國立臺灣大學電機資訊學院資訊網路與多媒體研究所

碩士論文

Graduate Institute of Networking and Multimedia

College of Electrical Engineering and Computer Science

National Taiwan University

Master’s Thesis

透過自動化程式碼分析與修改增強 Linux 系統呼叫的可靠性

Enhancing the Reliability of Linux System Calls through Automated Code Analysis and Modification

徐翊凌

Yi-Lin Hsu

指導教授: 黎士瑋 博士

Advisor: Shih-Wei Li, Ph.D.

中華民國 114 年 1 月

January 2025

doi:10.6342/NTU202500063

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

透過自動化程式碼分析與修改增強 Linux 系統呼叫的可靠性

Enhancing the Reliability of Linux System Calls through Automated Code Analysis and Modification

本論文係 徐翊凌 （學號 R11944042）在國立臺灣大學資訊網路與多媒體研究所完成之碩士學位論文，於民國 113 年 12 月 06 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 Graduate Institute of Networking and Multimedia on 06 December 2024 have examined a Master's Thesis entitled above presented by YI-LIN HSU (student ID: R11944042) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

黎士瑋	廖世偉	
（指導教授 Advisor）		
王紹睿	黃敬群	

系（所）主管 Director: 鄭卜壬

doi:10.6342/NTU202500063

致謝

感謝黎士瑋教授在我的研究過程中提供了許多的建議與深入的討論，讓我能夠完成研究。此外，也感謝黎士瑋教授在論文撰寫的最後階段不辭辛勞地細心檢閱並提出諸多修改建議，給了我在論文寫作上相當大的幫助。同時，也感謝口試委員們在口試過程中及前後的信件中提出的建議與提問，使我的論文討論更加完善。

徐翊凌

中華民國一百一十四年一月

doi:10.6342/NTU202500063

摘要

作業系統（OS）如 Linux 藉由系統呼叫介面向用戶提供服務，但其龐大的程式碼基礎（幾乎是使用不安全的程式語言撰寫）使其易於受到攻擊。2014 至 2024 年七月十三日，Linux 核心報告了 1,866 個新的 CVE，顯示了保護這些系統的挑戰。由於大部分作業系統核心採用單體式架構，一個漏洞即可危及整個系統。核心區隔化藉由執行最小特權原則，將核心分隔成不同區隔，限制每個元件僅能存取其功能所需要的資料跟程式碼，以減少風險。

過去的區隔化工具，如 µSCOPE 和 TyPM，雖嘗試自動化此過程，但仍存在局限性。這些工具要麼因不完整的動態程式碼分析導致區隔特權不足，要麼因基於資料型別授權導致區隔特權過多，使攻擊者得以利用漏洞去攻擊與該漏洞無關的系統元件。此外，這些工具仍需手動實作區隔化政策，導致採用過程費時且容易出錯。

此論文提出的框架，能自動將 Linux 核心元件進行細緻的區隔化。此框架支持針對特定程式路徑（如系統呼叫處理路徑）進行自訂區隔，並透過函式呼叫關係圖分析自動生成區隔化策略。框架定義了共享資料抽象 API，包括：(1) 用於分配資料至區隔的共享資料指派 API，以及 (2) 控制資料存取的共享資料存取 API（getter 和 setter）。自動化將記憶體操作的指令轉換為 API 呼叫，使監控器能夠強制執行共享資料存取的權限。

doi:10.6342/NTU202500063

我們的實作基於 MLTA 型別分析工具和 LLVM，能自動識別和修改 Linux v5.15 中的程式路徑上的程式碼。我們評估了此框架在多個系統呼叫（如 KVM_CREATE_VM）中的應用，證明其在減少手動修改的量之外，也能提升了核心安全性。

關鍵字：系統呼叫、靜態分析、系統可用性、核心區隔

doi:10.6342/NTU202500063

Abstract

Operating system (OS) kernels, such as Linux, expose a system call interface to users, but their extensive codebases—often written in unsafe languages—leave them vulnerable to exploitation. Between 2014 and July 13, 2024, 1,866 new CVEs were reported in the Linux kernel, illustrating the challenge of securing such systems. As most OS kernels are monolithic, a single vulnerability can compromise the entire system. Kernel compartmentalization, which conducts the principle of least privilege, mitigates this risk by dividing the kernel into isolated compartments, restricting access to only the data and code necessary for each component's function.

Past compartmentalization efforts, including μSCOPE and TyPM, have attempted to automate the process but remain limited. These tools either under-privilege compartments due to incomplete runtime analysis or over-privilege them by granting unnecessary access based on data types, which still allows attackers to exploit vulnerabilities to target parts of the system that are unrelated to the original vulnerabilities. Furthermore, they

doi:10.6342/NTU202500063

require manual implementation of the compartmentalization policies, making adoption time-consuming and error-prone.

This thesis introduces a framework for automatically compartmentalizing Linux kernel components with fine granularity. It supports custom compartmentalization of code paths, such as those handling system calls, using call graph analysis to generate compartmentalization policies. The framework defines a shared data abstraction API comprising: (1) shared data assignment APIs to allocate data to compartments, and (2) shared data access APIs (getters and setters) to enforce controlled data interaction. Automatic instrumentation transforms memory operations into API calls, enabling a monitor to enforce permissions on shared data access.

Our implementation, based on the type analysis tool MLTA and LLVM, automates the identification and instrumentation of code paths in Linux v5.15. The framework was evaluated by compartmentalizing several system calls, including KVM_CREATE_VM, demonstrating its effectiveness in reducing manual effort while enhancing kernel security.

Keywords: System Call, Static Analysis, System Availability, Kernel Compartmentalization

doi:10.6342/NTU202500063

Contents

doi:10.6342/NTU202500063

doi:10.6342/NTU202500063

doi:10.6342/NTU202500063

List of Figures

doi:10.6342/NTU202500063

List of Tables

doi:10.6342/NTU202500063

Chapter 1 Introduction

Operating system (OS) kernels provide services to users by exposing the system call interface. However, due to its extensive codebase, often written in unsafe languages, the system is constantly exposed to new vulnerabilities, making it susceptible to exploitation or paralysis by attackers who discover weaknesses in system calls. For instance, between 2014 and July 13, 2024, 1,866 new CVEs were reported in the Linux kernel over the decade [35]. While eliminating all vulnerabilities is unlikely, limiting their impact is critical for security. Most OS kernels, including Linux, are monolithic. This means a single vulnerability can compromise the entire system due to the lack of fault isolation among kernel components.

One effective method to address this issue is kernel compartmentalization [24, 33], which systematically enhances security by enforcing the principle of least privilege (PoLP) [34]. This approach divides the kernel into separate compartments, permitting each compartment to access only the data and execute the code essential to its specific function, thereby restricting unnecessary access and control flow. Compartmentalization reduces the attack surface to confine exploitation and the resulting damage across the system.

Past systems [6, 23, 24, 26, 27, 32, 36–38] attempted to compartmentalize components in the monolithic Linux. Some of them [26, 27, 37, 38], incorporate a monitor to

doi:10.6342/NTU202500063

enforce the isolation mechanism across compartments. The compartment needs to access shared data, such as variables in the data and heap sections, to ensure that changes to system-wide data are visible to multiple kernel components, including system calls, device drivers, and kernel modules. In the monitor-based compartmentalization mechanism, compartments must access shared data through the Monitor to ensure controlled, secure access, preventing unauthorized or arbitrary interactions with shared data. However, those works require manual effort to retrofit the system to achieve compartmentalization. In the Linux kernel, even a single system call like `KVM_CREATE_VM`, requires the developers to analyze around 2133 lines of code and 330 object fields. This makes compartmentalizing a large system difficult and requires significant developer effort because the compartments have to communicate with each other by accessing a large number of shared data.

To efficiently compartmentalize a large codebase like the Linux kernel, it is crucial to automatically analyze both the subjects—such as specific functions or lines of code—and the objects, which refer to shared data, that belong to each compartment. Additionally, we must automatically determine the permissions granted to subjects for operations on objects, such as load or store. We define the set comprising objects, subjects, and permissions as a compartmentalization policy. Moreover, we also have to provide a set of APIs to express what shared data should be assigned to the compartment and abstract access to different types of shared data, allowing us to express the compartmentalization policy in the system, restricting compartments to access shared data via the predefined shared data access API. We also have to transform the memory operations to shared data access API, allowing the monitor to enforce the read-write permission when compartment access the shared data. For instance, the monitor can verify whether a compartment is permitted to perform read or write operations on certain memory address ranges.

doi:10.6342/NTU202500063

Defining the compartmentalization policy for complex systems software like Linux requires high expertise, and it is time-consuming, error-prone, and lacks scalability. This can lead to inconsistent policies, where some compartments either lack access to necessary data or have unnecessary access to unrelated data, increasing the risk of overlooking critical data flows or dependencies. Therefore, automating the generation of the compartmentalization policy is critical. This enables developers to determine what code or shared data should be placed inside the compartment without manual effort. Recent works such as μSCOPE [33] and TyPM [21] focus on generating compartmentalization policies. However, they only provide analyses to determine which code and shared data should be included in each compartment. However, developers must manually modify Linux to implement the desired compartmentalization (e.g., creating APIs, enforcing isolation, and managing synchronization). This significantly limits the adoption of these tools.

The compartmentalization policies provided by existing tools like μSCOPE and TyPM are often imprecise. μSCOPE can lead to under-privileged compartments because it relies on dynamic analysis, which may not capture all possible data flows during runtime, resulting in compartments lacking necessary data access and causing execution errors. Conversely, TyPM can lead to over-privileged compartments because it grants permissions based on data types accessed by the compartment, potentially including all data of that type, which increases the risk of data misuse. Both challenges highlight the difficulty of achieving precise compartmentalization in real-world systems.

