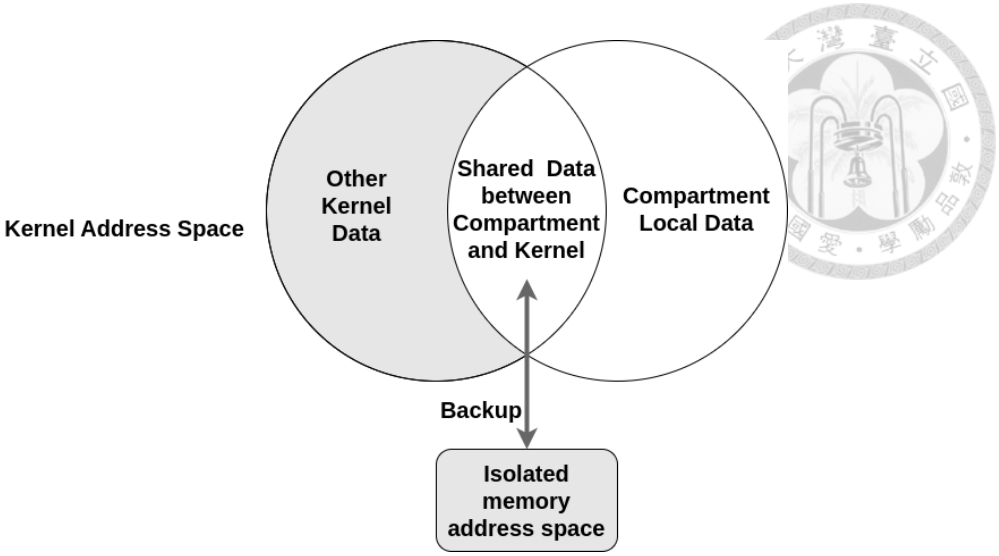


Figure 4.1: Memory Layout

recovery of the global data.

Furthermore, our framework does not address attacks aimed at creating system deadlocks. So exploiting attack to kernel and causing deadlock, such as ROP attack to a critical section, is out of scope. Lastly, we assume the system remains uncompromised during the boot process.

doi:10.6342/NTU202404466

Chapter 5 Design

Our objective is to maintain kernel availability in the event of errors that trigger exceptions and cause the CPU to enter an exception handler. If an error occurs in the compartment, the kernel can be restored to its previous state before the process entered the compartment. The system then informs the user that the system call encountered an error, allowing the user to decide whether to re-execute the system call.

During execution within the compartment, the process performs operations such as loading states from memory and registers, modifying these states, and writing the updated states back to memory and registers. These operations affect both memory and register states. To ensure proper recovery in the event of an error within the compartment, it is essential to save a copy of the memory and register states that the process accesses to an isolated memory address space before it enters the compartment. This enables the system to recover these states when an error occurs, preventing the kernel from crashing.

To summarize, several key challenges need to be addressed:

1. **What types of data need to be saved?** Identifying the critical memory and register states that must be saved as copies is essential for effective recovery.
2. **How do we save a copy of these data, particularly under context-switching scenarios?** Ensuring the correctness of data copies in a multi-process environment,

doi:10.6342/NTU202404466

where context switches occur, adds complexity to the saving mechanism.

3. **How can we ensure that the isolated memory address space remains uncompromised?** Protecting the integrity of the isolated memory space against potential attacks is critical for maintaining system security.

5.1 Motivation - Error Path Analyzing

During execution within the compartment, various types of data states may be modified, including common registers, system registers, local data, heap data, global shared data, and synchronization primitives. To address the first challenge—identifying which data states need to be saved for recovery—we analyze existing error-handling paths in the Linux kernel, with a focus on how the kernel manages and recovers from errors.

Our initial approach involves identifying a system call that is straightforward to analyze and has a well-defined error-handling path within the compartment. For this purpose, we randomly select a system call, the `kvm_create_vm` function in `kvm_main.c`, assuming it represents our compartment. We then examine the callee functions invoked within `kvm_create_vm` and their corresponding recovery functions in the error-handling path. Specifically, we investigate the memory operations performed by these functions, as these operations represent the memory addresses that need to be recovered if an error occurs within the compartment. Below are the pairs of callee functions within `kvm_create_vm` that we investigate:

After analyzing how these recovery functions restore memory states, we have identified several types of data that must be recovered, as well as those that do not require recovery.

doi:10.6342/NTU202404466

callee function	recovery function
`kvm_arch_alloc_vm(kvm)`	`kvm_arch_free_vm(kvm)`
`mmgrab(current->mm)`	`mmdrop(current->mm)`
`init_srcu_struct(&kvm->srcu)`	`cleanup_srcu_struct(&kvm->srcu)`
`refcount_set(&kvm->users_count, 1)`	`refcount_dec_and_test(&kvm->users_count)`
`kvm_alloc_memslots()`	`kvm_free_memslots(kvm, kvm->memslots[0])`
`kvm_arch_init_vm()`	`kvm_arch_destroy_vm(kvm)`

Table 5.1: Callee Function Pairs

- **Global Shared Data.** This includes data such as `vm_list`, which are accessed outside the compartment. Global shared data consists of statically allocated variables stored in the `.bss` or `.data` sections of an object file. Since these sections are not reset each time a process accesses them, failing to recover shared data may lead to errors in subsequent processes due to incorrect states. Therefore, it is essential to recover this data.

- **Heap Data.** Heap data refers to memory allocated during runtime using kernel memory allocation APIs, such as `kmalloc`. Recovery functions handle heap data by calling `kfree`, which prevents memory leaks by freeing the allocated memory.

- **Local Data.** Recovery functions typically do not restore local data. This is because local data is reinitialized with each invocation of functions like `kvm_create_vm`, eliminating the need for recovery.

Beyond the data types managed by the recovery functions, it is also crucial to consider the recovery of registers and synchronization primitives.

- **Register States.** This includes both system registers and common registers, which are modified within the compartment. As these register states are critical to the

doi:10.6342/NTU202404466

correct execution of processes, they must be preserved and restored in case of errors.

- **Synchronization Primitives.** These are essential for controlling access to shared data within the compartment. For example, in the function `kvm_create_vm`, locks such as `kvm_lock` are used when accessing `vm_list`. In our threat model, a process might crash while holding a lock, necessitating the recovery of locks to their state before entering the compartment to prevent deadlocks and ensure consistency.

In summary, the types of data states that must be saved to isolated memory address space include common registers, system registers, heap data, shared data, and synchronization primitives. By saving these states, the system can recover memory and registers to their state before entering the compartment, ensuring the kernel does not crash even if an error occurs within the compartment.

5.2 Time Interval-based Monitor

After identifying the critical memory and register states that need to be saved, we now address the second challenge: how to save copies of these states in scenarios involving context switching. Drawing inspiration from the paper by Narayanan et al. [6], which describes creating copies of shared data, isolating them in different memory address spaces, and automatically instrumenting the shared data, we designed our Time Interval-based Monitor to operate within an isolated memory address space. This approach provides security by ensuring that the monitor cannot be compromised. Our design involves instrumenting data that requires copying, thereby ensuring kernel availability even when errors occur.

doi:10.6342/NTU202404466

To facilitate this process, we explored different levels of instruction instrumentation granularity and ultimately chose to use LLVM Intermediate Representation (IR). This choice provides a better distinction of memory addresses in the compartment. We utilize the LLVM compiler from the repository available at this repository [10]. This compiler automatically identifies shared data, heap data, and shared locks in the compartment during compile time and instruments the relevant functions at the IR level, assisting us during the instrumentation phase.

With the instrumentation approach in place, our next objective is to discuss the specific monitor APIs we will design to facilitate the saving of data states during context switching. In section 5.1, we examined the `kvm_create_vm` system call and identified the critical data types that need to be saved for recovery: common registers, system registers, heap data, shared data, and synchronization primitives. We will now design corresponding monitor APIs to handle these tasks.

