# BULKHEAD: Secure, Scalable, and Efficient Kernel Compartmentalization with PKS

Yinggang Guo*†, Zicheng Wang*, Weiheng Bai†, Qingkai Zeng* and Kangjie Lu†
*State Key Laboratory for Novel Software Technology, Nanjing University, †University of Minnesota
{gyg, wzc}@smail.nju.edu.cn, bai00093@umn.edu, zqk@nju.edu.cn, kjlu@umn.edu

***Abstract*—The endless stream of vulnerabilities urgently calls for principled mitigation to confine the effect of exploitation. However, the monolithic architecture of commodity OS kernels, like the Linux kernel, allows an attacker to compromise the entire system by exploiting a vulnerability in any kernel component. Kernel compartmentalization is a promising approach that follows the least-privilege principle. However, existing mechanisms struggle with the trade-off on security, scalability, and performance, given the challenges stemming from mutual untrustworthiness among numerous and complex components.**

**In this paper, we present BULKHEAD, a secure, scalable, and efficient kernel compartmentalization technique that offers bi-directional isolation for unlimited compartments. It leverages Intel's new hardware feature PKS to isolate data and code into mutually untrusted compartments and benefits from its fast compartment switching. With untrust in mind, BULKHEAD introduces a lightweight in-kernel monitor that enforces multiple important security invariants, including data integrity, execute-only memory, and compartment interface integrity. In addition, it provides a locality-aware two-level scheme that scales to unlimited compartments. We implement a prototype system on Linux v6.1 to compartmentalize loadable kernel modules (LKMs). Extensive evaluation confirms the effectiveness of our approach. As the system-wide impacts, BULKHEAD incurs an average performance overhead of 2.44% for real-world applications with 160 compartmentalized LKMs. While focusing on a specific compartment, ApacheBench tests on `ipv6` show an overhead of less than 2%. Moreover, the performance is almost unaffected by the number of compartments, which makes it highly scalable.**

## I. Introduction

The Operating System (OS) kernel serves as the cornerstone of system software, with expanding functionality for a diverse range of user programs and hardware devices. Despite its significance, the huge codebase which is written in unsafe languages faces a continual influx of vulnerabilities. Illustratively, the Linux kernel, a widely utilized OS kernel, has witnessed a stark increase in Common Vulnerabilities and Exposures (CVE) reporting. In 2023, the incidence of CVEs surged by over 179% compared to 2013, cumulatively amounting to 2,814 CVEs over the decade [1].

Given that complete elimination of vulnerabilities remains an elusive goal, confining the impact of each vulnerability, including those yet undetected, is essential for system security. However, the upsetting fact is that most mainstream OS kernels, like the Linux kernel, are monolithic for performance and compatibility reasons, thus lack fault isolation. All kernel components share the supervisor privileges for data access and code execution, most of which are irrelevant to their intended duties. As a result, exploiting a single vulnerability in any component can potentially compromise the entire system.

A viable solution to this pervasive issue is kernel compartmentalization [2, 3], which offers principled and systematic protection against vulnerabilities through the principle of least privilege [4]. In this approach, kernel modules are isolated into separate compartments, with each compartment having access only to the data and code necessary for its functionality. Consequently, the impact of an exploited module is constrained, preventing systemic damage.

In the face of various vulnerabilities and powerful attackers, the kernel compartmentalization mechanism must achieve the trifecta of security, scalability, and performance. Particularly, in order to strictly confine any exploitation within compartment boundaries, the mechanism should guarantee not only data access but also control flow transfer. Besides, compartment interfaces must be well-protected against confused deputy attacks [5], which confuse a compartment to perform sensitive operations on behalf of the untrusted caller with malicious inputs. Scalability is another significant objective. Given the huge codebase of the OS kernel, fine-grained compartmentalization calls for a large number of isolated compartments. Finally, developers constantly trade off the benefits of protection against performance overhead. Only efficient mechanisms will bolster wide adoption in practice.

Unfortunately, existing works fail to fulfill these desirable objectives. Microkernels [6–9] minimize the attack surface by moving most kernel components into isolated user processes. Although providing strong protection, heavy inter-process communication (IPC) leads to low performance [8]. The complete redesign of the OS also hinders its general application. Software fault isolation (SFI)-based approaches [10–13] establish an isolated domain by instrumenting each memory access instruction with security checks during compilation time, which results in significant overhead. Virtualization-based approaches [14–19] employ a hypervisor to protect execution domains and can form different memory views with the extended page tables (EPTs). However, the additional layer makes the Trusted Computing Base (TCB) more complex. Running systems in virtual machines (VMs) results in extra overhead and nested virtualization restrictions. There are also some efforts [20–23] that utilize various hardware features to protect the kernel but cannot scale to multiple compartments because of the hardware limitations.

A more critical problem is that most previous works neglect the support for bi-directional isolation [25, 26] within the kernel space. They have to trust the core kernel for management

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230328
www.ndss-symposium.org

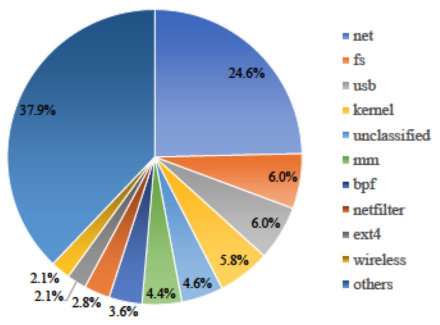


**Fig. 1:** Distribution of Linux kernel vulnerabilities reported by Syzkaller [24]. Only the top 10 subsystems with the most vulnerabilities are listed for demonstration.

and only enforce one-way access control between the core kernel and other untrusted components. The untrusted parts are restricted from accessing resources of the core kernel, while in the opposite direction, the trusted core kernel can arbitrarily access other components, leading to the monopoly over isolation [27]. Unfortunately, our analysis revealed that about 5.8% of Linux vulnerabilities are found in the core kernel (Figure 1), which is comparable to notorious subsystems like `ipv6` and `netfilter`.

In addition to direct exploitation in the core kernel, one-directional isolation leaves confused deputy attacks as the open problem [5, 28, 29]. Since the trusted part is unhindered, it can be confused by malicious inputs to break compartmentalization on behalf of the untrusted compartment. Therefore, both the source and the target of cross-compartment interactions should be validated bi-directionally.

This paper introduces BULKHEAD, a comprehensive framework for kernel compartmentalization that enforces bi-directional isolation with minimal performance overhead. BULKHEAD leverages Intel's recent hardware feature, Protection Keys for Supervisor-mode (PKS) [30], in a novel way to isolate data and code into mutually untrusted compartments. We exclude the core kernel from the TCB and tag it as well as other compartments with distinct protection keys (`pkeys`). A newly introduced lightweight in-kernel monitor is in charge of enforcing security invariants, including data integrity, execute-only memory (XOM), and compartment interface integrity that defends against confused deputy attacks according to developer-specified policies flexibly. With a specially designed switch gate table (SGT) and zero-copy ownership transfer, the compartment switching is performed securely and efficiently.

Employing PKS in kernel compartmentalization presents unique challenges due to the privileged environment. First, the memory access validation of PKS does not work on execution permissions, leaving the control flow not validated and unprotected [23]. Second, since all kernel code shares the privilege of modifying page tables (PTs) and the `PKRS` register, PKS-based isolation suffers from pitfalls [31–33] to be bypassed through PT tampering or instruction abusing. Third, the 4-bit `pkey` supports only up to 16 compartments, which is inadequate. The Linux kernel v6.1 has over 30 million lines of code and contains 6296 LKMs with the x86_64 generic configuration. The limited number of compartments makes each compartment too coarse to confine the impact of vulnerability exploitation.

To address these challenges, on the security hand, BULKHEAD introduces a lightweight in-kernel monitor through privilege separation for invariant enforcement. It restricts all kernel code pages as XOM, enables diversification schemes such as Kernel Address Space Layout Randomization (KASLR) [34], isolates private stacks, and enforces cross-compartment control flow integrity (CFI). These together substantially mitigate control flow hijacking. The page table and switch gates are exclusively controlled by the monitor to prevent PKS-based isolation from being bypassed. On the scalability hand, BULKHEAD proposes a locality-aware two-level scheme to support unlimited compartments. Different from the two-level compartmentalization in HAKC [2], which combines the ARM-specific features: Memory Tagging Extension (MTE) and Pointer Authentication (PA), we group the modules and split the address space based on the locality of module interactions. The first level is PKS-based intra-address space isolation, while the second level is locality-aware inter-address space isolation with Address Space Identifier (ASID).