KSplit [8] compartmentalizes untrusted device drivers to isolate them from the rest of Linux. It provides an automated method for partitioning the kernel and driver, making significant progress towards effortless compartmentalization of a large codebase. This approach eliminates manual efforts from programmers to modify the isolation target. How-

doi:10.6342/NTU202500063

ever, KSplit focuses on isolating device drivers from the rest of Linux. It provides a coarse-grained and inflexible compartmentalization policy. KSplit does not support compartmentalizing components with custom granularity within the kernel, such as functions used in a code path to handle system calls.

This thesis presents a framework for automatically compartmentalizing components in the Linux kernel at fine-grained granularity. Because system calls serve as the interface for the OS to provide its service to the users, they often become the target of the attacker to conduct the attack toward the OS, such as denial of service, privilege escalation, and so on. Therefore, we choose system calls as the primary target for automatic compartmentalization. This work supports compartmentalizing custom kernel components used by a selected call path of the system calls. The framework performs call graph analysis to automatically generate compartmentalization policies defining boundaries for individual compartments. Additionally, this framework provides a compartmentalization abstraction API, which defines the primitives governing the interaction between compartments and shared data. The compartmentalization abstraction API consists of two components: the shared data assign API, which defines the shared data belonging to the compartment, and the shared data access API in the form of getter and setter functions, allowing compartments to read from (getter) and write to (setter) various types of shared data. Further, the framework supports automatic instrumentation of compartments' code that accesses shared data. Our approach enables a monitor-based compartment system to monitor shared data accesses made by compartments.

Our implementation leverages the type-based analysis tool MLTA [22] to track all the functions used within a system call, as the LLVM [13] does not analyze the callees of indirect calls, which are prevalent in Linux. This approach eliminates the need for de-

doi:10.6342/NTU202500063

velopers to manually check the boundaries of the functions used in a code path to handle system calls. The current work implemented an LLVM Pass in LLVM version 14.0.6 to generate compartmentalization policies that define which shared data should belong to a compartment and instrument the memory operations for those shared data with shared data access APIs (getter and setter). The system leverages the compartmentalization mechanism through these APIs to enforce compartment policies. The current implementation includes a record-replay-based Monitor, designed to support crash recovery for Linux compartments [38].

The KVM hypervisor integrates the Linux kernel to leverage its operating system functionality for building the hypervisor. However, this integration increases the complexity of the KVM, potentially introducing more bugs and vulnerabilities [28–30]. Therefore, we used the prototype framework to compartmentalize several KVM system calls in Linux v5.15: `KVM_CREATE_VM` for creating a virtual machine, `KVM_CREATE_VCPU` for creating a virtual CPU, and `KVM_SET_GSI_ROUTING` for configuring the Global System Interrupt (GSI) routing table. Moreover, we implement a system call to demonstrate typical system call patterns and apply compartmentalization. Additionally, we compartmentalize the network packet transmit system call provided by the nullnet driver, which emulates a network adapter for packet transmission. Our prototype demonstrates the capability to automatically identify functions within a code path, define compartment boundaries, and compartmentalize components in Linux kernel v5.15 effectively.

In this thesis, we first introduced the background of compartmentalization, call graph analysis, and KVM in chapter 2. Then, we elaborated on the design of the automatic analysis and instrumentation framework in chapter 3. Afterward, we presented the implementation details in chapter 4. In the end, we show the evaluation of the KVM system calls

doi:10.6342/NTU202500063

after the automatic compartmentalization in chapter 5 and discuss the related works in chapter 6. The summary of the thesis is in chapter 7.

The primary contributions of this thesis are:

- We proposed the design of automatic analysis and instrumentation of system compartmentalization at fine-grained granularity.
- We proposed the API to allow the monitor-based compartmentalization system to monitor the access of the shared data made by the compartment.
- We implemented the framework of the automatic analysis and instrumentation based on existing LLVM compiler infrastructure.
- We demonstrated applicability by compartmentalizing the components in the code path of system calls in Linux kernel v5.15.
- We reduce the manual effort for compartmentalizing the components inside the Linux kernel.

doi:10.6342/NTU202500063

Chapter 2　Background

In this chapter, we will explain the concept of kernel compartmentalization, the technology of the call graph analysis, and the background of the KVM. The compartmentalization can allow the system to be executed following the PoLP. The call graph analysis can assist us in automatically identifying the component used on the call path of the system call. Our study treats KVM as the starting point for automatic compartmentalization because failures in other kernel components might only impact a single operating system, a failure in KVM can affect multiple virtual machines running on it. This makes KVM a critical component that requires compartmentalization even more than the other kernel components.

doi:10.6342/NTU202500063

2.1 Compartmentalization

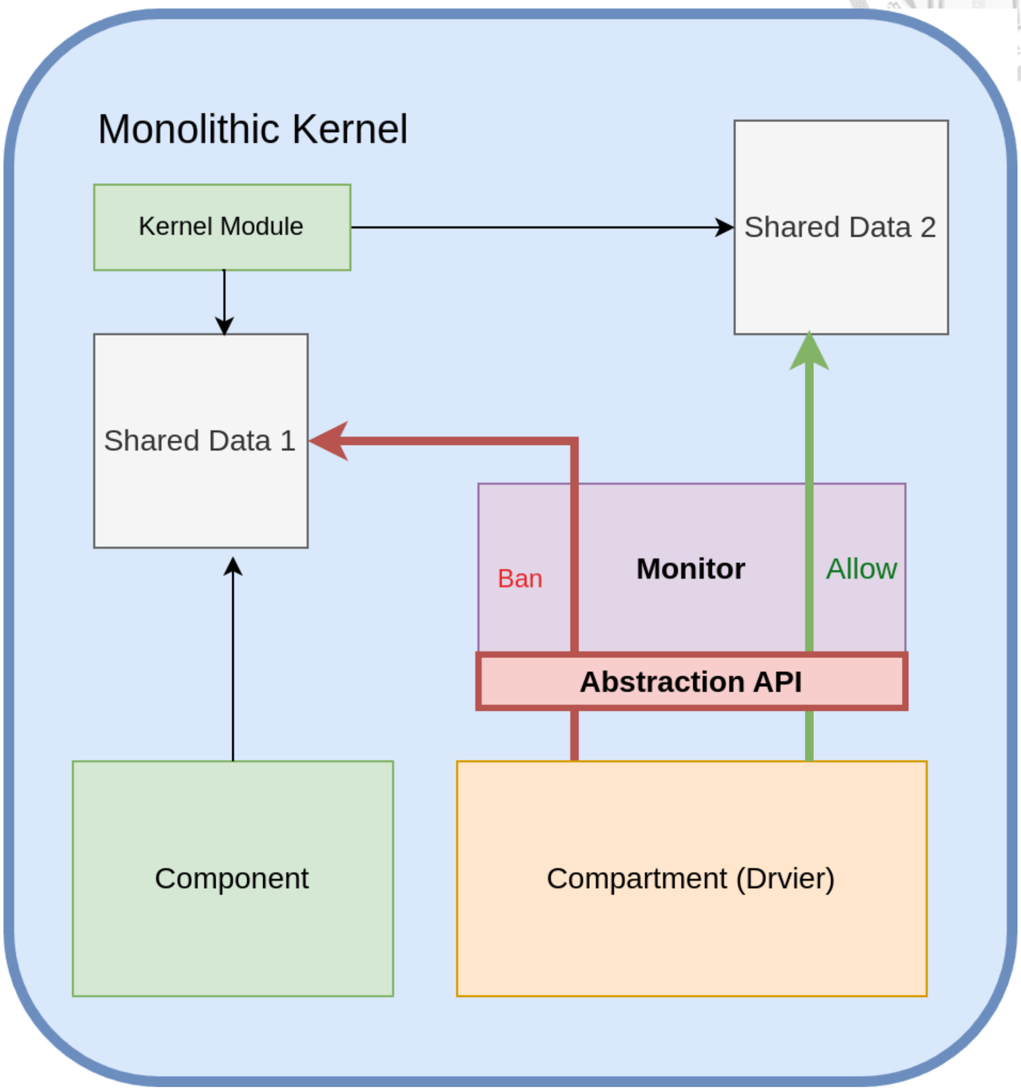

Figure 2.1: Compartmentalization System

Compartmentalization [14] is a security technique that divides a program into isolated parts, limiting each part to access only the shared data it needs, thereby enforcing the PoLP on large codebases like Linux.

Due to the Linux kernel's monolithic structure, a single vulnerability can compromise the entire system, and its large codebase makes manual compartmentalization impractical. Therefore, automating compartmentalization for the Linux kernel is essential.

doi:10.6342/NTU202500063

Compartmentalizing the Linux kernel involves addressing three key questions:

(1) How to determine which lines of code and shared data should be grouped together, and what access permissions (e.g., read or write) for divided code group to access the shared data. This question is addressed by the compartmentalization policy.

(2) How to express the defined policy in the program. This question is addressed by the compartmentalization abstraction.

(3) How to enforce the policy during runtime. This question is addressed by the compartmentalization mechanism.

2.1.1 Compartmentalization Policy

Before defining the compartmentalization policy, we first define the **subject**, **object**, and **permission** from Saltzer and Schroeder [34] and Miller [25]. The **subject** defines which lines of the code should be compartmentalized. The **object** defines the data, such as global variables or dynamically allocated variables, with which the subject is allowed to interact. The **permission** defines what actions that subject can perform on the object(eg., read, write).

The **compartment** is a set of the code (subject) that has the same permission to access the object. The **compartmentalization policy** is a concept of determining the subject, the object, and the permission for the compartment.

doi:10.6342/NTU202500063

2.1.2 Compartmentalization Abstraction

A compartmentalization abstraction defines the basic primitives to implement for expressing the separation policy in a program. Assigning the shared data to the compartment and allowing the compartment access to the shared data in the predefined permission, like reading and writing, need to be performed under the defined policy. Consider figure 2.1, this can prevent the compartment from arbitrarily obtaining ownership of the shared data and accessing them without control.