5.2.1 Shared Data

```
/*
    addr: addr of the shared data
*/
u64 comp_mem_get_data(void *addr);
/*
    addr: addr of the shared data
    val: the value of the shared data.
*/
void comp_mem_set_data(void *addr, u64 val);
```

Listing 1: Shared Data API

To enhance the availability of the kernel, we aim to enable processes to context switch within the compartment. Without our shared data API instrumentation, an error occurring during a context switch while accessing shared data can render the shared data state **un-**

doi:10.6342/NTU202404466

```c
int global_var1, global_var2, global_var3;
DEFINE_MUTEX(global_lock);

void increase_var_3(void){
    mutex_lock(&global_lock);
    global_var3 += 1;
    mutex_unlock(&global_lock);
}

void increase_var_2(void){
    mutex_lock(&global_lock);
    global_var2 += 1;
    mutex_unlock(&global_lock);
}

void increase_var_1(void){
    mutex_lock(&global_lock);
    global_var1 += 1;
    mutex_unlock(&global_lock);
}

int syscall2(void){
    /* PB */
    increase_var_3();
    increase_var_1();   // --- 1
    increase_var_1();   // --- 2
}
int syscall1(void){
    /* PA */
    int* err_ptr = (void *) 0;
    increase_var_1();   // --- 3
    increase_var_2();
    *err_ptr = 1;       // --- 4
}
```

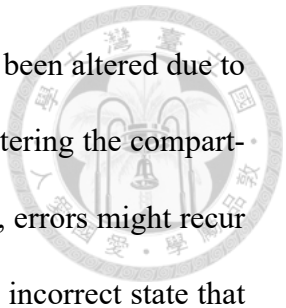
Listing 2: Case that *Dependency Issue* might occur

doi:10.6342/NTU202404466

```c
int global_var1, global_var2, global_var3;
DEFINE_MUTEX(global_lock);

void increase_var_3(void){
    com_mem_mutex_lock(&global_lock);
    com_mem_set_data(&global_var3, com_mem_get_data(&global_var3) + 1);
    com_mem_mutex_unlock(&global_lock);
}

void increase_var_2(void){
    com_mem_mutex_lock(&global_lock);
    com_mem_set_data(&global_var2, com_mem_get_data(&global_var2) + 1);
    com_mem_mutex_unlock(&global_lock);
}

void increase_var_1(void){
    com_mem_mutex_lock(&global_lock);
    com_mem_set_data(&global_var1, com_mem_get_data(&global_var1) + 1);
    com_mem_mutex_unlock(&global_lock);
}

int syscall2(void){
    /* PB */
    increase_var_3();
    increase_var_1();    // --- 1
    increase_var_1();    // --- 2
}
int syscall1(void){
    /* PA */
    int* err_ptr = (void *) 0;
    increase_var_1();   // --- 3
    increase_var_2();
    *err_ptr = 1;       // --- 4 // where null pointer deference error
    ↪ occurs
}
```

Listing 3: Case After Instrumentation Phase

doi:10.6342/NTU202404466



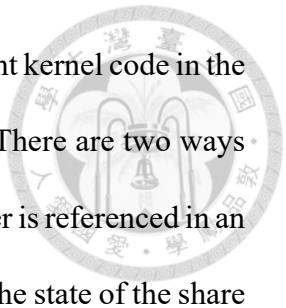
recoverable, which means a situation where the shared data state has been altered due to an error, and there is no mechanism to recover it to its state before entering the compartment. As a result, when other processes access this shared data again, errors might recur because the shared data state has become **dirty**, meaning it retains an incorrect state that could propagate errors to other processes.

For example, consider the case in Listing 2, where two system call control paths invoke the same function, `create_or_inc`. An error will occur if a process executes the program at line 33. Assume two processes, `PA` and `PB`, enter different system call control paths (`syscall1` and `syscall2`) on the same CPU core. The issue will manifest if the CPU executes instructions in the following order:

1. At line 31, `PA` calls the `increase_var_1` function, changing `global_var1`'s state in kernel address space from 0 to 1.
2. At line 25, `PB` calls the `increase_var_1` function, changing `global_var1`'s state from 1 to 2.
3. At line 33, `PA` encounters a null pointer dereference error.
4. At line 26, `PB` calls the `increase_var_1` function again, intending to change `global_var1`'s state. The state of `global_var1` should be 1, reflecting `PA`'s error; however, due to the absence of a recovery mechanism, `global_var1`'s state is still 2, and `PB` increases it to 3 instead of remaining at 2.

This example illustrates how shared data can become unrecoverable without proper instrumentation and recovery mechanisms in place, leading to further errors in other processes.

doi:10.6342/NTU202404466



To ensure kernel availability and resilience to errors, we instrument kernel code in the compartment with our monitor API to recover these memory states. There are two ways that a shared data is accessed in an instruction in the compartment, either is referenced in an instruction or is assigned a new state in an instruction. Both will load the state of the share data from the memory. Therefore, we created two monitor APIs, `comp_mem_get_data` and `comp_mem_set_data`, to monitor the state of shared data, as shown in Listing 1. When a shared data is referenced in an instruction in the compartment, we instrument the instruction with `comp_mem_get_data` during the instrumentation phase and pass the variable's address as a parameter to get a copy of its current state during the record phase. Conversely, when a shared data is assigned a new state in an instruction in the compartment, we instrument the instruction with `comp_mem_set_data` and pass the shared data address along with the new value as parameters to update the shared data state.

After instrumenting the two APIs during the instrumentation phase and defining their actions during the record phase, we focus on the monitor's actions during the restore and recovery phases. When the process exits the compartment, it calls the `comp_mem_exit_compartment` API, and the monitor copies the shared data state from the isolated memory address space back to the kernel address space during the restore phase. This ensures that the process can continue executing normally after exiting the compartment. If an error occurs in one of the processes within the compartment, the monitor enters the recovery phase. During this phase, the monitor does not copy the shared data state back to the kernel address space, preserving the shared data states as they were when the process initially entered the compartment.

However, if the monitor only recovers the process that encountered the error, a **dependency issue** may arise. This issue occurs when different processes access the same

doi:10.6342/NTU202404466

memory address within the compartment, and one process crashes and enters the recovery phase. For example, consider the case in Listing 2, where two system call control paths are within the compartment. The program, after instrumentation, is shown in Listing 3:

1. At line 31, `PA` calls the `increase_var_1` function, and the monitor changing `global_var1`'s state in the isolated memory address space from 0 to 1.

2. At line 25, `PB` calls the `increase_var_1` function, and the monitor changes `global_var1`'s state in the isolated memory address space from 1 to 2.

3. At line 33, `PA` triggers a null pointer dereference error and enters the recovery phase. In the recovery phase, the monitor deletes the `global_var1`'s node, resetting `global_var1`'s state to 0.

4. At line 26, `PB` calls the `increase_var_1` function again, intending to change `global_var1`'s state from 2 to 3. However, since the state has been reset by `PA` during its recovery routine, this causes an error.

In this case, we refer to `PA` and `PB` as dependent processes because both processes have accessed the same memory address `global_var1` in the compartment when `PA` crashed.

To effectively address the dependency issues discussed above, the design of the shared data API accounts for the three types of scenarios outlined below, as illustrated in Figure 5.1.

1. Process accesses the same shared data, instructions of accessing the shared data are instrumented, and the dependent processes are in the compartment.

   Processes accessing shared data where both the accessing instructions and depen-

doi:10.6342/NTU202404466

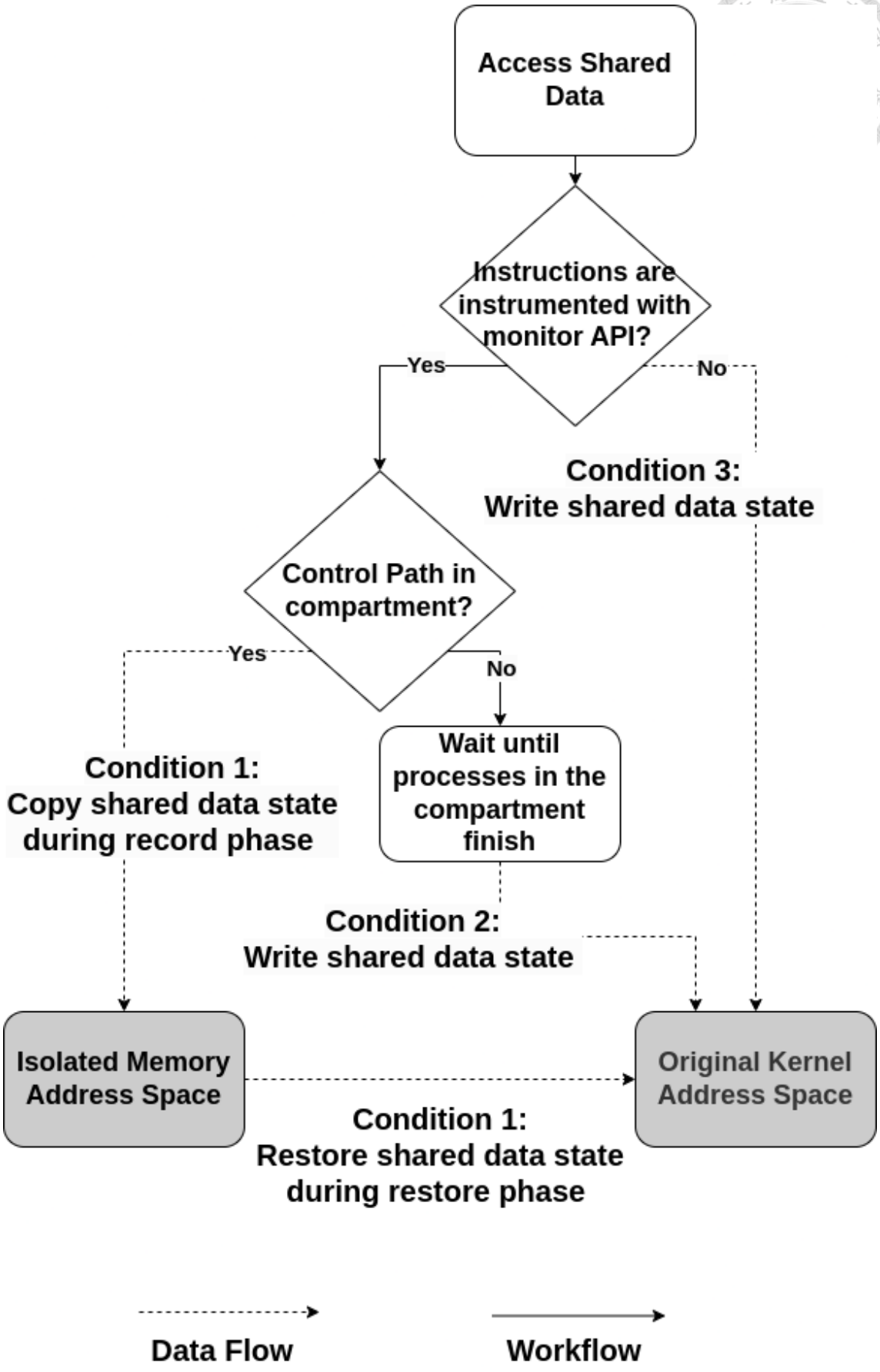
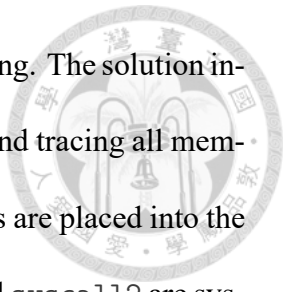