We implement a prototype system in Linux v6.1 and perform automated LKM isolation as a use case. In the case study, we specify security policies based on the boundary analysis, shared data analysis, and security check analysis, then, compartmentalize all 160 LKMs with `localmodconfig` supported by our machine to show the scalability. Security analysis with penetration tests confirms its comprehensive protection capabilities. Extensive performance evaluation with both micro-benchmarks and macro-benchmarks presents its efficiency. Specifically, BULKHEAD incurs an average overhead of only 2.44% on the whole-system benchmarks Phoronix [35] and ApacheBench [36] tests on the compartmentalized `ipv6` module show an overhead of less than 2%.

In summary, this paper makes the following contributions:

- A systematic analysis of kernel compartmentalization objectives and a secure, scalable, and efficient mechanism based on PKS that fulfills these objectives.
- A novel in-kernel monitor that supports bi-directional isolation and enforces multiple security-critical invariants, including data integrity, execute-only memory, and compartment interface integrity.
- A locality-aware two-level compartmentalization scheme that supports unlimited compartments.
- An implementation of the prototype system[1] for automated LKM isolation and extensive evaluation that demonstrates its security, scalability, and efficiency.

## II. Objectives For Kernel Compartmentalization

Kernel compartmentalization mechanisms should be characterized by a specific set of objectives, including security, scalability, performance, and compatibility issues. With a systematic survey (Table I), we analyze the limitations of related work and demonstrate how BULKHEAD meets these goals.

### A. *Kernel Vulnerability Analysis*

As inspiration for BULKHEAD, we analyzed all fixed vulnerabilities reported by Syzkaller [24] - one of the most popular kernel fuzzing tools developed by Google, with 5201 in total at the time of writing. According to the location of the

[1]Available at https://github.com/gyg128/BULKHEAD

| | Mechanisms | Security | | | | Scalability | Performance | | Compatibility |
|---|---|---|---|---|---|---|---|---|---|
| | | bi-directional isolation | data protection | control flow protection | interface protection | domain number | domain switch | data transfer | |
| seL4 [7] | Microkernel | No | Yes | Yes | No | Unlimited | Low | Low | Heavy redesign |
| UnderBridge [8] | Microkernel+PKU | No | Yes | Yes | No | 16 | High | High | Heavy redesign |
| LXFI [13] | SFI | No | Yes | Yes | Yes | Unlimited | Low | Low | Annotations |
| LVD [18] | Virtualization | No | Yes | Yes | No | 512 | High | Low | Nested Virtualization |
| KSplit [19] | Virtualization | No | Yes | Yes | No | 512 | High | Low | Nested Virtualization |
| xMP [37] | Virtualization | No | Yes | No | No | 512 | High | Low | Nested Virtualization |
| Nested Kernel [20] | WP bit | No | Yes | Yes | No | 2 | High | High | x86-64 |
| SKEE [38] | PT switching | No | Yes | Yes | No | 2 | Medium | Low | ARM |
| IskiOS [23] | PKU | No | No | Yes | No | 8 | High | High | SMAP/SMEP |
| HAKC [2] | MTE+PA | No | Yes | Yes | No | Unlimited | Medium | Medium | ARM |
| CHERI [39] | New architecture | No | Yes | Yes | Yes | Unlimited | Medium | Medium | New architecture |
| SecureCells [40] | New architecture | No | Yes | Yes | Yes | Unlimited | High | High | New architecture |
| DOPE [41] | PKS | No | Yes | No | No | 16 | High | High | Intel |
| **BULKHEAD** | **PKS** | **Yes** | **Yes** | **Yes** | **Yes** | **unlimited** | **High** | **High** | **Intel** |

**TABLE I:** Systematic analysis of kernel compartmentalization objectives.

patches, we investigate the distribution of vulnerabilities, as shown in Figure 1. For ease of demonstration, only the top 10 subsystems with the highest number of vulnerabilities are listed. As we can see, there are about 5.8% of Linux vulnerabilities found in the core kernel, which ranks fourth. Specifically, the number of issues from the core kernel is comparable to some notorious subsystems such as `ipv6` and `netfilter`. Given that any single vulnerability has the potential to destroy the entire system, simply trusting the core kernel is highly questionable.

## B. Security Objectives

To realize the security goal, a mechanism must enforce comprehensive protection of the isolated compartments.

*1) Bi-directional Isolation:* Despite the significant risks posed by vulnerabilities within the core kernel, most existing efforts in kernel compartmentalization [2, 18, 19] only perform one-directional isolation at the boundaries between the core kernel and other compartments. They grant arbitrary access to the core kernel and only restrict other compartments due to the challenge of intra-kernel privilege separation [20]. The reverse protection is hard because these approaches have to rely on the core kernel to manage and enforce access control policies. Besides direct vulnerability exploitation in the core kernel, the trusted part can also be exploited as a confused deputy, thereby necessitating a more robust strategy under the mutually untrusted threat model. Effective compartmentalization must ensure bi-directional isolation among all compartments and rigorously validate both the source and the target for each cross-compartment interaction. Several works have explored bi-directional protection at kernel-userspace boundaries [25], hypervisor-VM boundaries [42, 43], and application-enclave boundaries [26, 44]. While we enforce PKS-based bi-directional isolation within the kernel space, which breaks the monopoly of the core kernel without introducing any additional layer.

*2) Data Protection:* Sensitive data objects in the kernel, especially privilege-related objects like `cred`, are critical attack targets [45]. Kernel compartmentalization must isolate data into specific compartments and block any illegal access from irrelevant compartments. Existing works, such as IskiOS [23] and KCoFI [46], leave sensitive data protection out of consideration and only focus on control flow protection. However, data-oriented attacks could also indirectly change the control flow and even escalate the privilege [47]. BULKHEAD assigns different `pkeys` to objects of different compartments, which guarantees the data integrity of each compartment with hardware-enforced access control.

*3) Control Flow Protection:* Control flow hijacking is another major threat to kernel security. While demanding validity checks for control flow transfer, many kernel isolation works, such as xMP [37] and DOPE [41], rely on additional control flow protection mechanisms as assumptions, such as CFI [46]. Unfortunately, a simple combination of data protection and control flow protection will lead to significant performance overhead. As a countermeasure, BULKHEAD set all kernel code regions as XOM to mitigate code reuse attacks within compartments. While cross-compartment calls are guarded by carefully designed switch gates. Although XOM itself is not a strong defense, existing diversification schemes such as KASLR and data compartmentalization make memory disclosure [48] harder to break our protection.

*4) Compartment Interface Protection:* Compartment interface security is a long-neglected problem. Recent work [5] has emphasized the seriousness of interface-related vulnerabilities, which could be used to launch confused deputy attacks. With malicious inputs, attackers aim to indirectly exploit the privilege of the confused callee. Most existing works cannot defend against confused deputy attacks with one-directional isolation. Weak interfaces seriously reduce or even fully negate the security guarantee of compartmentalization. LXFI [13] proposed API integrity to protect kernel API, but it relies on the programmer's annotations, which require a lot of manual effort and are error-prone. CHERI [39] and SecureCells [40] also discussed interface checks. However, as new hardware architectures, it is still far from actual adoption in practice. BULKHEAD introduces compartment interface integrity with the help of developer-specified policies. With the novel switch gate table (SGT) protected by the in-kernel monitor, we record and validate both the source and the target compartment metadata. This bi-directional validation grants BULKHEAD the ability to protect compartment interfaces according to the policies specified by developers. For example, we define entry/exit points for compartment switches through boundary analysis and add validations for data transfer based on shared data and security check analysis.

### C. Scalability Objectives

*1) Scalable for Unlimited Compartments:* The security benefits of compartmentalization depend on the granularity of the compartments, which is limited by the number of supported domains. Nested Kernel [20] only constructs two domains based on the Write Protection (WP) bit. SKEE [38] creates a separate secure execution environment within the kernel through PT switching. Memory Protection Key (MPK)-based approaches suffer from the hardware limitation of 16 `pkeys`. Reserving `pkey` for other purposes will further minimize the domains available to the kernel [23]. The virtualization-based solutions [18, 19, 37] support up to 512 EPTs representing 512 different memory views. HAKC [2] proposed a two-level compartmentalization scheme combining the ARM-specific features MTE and PA. Although it theoretically supports unlimited compartments, HAKC evaluated only two compartments and got a linear growth of 14%-19% per compartment in overhead, attributed to the expensive cryptographic operations of PA. As a comparison, BULKHEAD proposes a locality-aware two-level compartmentalization scheme combining PKS and multiple address spaces. In addition to PKS-based intra-address space isolation, we also leverage locality-aware inter-address space isolation with ASID. This second level is more compatible and efficient than HAKC. Since each address space has its own 16 compartments, we can achieve support for unlimited compartments by address space switching.