2.1.3 Compartmentalization Mechanism

A compartmentalization mechanism is a technique to enforce the compartmentalization policy at runtime through the implementation of a compartmentalization abstraction. Notably, numerous past works [26, 27, 37, 38] have introduced an independent trusted component, referred to as the **Monitor**, as shown in figure 2.1.

The monitor is responsible for managing the metadata required to maintain the compartment execution. For instance, the monitor of the BULKHEAD [37] will maintain the metadata about what shared data should be assigned to the compartment and what permission, such as read or write, is allowed the compartment to perform on the shared data.

The monitor is responsible for enforcing the defined compartmentalization policy for the compartment when assigning the shared data to it and accessing the shared data. This is achieved through two fundamental methods: message passing and shared memory. Using message passing[26, 27], the monitor facilitates the assignment of shared data by transmitting it over a communication channel (e.g., pipes or POSIX sockets) under the defined

doi:10.6342/NTU202500063

permissions. For shared memory[37], the monitor ensures that only shared data explicitly permitted under the compartmentalization policy is accessible to the compartment. This process is supported by hardware features such as PKS[9]. The monitor will leverage this hardware feature to verify whether the shared data should be assigned to the compartment and to determine the access permissions, such as read or write, that the compartment can use when accessing the shared data.

2.2 Call Graph Analysis

The call graph is a graph that embodies the calling relationships among subroutines or functions in a computer program. Each node in the graph represents a function and the edge between the nodes represents a calling relationship among them.

The construction of the call graphs can help developers enforce the control flow integrity and detect the vulnerabilities of the large code base, like privilege escalation, buffer overflows, etc.

2.2.1 LLVM Call Graph Analysis

The LLVM compiler infrastructure [13] can construct the call graph[16] itself. The LLVM constructs a call graph to aid in interprocedural optimization by mapping out the calling relationships between functions within a module. In the graph, each function is represented as a node, which tracks the functions it calls. A call graph includes nodes for functions that are null, termed "external nodes", to account for control flows that are not analyzable within the module. These external nodes serve two primary purposes. First, they represent functions without internal linkage or those whose addresses are used for

doi:10.6342/NTU202500063

indirect calls, indicating potential invocations by external functions. Second, they capture outgoing calls from functions that are external or contain indirect calls, reflecting their ability to call any function without internal linkage or those with taken addresses.

This structure ensures that the call graph encompasses a conservative superset of all possible caller-callee relationships, which is essential for various compiler optimizations [17–19]. However, if the module contains an indirect call instruction, this callee of the indirect call will be mapped to an external node. Therefore, the target of the indirect call is missing and the call graph is incomplete.

2.2.2 First-Layer Type Analysis

To address the problem that indirect calls hinder the construction of a precise call graph, the type-based matching method, first-layer type analysis [5], collects all indirect calls and recognizes the function-pointer type of them. Then, it analyzes the entire program to gather address-taken functions containing the identical type as indirect calls' function-pointer type.

Although first-layer type analysis is computationally efficient and straightforward to implement, it frequently results in a high number of false positives—indirect calls that use generic parameters, such as void (*)(char *), often correspond to numerous unrelated function targets. This occurs because it does not consider the layered context in which function pointers are stored and utilized, leading to many irrelevant functions being included as potential targets. Consequently, first-layer type analysis offers only a basic level of precision due to its lack of accuracy. This limitation underscores the need for more sophisticated analysis techniques that can account for the full context of function pointer usage.

doi:10.6342/NTU202500063

2.2.3 Multi-Layer Type Analysis

```c
struct kvm_pgtable_mm_ops {
  void*   (*zalloc_page)(void *arg);
        ...
};

struct stage2_map_data {
        ...
  struct kvm_pgtable_mm_ops *mm_ops;
};

struct hyp_map_data {
        ...
  struct kvm_pgtable_mm_ops *mm_ops;
};

static void *kvm_hyp_zalloc_page(void *arg)
{
  return (void *)get_zeroed_page(GFP_KERNEL);
}

static void *stage2_memcache_zalloc_page(void *arg)
{
  struct kvm_mmu_memory_cache *mc = arg;

  /* Allocated with __GFP_ZERO, so no need to zero */
  return kvm_mmu_memory_cache_alloc(mc);
}

static struct kvm_pgtable_mm_ops kvm_hyp_mm_ops = {
  .zalloc_page    = kvm_hyp_zalloc_page,
        ...
};

static struct kvm_pgtable_mm_ops kvm_s2_mm_ops = {
```

doi:10.6342/NTU202500063

```
  .zalloc_page    = stage2_memcache_zalloc_page,
        ....
};

struct hyp_map_data map_data = {
    .mm_ops = &kvm_hyp_mm_ops,
    ....
};

struct stage2_map_data map_data = {
    .mm_ops   = &kvm_s2_mm_ops,
    ...
};

static int stage2_map_walk_leaf( ..., struct stage2_map_data *data)
{
  struct kvm_pgtable_mm_ops *mm_ops = data->mm_ops;
  kvm_pte_t *childp = mm_ops->zalloc_page(...);
        ...
  return 0;
}
```

Listing 2.1: Type Analysis based Code Example

To obtain a more precise indirect call target function, Multi-Layer Type Analysis (MLTA) [22] operates by utilizing the layered storage of types in system software to identify indirect call targets more accurately. The core insight of MLTA is that function pointers are commonly stored in instances with multi-layer type hierarchies, and before an indirect call is made, these pointers are loaded through multiple layers of types. Therefore, when the MLTA encounters the indirect, it perform the following operations.

(1) It identifies all the address-taken functions that are the same type as the indirect call's function pointer. Consider listing 2.1, `stage2_memcache_zalloc_page` and

doi:10.6342/NTU202500063

`kvm_hyp_zalloc_page` both are the address-taken functions because both of their address is assigned to the global variable. It compares their function type with the indirect call's function pointer type in line 52. It concludes that both of them are the possible callees for this function pointer.

(2) It refines possible callees by comparing the type of instance that holds the function with the type of instance that holds the function pointer layer by layer. Consider listing 2.1. It found the instance that holds `stage2_memcache_zalloc_page` with the layer of type, `struct kvm_pgtable_mm_ops` and `struct stage2_map_data`. Then, It compares it with the type of instance that holds the function pointer at line 52 layer by layer. It found that their types are consistent. It concludes that `stage2_memcache_zalloc_page` is the callee of the indirect call in line 52. In contrast, the instance holding `kvm_hyp_zalloc_page` does not have the same layer of type as the indirect call in line 52. As a result, this function can be excluded as a possible callee.

MLTA offers two notable advantages. First, multi-layer types impose stricter constraints compared to single-layer types, effectively reducing false positives. The target set identified by single-layer analysis represents a broader collection than the refined set obtained through multi-layer analysis, as the latter narrows down the possibilities. Second, multi-layer type analysis provides flexibility to mitigate possible false negatives. While a type containing additional layers offers greater precision in limiting indirect call targets, MLTA can adapt by falling back to more general sub-types when detailed type information is unavailable, ensuring accurate results without increasing false negatives.

doi:10.6342/NTU202500063

2.3 Kernel-based Virtual Machine

Kernel-based Virtual Machine (KVM) is a type-2 hypervisor integrated into the mainline Linux kernel in 2007, enabling the kernel to serve as a hypervisor. By leveraging existing kernel components, such as memory management, scheduling, and so on, KVM can efficiently run multiple virtual machines on a single physical host.

On x86 platforms, KVM leverages hardware virtualization features such as Intel VMX to partition CPU operations into two distinct modes: root mode for the host and non-root mode for virtual machines. Single instructions like VM Entry and VM Exit handle the saving and restoring of the processor's state directly in hardware, enabling efficient transitions between the host and VMs. In contrast, KVM on ARM must address the limitation that the host OS cannot operate in EL2, the privilege level designated for virtualization. To overcome this, KVM ARM employs split-mode virtualization [4], placing most hypervisor functionality in EL1 alongside the host kernel while maintaining a minimal lowvisor at EL2. Since ARM does not provide a single hardware mechanism to save and restore the full CPU state, KVM ARM must manually switch contexts in software, which introduces additional overhead compared to the virtualization hardware support on x86.

Recently, KVM ARM introduced VHE mode, which allows the Linux kernel to run entirely in EL2. VHE mode eliminates the need to separate KVM ARM into different CPU privilege levels. By operating the Linux kernel fully in EL2, guest EL1 states do not need to be restored during VM entry or exit, thereby reducing overhead. KVM ARM now supports both VHE mode and continues to maintain split-mode virtualization as NVHE mode.

doi:10.6342/NTU202500063

Chapter 3 Design

Modifying a large codebase like Linux to integrate a compartmentalization system is a complex undertaking that demands significant engineering effort. This framework aims to automate the compartmentalization of components within Linux, minimizing the need for developer effort. In this section, we first define the essential elements that constitute compartmentalization for this framework. Based on this definition, this framework demonstrates how to analyze these elements within the kernel component targeted for compartmentalization and then automatically instrument the component to achieve compartmentalization.

3.1 Compartmentalization Policy of the Framework

The compartmentalization policy can be defined with variety according to the demand or the goal of the developer. we will first define the compartmentalization policy for this framework by specifying the subject, the permissions, and the objects to be analyzed.