Figure 5.1: Three Type of Conditions For Shared Data API

doi:10.6342/NTU202404466



dent processes are within the compartment require careful handling. The solution involves identifying dependent processes of the crashed process and tracing all memory addresses they accessed. Subsequently, dependent processes are placed into the recovery phase. For example, in Listing 2, if both `syscall1` and `syscall2` are system call control paths in the compartment and share `global_var1`, `global_var2`, `global_var3`, and `global_lock`, the state of these shared variables and shared locks is recovered to prevent dependency issues. This ensures addresses that these dependent processes has accessed recovered to their states prior to entering the compartment.

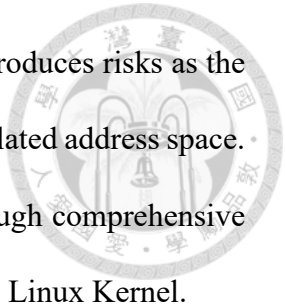
2. Process access the same shared data, instructions of accessing the shared data are instrumented, but dependent processes are not within the compartment.

   When processes accessing shared data have some control paths within the compartment and others outside, preemptive scheduling and CPU rescheduling are employed. This approach ensures that processes before entering the compartment are preempted until processes within the compartment exit the path successfully. For instance, in Listing 2, if `increase_var_1` is accessed by both `syscall1` (in compartment) and `syscall2` (not in compartment), `syscall2` is delayed from accessing `global_var1` until `syscall1` exits the compartment and completes its recovery phase.

3. Process accesses the same shared data, and instructions of accessing the shared data are not instrumented.

   In cases where shared data access instructions are not instrumented, a straightforward approach is adopted. The shared data state is not recovered if the processes within the compartment encounter error. Processes within the compartment directly

doi:10.6342/NTU202404466



modify the data state in kernel address space. This scenario introduces risks as the kernel may encounter errors if the data state isn't recorded in isolated address space. However, such occurrences are expected to be minimized through comprehensive analysis and instrumentation of all relevant variables within the Linux Kernel.

These solutions aim to mitigate dependency issues arising from shared data access across different control paths, ensuring robustness and security in system operations within compartmentalized environments. Additionally, if one of the dependent processes exits the compartment without errors, the monitor must wait for the other dependent processes to exit and restore them together. This ensures that, even if an error occurs in one of the dependent processes, the monitor can still recover all dependent processes as they remain monitored within the compartment.

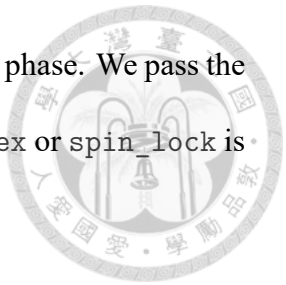
5.2.2 Synchronization Primitives

```
1 void comp_mem_mutex_lock(struct mutex *lock);
2 void comp_mem_mutex_unlock(struct mutex *lock);
3 void comp_mem_spin_lock(spinlock_t *lock);
4 void comp_mem_spin_unlock(spinlock_t *lock);
```

Listing 4: Shared Lock API

In many cases, shared data within a compartment is protected by synchronization primitives such as `mutex` and `spin_lock` to ensure consistency and prevent race conditions. For example, in Listing 2, each `global_var` is protected by critical sections created by `global_lock`. However, in our threat model, a process may crash while holding locks. Therefore, during the recovery phase, it is necessary to release any locks that were held by the process at the time of the crash. To achieve this, we instrument the kernel lock/unlock APIs for each `mutex` and `spin_lock` in the compartment by replacing them with

doi:10.6342/NTU202404466



our shared lock API, as shown in Listing 4, during the instrumentation phase. We pass the lock address to the isolated memory address space to track which `mutex` or `spin_lock` is held or released.

If the original kernel code creates a critical section using synchronization primitives to protect shared data within the compartment, it is essential to ensure that this critical section remains intact after instrumenting the kernel lock/unlock APIs with our shared lock API during the record phase. Additionally, the lock state must be recovered during the recovery phase if an error occurs within the compartment. To protect our monitor APIs and ensure that the critical section cannot be executed by multiple processes simultaneously, we use `monitor_mutex` in the monitor.

To address the three types of scenarios in the shared data API design, as shown in Figure 5.1, we design the shared lock API to avoid directly calling the kernel synchronization primitives APIs. Instead, we save the lock state to the isolated memory address space during the record phase. When a process attempts to acquire a synchronization primitive, it calls `comp_mem_mutex_lock` or `comp_mem_spin_lock` to check if any other process currently holds the synchronization primitive. If the primitive is held, the API yields the process and waits until the primitive is released before retrying. If the synchronization primitive is not held, the API updates the synchronization primitive's state in the isolated memory address space, allowing the process to acquire the lock. Since the monitor APIs are also protected by `monitor_mutex`, the original critical section remains protected after instrumenting the shared lock API.

During the restore phase, we ensure that the synchronization primitives in the kernel address space reflect their state within the compartment. If a synchronization primitive

doi:10.6342/NTU202404466

was released in the compartment, it is released in the kernel address space; if the synchronization primitive was not released, it is held. In the recovery phase, we adjust the synchronization primitives based on their state when the process entered the compartment. Synchronization primitives that were not held before entering the compartment are released, and those that were not released are retained. This approach helps prevent deadlocks after the restore or recovery phase.

5.2.3 Heap Data

```
1    /*
2        size: size of memory object need to allocate
3    */
4    void *comp_mem_kmalloc(int size);
5
6    /*
7        addr: addr of the memory object that need to be freed
8    */
9    void comp_mem_kfree(void *addr);
```

Listing 5: Memory Allocation API

In the compartment, some memory in the kernel might be dynamically allocated from the heap using kernel memory allocation APIs like `kmalloc`. To prevent memory leaks, this heap memory must be freed by calling `kfree`. However, in our threat model, a process might crash before executing the `kfree` API within the compartment, leading to potential memory leaks. Therefore, we instrument these APIs with our monitor to manage memory deallocation and provide a recovery mechanism in case of errors. As shown in Listing 5, we instrument `kmalloc` as `comp_mem_kmalloc` and `kfree` as `comp_mem_kfree` in the compartment.

There are three scenarios we need to consider for heap data in the compartment, based on whether the memory is allocated or freed within the compartment:

doi:10.6342/NTU202404466

1. **Allocated and freed within the compartment:** This scenario occurs when both allocation and deallocation happen within the compartment.

2. **Allocated within the compartment but not freed within the compartment:** An example of this scenario is a memory allocation performed by the `malloc` system call, where the deallocation occurs outside the compartment.

3. **Not allocated within the compartment but freed within the compartment:** This scenario is exemplified by the `free` system call, where the memory was allocated outside the compartment but is deallocated within it.

To handle all three scenarios, maintain compatibility with the kernel memory allocator, and provide a recovery mechanism, we implement a **lazy free** mechanism for heap data within the compartment. During the record phase, the `comp_mem_kmalloc` API calls the kernel memory allocation API `kmalloc`, passing the size of the memory object as a parameter. The address returned from the kernel memory allocation API is then saved to the isolated memory address space. On the other hand, during the record phase, the `comp_mem_kfree` API only marks the heap data as *invalid* in the isolated memory address space and does not immediately free the memory using the kernel memory allocation API `kfree`. The actual memory deallocation is deferred to the restore phase.

This approach is necessary to address the third type of heap data scenario, where the data might need to be recovered during the recovery phase. If the memory were freed immediately by calling `kfree` when `comp_mem_kfree` is invoked during the record phase, the memory address could be reused by other processes. This would cause null pointer dereference errors when the heap data is accessed after the recovery phase. By deferring the deallocation, we ensure that the memory state remains intact and can be properly

doi:10.6342/NTU202404466

managed during recovery and restoration phase.