### D. Performance Objectives

*1) Fast Compartment Switches:* Since the kernel serves a central role in the system, it is extremely sensitive to performance overhead. Compartment switches are essential and frequent for interactions, which dominate the performance. Microkernels [7, 8] suffer from time-consuming IPC. The inserted security checks for every memory access slow down the performance of SFI-based approaches [13]. Although the `vmfunc` instruction accelerates EPT switching [18, 19, 37], due to nested paging and I/O virtualization, running systems in VMs additionally incurs extra overhead [49, 50]. BULKHEAD benefits from efficient permission validations and switches with PKS. The hardware-based compartment switch only involves a specific register update.

*2) Zero-copy Data Transfer:* Data copying across compartments can overwhelm the performance-critical OS kernel. Unfortunately, PT/EPT switching-based approaches cannot support secure zero-copy communication across different memory mapping and require complex synchronization of shared states [18, 19, 38]. In contrast, tag-based approaches inherently support zero-copy data transfer. BULKHEAD tags data with a specific `pkey`, which enforces single ownership of objects. As a response to compartment switching, access from the target compartment will trigger a page fault. Then, the monitor validates the shared data and just updates its `pkey` in the dedicated handler, without copying data across compartments.

### E. Compatibility Objectives

*1) Compatibility Issues:* Besides security, scalability, and performance, ideal kernel compartmentalization mechanisms also should be compatible with real-world machines and complex production environments. Different from related work, BULKHEAD utilizes the emerging hardware feature of the widely-used Intel architecture. It requires neither heavy code redesign like the microkernel [7, 8] nor extensive annotations like LXFI [13]. Moreover, the compartmentalized kernel based on PKS could be directly used in the cloud environment without nested virtualization restrictions [18, 19, 37].

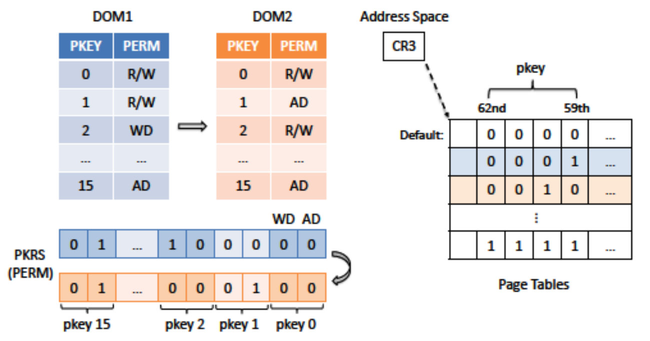


**Fig. 2:** Working principle of MPK, where WD and AD stand for write-disable and access-disable permissions, respectively.

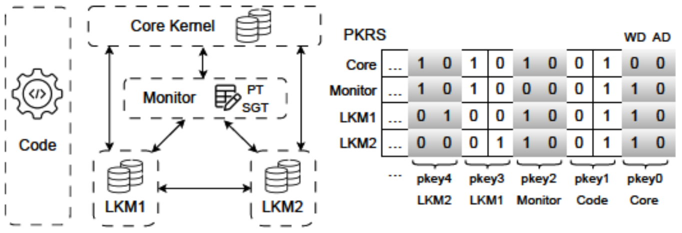
## III. Background

### A. Memory Protection Keys

MPK [30] is an Intel hardware feature that enforces per-thread access control but without requiring PT modification for domain switching (Figure 2). In addition to the permission bits (R/W bit and NX bit) in the page table entries (PTE), it tags pages with a 4-bit `pkey` (bits [59:62]), which partitions the address space into at most 16 memory domains. The permissions of pages with a specific `pkey` are stored in a dedicated per-thread register as a 2-bit notation (WD, AD), where WD means write-disable and AD means access-disable. The hardware-based access checks incur nearly zero runtime overhead [8], while permission change is simply done by updating the register. Since no PT walk or TLB flush is required, MPK is known for the fast domain switching [51, 52].

According to the User/Supervisor (U/S) bit in the PTE, MPK has two variants, namely Protection Keys for User-mode (PKU) and Protection Keys for Supervisor-mode (PKS), using protection key rights register for user pages (`PKRU`) and protection key rights register for supervisor pages (`PKRS`, i.e., `MSR 0x6E1`) respectively. PKU has been explored in depth for intra-process isolation and has shown outstanding results [51, 52]. Before PKS was available, several works attempted to utilize PKU for kernel isolation by setting the User bit of kernel pages, such as UnderBridge [8] and IskiOS [23]. This unconventional application requires additional consideration for kernel-user isolation and is incompatible with other security features based on the U/S bit, such as Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Prevention (SMEP). Multiple untrusted compartments and the privileged environment also pose challenges to PKU-based kernel isolation, especially the switch gate design.

The long-awaited feature PKS has recently become available on 12th Core and 4th Xeon CPUs [30], yet it remains underexplored in existing works. There are some PKS-based studies concurrent to BULKHEAD. Among them, KDPM [53] and DOPE [41] only target kernel data protection, neglecting other security objectives. MOAT [54] and eBPF-sandbox [55]



| PKRS | ... | pkey4 | | pkey3 | | pkey2 | | pkey1 | | pkey0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | ... | LKM2 | | LKM1 | | Monitor | | Code | | Core | |
| | | WD | AD | WD | AD | WD | AD | WD | AD | WD | AD |
| Core | ... | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| Monitor | ... | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| LKM1 | ... | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| LKM2 | ... | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

**Fig. 3:** The overview of BULKHEAD.

are specific to eBPF program isolation and cannot generalized to other kernel components. To the best of our knowledge, BULKHEAD is the first work that achieves comprehensive kernel compartmentalization with PKS, addressing both security and scalability challenges systematically as detailed in §V.

## IV. Threat Model

We assume that vulnerabilities are present throughout the kernel, encompassing both the core kernel and LKMs. A non-root adversary can exploit these to launch various attacks, such as data-oriented attacks, control flow hijacking, and confused deputy attacks. The ultimate goal of this potential attacker is to spread the impact of vulnerabilities and finally take control of the entire system. Consequently, kernel components are mutually untrusted and the core kernel is also excluded from the TCB. We trust the lightweight in-kernel monitor for security invariant enforcement, and its correctness can be formally verified. We trust the system developer who specifies prior compartmentalization policies. However, users who load compartments and register gates at runtime could be malicious and their behaviors are validated by the monitor against the predefined policies. We also assume the secure boot mechanism [56, 57] and trust the underlying hardware.

We focus on software attacks and do not consider physical attacks such as cold boot attacks [58] and RawHammer attacks [59]. Denial-of-service (DoS) and side-channel attacks like Spectre [60] and Meltdown [61] are also excluded from our consideration. The defense mechanisms against these attacks are orthogonal to kernel compartmentalization and can be applied to protect the system further.

## V. BULKHEAD Design

### A. Overview

BULKHEAD provides strong protection by adhering to the principle of least privilege. As shown in Figure 3, the kernel memory is partitioned and tagged with different `pkeys`, and the per-thread `PKRS` indicates the access permissions for each compartment. Cross-compartment access is controlled bi-directionally, thus all compartments, including the core kernel and the monitor are restricted to accessing only what is necessary for their operations (§V-B). The data of each compartment can only be modified by itself. All kernel code pages are tagged as XOM. It is hard to reuse gadgets without a readable code layout. While the lightweight in-kernel monitor (§V-C) serves as a special compartment for security invariant enforcement (§V-D). Specifically, the page table (PT) and a novel switch gate table (SGT) are write-protected by the monitor, and compartment switches are performed securely and efficiently based on the SGT. Although we take two LKMs as an example in Figure 3, BULKHEAD incorporates a locality-aware two-level compartmentalization scheme to support unlimited compartments (§V-E).