The granularity of the subject can be a function, a system call, a c file, or a system call. Most of the past works [8, 24, 27, 37] choose coarse-grained granularity, driver, or c files, as their subject's granularity because this granularity can be easily defined by the

doi:10.6342/NTU202500063

developer without further analysis. For instance, if the developer defines the driver as the subject to compartmentalize, they can simply select the c file related to the driver to instrument. However, if they defined the subject as the more fine-grained level, such as system level, they have to identify the functions that are called during the system call execution through either manual analysis or static analysis method [5, 7, 16, 22]. Moreover, the coarse-grained granularity of the subject only stretches the surface of the PoLP. Therefore, although this framework can compartmentalize Linux in any granularity, it chooses functions used in a code path to handle system calls as the granularity of the subject to demonstrate its ability to analyze and instrument kernel at a more fine-grained level. The analysis methods will be detailed in section 3.4.1

The shared data are system-wide visible to the components in Linux, such as kernel modules, device drivers, and system calls, etc. In the Linux kernel, per-CPU data is designed to provide each CPU with its own private copy of the data. This design eliminates the need for synchronization primitives when a processor accesses its own copy, thereby avoiding performance degradation caused by synchronization overhead. Although PER-CPU data are not shared directly across processors, they remain system-wide visible in the Linux kernel. As a result, it is still classified as shared data within the system. In this framework, the object is defined as the shared data that a subject interacts with. The permission for the subject to access the object is defined by whether the subject has the load or store instructions that operate on the object. The analysis in section 3.4.2 will detail how to determine which shared data the compartment is allowed to access precisely and what permission, write or read, is given to the compartment for access to the shared data.

doi:10.6342/NTU202500063

Table 3.1: Shared Data Assign API

Shared Data Assign API	**Description**
`int new_shared_data(void *var_addr, u64 size)`	`var_addr`: the address of shared data `size` : bytes of shared data

3.2 Compartmentalization Abstraction API

The compartmentalization abstraction API is a set of APIs that express the compartmentalization policy in the program. The compartmentalization abstraction API here comprises two types of the API. The Shared Data Assign API defines the shared data that should be assigned to the compartment and the Shared Data Access API defines the access permission, read or write, that the compartment can perform on these shared data.

3.2.1 Shared Data Assign API

In the design of a monitor-based compartmentalization mechanism, The monitor is responsible for assigning shared data to the compartment. This allows the monitor to restrict the data the compartment can access, ensuring that compartments only have privileges for objects explicitly assigned to them. This framework defines Shared Data Assign API in table 3.1 to define which shared data should belong to the compartment.

The `new_shared_data` provides the interface for instructing the monitor to assign the shared data to the compartment. The first argument provides the monitor with the address of the original shared data, and the second argument specifies the number of bytes of shared data to be assigned. Using this information, the monitor can retrieve the data value from the shared data address within the defined size and then allocate it to the compartment.

doi:10.6342/NTU202500063

Table 3.2: Shared Data Access API

Shared Data Access API
void set_shared_data(void *shared_data_addr, u64 value)
u64 get_shared_data(void *shared_data_addr)

This API interface supports a variety of shared data. For primitive types, it simply assigns the entire variable to the compartment. For composite types, it allows defining the granularity of the shared data to be assigned to the compartment. Specifically, it can assign a field of a composite variable to the compartment by using the field's address as the `var_addr` argument and the field's byte size as the `size`.

3.2.2 Shared Data Access API

Once the shared data is assigned to the compartment, the compartment should access it under the monitor's control. This framework provides Shared Data Access API table 3.2, getter, and setter APIs, for the compartment to get and set the shared data managed by the monitor. `set_shared_data` serves as the setter to allow the compartment to set the shared data. The first argument, `shared_data_addr`, specifies the address of the shared data the compartment intends to write, and the second argument specifies the value the compartment intends to write to the shared data. The `get_shared_data` serves as the getter to authorize the compartment to read the value of shared data. The first argument specifies the address of the shared data the compartment intends to read.

3.3 Support for Monitor-Based Mechanism

In our framework, we aim to provide an automated analysis and modification method for integrating a monitor-based compartmentalization mechanism into the system, elimi-

doi:10.6342/NTU202500063

nating the need for manual code modifications in the components the developer seeks to compartmentalize. The monitor-based mechanism [26, 27, 37, 38] can leverage the abstraction API to regulate the compartments' access to shared data under the predefined compartmentalization policy. The monitor determines which shared data should be assigned to a compartment using the Shared Data Assign API. When the compartment attempts to access the shared data, the monitor controls access through the Shared Data Access API. The framework also provides the analysis and instrumentation to support the monitor-based compartmentalization mechanism.

3.4 Compartmentalization Analysis and Instrumentation

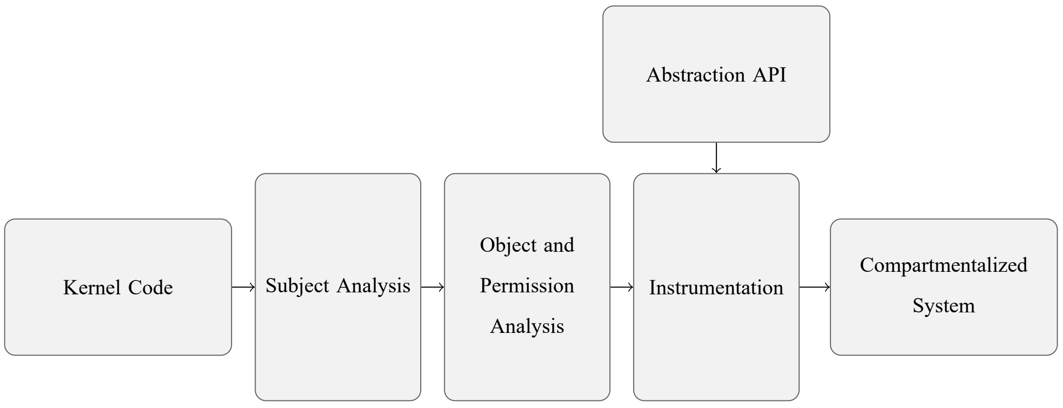

Figure 3.1: Compartmentalization Analysis and Instrumentation Workflow

This section begins by analyzing the subject, object, and permission that defines the compartment. We then instrument components within Linux to transform them into the compartment. Consider figure 3.1, the framework first analyzes the subject of the compartment, followed by analyzing the object and the permissions associated with it. Finally, it instruments the API into the kernel code, transforming it into a compartmentalization system.

```
struct B {
```

doi:10.6342/NTU202500063

```c
    int status;
    int flag;
};

int counter = 0;

int sys_call(unsigned int type)
{
    struct B* b;
    int r = 0;

    counter++;

    b = kzalloc(sizeof(*b), GFP_KERNEL_ACCOUNT);
    if (!b){
        return -1;
    }

    b->flag = 2;
    r=b->flag;

    ....
    return r;
}

int instrumented_sys_call(unsigned int type)
{
    struct B* b;
    int r = 0;
    new_shared_data(&counter, sizeof(counter));
    //counter++;
    int tmp = get_shared_data(&counter);
    set_shared_data(&counter, tmp + 1);

    b = kzalloc(sizeof(*b), GFP_KERNEL_ACCOUNT);
    if (!b){
```

doi:10.6342/NTU202500063

```c
        return -1;
    }
    new_shared_data(&b->flag, sizeof(b->flag));
    //b->flag = 2;
    set_shared_data(&b->flag, 2);
    //r=b->flag;
    r = get_shared_data(&b->flag);
    ....
    return r;
}
```

Listing 3.1: Instrumentation Code Example

3.4.1 Compartmentalization Subject Analysis

According to the discussion in section 3.1, we define the granularity of the subject as the functions used in a code path to handle system calls. To compartmentalize a system call, the first step is to identify the functions that are used by the target system call. To achieve this, we need to generate a comprehensive control flow graph (CFG) of the Linux kernel. LLVM can automatically generate a program's CFG. However, because Linux developers emphasize code reusability and aim to leverage object-oriented programming principles within the kernel to increase the flexibility of the kernel code [31], they frequently use function pointers to create callback functions, resulting in many indirect calls. For instance, the KVM hypervisor code that walks the hypervisor's page table often involves callbacks to trigger different functionalities during the page table walk.

In our framework, we utilize indirect call target analysis to identify the subjects of the compartment, showcasing a novel use case for this analysis.

Because the limitation of LLVM is its inability, by default, to infer the set of target

doi:10.6342/NTU202500063

functions for these indirect calls, when we encounter an indirect call, we perform a type-based analysis in section 2.2.3 to infer the callee of an indirect call.

After finishing the analysis process, we obtain a comprehensive control flow graph of the Linux kernel. Starting from the function that invokes the system call, we traverse the control flow graph to identify all functions called along this path. The functions collected through this traversal are then considered as the subjects of the compartment.

3.4.2 Compartmentalization Object and Permission Analysis

After analyzing the subject of the compartment at the compartmentalization subject analysis phase in section 3.4.1, we obtain the set of the functions used in a code path to handle system calls as the subject of the compartment. In the analysis phase, we identify the field of shared data that must be assigned to the compartment, defining it as the compartment's object. Additionally, we analyze the permissions—either read or write—that the compartment is allowed to operate on for each assigned object.

The first step is to iterate through each load and store instructions within the subject of the compartment because those memory operation instructions can potentially access the shared data.

In the second step, we analyze those memory operation instructions through the following static analysis to identify whether they access the shared data. We perform the following steps:

(1) We build a working list and select variables used in memory operation instructions as the initial variable for analysis. For load instructions, we choose the variables from which they load, and for store instructions, we select the variables to which they store to.

doi:10.6342/NTU202500063

(2) We pop the working list to obtain the variable to analyze and set it as the current variable. We add all the variables used to define the current variable into the working list and add the current variable to the use define chain set.

(3) If the current variable is a function argument, we use the call graph to locate the caller of this function, add the corresponding argument of the caller to the working list, and then proceed to step 2.