5.2.4 Shared Field

```
/*
    field_addr: addr of the field data
    base_addr: the structure's base address that the field belongs to
*/
u64 comp_mem_get_field(void *field_addr, void *base_addr);

/*
    field_addr: addr of the field data
    base_addr: the structure's base address that the field belongs to
    val: the value of the field data
*/
void comp_mem_set_field(void *field_addr, void *base_addr, u64 val);
```

Listing 6: Field Data API

To save the state of data in the isolated memory address space, we create a sequence of data nodes that maintain these states. Each data node is a fixed-length structure, 64 bits long, and is organized as a key/value pair, where the data address serves as the key and the corresponding data state as the value. This design ensures that multiple nodes are not created for the same data address. Additionally, each data node contains a field that records the process IDs of the processes that have accessed the data within the compartment. This information is used during both the restore and recovery phases to identify dependent processes.

However, some data might be composite types, such as structures, rather than primitive types, leading to variable-sized states. To handle this, we implement a field data API to record each field of a structure separately, allowing these data to be saved in fixed-length data nodes. The field data API is instrumented when a structure's field is accessed within the compartment during the instrumentation phase and records data state during the record phase, similar to the shared data API. The key difference, as shown in Listing 6,

doi:10.6342/NTU202404466

is that we pass an additional argument `base_addr`, which represents the base address of the structure containing the field. This approach ensures that we only copy the state of the specific field that is modified, rather than the entire structure, reducing the time and space overhead of recovery.

The shared field API is also applied when dealing with heap data that are composite types. When such heap data is allocated or freed, the monitor ensures that all relevant fields are restored or recovered together, maintaining consistency across the data state. This design optimizes both recovery efficiency and data usage.

5.2.5 Registers

To recover the system state to what it was before entering the compartment when a process crashes, it is necessary to restore not only the memory state but also the register state, as both can be modified during execution in the compartment. There are two types of registers that may be altered in the compartment: common registers and system registers. The monitor saves these register states to an isolated memory address space and adds the current process ID to the monitor to indicate that the process has entered the compartment through the `comp_mem_enter_compartment` API.

The mechanism for saving registers is analogous to the principle of virtual machine (VM) switching, where the process traps into the hypervisor mode (EL2) and copies the states of the common registers and EL1 system registers to the isolated memory address space. If an error occurs, the process enters the recovery phase, trapping into EL2 to restore the register state saved before entering the compartment. Once the registers state are restored, and the recovery phase is complete, the process returns to EL1 and resumes

doi:10.6342/NTU202404466

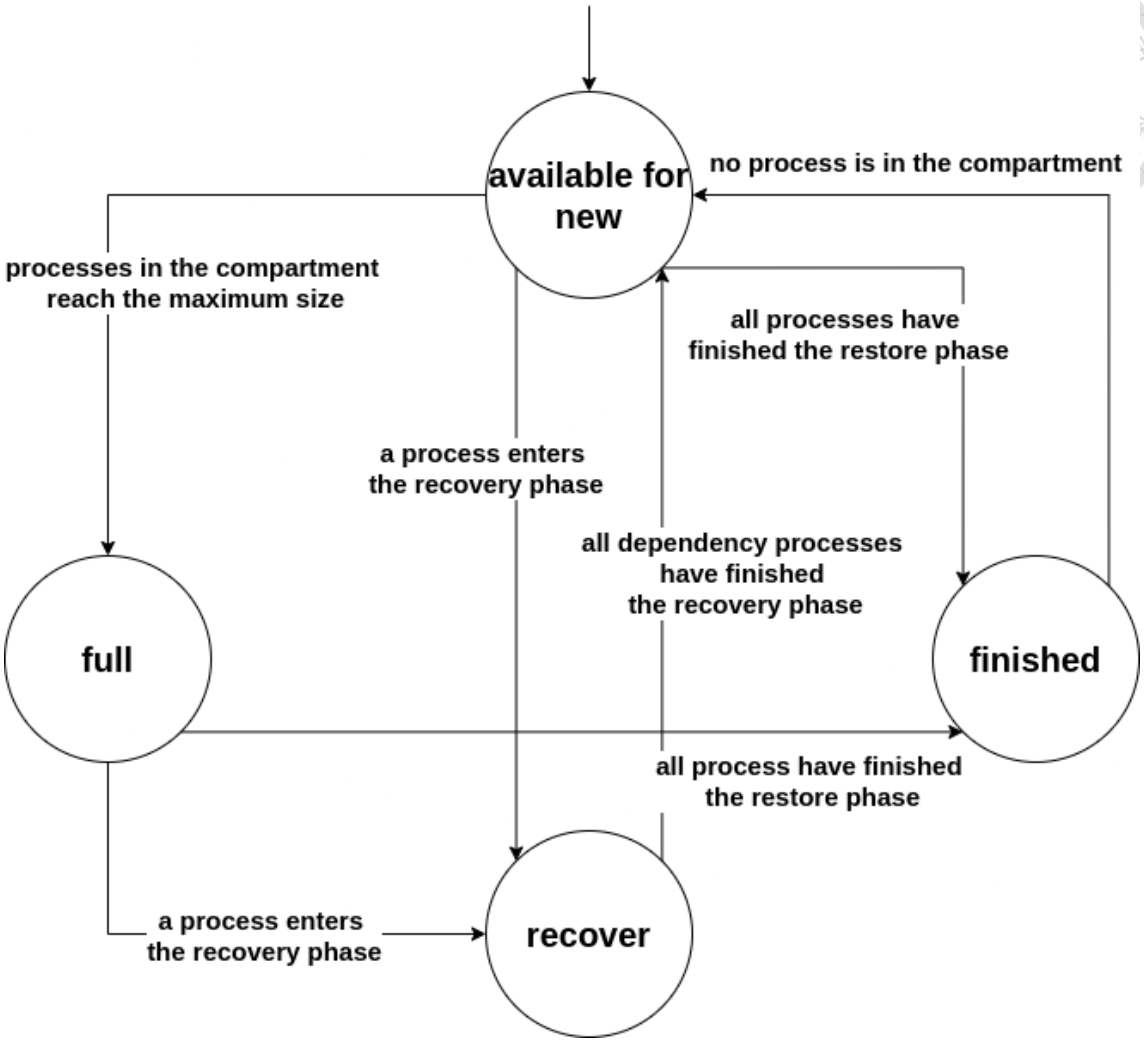


Figure 5.2: Finite State of Monitor

execution from the point where it trapped into EL2. This ensures that the process can continue running with the correct register state, preventing further errors after recovery.

5.2.6 Monitor State

In Listing 3, we demonstrated that if a process crashes within the compartment, all dependent processes must also be recovered. In our implementation, the isolated memory address space has limited capacity and cannot accommodate an unlimited number of processes. Therefore, we must limit the number of processes allowed in the compartment. To address these challenges and facilitate communication between processes within