### B. PKS-based Bi-directional Isolation

Bi-directional isolation handles mutual distrust through cross-compartment access control. Each compartment is restricted to accessing only necessary resources, even for the core kernel and the monitor. As Figure 3 shows, the memory of each compartment is tagged with a distinct `pkey`. All kernel code pages are attached with `pkey1` to realize XOM. The default `pkey0` represents the core kernel's data, while `pkey2` is assigned to the monitor. The PT and SGT are exclusively owned by the monitor and read-only for other compartments. BULKHEAD configures the (WD, AD) notations in the per-thread `PKRS` for bi-directional access control. The `PKRS` value uniquely identifies the current compartment by only one (0, 0) notation which means full access to memory with the respective `pkey`. Notations of other `pkeys` all have the WD or AD bits set to enforce the compartmentalization policy. As a result, compartments, including the core kernel, are mutually untrusted, and no one can arbitrarily access each other's memory. Any attempt to break the access permissions will result in a protection key violation fault. Compartment switches can only be performed by switch gates registered in the SGT that alter the `PKRS` to disable access to the source compartment and grant access to the target compartment. Both the source and the target are validated according to the metadata recorded in the write-protected SGT, thus mitigating confused deputy attacks.

### C. In-kernel Monitor

The lightweight monitor is a special compartment responsible for guaranteeing bi-directional isolation. It manages critical metadata such as the PT for permission restriction and the SGT for secure switching. Unlike virtualization-based monitors[18, 19], BULKHEAD does not rely on an additional layer and thus breaks the "turtles all the way down" paradigm. The monitor is constructed through privilege separation within the same level of the kernel. We protect the memory resources of the monitor by PKS-based isolation while securing the instruction and register resources by depriving other compartments' privileges.

*1) Memory Isolation for In-Kernel Monitor:* With the help of PKS, the monitor memory is isolated from the rest of the kernel. To be specific, the data part especially the PT and the SGT is tagged with `pkey2`, thus write-protected against other compartments by the WD bit in `PKRS`. Only the monitor has the privilege to update these metadata related to access control. On the other hand, the code part is tagged with `pkey1` as XOM so that attackers cannot reuse the privileged code maliciously. Note that although the monitor is trusted, it also cannot access other compartments directly due to bi-directional isolation.

*2) Instruction Deprivation:* Besides memory resources, instruction and register resources can also be abused to break the compartmentalization [62], such as malicious `PKRS` updates. Moreover, since x86 does not require instruction alignment, the unintentional occurrences of privileged instructions, such as part of a longer instruction or spanning two consecutive instructions, are also hazardous. BULKHEAD should restrict

| Instruction Prefixes | Opcode | ModR/M | SIB | Displacement | Immediate |
|---|---|---|---|---|---|
| Prefixes of 1 byte each (optional) | 1-, 2-, or 3-byte opcode | 1 byte (if required) | 1 byte (if required) | Address displacement of 1, 2, or 4 bytes or none | Immediate data of 1, 2, or 4 bytes or none |

**Fig. 4:** Intel 64 and IA-32 architectures instruction format.

```
750f          jne 0xf(%rip)
30c0          xor %al,%al
```

```
750f          jne 0xf(%rip)
90            nop
30c0          xor %al,%al
```

(a) nop insertion

```
488b440f30    mov 0x30(%rdi,%rcx,1),%rax
```

```
52            push %rdx
4889ca        mov %rcx,%rdx
488b441730    mov 0x30(%rdi,%rdx,1),%rax
5a            pop %rdx
```

(b) Register reassignment

```
41bd0f300000    mov $0x300f,%r13d
```

```
41bd00300000    mov $0x3000,%r13d
4183c50f        add $0xf,%r13d
```

(c) Data adjustment

**Fig. 5:** Some examples of eliminating unintended `wrmsr` (`0x0f30`).

other compartments' behaviors that run with the monitor in the same privilege level (Ring-0) by instruction deprivation.

Benefiting new advances in binary rewriting [51, 63], we eliminate privileged instructions in compartments other than the monitor. At a high level, unintended occurrences are replaced with functionally equivalent instructions, while intended occurrences are replaced with switch gates to the monitor. As a result, only the lightweight monitor has the capability to execute the security-critical instructions after validation. Even if attackers attempt to reuse these occurrences in the monitor, they cannot locate the useful gadgets because of the read-disable protection of XOM. Specifically, we focus on three categories of instructions in our prototype. First, the instruction that changes the value of `PKRS`, i.e., `wrmsr`. Second, instructions that write control registers, i.e., `mov-to-CRn`. For example, attackers can forge the page table with a malicious value of `CR3` or disable PKS by clearing the PKS bit in `CR4`. Third, instructions that store system registers like `IDTR`, `GDTR`, `LDTR`, etc. It is notable that malicious interrupts are defended by the atomic switch gate described in §V-D3 instead of eliminating the 1-byte interrupt enabling/disabling instruction.

For unintended instructions, we apply different binary rewriting strategies according to the fields with which the unintended subsequence overlaps. The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 4 [30]. An instruction consists of: (1) an optional instruction prefix, (2) a primary opcode field (up to three bytes), (3) a ModR/M field that determines the addressing form and includes a register operand, (4) a SIB (Scale-Index-Base) field that specifies registers for indirect memory addressing, (5) a displacement and/or (6) an immediate data field that specify constant offsets. If an unintended sequence spans two or more instructions, it could be broken by inserting a 1-byte `nop` (`0x90`) or reordering instructions. Otherwise, unintended instructions may appear entirely within a longer instruction. For these cases, we eliminate them by register reassignment, data adjustment, or replacement with other functionally equivalent instructions. These strategies are applied iteratively until there are no unintended instructions. Figure 5 shows some concrete examples of eliminating unintended occurrences of `wrmsr` (`0x0f30`).

For intended instructions, we replace each occurrence with a switch gate to the monitor and delegate the monitor to execute the instruction after security validation. The validation of the switch gate relies on developer-specified policies and is flexible to perform various checks. In particular, we record rich metadata on the source and the target as described in §V-D3. Thus, except for the lightweight monitor protected by XOM, other domains are deprived of the capability to bypass isolation through privileged instructions. Although BULKHEAD could benefit from hardware extensions [62] for instruction deprivation, binary rewriting offers compatibility with existing hardware.

## D. Invariant Enforcement

With intra-kernel privilege separation, the monitor enforces a series of invariants to fulfill the security objectives: (1) data integrity that defends data-oriented attacks, (2) execute-only memory that mitigates control flow hijacking, and (3) compartment interface integrity targets confused deputy attacks.

*1) Data Integrity: Data of each compartment can only be modified by itself.* ***(I1)***

BULKHEAD determines the ownership of data according to the `pkey`. By assigning different `pkeys` to different compartments in PTEs, the monitor guarantees no compartments share the same `pkey`. Since the PTEs contain `pkeys` as well as other permission control bits, the page table is a natural prime target of attackers. To prevent arbitrary tampering, the page table is read-only for other compartments. All PT updates are exclusively delegated to the monitor, and only those adhering to the compartmentalization policy can be performed. With instruction deprivation, attackers also cannot forge page tables by malicious `CR3` value. As a result, BULKHEAD enforces PT integrity to support data integrity: *the page table can only be modified by the monitor.* ***(I1.1)***

BULKHEAD comprehensively protects the compartment's memory, including heap, stack, physmap, and MMIO regions. For the objects allocated dynamically at runtime, we create a private heap for each compartment. An intuitive way is to modify the `pkey` in the PTE after allocation. However, this dynamic updating through page walk is expensive. Instead, during the initialization of a specific compartment, we reserve a tagged memory pool for it. Then, the allocators are modified to use the reserved cache without the overhead of dynamic tagging. With the private heap, cross-compartment heap corruption becomes impossible. Similarly, we also provide private stacks for compartments and switch the stack during compartment switching. As a result, attackers cannot disturb other compartments' stacks to hijack the control flow. Besides, the direct map of physical memory (physmap) and the memory-mapped I/O (MMIO) regions are also tagged with `pkey`. Access to these areas goes through PKS hardware validation as well.