(4) The algorithm will finish until all the variables in the working list finish analysis.

(5) We check the use define chain set to find whether the variable inside it has the global variable or the variable allocated by the kernel allocation function. If so, we consider these shared data variables as the object of the compartment. The memory operations identified in our analysis are defined as the permissions, read or write, the compartment is allowed to perform on those objects.

Consider listing 3.1. In line 20, this line of code is to store a value to the variable. After the analysis process, we found that this line of code performs the store action on the allocated heap data defined at line 15. We define that the field, `flag`, of the variable, `b`, should be the object of the compartment, and the compartment has the permission to write the field, `flag`, of the variable, `b`.

3.4.3 Compartmentalization Instrumentation

After the previous analysis phase in section 3.4.2, we automatically instrument with the compartmentalization abstraction API defined in section 3.2. For the objects we analyze in the section 3.4.2, we instrument Shared data Assign API before the object is used to notify the monitor to assign the shared data to the compartment. For instance, con-

doi:10.6342/NTU202500063

sider listing 3.1, before writing the global variable, `counter`, in line 13, and the field of the dynamically allocated memory object that the pointer `b` points to, we instrument the `new_shared_data` API to instruct the monitor to assign the whole shared data or a field of shared data to the compartment.

According to the section 3.4.2 phase, we replace the load and store operations with the getter and setter APIs, as detailed in table 3.1. Consider listing 3.1, after this replacement, the compartment can only access shared data through the predefined API, ensuring the Monitor can strictly control all shared data usage. For instance, line 20 and the line 21 would be replaced as the `set_shared_data` and the `get_shared_data` respectively. The `instrumented_sys_call` in listing 3.1 is the result of the code after instrumentation.

doi:10.6342/NTU202500063

Chapter 4 Implementation

We implement the framework based on the design. The framework comprises the static analysis tool MLTA [22] to analyze the subject of the compartment, and the extended LLVM of version 14.0.6 to compartmentalize the functions on the code path to handle the system call inside the Linux kernel automatically, which consists of 1840 LOC. LLVM version leveraged to build MLTA analysis tool [12] is version 14.0.6. LLVM version 14.0.6 is compatible with the existing build process of Linux kernel [15] to produce LLVM IR from the Linux kernel source for our framework. We select the record-replay-based monitor [38] as the compartmentalization mechanism of this framework. Therefore, this framework also provides the customized shared data access API in section 4.1.3 for the record-replay-based monitor [38] to support its record and replay functionality.

4.1 Customized Compartmentalization Abstraction API

We will first present the compartmentalization mechanism provided by the record-replay-based monitor [38]. Following this, we will introduce the customized shared data assign API specifically designed for the record-replay-based monitor, enabling seamless integration with this framework.

doi:10.6342/NTU202500063

4.1.1 Record-Replay-Based Monitor

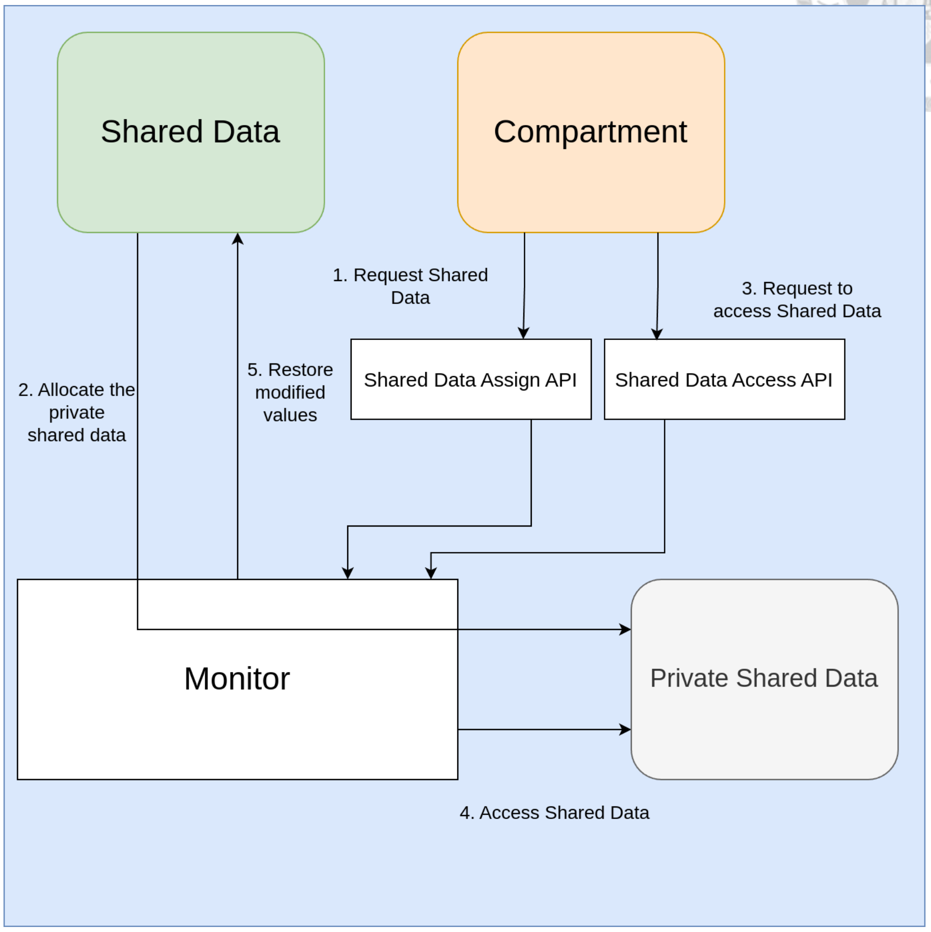

Figure 4.1: Record-Replay-Based Monitor Workflow

The monitor [38] provides the mechanism to allow the shared data that the compartment used inside to remain unmodified until the compartment finishes execution. Consider the workflow in figure 4.1, when the compartment starts to execute, the monitor allocates the private shared data for the compartment according to the shared data assigned to it. Then, the monitor copies the value of an original shared data to private shared data. The monitor will handle all access to the private shared data requested from a compartment. Once the compartment finishes execution without any errors that cause the compartment to crash, such as the null pointer dereference or Use-after-free, the monitor will copy the value on the private shared data back to the original shared data.

doi:10.6342/NTU202500063

Table 4.1: Shared Data Assign API

Shared Data Assign API
int new_global(void *var_addr, u64 size)
int set_mem_obj(void *obj_addr)
int free_mem_obj(void *obj_addr);
int new_field(void *base_addr, void *field_addr, u64 size)

4.1.2 Concurrency Model

Regarding the concurrency model, our design leverages existing critical sections within the kernel to ensure that the getter and setter operations on shared data are executed within a critical section. For atomic read and atomic write operations, the getter and setter disable interrupts during execution to guarantee atomicity.

However, from the perspective of our monitor implementation, to support record and replay functionality, all modifications to shared data are written back only after the compartment finishes execution. This imposes certain implementation constraints: the system must operate in a non-preemptive configuration to prevent scenarios where a compartment exits a critical section but gets preempted before its modifications to shared data are written back. Additionally, a single-core environment is required to avoid race conditions arising from tasks on different cores competing for the same critical section. For instance, if one compartment exits a critical section without writing back its shared data changes and another task on a different core acquires the same critical section, it could lead to inconsistent shared data and race conditions.

4.1.3 Customized Shared Data Assign API

According to the mechanism of the record-replay-based monitor, when shared data is allocated in the heap section, the following scenarios may occur:

doi:10.6342/NTU202500063

(1) The shared data is allocated outside the compartment, and be deallocated inside the compartment.

(2) The shared data is allocated inside the compartment, and the shared data is then assigned back to the core kernel.

Therefore, the monitor must be aware of the lifetime of the shared data; otherwise, it risks reassigning deallocated shared data back to the core kernel or failing to assign newly created shared data to the core kernel after the compartment completes execution. This framework provides the **Object Lifetime API** to allow the monitor to track the lifetime of the shared data. The `set_obj_mem` will inform the monitor that new shared data are allocated. The `free_obj_mem` will inform the monitor that the shared data is deallocated.

A type of shared data is composite, containing fields of varying sizes to store different values. To handle this, the framework provides a `new_field` API, enabling the monitor to assign shared data to a compartment at the field level rather than as an entire one. The `new_field` includes an additional argument, `base_addr`, which specifies the starting address of the shared data containing the field. This information allows the monitor to track the field’ s lifetime, ensuring it does not reassign a deallocated field back to the core kernel after the compartment completes execution.