doi:10.6342/NTU202404466

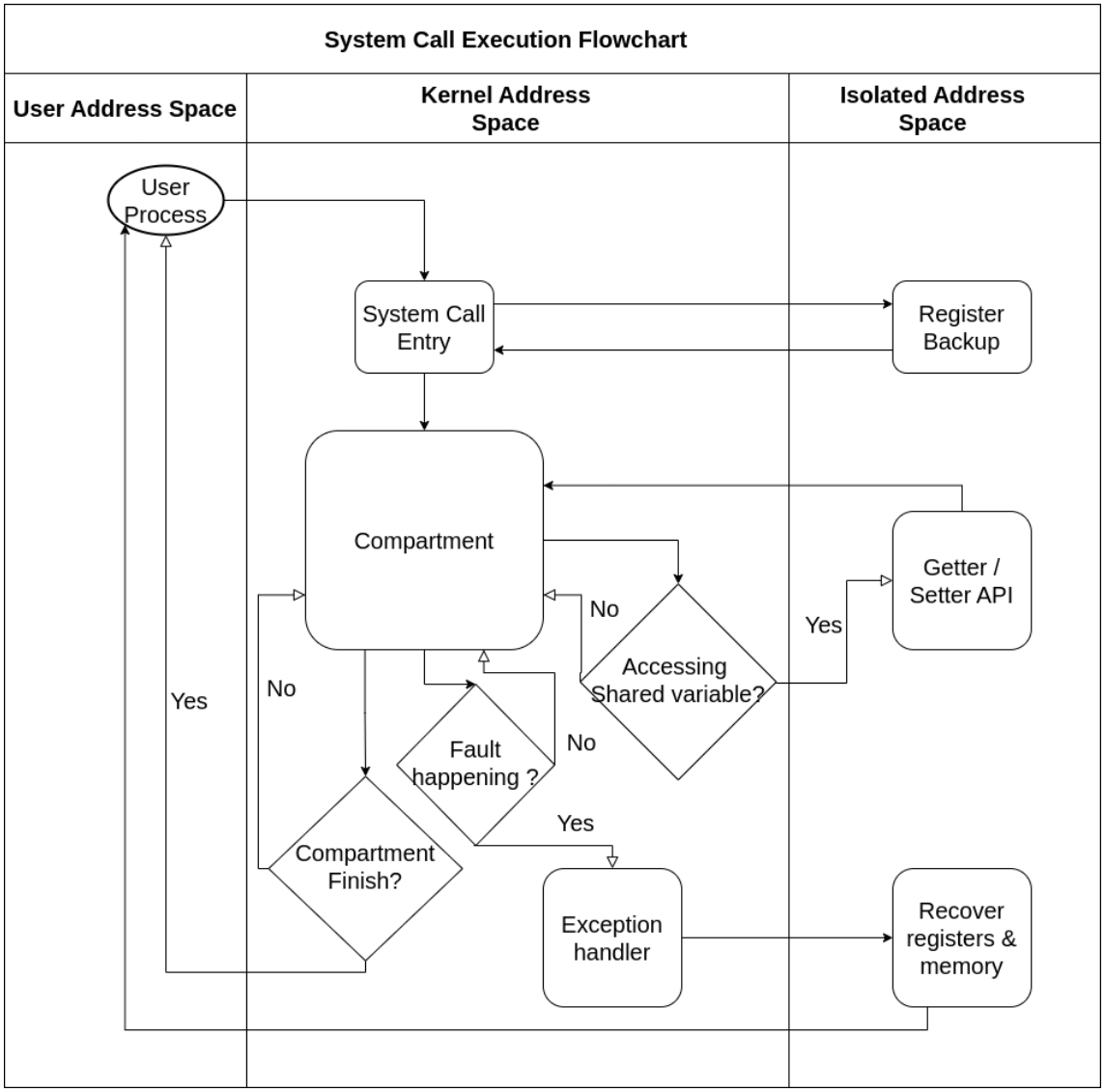


Figure 5.3: Detailed Runtime Workflow

doi:10.6342/NTU202404466

Algorithm 1 Monitor State Transitioning Condition

$monitor\_state \leftarrow AVAILABLE\_FOR\_NEW$
function CHANGE_STATE($void$)
  if $monitor\_state = AVAILABLE\_FOR\_NEW$ and pid_list is full **then**
    $monitor\_state = FULL$
  else if $monitor\_state = AVAILABLE\_FOR\_NEW$ and all pid.done $== 1$ **then**
    $monitor\_state = FINISHED$
  else if $monitor\_state = FULL$ and all pid.done $== 1$ **then**
    $monitor\_state = FINISHED$
  else if $monitor\_state = AVAILABLE\_FOR\_NEW$ and one of the process failed **then**
    $monitor\_state = RECOVER$
  else if $monitor\_state = FULL$ and one of the process encounter an error **then**
    $monitor\_state = RECOVER$
  else if $monitor\_state = RECOVER$ and pid_list is empty **then**
    $monitor\_state = AVAILABLE\_FOR\_NEW$
  else if $monitor\_state = FINISHED$ and pid_list is empty **then**
    $monitor\_state = AVAILABLE\_FOR\_NEW$
  end if
end function

the compartment, we define several states for the monitor. These states help manage each process's phase, and the monitor takes appropriate actions when a process transitions from one phase to another. Ultimately, these states and transitions form a finite state machine shown in Figure 5.2. A brief description of each monitor state is provided below, with the monitor initialized to the *available_for_new* state after kernel boot.

In Listing 3, we demonstrated that if a process crashes within the compartment, all dependent processes must also be recovered. Since the isolated memory address space has limited capacity and cannot accommodate an unlimited number of processes, it is necessary to limit the number of processes allowed in the compartment. To address these challenges and facilitate communication between processes within the compartment, we define several states for the monitor. These states help manage each process's phase, and the monitor takes appropriate actions when a process transitions from one phase to another. Ultimately, these states and transitions form a finite state machine, as shown in Figure 5.2.

doi:10.6342/NTU202404466

A brief description of each monitor state is provided below, with the monitor initialized to the *available_for_new* state after the kernel boots.

- *available_for_new*: The monitor is available for new processes to enter.
- *full*: The monitor has reached its maximum capacity for processes.
- *finished*: All processes have finished the execution in the compartment.
- *recover*: One of the processes has encountered an error and entered the recovery phase.

Whenever a process calls the monitor API, the API first invokes the `monitor_state_check_and_change` function to update the monitor's state based on the process's current state, and then performs actions according to the monitor's state using different APIs. Below are the transition conditions for the monitor states, and Algorithm 1 provides a brief overview of the algorithm used to transition states in our implementation.

Entering the Compartment: When a process enters the compartment, it calls the `comp_mem_enter_compartment` API. If the monitor's current state is *available_for_new* and there is still capacity for new processes, the process creates a data structure in the `pid_list` in the isolated memory address space. This data structure stores the process ID and tracks the process's current phase. If the compartment is full, the monitor state changes to *full*, and any additional processes attempting to enter will yield until the monitor transitions back to the *available_for_new* state.

Completing Execution: When a process completes execution in the compartment without encountering errors, it labels itself as **done** and checks whether all dependent processes have finished. If the monitor's current state is *available_for_new* or *full*, and all

doi:10.6342/NTU202404466

dependent processes are marked as **done**, the monitor transitions to the *finished* state. At this point, all data states accessed by the dependent processes are restored, and their data structures are removed from the `pid_list`.

Exiting the Compartment: If a process is labeled as **done** and sees that the monitor is in the *finished* state, it exits the compartment by cleaning its data structure from the `pid_list`.

Encountering an Error: If a process encounters an error, the monitor state transitions to *recover*. The monitor then cleans all recorded states in the isolated memory address space and removes the process's data structure from the `pid_list`. Once other dependent processes detect the transition to the *recover* state, they clean their data structures as well.

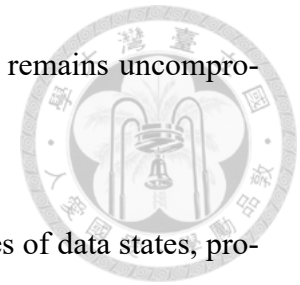
Re-entering the Compartment: When a new process attempts to enter the compartment and finds that the monitor state is either *finished* or *recover*, and there are no processes recorded in the `pid_list`, the monitor state changes to *available_for_new*, allowing the process to enter the compartment.

This design ensures that the monitor effectively manages the phases of each process within the compartment, handling transitions and dependencies to maintain kernel stability and availability in the event of errors. We summarize the detailed runtime workflow in Figure 5.3.

5.3 Isolated Memory Address Space

After detailing how our monitor is designed to save copies of data to the isolated memory address space under context-switching scenarios, we now address the final chal-

doi:10.6342/NTU202404466



lenge: How can we ensure that the isolated memory address space remains uncompromised?

As mentioned in previous sections, the data nodes storing copies of data states, process information saved in the `pid_list`, and the monitor state itself are all saved in the isolated memory address space. Therefore, protecting this space is crucial. To secure the isolated memory address space, we map the monitor's data and text sections from the EL1 virtual address space to the EL2 address space using the `create_hyp_mappings` function. This function is executed in EL1 during KVM initialization and sets up the EL2 page tables before transitioning to EL2. We have modified the function to support memory block duplication in hypervisor mode (EL2). Since EL2 has higher privileges than EL1, attackers cannot directly compromise the monitor or the isolated memory address space, as access is restricted to monitor APIs.

This approach ensures that the isolated memory address space remains secure and protected from potential attacks, maintaining the integrity of the monitor's operations.

doi:10.6342/NTU202404466

Chapter 6 Evaluation

6.1 Performance

```c
int global_var = 0;
#define EXEC_TIMES 10000000
int demo_syscall(int a, int b)
{
    int i;
    for (i = 0; i < EXEC_TIMES; i++) {
        /* monitor API / original instruction is putted in this section */
        comp_mem_set_data(&global_var, i + 1);
        /* monitor API / original instruction is putted in this section */
    }
    return 0;
}

asmlinkage long __sys_compartment_test(int para_a, int para_b)
{
    if (!comp_mem_enter_compartment()) {
        return 0;
    }
    ret = demo_syscall(para_a, para_b);
    comp_mem_exit_compartment();
}
```

Listing 7: API Cost Time Evaluation Program

We implemented our prototype based on the design proposed in chapter 5, using the Linux 5.15 SeKVM kernel architecture [8, 9] as the foundation. SeKVM already provides isolated memory address spaces and register copying functions in EL2, ensuring the robustness of isolated memory spaces, which makes it a reliable base for our design. Our

doi:10.6342/NTU202404466

API type	executed times	original instruction (sec)	monitor API (sec)	ratio (monitor API / original instruction)
kmalloc / kfree pair	10^6	0.249	37.995	152.59
mutex_lock / mutex_unlock pair	10^7	0.273	1.463	5.359
spin_lock / spin_unlock pair	10^7	2.992	1.683	0.563
set_data	10^7	0.008	6.137	767.125
get_data	10^7	0.015	6.154	410.267
set_field	10^7	0.011	5.932	539.273
get_field	10^7	0.013	6.215	478.077

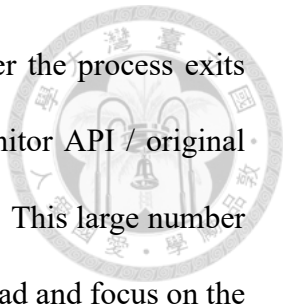
Table 6.1: API Cost Time

implementation includes our time-interval-based monitor, with an additional 1,153 lines of kernel code beside SeKVM's codebase. We evaluated the performance of the monitor APIs by comparing their execution time to that of the original kernel APIs, using QEMU version 7.0.0. All experiments were conducted on a prototype with a single 64-bit ARM Cortex-A57 CPU core and 2GB of memory. The host running QEMU was configured with Linux 5.15 and had 16 Intel i7-12700K CPU cores and 16GB of memory.