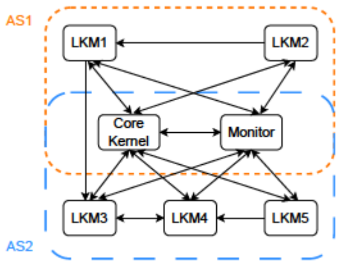
*2) eXecute-Only Memory (XOM): All kernel code cannot be read or written by anyone.* ***(I2)***

XOM is a lightweight but effective control flow protection [64]. Although PKS does not perform hardware-enforced checks on execution permission, we set the AD bit in `PKRS`

```
get_metadata(gate_id);
verify(source_addr);
if (target_pgdir != source_pgdir)
    load_new_mm_cr3(target_pgdir, target_asid);
if (target_pkrs != current_pkrs)
loop:
    write_pkrs(target_pkrs);
if (current_pkrs != target_pkrs)
    goto loop;
switch_stack(target_stack);
jump(target_addr);
```

| **gate id** | |
|---|---|
| *source* | *target* |
| address | |
| pgdir | |
| asid | |
| pkrs | |
| stack | |

**Fig. 6:** The switch gate pseudocode (left) and the metadata format of a SGT entry (right).



| **Allowed LKM Transitions** | | |
|---|---|---|
| gate id | source | target |
| ... | ... | ... |
| 22 | LKM1 | LKM3 |
| 24 | LKM2 | LKM1 |
| 26 | LKM3 | LKM4 |
| 27 | LKM4 | LKM3 |
| 28 | LKM5 | LKM4 |

**Fig. 7:** An example of compartment transition policies. The core kernel and the monitor are shared by all address spaces. AS1 contains LKM1 and LKM2, while AS2 contains LKM3-5. Edges between compartments are allowable transitions. Transitions with the core kernel and the monitor are omitted from the table.

for all kernel code regions to disable any write or read access, which is required by code injection or code reusing.

With the $W \bigoplus X$ policy in the kernel, all executable regions are not writable, which prevents code injection attacks. While code reuse attacks (e.g., ROP [65]) highly rely on information about where and how code has been placed in memory. Combined with diversification schemes like KASLR, read-disabling hinders the attacker from finding useful gadgets. As a special case, a few code pages need dynamic updates at runtime. For example, the code generated by the BPF Just-In-Time (JIT) compiler. These update requests are also directed to the monitor. After validation, BULKHEAD will temporarily grant the monitor access to these pages. Albeit whole kernel CFI provides stronger security, it comes at the cost of performance [46, 66]. Besides, constructing a precise global control flow graph (CFG) for the huge kernel is still challenging [67, 68]. As a trade-off, we enforce compartment interface integrity for the cross-compartment call. Cross-compartment CFI and private stack make XOM offer better security than itself.

*3) Compartment Interface Integrity (CII): Compartment switches must occur at the predefined entry/exit points and pass data according to security policies. (I3)*

Previous work [5, 28, 29] has assessed the serious impact of compartment interface vulnerabilities (CIVs), where a compartment can be exploited as a confused deputy to control the execution or corrupt data of other compartments. Accordingly, we propose CII against these threats on the basis of bi-directional isolation. First, the control flow is constrained by the predefined entry/exit points. Second, the shared data is validated according to security policies.

*Secure Switching.* BULKHEAD's monitor maintains a software SGT inspired by some hardware extensions for secure domain switching [40, 62]. Different from other user-mode switch gate designs, such as ERIM [51] and Hodor [52], the unique challenges of kernel compartmentalization come from multiple untrusted compartments and the privileged environment. As shown in Figure 6, the novel SGT contains informative metadata of all registered gates and directs each switch to the target address of the legal compartment. Secure switching implies that all switch gates satisfy the properties of atomicity, determinism, and exclusivity, which prevent malicious exploitation of interfaces and thus support CII.

***P1.** Atomicity: Compartment switches through switch gates are executed atomically.*

Atomicity means that the series of operations shown in Figure 6 cannot be exploited separately. The switch gate first uses a registered gate id to retrieve the gate metadata from the write-protected SGT. According to the metadata, it verifies the source address and updates `CR3` if the source and the target compartment are in different address spaces (i.e., two-level compartmentalization described in §V-E). Then, it changes `PKRS` to the target value on demand by **`wrmsr`**, switches to the private stack of the target compartment, and finally jumps to the target address recorded in the SGT. Here the most crucial step is the `PKRS` update, which implies the permission transfer.

Attackers can hijack the execution of the switch gate in two ways: (1) interrupting the sequence, (2) jumping to the middle of the sequence. First, malicious interrupts after `PKRS` updates can make the attacker gain the privilege of the original target compartment. To prevent this, we update `PKRS` to the default restricted value as the core kernel's privilege at each interrupt entry so that potentially illegal `PKRS` will not work for the interrupt context. Second, since the value of `PKRS` is not updated directly by an immediate operand but taken from the **`eax`** register, attackers may jump to just before the **`wrmsr`** instruction with forged **`eax`** value. As a countermeasure, we add a security check after **`wrmsr`** to guarantee the updated value is the same as the metadata in the SGT. If not, we loop back and update `PKRS` again, thus making sure the permission can only be transferred to the appointed compartment.

***P2.** Determinism: The switch gate behavior is uniquely determined by the gate id.*

Although the switch gate can be called from any compartment, its behavior cannot be influenced by attackers. To perform a compartment switch, the only accepted parameter is the gate id, and its behavior strictly follows the retrieved metadata. All gates must be registered first based on transition policies and do not trust the source or the target compartment. We also provide an API to register switch gates at runtime when loading new compartments. The registration requests the monitor to store metadata in the write-protected SGT. Each gate entry contains both the source and the target information as shown in Figure 6, including the address space (pgdir and ASID), the entry/exit address, the `PKRS` value, and the private stack pointer. The entry index is used as a unique gate id. Each switch gate validates the source compartment information and atomically switches to the target compartment determined by the metadata. As a result, there is no chance for attackers to disturb the switch with compromised input, even payloads on the stack. The informative metadata guarantees the integrity of permission transfer and control flow transfer.

Figure 7 demonstrates an example of compartment transition policies. BULKHEAD expects developers to specify flexible policies according to their requirements. Then allowed switch gates will be registered into the SGT. For instance, gate 22 allows the transition from LKM1 to LKM3, while the opposite direction (gate 23) is not allowed. If LKM5 is loaded at runtime and the user requests the monitor to register switch gates for it through our API. The monitor will check the gate to be registered against predefined policies and reject illegal requests. It guarantees that switch gates from LKM5 to modules other than LKM4 cannot be registered. As a result, an adversary cannot abuse the provided API to modify the policy maliciously.

*P3. Exclusivity: Compartment switches are exclusively possible via switch gates.*

With bi-directional isolation, the only way to access resources of other compartments is through secure compartment switches. Since switch gates are strictly based on the SGT controlled by the monitor, attackers may attempt to perform switches in other ways, such as abusing instructions for `PKRS` updates. However, thanks to instruction deprivation, all instructions that can be used to break access control are validated by the monitor. As a result, switch gates exclusively guard compartment switches to support CII.

*Zero-Copy Ownership Transfer.* To achieve secure and fast communication, BULKHEAD incorporates zero-copy ownership transfer for sharing data across compartments. The validation based on policies during ownership transfer thwarts confused deputy attacks, while zero-copy improves efficiency.

Data copying across compartments is a headache from multiple aspects. First, simply trusting shared data violates bi-directional isolation and leads to CIVs. Second, since the Linux kernel utilizes many tricky programming idioms like sentinel arrays and recursive structures, the synchronization between compartments requires complex and error-prone analysis and marshaling [19]. Lastly, passing large buffers will cause significant performance overhead [40]. Unlike PT/EPT switching-based approaches, PKS-based compartmentalization supports zero-copy communication due to its tagging mechanism.

BULKHEAD enforces single ownership of objects. As described in ***I1*** (§V-D1), all kernel data is tagged with a `pkey`. Besides, each compartment has its own private heap. At any time, an object is owned and can be accessed by exactly one compartment. There are three typical ways to share data: global variables, cross-compartment function call arguments and return values. Either way, access from the target compartment will trigger a page fault first. The dedicated page fault handler allows the monitor to validate the shared data to avoid confused deputy attacks. If the shared data conforms to the developer-defined policy, its `pkey` will be updated to the target's, which means the ownership transfer without data copy. We provide an API to define the data transfer policy, including the source and target of shared data and its legal value ranges. So developers can deploy different compartmentalization policies flexibly and perform security checks in the page fault handler on demand.

Taking CVE-2022-1015 in Listing 1 as an example, `attr` is data shared between `nf_tables` and other compartments. `nft_parse_register` parses a register value from the netlink attribute `attr`. However, improper `reg` validation in

```
static int nft_validate_register_load(enum nft_registers
↪  reg, unsigned int len)
{
        if (reg < NFT_REG_1 * NFT_REG_SIZE /
        ↪  NFT_REG32_SIZE)
                return -EINVAL;
        if (len == 0)
                return -EINVAL;
        if (reg * NFT_REG32_SIZE + len >
        ↪  sizeof_field(struct nft_regs, data))
        /* A large value of reg could overflow the integer
        ↪  and bypass the check */
                return -ERANGE;
        return 0;
}

int nft_parse_register_load(const struct nlattr *attr, u8
↪  *sreg, u32 len)
{
        u32 reg;
        int err;
        reg = nft_parse_register(attr);
        err = nft_validate_register_load(reg, len);
        ...
}
EXPORT_SYMBOL_GPL(nft_parse_register_load);
```

**Listing 1:** CVE-2022-1015: improper `reg` validation could lead to integer overflow in net/netfilter/nf_tables_api.c.

`nft_validate_register_load` could lead to integer overflow and thus out-of-bounds write issues. Based on constraints on `reg` in `nft_parse_register` and the patch for this vulnerability, we add security checks during `attr` transfer. The dedicated page fault handler guarantees the register value included in `attr` is within legal ranges: `NFT_REG_VERDICT...NFT_REG_4` or `NFT_REG32_00...NFT_REG32_15`.