Since the lifetime of a global variable persists until the system shuts down, the monitor does not need to manage its lifetime. For shared data that is a global variable, we provide the `new_global` API. This API informs the monitor that the shared data is a global variable, so the monitor does not need to manage its lifetime.

doi:10.6342/NTU202500063

4.2 Compartmentalization Analysis and Instrumentation Framework

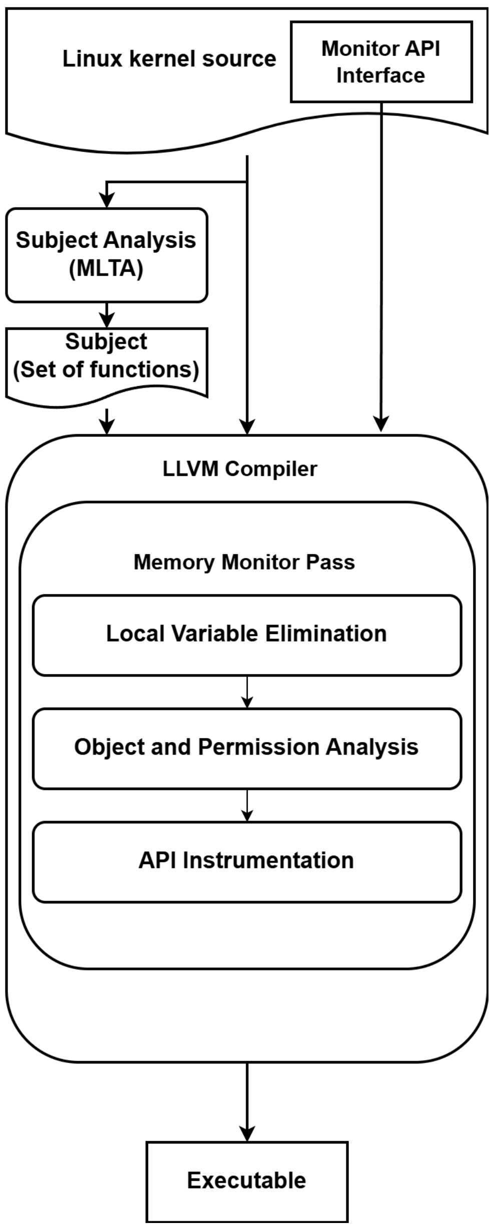

Figure 4.2: Compartmentalization Analysis and Instrumentation Architecture

This framework in figure 4.2 comprises three components. The first component is the static analysis tool MTLA [22], which is responsible for analyzing the subject of the compartment as discussed in section 3.4.1. The second component is the monitor API

doi:10.6342/NTU202500063

interface, which provides the API for the compartment to interact with the monitor as the discussion in section 4.1.3 and section 3.2. The third component is our self-implemented LLVM pass, called the memory monitor pass. This LLVM pass is in charge of compartmentalization analysis and instrumentation section 3.4. We integrate the memory monitor pass into the middle end of the LLVM 14.0.6, which will perform the analysis and the instrumentation on LLVM intermediate representation[20]. The following section explains how the components interact and outlines the workflow of the framework.

4.2.1 Workflow of the Framework

In the first step, subject analysis constructs a call graph of the entire Linux kernel source. Starting from the function that invokes the target system call, we traverse the call graph to identify the set of functions along the code path handling the system call. This set of functions is defined as the subject. In the second step, we input the subject and the function name of the monitor API interface, as we defined in section 4.1.3 and table 3.2, to our self-implemented memory monitor pass. The memory monitor pass will have information about the subject to be analyzed and the monitor API interface to be instrumented. In the fourth step, before performing the analysis and instrumentation, the memory monitor pass will first perform the local variables elimination to improve the efficiency of the following analysis. The detail will be explained in section 4.2.2. In the fifth step, the memory monitor pass analyzes the shared data used by the subject, referred to as the object, and determines the permissions (read or write) that a subject is allowed to perform on the shared data according to the algorithm in section 3.4.2. In the sixth step, the memory monitor pass will instrument the subject with the customized shared data assign API in section 4.1.3. The customized shared data assign API will define the object(shared data) that should be assigned to the compartment. The memory monitor pass will also instrument the shared data access API(getter and setter) in table 3.2 to define what permission the compartment can perform on the object. In the final step, the executable Linux kernel with a compartmentalized system call will be the output.

4.2.2 Local Variables Elimination

```c
struct A {
    int cnt;
    int flag;
};

struct A a;

int sys_call_before() {
    struct A *tmp = &a;
    int i = 0;
    i = 3;
    tmp->cnt = 1;
    tmp->flag = 3;
    return tmp->cnt;

}

int sys_call_after() {
    a.cnt = 1;
    a.flag = 3;
    return a.cnt;
}
```

Listing 4.1: Local Variables Elimination C Example

In the Linux kernel, developers may declare local variables to store the address of shared data. When leveraging the analysis in the section 3.4.2 to determine whether load

doi:10.6342/NTU202500063

Algorithm 1: Local variable Elimination Algorithm.

```
Function elimiateLocalVariable(AllocaInst):
    Stores = {};
    foreach Users of AllocaInst do
        if User is StoreInst then
            add User to Stores;
        end
        if User is LoadInst then
            store = Find the nearest preceding StoreInst in Stores;
            Value = the value that StoreInst store;
            replace the use of LoadInst with Value;
            delete LoadInst;
        end
    end
    delete all the StoreInst in stores;
    delete the AllocaInst;
```

or store instructions operate on shared data, we would encounter instructions that reference shared data via local variables in the CFG. Consequently, we must also analyze where the definition of these local variables comes from by analyzing the store instructions that store the address of shared data to local variables, which increase the analysis overhead in the section 3.4.2 because the more local variables are used to propagate the address of shared data, the more local variables are needed to be analyzed to find where their definition from.

For example, consider listing 4.1, when we leverage the analysis in the section 3.4.2 to analyze whether the store operation in line 12 stores a value to the shared data, we will encounter the local variable in line 9 first according to the CFG. Therefore, we also need to analyze where the definition of the local variable in line 9 comes from. In contrast, the store operation in line 19 of `sys_call_after` would not encounter the local variable and does not need to analyze where the definition of the local variable from because all the local variables in `sys_call_after` are all eliminated. Therefore, the overhead of analysis in section 3.4.2 decreases when analyzing the store operation inside the `sys_call_after` in line 19. Thus, we apply the algorithm 1 to eliminate local variables ahead of the analysis

doi:10.6342/NTU202500063

phase in section 3.4.2.

The algorithm 1 collects all store instructions that write values to the allocated local variable. For each load instruction, it first finds the nearest preceding store instruction, extracts the value stored by that instruction, and replaces all instances where the loaded value is used with the extracted value. The load instruction is then deleted. Once all load instructions are processed, the function deletes all store instructions and the original allocation. In the listing 4.1, the function `sys_call_before` represents the code before erasing local variables, while `sys_call_after` shows the function after processing with the algorithm 1.

doi:10.6342/NTU202500063

Chapter 5 Evaluation and Discussion

5.1 Performance Evaluation

We evaluated system calls in Linux v5.15, configured for the arm64 architecture. The host Linux system runs on the Raspberry Pi 4 Model B board, featuring the Broadcom BCM2711 system-on-chip with a quad-core Cortex-A72 (Armv8-A) processor and 4GiB of RAM. The system calls were executed in a tightly controlled setup, confined to a single CPU core to eliminate the potential impact of parallel processing. Furthermore, preemption was disabled to maintain consistent timing and prevent context switches during execution. This configuration ensured that we could precisely measure the performance and behavior of each system call in isolation, free from interference caused by multitasking or other system activities. The configuration is included in chapter 7. For the monitor that provides the compartmentalization mechanism, we leverage a monitor that can dynamically allocate memory for compartments, allowing the compartment to maintain its copy of the shared data.

We selected four system calls in the Linux kernel that frequently interact with shared variables, utilizing common C patterns to read from or write to various data types. These include primitive types (integer) and fields within composite data structures (like struct). For the time measurement, we utilize the timer functionality from the C standard library

doi:10.6342/NTU202500063

Table 5.1: System Call Performance Evaluation

System Call	Before	After
Demo_sys_call	0.000021s	0.000067s (3.19x)
KVM_CREATE_VM	0.000695s	0.002935s (4.62x)
KVM_CREATE_VCPU	0.000068s	0.001329s (19.54x)
KVM_SET_GSI_ROUTING	0.000385s	0.029363s (76.26x)

The table shows system calls' execution time before and after instrumentation.

Table 5.2: LOC Increment

System Call	Before	After
Demo_sys_call	37	74 (2x)
KVM_CREATE_VM	2133	5050 (2.37x)
KVM_CREATE_VCPU	1487	2716 (2.37x)
KVM_SET_GSI_ROUTING	330	611 (1.8x)

The table shows the increase in lines of code before and after the instrumentation.

to measure the execution time of system calls both with and without instrumentation. The system calls' execution time, with and without instrumentation, are presented in table 5.1.

5.1.1 Demo system call

```c
DEFINE_MUTEX(global_mutex);
DEFINE_SPINLOCK(global_spin_lock);
struct a_struct {
  int first, second;
};

struct a_struct var_a = { 1, 1 };
int global_var1 = -234;

noinline long doSomethingDemo(void) // normal syscall
{
  int i;
  int *lll;
  for (i = 0; i < 3; i++) {
    mutex_lock(&global_mutex);
```

doi:10.6342/NTU202500063

```

    global_var1 = global_var1 + 1;
    var_a.first = var_a.first + 1;

    mutex_unlock(&global_mutex);;
  }
  lll = kzalloc(sizeof(int), GFP_KERNEL);

  if (!lll) {
    pr_err("!!![%d][demo_syscall_a] kzalloc failed!!!\n", current->pid);
  }
  spin_lock(&global_spin_lock);
  spin_unlock(&global_spin_lock);

  kvfree(lll);

  return 5;
}

SYSCALL_DEFINE0(demo_syscall) {
    doSomethingDemo();
    return 0;
}
```

Listing 5.1: Demo system call

The Demo system call in listing 5.1 is a custom-designed system call that we implemented to emulate the behavior and patterns of conventional system calls within an operating system. This system call is specifically designed to simulate the core functionalities of standard system calls, particularly in terms of how they interact with and manipulate shared data. In practice, the Demo system call performs both write and read operations on a variable, which can be allocated either in the heap section or the data section of memory. This design allows us to closely mirror the memory management strategies typically

doi:10.6342/NTU202500063

employed by real system calls.

The overhead after instrumentation primarily arises from the additional instructions inserted to support the execution of compartmentalization. These extra instructions include generating the necessary arguments for getter and setter functions, loading the values of shared variables for the Monitor to record, and inserting APIs to collect information regarding the allocation status of these variables. As shown in Table , these additional instructions significantly contribute to doubling the lines of code.