Our analysis focuses on evaluating the overhead introduced when system operations in the compartment are replaced by our monitor APIs. The results in Table 6.1 were measured by implementing a custom system call (shown in Listing 7) in the Linux kernel. We then measured the system execution time required for the original operations, as well as the execution time after instrumenting the system with our monitor APIs. The custom system call used for evaluation includes 224 lines of code and assumes that all instructions accessing shared data are fully instrumented, meaning that it does not cover scenarios where only partial instrumentation is applied.

The execution time for each API in Listing 7 represents the total time taken from when the process enters the system call to when it exits. Therefore, the measured time can be expressed as: "Time before the process enters the `demo_syscall` function" +

doi:10.6342/NTU202404466



”Time spent on the monitor API / original operation” + ”Time after the process exits the `demo_syscall` function.” To isolate the time spent on the ”monitor API / original operation”, we executed the APIs in our system call 10^6 to 10^7 times. This large number of executions allows us to minimize the influence of unrelated overhead and focus on the performance impact of the APIs themselves.

By analyzing the data in Table 6.1, we can draw the following conclusions:

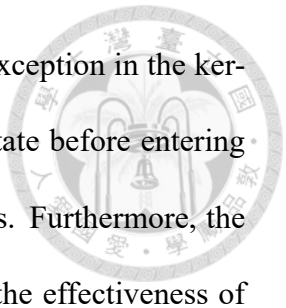
1. **Getter/setter APIs and kmalloc/kfree APIs:** These APIs introduce significant overhead, ranging from 100x to 700x, after replacing them with our monitor API. The substantial increase is primarily due to hypercall overhead, the cost of saving the state of the data, additional checks and updates to the monitor state, and the need to acquire mutex locks within the monitor. These factors contribute to a marked increase in execution time.

2. **Mutex API overhead:** The overhead for the mutex APIs is relatively lower because our monitor already uses an internal mutex to prevent multiple processes from accessing the monitor simultaneously. This means that when the kernel’s original mutex APIs are replaced with our monitor’s mutex APIs, only one additional lock needs to be acquired, which limits the performance impact.

3. **Spinlock API overhead:** Similar to the mutex API, we do not directly hold the spinlock within the monitor. Instead, we label the data node to indicate that the lock is held by another process. By avoiding the busy waiting typical of spinlocks, our monitor API reduces the performance overhead, resulting in better performance than the original spinlock implementation.

Additionally, we tested the correctness of our design by executing a null pointer def-

doi:10.6342/NTU202404466



erence bug system call within the compartment, which triggered an exception in the kernel. In this scenario, the process was successfully recovered to its state before entering the compartment and returned to user space without any kernel errors. Furthermore, the kernel continued operating without showing any errors, confirming the effectiveness of the recovery mechanism.

In summary, while the overhead introduced by the monitor APIs is significant for some operations (especially getter/setter and memory allocation APIs), the performance remains acceptable for operations like mutex and spinlock handling. These results highlight the trade-off between increased security (through memory isolation and data state recovery) and performance.

6.2 Security Analysis

Our protection mechanism involves saving copies of memory and register states in an isolated memory address space that is secure from compromise. This allows us to recover these states in the event of errors that trigger exceptions and cause the CPU to enter an exception handler.

Since 2014, there have been 1,658 vulnerabilities identified within the Linux kernel, with 1,345 of these classified as memory corruption vulnerabilities, constituting 85% of the total. In many cases, memory corruption is caused by unchecked code, leading to errors that trigger exceptions. To evaluate the effectiveness of our design in recovering from such errors, we analyze several CVEs as examples, demonstrating how our protection mechanism can mitigate these specific vulnerabilities.

CVE-2024-26598 is a vulnerability in the Linux Kernel Virtual Machine (KVM)

doi:10.6342/NTU202404466

on the ARM64 architecture, specifically related to the Virtual Generic Interrupt Controller (VGIC) Interrupt Translation Service (ITS) subsystem. The issue is a potential use-after-free (UAF) scenario in the Local Peripheral Interrupt (LPI) translation cache. The vulnerability occurs when there is a race condition between an LPI translation cache hit and an operation that invalidates the cache. The core problem is that the function `vgic_its_check_cache()` fails to properly elevate the reference count (refcount) on the `vgic_irq` structure before releasing a lock that serializes changes to the refcount. This could lead to the `vgic_irq` structure being freed while it is still in use, causing a use-after-free condition. Our design protects against this type of vulnerability by saving a copy of the data state before it is freed. Specifically, when a process or thread accesses shared data that could be involved in race conditions, such as the codevgic_irq structure, our mechanism records the state in an isolated memory address space. In the event of an error, such as a use-after-free condition, the system can recover the data state from the isolated memory address space, preventing the exploitation of the freed memory. This approach helps to maintain the integrity and availability of the kernel, even in the presence of such vulnerabilities.

CVE-2019-6974 is another vulnerability in the Linux Kernel Virtual Machine (KVM) subsystem, specifically related to the `kvm_ioctl_create_device` function in `virt/kvm/kvm_main.c`. The issue is a use-after-free (UAF) vulnerability as well caused by mishandling reference counting due to a race condition. The vulnerability arises during the process of creating a virtual device in KVM. The `kvm_ioctl_create_device` function performs several steps:

1. **Device Creation:** It creates a device that holds a reference to the VM (Virtual Machine) object, but this reference is a ”borrowed” one, meaning that the VM’s refer-

doi:10.6342/NTU202404466

ence count has not been increased yet.

2. **Device Initialization:** The device is initialized.

3. **Ownership Transfer:** The reference to the device is transferred to the caller's file descriptor table.

4. **Reference Count Update:** The function then calls `kvm_get_kvm()` to convert the borrowed reference into a real reference by increasing the VM's reference count.

This sequence allows for a race condition that leads to a use-after-free scenario, potentially causing undefined behavior, such as kernel crashes or memory corruption. Our design addresses this by saving copies of critical memory and register states in an isolated memory address space that cannot be compromised. Even if an error that trigger exceptions occurs, our system can restore the previous valid states from the isolated memory, maintaining kernel integrity and preventing exploitation or crashes due to such memory corruption vulnerabilities.

CVE-2017-15274 is a null pointer dereference vulnerability in the Linux kernel's key management subsystem (`security/keys/keyctl.c`). This vulnerability is triggered when the `sys_add_key()` function or the `KEYCTL_UPDATE` operation of the `sys_keyctl()` function is called with a null payload pointer and a nonzero length value. Several key types in the kernel do not handle this situation correctly, leading to a null pointer dereference and a resulting kernel panic (OOPS). Our design ensures that if such an error occurs, the system can restore the memory and register states before the `add_key` or `keyctl` system calls from the isolated memory address space. This capability prevents the system from crashing and maintains kernel availability.

doi:10.6342/NTU202404466

In summary, these examples demonstrate that our monitor design, through the instrumentation of code and the use of isolated memory for state saving, can effectively recover from memory corruption vulnerabilities like those described in the selected CVEs, thereby preventing kernel crashes and maintaining system stability.

doi:10.6342/NTU202404466

doi:10.6342/NTU202404466

Chapter 7 Related Work and Future Work

7.1 Related Work

In this section, we review and compare notable works with similarities and differences to our design, focusing on key dimensions such as the target of protection, the granularity of the recovery mechanism, and the overall approach to crash recovery.

Works	Protect Target	Error recovery machnism
Our Design	kernel data accessed in the system call control path	recover to the state before entering the compartment
Ksplit	driver	no
Driverlets	driver/device interactions	replay interactions
HAKC	kernel compartment	no
ACES	compartment on bare-metal system	no
QEMU	virtual machine	checkpoint and replay the entire VM state
CRAK	user process	process migration focus
Kckpt	user process	user-space recovery

Table 7.1: Comparison Between Related Works

doi:10.6342/NTU202404466

7.1.1 Isolation Techniques

The following works serve as prominent examples of isolation techniques designed to prevent faults in specific components of the Linux kernel.