## E. Two-level Compartmentalization

The Linux kernel contains thousands of LKMs developed by programmers with varying levels of expertise, requiring fine-grained compartmentalization to constrain potential vulnerabilities. However, the 4-bit `pkey` limits the available compartments to a maximum of 16. Considering that we reserve `pkey0` for the core kernel, `pkey1` for XOM, and `pkey2` for the monitor, the number of compartments available for other kernel modules is further reduced to 13, which cannot satisfy the compelling need for more isolated compartments.

To fulfill the scalability objective (§II-C), we utilize a two-level compartmentalization scheme. The first level is PKS-based intra-address space isolation described in §V-B, while the second level is locality-aware address space switching with ASID. We group modules into multiple address spaces according to the locality of module interactions. With different memory mapping, each address space has its own 16 `pkeys`, thus supporting 16 new compartments.

HAKC [2] proposed a fundamentally different two-level compartmentalization approach for ARM, which does not work for BULKHEAD. Its first level is address space coloring via MTE, while the second level recycles colors through cryptographic hashes based on PA. The Intel architecture does not feature hardware primitives like PA. In addition, cryptographic operations are too expensive for frequent compartment switches. Therefore, we have to design a more compatible and efficient way to overcome the limitation of few tag bits.

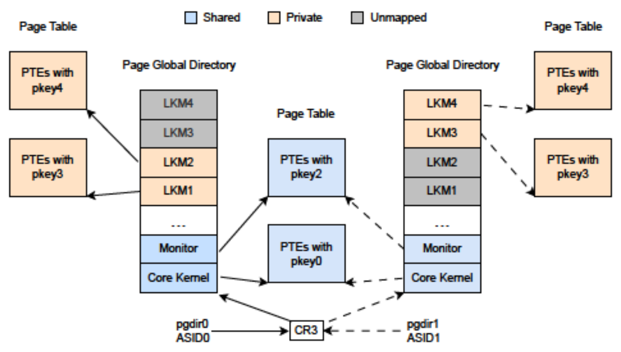


**Fig. 8:** The illustration of two-level compartmentalization. For example, LKM1 and LKM2 are isolated within the same address space through different `pkeys`, while LKM1 and LKM3 are isolated in different address spaces and reuse the same `pkey`.

BULKHEAD takes lessons from existing solutions on the userspace PKU scalability. Besides hardware extensions [69, 70], these approaches roughly fall into two categories: dynamic PT modification [71] and multiple address spaces [72, 73]. By modifying the PTE, the corresponding `pkey` can be evicted and then allocated to another domain. It is essentially a kind of multiplexing and suffers from significant performance degradation with increasing domains due to TLB flushes and busy waiting. In contrast, address space switching is an efficient way to support unlimited domains. Nevertheless, PKU virtualization approaches [72, 73] all rely on the underlying kernel for scheduling and management, and thus cannot migrate to the kernel space directly. MOAT [54] isolated BPF programs into different address spaces, but universally splitting the monolithic kernel is still a challenge.

However, we found that scalable kernel compartmentalization does not require complex address space splitting, based on two observations. First, interactions between kernel modules show a pattern of locality. Although there are thousands of LKMs in Linux, each module only needs to interact with a limited number of modules for its functionality. For example, in Linux kernel v6.1 with x86_64 generic Kconfig, only 54 of 6296 LKMs depend on more than 12 modules, which include indirect dependencies. Thus we can split the address space based on module dependencies specified by `modules.dep` to reduce address space switches. Notably, the dependencies do not mean running all dependent modules strictly at the same time. With more address space switching, we can support unlimited compartments as a kind of multiplexing, even for the 54 cases. In Figure 8, we assume that LKM2 depends on LKM1 and map them in the same address space, while the unrelated modules like LKM3 and LKM4 can be isolated in another address space to reuse `pkeys`.

Second, the Linux kernel employs hierarchical PTs for memory mapping and we do not have to duplicate all the PTs for multiple address spaces. Figure 8 shows an example with two-level PTs. We divide the PTs into the shared and private parts. Since all modules need to interact with the core kernel and the monitor, all address spaces share the same mapping of them. On the other hand, the memory reusing the same `pkeys` with other address spaces is private to prevent collision. During address space switching, we unmap the source space's private part and remap the private part of the target. Thanks to the hierarchical structure, PT updates mainly occur at the low-level PTs. Thus we can adjust the memory mapping efficiently to maintain different views. Besides, we attach each address space with an address space identifier (ASID), also called process context identifier (PCID) for x86, to avoid TLB flushes as optimization. The address translation only uses TLB entries with the same ASID as the current page table base register (`CR3`). Specifically, address space switches are also advised by the monitor based on the SGT. The registered gate entry records the source and target page global directory addresses (`pgdir`) as well as ASIDs (Figure 6). Due to the secure properties of gates, attackers cannot forge a malicious address space to bypass isolation. Thus, we realize secure and locality-aware two-level compartmentalization that supports unlimited compartments.

## VI. Implementation

To test the effectiveness of our design, we have implemented a prototype based on the Linux kernel v6.1, the newest long-term support (LTS) version when we started this work, and LLVM version 14.0.0. The BULKHEAD system consists of a set of kernel patches, a lightweight monitor module, and LLVM passes for instruction deprivation and automated gate instrumentation. Since the Linux kernel did not have support for Intel PKS at the time of writing, we implemented a series of patches to provide PKS-related API like existing API for PKU, such as `pkey` allocation and `PKRS` updates. This part includes 869 lines of addition and 90 lines of deletion. On the basis of these APIs, we implemented the in-kernel monitor as a separate LKM for portability, which only contains 2759 lines of code. The small codebase makes formal modeling and verification possible [74], which is one of our ongoing works.

Although BULKHEAD is implemented over Linux for its open-source nature, we expect the main design principles like bi-directional isolation and two-level compartmentalization could also be extended to other commodity OSs with PKS support. Here we highlight some key points.

*PKRS State.* BULKHEAD manages the `PKRS` state by the monitor. Unlike `PKRU`, the per-thread `PKRS` register is not XSAVE-supported by hardware. Therefore, our software implementation needs to save and restore its state on context switches and exceptions. However, attackers could overwrite the stored `PKRS` in memory to gain control over the register indicating access permissions. As a countermeasure, we store the `PKRS` state in a specific region write-protected by the monitor, which ensures that attackers cannot tamper with its value.

*Multi-threading Support.* BULKHEAD supports multi-threading securely. Since `PKRS` is per-thread and its state is write-protected, the access control inherently supports multi-threading. Concurrent access to shared data may lead to vulnerabilities, such as time-of-check to time-of-use (TOCTTOU) attacks. In response, we enforce exclusive access to shared data through single ownership, which means that concurrent compartments other than the owner cannot modify the data.

*Write-Protected Page Tables.* We reserve a page pool tagged with the monitor's `pkey` for PTs. To solve the page split issue of the direct map region, the pool remains enough pages to break down huge pages into the 4 KB size. Then we identify all kernel functions that allocate or update PTs. For PT allocation, a wrapper function is added to allocate from the reserved page pool. For PT updates, we insert switch gates to the monitor to access the PT securely. In order to prevent omissions, we apply a checker that scans for unprotected PT pages. PTs that are allocated before BULKHEAD initialization are also caught by the checker and then tagged with the `pkey`. As a result, we guarantee all PTs are write-protected by the monitor.

*Private Heap.* We implement private heaps to protect dynamically allocated objects of compartments. Similarly to the PT, dedicated memory caches with specific `pkey` are provided for private object allocation. We extend both the buddy allocator and the SLAB/SLUB allocator to return objects tagged with the desired `pkey` from the caches.

*Page Alignment.* Since `pkey`s are associated with PTEs, BULKHEAD can only protect memory at the page granularity, which is generally 4KB. Objects may interleave with other compartments across the page boundaries and lead to imprecise access control. To prevent this, we force all compartments to allocate objects from their private heaps. The reserved page pool ensures the compartment memory is page-aligned.