Additionally, the overhead is further compounded by the internal functionality of the APIs provided by the Monitor. These APIs are responsible for several critical operations, such as allocating compartmentalized memory for shared variables, searching the hash table to either retrieve or modify data stored in the compartmentalized memory and, at the end of the execution of the compartmentalized system call, writing the updated values of shared variables from the compartmentalized memory back to their original memory locations.

The combined effect of these two factors—the increased number of instructions and the internal operations of the compartmentalization abstraction APIs result in a 319% increase in overhead, as shown in Table .

5.1.2 KVM Virtual Machine Creation System Call

This system call is responsible for creating virtual machine instances and initializing the metadata necessary for its operation. In general, most of the code patterns within this system call are focused on loading and storing values in the fields of composite-type shared variables. After instrumentation, the increase in overhead is largely attributed to the additional inserted instructions, similar to the overhead factors discussed in the Demo_System_Call. However, this system call introduces added complexity, as it involves iterative and recursive patterns.

For instance, during the initialization of the virtual machine’ s memory slots, a for loop is used to allocate and initialize each instance of a memory slot. This pattern introduces an iterative approach to memory initialization, which inherently increases overhead. In addition, the system call includes functions designed to create mappings for the hypervisor to access EL1 data, such as instances of the KVM structure. These functions are implemented as recursive algorithms to traverse the hypervisor’s page table and initialize each page table entry accordingly.

The overhead introduced by these iterative and recursive patterns is amplified because both patterns repeatedly invoke getter and setter functions within the loops and recursive calls. Consequently, as shown in table 5.1, the performance overhead for this system call is significantly higher than that of the Demo_System_Call, which primarily involves simple memory operations.

5.1.3 Cases of Significant Performance Overhead

In our evaluation, we found the overhead of the KVM_CREATE_VCPU, and KVM_SET_GSI_ROUTING are relatively higher than the Demo_System_Call and KVM_CREATE_KVM. The KVM_CREATE_VCPU system call is responsible for creating a virtual CPU (VCPU) instance, allowing the virtual machine to execute tasks. On the other hand, KVM_SET_GSI_ROUTING configures the Global System Interrupt (GSI) routing table, managing and replacing the routing table entries for interrupts.

doi:10.6342/NTU202500063

Both system calls involve shared variables structured as arrays and make extensive use of loops to modify the entries within these arrays. For instance, the `KVM_CREATE_VCPU` system call contains structures like the Virtual Performance Monitor Unit (PMU) and the Virtual Generic Interrupt Controller (VGIC), both of which rely on array-like data structures. Similarly, `KVM_SET_GSI_ROUTING` uses an Interrupt Request (IRQ) table for the virtual machine, which is also an array structure. In both cases, the system modifies or initializes these array entries by iterating through each element in the array.

Although the increase in code size for these system calls, as shown in table 5.2, is similar to or even smaller than the previously discussed system call, the frequent use of iteration patterns results in much higher overhead. This is because the getter and setter functions, which incur high overhead, are repeatedly called within these loops. Consequently, the performance overhead is significantly higher, as shown in table 5.1.

The performance overhead for `KVM_SET_GSI_ROUTING` is particularly greater than for `KVM_CREATE_VCPU` due to its use of multidimensional arrays, which are modified through nested loops. The higher number of array entries and the deeper level of iteration in `KVM_SET_GSI_ROUTING` result in a more substantial performance increase. From this, we can conclude that the increase in code size is not the primary factor causing the performance degradation. Rather, iterative and recursive patterns, which amplify overhead by repeatedly invoking high-overhead getter and setter functions, are the main contributors to the increased performance overhead.

doi:10.6342/NTU202500063

5.2 Discussion

5.2.1 Generalization

Most components within the Linux kernel can be compartmentalized using our automation framework. We will discuss the situation that our automation framework can not analyze and instrument.

5.2.1.1 Analysis Limitation

In the KVM module, certain system calls utilize a hypercall, implemented in assembly code, to invoke functions of the lowvisor. However, since our subject analysis process analyzes the intermediate representation of the Linux kernel, we cannot determine the callee for functions invoked through assembly. Consequently, if a system call involves a hypercall, we are unable to analyze the complete set of functions associated with handling that system call.

Even if we manually identify the functions invoked by the hypercall, the arguments passed to these functions via registers using inline assembly make the analysis relies on intermediate representation impossible. The object and permission analysis process section 3.4.2 relies on a complete caller-callee relationship. Without this relationship, the algorithm fails to determine whether these memory operations, such as load or store instructions, access the shared data, leaving the analysis incomplete.

doi:10.6342/NTU202500063

5.2.1.2 Instrumentation Limitation

Linux provides memory manipulation functions like `memcpy`, `memset`, and others, which are implemented in assembly code and cannot be instrumented directly by our framework. For example, the `KVM_GET_DIRTY_LOG` system call, which captures a snapshot of the dirty page, uses `memset` to clear the dirty bitmap to zero. In such cases, we can simply rewrite `memset` as a for loop to clear each entry in the dirty bitmap individually. This allows our prototype compiler to modify the operation with a setter.

Similarly, atomic primitives are low-level operations used in concurrent programming to ensure that updates to shared data are performed atomically. In modern device drivers, atomic primitives such as `atomic_inc()` and `atomic_set()` are commonly used to update shared state safely in a multi-CPU environment. These atomic primitives are implemented as inline assembly, which makes it challenging to track the data flow through them for automated modification. For example, the packet transmit system call provided by the nullnet driver includes atomic primitives when updating the net device structure's statistical data. In this scenario, we can simply replace these atomic operations with the getter and setter APIs provided by our framework. Since there are only two atomic primitives that need to be replaced, the manual effort required is minimal.

5.2.2 Concurrency Bug Discussion

In this framework, we reuse the concurrency primitives, such as the spin lock, mutex, etc., so the getter and setter in our framework would also be instrumented in the critical section and wouldn't incur any concurrency bugs.

Atomic operations, e.g., `atomic_inc()` require three steps: read, modify, and write,

doi:10.6342/NTU202500063

to be performed atomically. However, our framework does not support atomic operations because it provides only the getter and setter APIs separately and lacks an API to execute these steps as a single atomic action. As a result, we cannot ensure that a series of instructions accessing shared data (such as `atomic_inc`, which involves both load and store steps) are executed atomically. In the future, we will provide a set of APIs in charge of atomic operations, ensuring that compartments can safely perform atomic accesses to shared data.

However, other types of concurrency bugs, such as forgetting to use lock primitives or a deadlock, cannot be prevented by our framework.

5.2.3 How to ensure the code after instrumentation works as it should be?

We evaluate whether the system calls after instrumentation functions correctly in two ways. First, we identify which load and store instructions need to be converted into getter and setter functions, then verify that every targeted instruction is indeed transformed by our framework. If all relevant instructions are successfully converted, we conclude that the instrumented code can execute as intended. Second, we test the system call after instrumentation at runtime by leveraging the user space program. For instance, for the compartmentalized system calls in KVM, we spawn a VM via QEMU [3] to trigger them. We found that VM boots and functions correctly without crashing.

doi:10.6342/NTU202500063

5.2.4 Effect of Memory Ordering

A weakly-ordered memory [11] model enhances performance by allowing memory operations to be executed, observed, and finalized in a sequence that does not necessarily match the program's defined order. The Armv8-A is an example of implementing a weakly-ordered memory model. At runtime, reordering may occur if, for example, the CPU encounters a cache miss and then attempts to execute a subsequent memory operation ahead of schedule. Nevertheless, because all memory operations are replaced with function calls, these calls are not affected by runtime memory ordering during execution. Consequently, runtime memory ordering would not happen.

5.2.5 Effect of Compiler Optimization

LLVM usually performs code-transforming optimizations such as function inlining or dead argument elimination in the middle-end optimization phase. Additionally, LLVM also can perform Link Time Optimization. Link Time Optimization in LLVM process that merges multiple object files into a single, unified program representation during the linking phase. By gathering all modules at once, LLVM can perform advanced cross-module optimizations, such as dead-code elimination, and constant merging, which ultimately improves runtime performance and may reduce the overall code size.

Our framework begins instrumenting the LLVM IR generated after LLVM's middle-end optimization phase, so the IR it receives has already been optimized and will not be transformed further. Therefore, the LLVM IR after being instrumented by our framework wouldn't be inadvertently removed by LLVM.

doi:10.6342/NTU202500063

Chapter 6 Related Work

6.1 Related Work

6.1.1 Driver Automated Isolation

KSplit[8] primarily focuses on isolating device drivers from the rest of the Linux kernel, offering a coarse-grained and inflexible compartmentalization policy. It lacks support for compartmentalizing components with custom granularity, such as isolating individual functions involved in handling system calls. Additionally, KSplit computes shared data by performing data-flow analysis on both the driver and kernel sides. This approach, however, is time-consuming and struggles to scale effectively to gigantic kernel codebases, limiting KSplit's ability to analyze the entire kernel. Instead, it can only analyze a subset of the code, which requires manual selection.

Moreover, KSplit lacks a standardized method for accessing different types of shared data, complicating its ability to handle certain low-level C idioms. For example, sentinel-terminated arrays, which are terminated by a NULL value, present challenges for KSplit, as it cannot automatically infer the size of such arrays or determine if a structure is indeed an array. This makes it difficult for KSplit to handle these constructs accurately.

doi:10.6342/NTU202500063

6.1.2 Compartmentalization Policy Analysis

Compartmentalization policy analysis works such as µSCOPE[33] and TyPM[21] often produce imprecise policies, which limits their practicality in real-world systems. µSCOPE relies on dynamic analysis, which can lead to under-privileged compartments. Since dynamic analysis only captures data flows during the runtime scenarios that are actually executed, certain potential interactions may be missed. For example, a driver handling both read and write operations may only execute read paths during analysis, resulting in the compartment lacking the necessary access to shared data for untested write operations. This incomplete analysis can cause execution errors and restrict the correct functioning of compartments to only those scenarios that were observed during runtime analysis.