KSplit [6] is designed to isolate device drivers from the rest of the Linux kernel. It works by separating the shared memory space between the device driver and the kernel, thereby preventing faulty drivers from corrupting critical kernel data. KSplit isolates driver-related faults through instrumentation and synchronization of shared data but does not provide a checkpointing or recovery mechanism. It prevents errors from propagating within the kernel but lacks the ability to revert to a prior system state after a failure. In contrast, our work not only isolates faults at the system call level but also provides a recovery mechanism, checkpointing and restoring the system call states to maintain kernel availability.

Driverlets [4], introduced in the paper Minimum Viable Device Drivers for ARM TrustZone, takes a fine-grained approach by isolating specific IO operations in device drivers. These IO operations typically involve sensitive device communication (such as sending and receiving data from hardware peripherals) and are critical to ensuring the correctness of driver behavior. Driverlets use a replay-based mechanism to ensure IO correctness, and they run within the ARM TrustZone OP-TEE environment to securely handle interactions between untrusted operating systems and trusted devices. This design isolates sensitive operations, ensuring that even if the OS is compromised, the IO operations remain secure. However, Driverlets are limited to IO interactions and do not extend to broader kernel operations or offer recovery from system call errors. Our work, on the other hand, focuses on system calls across the entire kernel and provides a recovery

doi:10.6342/NTU202404466

mechanism for restoring the kernel to a stable state after memory corruption issues.

HAKC (Hardware-assisted Kernel Checker) [13] focuses on detecting and preventing control-flow violations at runtime using hardware features to enforce control-flow integrity (CFI). HAKC helps in detecting potential vulnerabilities like return-oriented programming (ROP) attacks by monitoring kernel code execution. However, it is primarily focused on detection and prevention of control flow errors rather than recovery from kernel memory corruption errors. Unlike our system, HAKC does not address null pointer dereference or use-after-free errors, nor does it offer a recovery mechanism to restore system stability after such failures. Our work takes a broader approach by targeting memory corruption recovery and addressing system calls specifically within the kernel, ensuring availability even after errors occur.

ACES (Asymmetric Cores for Error-tolerant Systems) [2] leverages asymmetric core architectures to isolate and tolerate errors in system execution. ACES is focused on bare-metal embedded systems where advanced hardware isolation mechanisms like Memory Management Units (MMUs) are often unavailable. It aims to isolate components (e.g., code, data, peripherals) from one another in a very fine-grained manner, preventing faults or vulnerabilities in one compartment from compromising the entire system. Our design includes active recovery mechanisms, recovering memory and register states in response to errors (such as null pointer dereference or use-after-free). ACES, in contrast, is more about fault isolation rather than recovery. ACES is tailored for embedded systems without advanced hardware, while our design focuses on general-purpose kernels (e.g., Linux) with SeKVM, applying more specifically to system calls.

REWIND [17] is a platform designed to secure serverless function execution by pro-

doi:10.6342/NTU202404466

viding isolation between consecutive requests. REWIND is on the serverless computing environment, operates at the container level, restoring the memory, file system, and processes after each serverless function request. It focuses on ensuring that each function execution starts in a clean state, preventing the leakage of data between different invocations. It's recovery mechanism will resets the entire container state, providing a coarse-grained recovery mechanism for the entire container. On the other hand, our design's recovery mechanism restores individual memory and register states within the kernel when a system call encounters errors.

In contrast to these isolation techniques, our design not only isolates faults at the system call level but also offers a recovery mechanism that checkpointed system calls and restored them after a failure. This ability to both contain and recover from faults is what distinguishes our approach from isolation-centric techniques.

7.1.2 Crash Recovery Mechanisms

Several checkpointing mechanisms such as QEMU' s record/replay, CRAK, and Kckpt provide system recovery at varying levels of granularity.

QEMU's record/replay [14–16] mechanism captures the entire state of a virtual machine (VM) at specific intervals, allowing deterministic replay in the event of a failure. This approach focuses on VM-level recovery, which is coarse-grained since it records the entire VM state, including memory, CPU registers, and device states. When an error occurs, the system rolls back the entire VM to a previous state. While effective for full-system recovery, this method incurs significant overhead due to the need to revert the entire system. In contrast, our design is more fine-grained, targeting kernel-level recovery by

doi:10.6342/NTU202404466

recording individual memory and register operations for system calls. This enables faster and more targeted recovery without the need to revert the entire system state.

CRAK (Checkpoint/Restart as a Kernel Module) [22] offers transparent checkpointing and restart capabilities for Linux networked applications, preserving user-space data such as memory, registers, file descriptors, and IPC structures. It is particularly useful for process migration and parallel application scenarios, but it does not target kernel-level crash recovery. CRAK's focus is on process migration, enabling processes to be moved between systems while retaining their state. Our design differs in that it aims to maintain kernel availability by providing recovery for system calls in case of memory corruption or system crashes. Instead of focusing on user-space processes, we ensure that kernel-level operations can recover from errors without causing the kernel to crash.

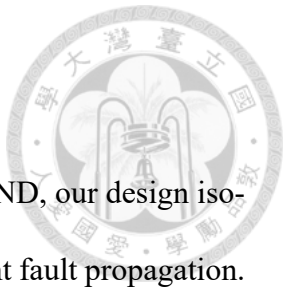
Kckpt [5], another checkpointing mechanism for UnixWare, captures process state for user-space processes, offering both user-directed and automatic checkpoints. Similar to CRAK, Kckpt focuses on user-space data, such as open files and system calls that affect user processes. While Kckpt effectively restores user-space processes after a crash, it does not address kernel-level recovery. In contrast, our work is focused on kernel-space recovery, specifically targeting memory corruption errors related to system calls. While Kckpt restores processes, our system restores kernel memory and register states accessed by those processes.

7.1.3 Combining Isolation and Recovery

Our work is unique in that it combines isolation and recovery mechanisms at the kernel level, offering both fault containment and the ability to restore system call states in

doi:10.6342/NTU202404466

case of failure.



- **Isolation**: Similar to KSplit, Driverlets, HAKC, ACES, REWIND, our design isolates faults in critical parts of the kernel (system calls) to prevent fault propagation. However, unlike these systems, which focus on fault isolation for device drivers or IO operations, our work isolates errors within system calls, affecting the entire kernel.

- **Recovery**: Like QEMU, CRAK, and Kckpt, we provide checkpointing mechanisms for system recovery. However, our approach is more fine-grained, targeting system calls rather than user-space processes or entire VMs. This enables efficient recovery without the overhead of rolling back the entire system.

- **Strength of Our Work**: By combining both isolation and recovery, we offer an efficient, low-overhead solution for handling common kernel memory corruption errors, such as null pointer dereference and use-after-free vulnerabilities. Our time-interval-based monitoring system precisely captures and restores kernel states at the system call level, ensuring that the system remains available and stable even after errors occur.

In conclusion, as shown in Table 7.1, while KSplit, Driverlets, HAKC, and ACES provide isolation, and tools like QEMU, CRAK, and Kckpt offer broader checkpointing solutions, our design fills a gap by offering a fine-grained, kernel-level recovery mechanism. Our work focuses on recovering from memory corruption errors specifically within system calls, ensuring that the kernel remains stable and operational without the need for full system rollbacks. This combined approach of isolation and recovery provides a robust solution for kernel error handling.

doi:10.6342/NTU202404466

7.2 Future Work

```
struct demo_entry {
    struct list_head list;
    int val;
};
LIST_HEAD(demo_list);
void list_val_init(void)
{
    struct demo_entry *entry;
    entry = kmalloc(sizeof(*entry), GFP_KERNEL);
    if (!entry)
        return -ENOMEM;
    entry->val = 0;
    list_add(&entry->list, &demo_list);
    return 0;
}
```

Listing 8: Data That Might not Need to Recover

The recovery mechanism is protected because each system call control path must first invoke the `enter_compartment` API. This API registers the process ID in an isolated memory address space, marking the process as operating within the compartment. If an attacker attempts to invoke the record/recovery API without properly entering the compartment (i.e., without calling `enter_compartment`), the process will not be recognized by the API. As a result, the attacker will be unable to alter the compartment's data state. This ensures that only legitimate processes within the compartment can trigger recovery actions, safeguarding the integrity of the monitor.

However, if an attacker employs a Return-Oriented Programming (ROP) or a Jump-Oriented Programming (JOP) attack to invoke `enter_compartment`, the system will mistakenly treat the process as safely entering the system call. This could allow the attacker to make unauthorized calls to the monitor. To prevent this type of attack, in our future work, it is necessary to enforce ARM Pointer Authentication Code (PAC) and Branch Target Iden-

doi:10.6342/NTU202404466

tification (BTI) techniques. These hardware-level protections ensure that indirect calls, such as those made through ROP or JOP, are validated, providing an additional layer of security against such exploits. Similarily, ARM's Memory Tagging Extension (MTE) and Kernel Address Sanitizer (KASAN) can label memory regions within the Linux kernel, triggering exceptions if an attacker attempts to access invalid or misused memory addresses. This mechanism would prevent attackers from leveraging compartments to corrupt memory outside of the compartment.