Data sharing across compartments also poses a challenge. As described in §V-D3, zero-copy data transfer performs at the granularity of a page, which means that potentially more data gets shared and thus increases attack surfaces. One solution is to include only the objects to be shared on the transferred page. However, it may waste a lot of memory. As a trade-off between security and memory overhead, we group shared objects into different privilege classes according to the source and the target compartments, then put objects with the same privilege on a page. For instance, all objects shared between LKM1 and LKM2 could be gathered on the same page, while objects shared between LKM1 and LKM3 are located on another page. With this kind of object grouping, even over-sharing does not give a compartment access to resources it should not have.

*Switch Gate Registration and Instrumentation.* Switch gates are registered in pairs for bi-directional isolation. If a gate id $x$ represents the entry to a specific compartment, then $x + 1$ represents the return, and corresponding SGT entries contain the exact opposite of source and target information. Thus, the paired gates enforce CII and thwart ROP-like attacks on cross-compartment calls. BULKHEAD requires the trusted policy developers to define a set of legal transitions like Figure 7. During the initialization, we register two special gates as the entry and exit of the monitor. After that, the SGT is exclusively owned by the monitor and all registrations should transfer to it first. Then, the monitor will check the gate metadata against the policy and add the validated one to the SGT. Although users may attempt to register gates at runtime via an API for newly loaded compartments, these requests are also guarded by the monitor from violating the prior policy. To perform registered switches, the insertion of switch gate calls with the gate id is needed. For usability, we implement an LLVM pass that automatically inserts switch gate registrations and switch gate calls at a list of interfaces based on the boundary analysis.

### A. Use Case

We choose LKM isolation as the use case of BULKHEAD for security and practicability reasons. First, LKMs cause the majority of vulnerabilities as shown in Figure 1. Second, diverse LKMs provide specialized functionality, which is suitable for privilege separation. Third, the development conventions make LKM boundaries relatively clear, and we can benefit from existing static analysis methods to perform automated isolation via LLVM passes.

For compartmentalization policy generation, we apply several analyses on the LLVM intermediate representation (IR) of the kernel. The LKM's interfaces and shared data are identified by KSplit [19] static analysis, while security checks are specified by a constraint analysis [75]. Specifically, a boundary analysis will collect all interface functions between the LKM and other kernel compartments, including exported symbols, registered function pointers, and interrupt handlers. It is a bi-directional analysis and the interfaces will be classified as entries or exits. Then we identify the data accessible from both sides of the isolation boundaries. This shared data analysis is performed on the program dependence graph (PDG) built by SVF [76]. For data validation, we collect constraints on shared data along the use-define chains from interfaces and generate security checks for them. With these boundaries, shared data, and security checks, we can insert switch gates for secure switching and cross-compartment communication.

Given a specific LKM that needs to be isolated, we first allocate a `pkey` and tag its memory with the `pkey` during the module initialization. Then we register and insert the LKM entry/exit gates according to policies before loading it into the kernel. Different from KSplit, BULKHEAD employs zero-copy ownership transfer and thus does not need complex synchronization for data sharing. As a result, we write a simple LLVM pass instead of a specific interface definition language (IDL) compiler to realize automated LKM isolation with PKS.

## VII. Evaluation

This section answers the following questions through comprehensive evaluations of BULKHEAD for security, performance, and scalability:

**RQ1:** Can BULKHEAD defend against potential attacks under the proposed threat model (§VII-A)?

**RQ2:** How much overhead does BULKHEAD introduce to micro-benchmarks (§VII-B1)?

**RQ3:** What performance overhead does BULKHEAD impose on real-world applications (§VII-B2)?

**RQ4:** Is the performance of BULKHEAD scalable for multiple compartments (§VII-B3)?

**RQ5:** What's the memory overhead of BULKHEAD (§VII-C)?

### A. Security Analysis

We assess how BULKHEAD enforces security invariants ***I1-I3*** (§V-D) against attack vectors under our threat model for ***RQ1***. Furthermore, to evaluate the effectiveness of BULKHEAD in real-world situations, we performed some penetration tests and investigated real vulnerabilities as case studies.

*1) Data-oriented Attacks:* Attackers exploiting vulnerabilities within a compartment may attempt to get arbitrary read-and-write primitive and corrupt data of other compartments. BULKHEAD thwarts this threat by tagging compartments' memory with different pkeys and thus enforces data integrity (***I1***). The PKRS indicates the current compartment's permission and only allows access to its own data, adhering to the principle of least privilege. Since the access control by PKRS is thread-local, BULKHEAD also prevents cross-thread attacks inherently. We comprehensively isolate all types of kernel memory regions, including the heap, stack, physmap, and MMIO regions. The private heap guarantees the single ownership of heap objects, while the private stack hides the execution context from other compartments. PKS hardware validation also works on the physmap and MMIO regions, while malicious direct memory access (DMA) actions [77] are guarded by IOMMU [20]. With bi-directional isolation, even vulnerabilities in the core kernel cannot be exploited to compromise other subsystems.

Attackers may also seek ways to bypass or disable the PKS-based protection, either tampering with the permission bits in PTs, or abusing control registers. We block these attempts by write-protected PTs and instruction deprivation. PT updates and sensitive instructions can only be performed by the monitor with validation. Besides, the PKRS state is also exclusively managed by the monitor. Notably, PKU pitfalls [31–33] like syscall abuses are under a weaker threat model focusing on the userspace. The low-level countermeasures we take for the privileged kernel will make these attacks no sense.

*2) Control Flow Hijacking:* In addition to data-oriented attacks, control flow hijacking is another traditional threat to kernel security. Besides the private stack mentioned above, we mitigate it broadly in a lightweight but effective way, XOM (***I2***). On the basis of the $W \bigoplus X$ policy, all kernel code regions are unwritable and unreadable. Therefore, attackers cannot inject malicious code directly, and reusing the existing code also becomes difficult without layout information. With existing diversification schemes like KASLR and data compartmentalization, it is hard to break XOM through memory disclosure. Instead of expensive whole kernel CFI, we focus on the cross-compartment calls and enforce CII (***I3***). The in-kernel monitor manages the SGT to make switches atomic, deterministic, and exclusive, which prevents attackers from abusing the gates to jump to illegal compartments and gain elevated permissions.

*3) Confused Deputy Attacks:* Compartmentalization faces a special challenge of confused deputy attacks at the compartment interfaces. To address this problem, we highlight bi-directional isolation under the mutually untrusted threat model and enforce CII (***I3***) with the help of developer-specified policies. Specifically, with distrust in mind, the monitor will check both the source and the target metadata at interfaces. Cross-compartment communication is performed in the way of zero-copy ownership transfer. Any compartmentalization policy violation will be detected in the page fault handler.

We trust the system developer who specifies prior policies. Although an adversary may attempt to register malicious switch gates at runtime for newly loaded compartments, these requests are also guarded by the monitor from violating the policy.

*4) Penetration Tests:* We instantiated the above attack vectors on the LKM isolation use case to test the security of BULKHEAD. We simulate an attacker to (1) modify the heap object of other compartments, (2) tamper the PT directly, (3) forge PTs by mov-to-CR3, (4) update PKRS directly, (5) abuse the switch gate to hijack control flow, and (6) pass malicious data through interfaces. All attempts are detected by protection key violation or rejected by the monitor, which shows that BULKHEAD is immune to these attacks.

| CVE ID | Root Cause | Compartment | Countermeasures |
|---|---|---|---|
| 2023-4147 | use-after-free in net/netfilter/nf_tables_api.c | nf_tables | The private heap prevents the compartment from corrupting other kernel objects. |
| 2022-24122 | use-after-free in kernel/ucount.c | core kernel | |
| 2022-27666 | heap out-of-bounds write in net/ipv6/esp6.c | esp6 | |
| 2022-25636 | heap out-of-bounds write in net/netfilter/nf_dup_netdev.c | nf_dup_netdev | |
| 2021-22555 | heap out-of-bounds write in net/netfilter/x_tables.c | x_tables | |
| 2018-5703 | heap out-of-bounds write in net/ipv6/tcp_ipv6.c | ipv6 | |
| 2023-0179 | stack buffer overflow in net/netfilter/nft_payload.c | nf_tables | The private stack blocks cross-compartment stack corruption. |
| 2018-13053 | integer overflow in kernel/time/alarmtimer.c | core kernel | |
| 2022-1015 | improper input validation in net/netfilter/nf_tables_api.c | nf_tables | The monitor-enforced interface checks thwart confused deputy attacks. |
| 2022-0492 | missing authorization in kernel/cgroup/cgroup-v1.c | core kernel | |
| 2017-18509 | improper input validation in net/ipv6/ip6mr.c | ipv6 | |

**TABLE II:** Representative Linux kernel CVEs, their root causes, the located compartment, and the countermeasures of BULKHEAD.