On the other hand, TyPM can lead to over-privileged compartments. TyPM bases its compartmentalization on the types of data accessed by the compartment, often resulting in broad permissions that include all instances of a given data type. For instance, if a compartment only needs access to a specific configuration parameter, TyPM may grant access to all instances of that parameter type, leading to unintended modifications to unrelated system-wide data. This over-granting of permissions increases the risk of data misuse, compromising system security and reducing the effectiveness of the compartmentalization.

Both under-privileged and over-privileged compartments present significant challenges for achieving precise compartmentalization in large systems like the Linux kernel. While µSCOPE and TyPM represent advances in compartmentalization policy analysis, their limitations underscore the need for more precise, automated solutions that can dy-

doi:10.6342/NTU202500063

namically adapt to the varying requirements of different system components while maintaining effective security and isolation boundaries.

6.1.3 Manual Compartmentalization Approaches

Past systems[6, 23, 24, 26, 27, 32, 36–38] have made substantial efforts to compartmentalize components within the monolithic Linux kernel. The primary goal of these strategies is to isolate subsystems, particularly device drivers, which are often developed by third-party vendors prone to introducing vulnerabilities. Nooks, for example, isolate drivers using protection domains but require extensive manual setup. VirtuOS uses virtualization to create service domains, but this method adds significant complexity, requiring developers to manually modify the code into compartments. LXFI isolates kernel modules through API integrity, but its reliance on developer annotations makes it cumbersome and limits scalability. These software compartmentalization approaches introduce significant performance overhead primarily due to frequent context switching and cross-domain communication, which increase latency and reduce efficiency.

Recent efforts like BULKHEAD, LVD, and HAKC utilize hardware features[1, 2, 9, 10] to reduce performance overhead while enforcing compartmentalization. Nevertheless, these solutions still require manual intervention for setup, policy definition, and code partitioning, making them error-prone and difficult to maintain. As the Linux kernel evolves rapidly, this lack of automation hinders the effectiveness and adaptability of these strategies. Our research addresses this limitation by automating kernel compartmentalization. By generating compartment boundaries and modifying code automatically.

doi:10.6342/NTU202500063

Chapter 7 Conclusions

In conclusion, we present a comprehensive approach for automatically compartmentalizing components within the Linux kernel at a fine-grain granularity. This framework leverages type-based analysis to identify functions involved in the call path of system calls, thus enabling precise and effective isolation of kernel components. By utilizing call graph analysis, the framework automatically generates compartmentalization policies, defining boundaries for each compartment while minimizing the need for manual intervention.

The framework enforces these compartment boundaries using the getter and setter API to manage shared data access. Additionally, the implementation includes automatic instrumentation of the compartment code, replacing memory operations such as load and store instructions with corresponding API invocations. This enables a monitor-based system that consistently supervises shared data access, enhancing the integrity and security of the system.

In our framework, we define a compartment by identifying the functions that handle a given system call. However, shared data may also be used by other system calls or functions outside the compartment, leading to frequent data transfers and increased overhead. In the future, we can partition the kernel according to whether components access shared data, thus reducing the overhead of data transfers [33].

doi:10.6342/NTU202500063

References

[1] ARM. Memory Tagging Extension. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety [Accessed: JUL 13, 2024].

[2] ARM. pointer authentication instructions. https://developer.arm.com/documentation/109576/0100/Pointer-Authentication-Code/Introduction-to-PAC [Accessed: JUL 13, 2024].

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, page 41, USA, 2005. USENIX Association.

[4] C. Dall and J. Nieh. KVM/ARM: the design and implementation of the linux ARM hypervisor. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, page 333–348, New York, NY, USA, 2014. Association for Computing Machinery.

[5] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi. On the Effectiveness of Type-based Control Flow Integrity. In Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC '18, page 28–39, New York, NY, USA, 2018. Association for Computing Machinery.

doi:10.6342/NTU202500063

[6] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, page 168–178, New York, NY, USA, 2008. Association for Computing Machinery.

[7] M. W. Hall and K. Kennedy. Efficient call graph analysis. ACM Lett. Program. Lang. Syst., 1(3):227–242, Sept. 1992.

[8] 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, Carlsbad, CA, July 2022. USENIX Association.

[9] Intel. Intel. (2023, Dec.) Intel 64 and IA-32 architectures software developer manuals. [Online]. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html [Accessed: JUL 13, 2024].

[10] Intel. Intel® 64 and IA-32 Arch. SDM. Vol. 3C, Sec. 26.5.6: FEATURES SPECIFIC TO VMX NON-ROOT OPERATION: VM Functions. Version: December 2022. [Accessed: JUL 13, 2024].

[11] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. SIGPLAN Not., 52(1):175–189, Jan. 2017.

[12] Kangjie Lu. MLTA LLVM 14.0.6 Source code. https://github.com/seclab-ucr/Unias/tree/main/src/mlta.

[13] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program

doi:10.6342/NTU202500063

analysis transformation. In International Symposium on Code Generation and Optimization, 2004. CGO 2004., pages 75–86, 2004.

[14] H. Lefeuvre, N. Dautenhahn, D. Chisnall, and P. Olivier. SoK: Software Compartmentalization, 2024.

[15] Linux Developer Group. Minimal requirements to compile the Kernel. https://elixir.bootlin.com/linux/v5.15/source/Documentation/process/changes.rst.

[16] LLVM Developer Group. LLVM Call Graph. https://llvm.org/doxygen/CallGraph_8h_source.html [Accessed: DEC 19, 2024].

[17] llvm.org. Argument Promotion. https://llvm.org/doxygen/ArgumentPromotion_8cpp.html [Accessed: DEC 21, 2024].

[18] llvm.org. Hot Cold Splitting. https://llvm.org/doxygen/HotColdSplitting_8cpp.html#details [Accessed: DEC 21, 2024].

[19] llvm.org. Inliner. https://llvm.org/doxygen/Inliner_8cpp.html [Accessed: DEC 21, 2024].

[20] llvm.org. LLVM Intermediate Representation . https://releases.llvm.org/14.0.0/docs/LangRef.html. [Accessed: JUL 13, 2024].

[21] K. Lu. Practical Program Modularization with Type-Based Dependence Analysis. In 2023 IEEE Symposium on Security and Privacy (SP), pages 1256–1270, 2023.

[22] K. Lu and H. Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 1867–1881, 2019.

doi:10.6342/NTU202500063

[23] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with API integrity and multi-principal modules. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, page 115–128, New York, NY, USA, 2011. Association for Computing Machinery.

[24] 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.

[25] M. S. Miller. Robust composition: towards a unified approach to access control and concurrency control. PhD thesis, JohnsHopkins University, USA, 2006. AAI3245526.

[26] V. Narayanan, A. Balasubramanian, C. Jacobsen, S. Spall, S. Bauer, M. Quigley, A. Hussain, A. Younis, J. Shen, M. Bhattacharyya, and A. Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 269–284, Renton, WA, July 2019. USENIX Association.

[27] V. Narayanan, Y. Huang, G. Tan, T. Jaeger, and A. Burtsev. Lightweight kernel isolation with virtualization and VM functions. In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '20, page 157–171, New York, NY, USA, 2020. Association for Computing Machinery.

[28] National Institute of Standards and Technology. National Vulnerability Database. CVE-2018-18021. https://nvd.nist.gov/vuln/detail/cve-2018-18021 [Accessed: DEC 26, 2024].

[29] National Institute of Standards and Technology. National Vulnerability Database.

doi:10.6342/NTU202500063

CVE-2019-6974. https://nvd.nist.gov/vuln/detail/cve-2019-6974 [Accessed: DEC 26, 2024].

[30] National Institute of Standards and Technology. National Vulnerability Database. CVE-2021-4095. https://nvd.nist.gov/vuln/detail/CVE-2021-4095 [Accessed: DEC 26, 2024].

[31] Neil Brown. Object-oriented design patterns in the kernel. https://lwn.net/Articles/444910/ [Accessed: DEC 22, 2024].

[32] R. Nikolaev and G. Back. VirtuOS: an operating system with kernel virtualization. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, page 116–132, New York, NY, USA, 2013. Association for Computing Machinery.

[33] N. Roessler, L. Atayde, I. Palmer, D. McKee, J. Pandey, V. P. Kemerlis, M. Payer, A. Bates, J. M. Smith, A. DeHon, and N. Dautenhahn. µSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses, RAID '21, page 296–311, New York, NY, USA, 2021. Association for Computing Machinery.

[34] J. Saltzer and M. Schroeder. The protection of information in computer systems. Proceedings of the IEEE, 63(9):1278–1308, 1975.

[35] SecurityScorecard. Linux kernel vulnerabilities. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33 [Accessed: JUL 13, 2024].

doi:10.6342/NTU202500063

[36] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers. Nooks: an architecture for reliable device drivers. In Proceedings of the 10th Workshop on ACM SIGOPS European Workshop, EW 10, page 102–107, New York, NY, USA, 2002. Association for Computing Machinery.

[37] W. B. Q. Z. K. L. Yinggang Guo, Zicheng Wang. BULKHEAD: Secure, Scalable, and Efficient Kernel Compartmentalization with PKS. In NDSS, 2025.

[38] You-Ting Lee. Time interval base crash recovery system. https://github.com/ntu-ssl/linux-compartment [Accessed: DEC 19, 2024].

doi:10.6342/NTU202500063

Appendix A — Configuration

```
CONFIG_PREEMPT_NONE=y
# CONFIG_PREEMPT_VOLUNTARY is not set
# CONFIG_PREEMPT is not set
```

Listing A.1: Linux Kernel Configuration

doi:10.6342/NTU202500063