This approach hardens the fault recovery mechanism and protects the system from ROP or JOP attacks that could manipulate the system call flow.

In our current design for managing heap data (as mentioned in subsection 5.2.3), we rely on the kernel's memory allocation API (`kmalloc`) for allocating memory. This means the kernel memory allocator must be trusted and free from compromise. To further reduce our Trusted Computing Base (TCB), we propose developing an isolated memory allocator within the isolated memory address space, which would handle memory allocations without needing to rely on the kernel's allocation APIs. Combining this with PAC and BTI techniques, the kernel address space can be excluded from our TCB, ensuring that even if an attacker compromises the kernel's control flow through function pointer manipulation or code-reuse attacks (e.g., ROP or JOP), the isolated memory and compartment remain secure.

Regarding the dependency issues mentioned in subsection 5.2.1, we assume that if shared data are not "fully instrumented"—meaning not all instructions referencing the shared data are instrumented—kernel recovery cannot be guaranteed after an error occurs. This is because if we cannot fully record the data state in the isolated memory address

doi:10.6342/NTU202404466

space, it becomes impossible to identify dependent processes and recover them when one process crashes. In future work, we plan to analyze which types of shared data, if not recovered, will not compromise kernel stability or lead to crashes. For instance, in Listing 8, if a `demo_entry` node is added to the linked list `demo_list` in the function `list_val_init`—with `demo_list` being a shared data structure—then all instructions referencing the linked list must be instrumented with our monitor API to enable recovery. If a process that accesses this linked list crashes and the shared data `demo_list` is not recovered, it may lead to a memory leak. However, if the data is initialized correctly, the kernel may not crash despite the memory leak.

A promising approach to addressing this issue is through semantic analysis of shared data, allowing us to determine which shared data may not critically impact the kernel if not recovered. By doing so, we can reduce dependency issues and avoid the need to recover all data.

doi:10.6342/NTU202404466

doi:10.6342/NTU202404466

Chapter 8 Conclusions

This thesis introduces a software-based solution to address critical kernel vulnerabilities, particularly those caused by memory corruption, such as errors that trigger exceptions and cause the CPU to enter an exception handler. We developed a time-interval-based monitor that operates at the kernel level, which saves copies of memory and register states in an isolated memory address space. This mechanism enables efficient recovery in the event of kernel errors, ensuring system availability without the need for hardware-specific features. The proposed solution offers flexibility in deployment across different environments by focusing on kernel-level recovery.

Our evaluation demonstrated the system's effectiveness in mitigating real-world vulnerabilities, all while maintaining minimal performance overhead. This approach ensures system resilience and preserves kernel integrity in the presence of memory corruption errors. In conclusion, our work contributes a flexible, software-based recovery solution that enhances kernel availability and addresses critical vulnerabilities in modern operating systems.

doi:10.6342/NTU202404466

doi:10.6342/NTU202404466

References

[1] H.-S. Chen. Toward record replay of virtual machines on linux kvm for arm. In Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, 2024.

[2] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer. Aces: automatic compartments for embedded systems. In Proceedings of the 27th USENIX Conference on Security Symposium, SEC'18, page 65–82, USA, 2018. USENIX Association.

[3] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong. Execrecorder: Vm-based full-system replay for attack analysis and system recovery. In Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID '06, page 66–71, New York, NY, USA, 2006. Association for Computing Machinery.

[4] L. Guo and F. X. Lin. Minimum viable device drivers for arm trustzone. In Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, page 300–316, New York, NY, USA, 2022. Association for Computing Machinery.

[5] G. Hong, S. J. Ahn, S. C. Han, T. Park, H. Yeom, and Y. Cho. Kckpt: checkpoint

doi:10.6342/NTU202404466

and recovery facility on unixware kernel. In Proceedings of the 15th International Conference on Computers and Their Applications (ISCA). Citeseer, 2000.

[6] Y. Huang, V. Narayanan, D. Detweiler, K. Huang, G. Tan, T. Jaeger, and A. Burtsev. {KSplit}: Automating device driver isolation. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 613–631, 2022.

[7] kernel contributors. Memory Management APIs —The Linux Kernel documentation.

[8] S.-W. Li, J. S. Koh, and J. Nieh. Protecting cloud virtual machines from hypervisor and host operating system exploits. In 28th USENIX Security Symposium (USENIX Security 19), pages 1357–1374, 2019.

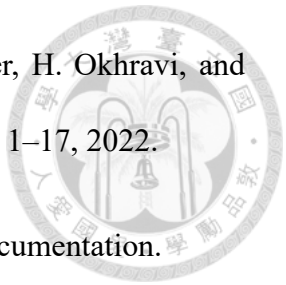
[9] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. Formally verified memory protection for a commodity multiprocessor hypervisor. In 30th USENIX Security Symposium (USENIX Security 21), pages 3953–3970, 2021.

[10] Lilihsu. GitHub - lilihsu/llvm-project: The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

[11] D. Lomet and G. Weikum. Efficient transparent application recovery in client-server information systems. In Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, SIGMOD '98, page 460–471, New York, NY, USA, 1998. Association for Computing Machinery.

[12] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and M. Rosenblum. Towards practical default-on multi-core record/replay. ACM SIGPLAN Notices, 52(4):693–708, 2017.

doi:10.6342/NTU202404466



[13] D. P. McKee, Y. Giannaris, C. Ortega, H. E. Shrobe, M. Payer, H. Okhravi, and N. Burow. Preventing kernel hacks with hakcs. In NDSS, pages 1–17, 2022.

[14] qemu contributors. CheckPoint and Restart (CPR) —QEMU documentation.

[15] qemu contributors. Documentation/CreateSnapshot - QEMU.

[16] qemu contributors. Record/replay —QEMU documentation.

[17] J. Song, B. Kim, M. Kwak, B. Lee, E. Seo, and J. Jeong. A secure, fast, and {Resource-Efficient} serverless platform with function {REWIND}. In 2024 USENIX Annual Technical Conference (USENIX ATC 24), pages 597–613, 2024.

[18] J. P. Walters and V. Chaudhary. Application-level checkpointing techniques for parallel programs. In University at Buffalo, The State University of New York, pages 222–233, 2006.

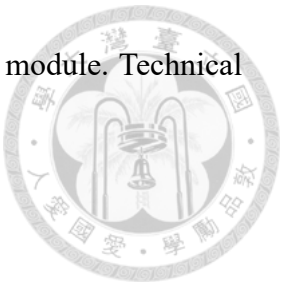
[19] S. Wan, M. Sun, K. Sun, N. Zhang, and X. He. Rustee: Developing memory-safe arm trustzone applications. In Proceedings of the 36th Annual Computer Security Applications Conference, ACSAC '20, page 442–453, New York, NY, USA, 2020. Association for Computing Machinery.

[20] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In 2022 IEEE Symposium on Security and Privacy (SP), pages 352–369. IEEE, 2022.

[21] Z. Yedidia. Lightweight fault isolation: Practical, efficient, and secure software sandboxing. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pages 649–665, 2024.

doi:10.6342/NTU202404466

[22] H. Zhong and J. Nieh. Crak: Linux checkpoint/restart as a kernel module. Technical report, Citeseer, 2001.



doi:10.6342/NTU202404466

Appendix A — Introduction

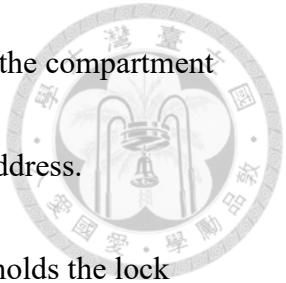
A.1 Data Structures in the Monitor

```
struct comp_mem_record_node{
    int pid[COMPARTMENT_MAX_PROCESS];
    int pid_num;
    enum comp_mem_node_type type;
    u64 addr;
    u64 value;
    u64 original_value;
    bool is_freed_inside;
    bool is_alloc_inside;
    u64 base_addr;
    int lock_hold_by;
    struct hlist_node node;
}
```

Listing 9: Record Node

- pid: which pids have accessed to this record node.
- type: type of the data node, including {global, field, spin_lock, mutex}
- addr: addr of the global shared memory
- value: value we record in the compartment
- original_value: data value in kernel address space before getting into system call
- is_freed_inside: check if the memory address is free inside the compartment

doi:10.6342/NTU202404466



- is_alloc_inside: check if the memory address is allocted inside the compartment
- base_addr: only used for field node, the parent structure base address.
- lock_hold_by: only used by lock type node, track the pid who holds the lock

We use an array to record pids that have used this memory address. So if there are over 1 process have accessed to this node, this node will be a shared variable according to our definition.

```
struct comp_mem_pid_list {
    u64 pid;
    int finished;
    int recovered;
};
```

Listing 10: Pid List

- pid: The structure belongs to which pid. Each pid tracked in the monitor will have one.
- finished: The process for this pid has finished the compartment successfully or not.
- recovered: The process has recovered all of it's record node or not.

For pid_list, when each process first enters the compartment, it will create its pid_list space by monitor, which will record its current process status.

doi:10.6342/NTU202404466