*5) Real-world Vulnerabilities:* To evaluate how BULKHEAD mitigates real-world vulnerabilities, we select 11 representative Linux kernel CVEs according to the following criteria: (1) they should contain multiple error types and cover attack vectors we considered in §VII-A1 - §VII-A3; (2) they should be distributed in diverse compartments, including the core kernel and other LKMs; (3) their Proof-of-Concept (PoC) programs or exploits are available; (4) we prefer recent vulnerabilities with high severity, i.e., CVSS 3.x rating *7.8* or higher except for CVE-2018-13053 (*3.3*) and CVE-2022-1015 (*6.6*). Table II illustrates the CVE set with their root causes, the located compartment, and the countermeasures.

Since the compartment can only access its own objects tagged with the corresponding pkey, the private heap confines the damage of both heap use-after-free (UAF) vulnerabilities like CVE-2023-4147 and heap out-of-bounds (OOB) vulnerabilities like CVE-2022-27666. Stack buffer overflow (CVE-2023-0179) or integer overflow (CVE-2018-13053) can be exploited to corrupt stack variables or tamper with the return addresses. BULKHEAD mitigates these threats by maintaining a private stack for each compartment. In general, the data integrity (***I1***) blocks any attempts of memory corruption, which is the dominant threat to the Linux kernel. Vulnerabilities with insufficient parameter validation lead to confused deputy attacks, which is a challenge for traditional data protection techniques [78]. We address this problem through bi-directional isolation. The in-kernel monitor checks both the source and the target information during switching to enforce CII (***I3***). Thus, only legal parameters can be transferred across compartments. For example, BULKHEAD will strictly validate the input register values of nft_parse_register_load function to mitigate CVE-2022-1015 (Listing 1). Besides, many listed vulnerabilities, such as CVE-2021-22555, CVE-2022-1015, and CVE-2022-25636, may be further exploited to launch ROP attacks. As a principled countermeasure, XOM (***I2***) makes it extremely hard to find usable gadgets. Overall, BULKHEAD provides comprehensive protection against various kinds of real-world vulnerabilities to fulfill security objectives (§II-B).

## *B. Performance Evaluation*

We first describe our experiment setup. All evaluations were conducted on a machine with Intel Core i7-12700H CPU, 16 GB memory, and 500GB disk, running Ubuntu-22.04 with Linux kernel v6.1. Further, we set the kernel to performance mode and locked the CPU frequency to avoid randomness.

There are five types of configuration: (1) *monitor*, which only enables BULKHEAD in-kernel monitor for the PT and SGT protection; (2) *ipv6* with the compartmentalized `ipv6` module; (3) *ipv6-nft*, which isolates `ipv6` and `nf_tables` into separate compartments; (4) *lkm-20*, which isolates 20 LKMs into different address spaces detailed in §VII-B3; (5) *lkm-160*, which compartmentalizes all 160 LKMs with `localmodconfig` on the experimental machine. We specify compartmentalization policies for each setting based on the boundary analysis, shared data analysis, and security check analysis described in §VI-A.

*1) Micro-benchmarks:* For ***RQ2***, we measure the CPU cycles of the compartment switch and then use LMbench [79] to evaluate the latency and bandwidth overhead of BULKHEAD on micro-operations.

| Operation | Cost (cycles) |
|---|---|
| `PKRS` update by `wrmsr` | 185.31 |
| compartment switch | 224.31 |
| com-switch without stack switch | 203.96 |
| com-switch with address space switch | 523.69 |
| syscall null | 293.86 |
| hypercall null | 2894.06 |
| `vmfunc` (EPT switch) | 198.82 |

**TABLE III:** Cost comparison of BULKHEAD micro-operations.

We list the cost of the compartment switch operations in Table III. The CPU cycles are measured by the `rdtscp` instruction, and we average ten runs of 10000 invocations. A single `PKRS` update by `wrmsr` takes about 185.31 cycles, while a secure compartment switch through the gate takes 224.31 cycles. As a reference, the null syscall on the experimental machine costs 293.86 cycles, and a simple hypercall into the hypervisor costs 2894.06 cycles. Besides, the `PKRS` update is even faster than the `vmfunc` instruction, which is well-known for its efficient EPT switch. Considering the security guarantees, virtualization-based approaches will suffer worse performance overhead due to nested paging and I/O virtualization. Although the additional address space switch increases the cost to 523.69 cycles, the locality-aware two-level compartmentalization scheme makes this case rarely happen and almost does not affect performance, which is shown in the following evaluation.

Table IV illustrates the LMbench evaluation results, with vanilla kernel v6.1 as the baseline. To achieve stable results, we run each benchmark 100 times and take the average.

| Benchmarks | monitor | ipv6 | ipv6-nft | lkm-20 | lkm-160 |
|---|---|---|---|---|---|
| | | | ***Latency*** | | |
| syscall null | -0.37 | -0.28 | -0.30 | 0.09 *(4)* | 0.36 *(12)* |
| simple read | -2.65 | -1.68 | -1.42 | -1.92 *(4)* | -1.90 *(12)* |
| simple write | -2.30 | -2.34 | -2.08 | -2.00 *(4)* | -1.58 *(12)* |
| simple stat | -0.03 | 0.58 | 0.05 | -0.17 *(4)* | 0.80 *(12)* |
| simple fstat | 0.22 | -0.74 | -1.20 | -0.74 *(4)* | -0.51 *(12)* |
| open/close | 0.85 | 0.87 | 0.79 | 0.07 *(4)* | 1.24 *(12)* |
| select on fd's | -1.26 | -1.20 | -1.38 | -0.83 *(4)* | -1.18 *(12)* |
| select on tcp fd's | -0.64 | -0.41 | -0.57 | -0.30 *(4)* | -0.53 *(12)* |
| signal install | -0.01 | -0.19 | 0.39 | 0.50 *(4)* | 0.43 *(9)* |
| signal handler | -0.21 | -0.26 | 1.53 | -1.96 *(4)* | -0.14 *(9)* |
| proc fork+exit | 26.88 | 27.10 | 27.79 | 27.84 *(4)* | 28.30 *(9)* |
| proc fork+exec | 15.86 | 15.80 | 16.06 | 15.86 *(4)* | 16.26 *(9)* |
| proc shell | 14.59 | 14.47 | 14.77 | 14.63 *(4)* | 15.25 *(9)* |
| page fault | 39.71 | 39.67 | 39.76 | 39.89 *(6)* | 38.64 *(9)* |
| pipe latency | 0.68 | 1.09 | 1.77 | 1.02 *(4)* | 1.89 *(11)* |
| UDP latency | 0.96 | 0.78 | 1.41 | 2.50 *(4)* | 3.09 *(11)* |
| TCP latency | 0.35 | 0.62 | 1.31 | 1.57 *(6)* | 3.54 *(11)* |
| | | | ***Bandwidth*** | | |
| file write bandwidth | -1.75 | -3.99 | -3.57 | -2.78 *(6)* | -3.05 *(9)* |
| pipe bandwidth | 1.42 | -0.35 | 0.77 | -0.11 *(4)* | 0.18 *(11)* |
| AF_UNIX bandwidth | 0.34 | 0.58 | 0.23 | 0.35 *(4)* | 0.23 *(11)* |

**TABLE IV:** BULKHEAD performance overhead (in % over the vanilla kernel) on LMbench. The numbers in parentheses represent the number of compartments traversed for each benchmark.

The table reveals two main observations. Focusing on the columns, BULKHEAD introduces negligible overhead of less than 1% in most benchmarks, except for the process operations and page faults. The increased runtime overhead stems from compartment switches with the monitor due to the PT protection. However, as we will see from the macro-benchmarks, it does not have a noticeable impact on real-world applications. While looking at the table by rows, we can observe that the system-wide micro-benchmark is almost unaffected by LKM isolation, regardless of the number of involved compartments. This is because the switches between LKMs occur rarely on LMbench. Especially, the results of *lkm-20* and *lkm-160* show the scalability of BULKHEAD (***RQ4***). A few benchmarks perform slightly better than the vanilla kernel within a reasonable margin of fluctuation. It comes down to the system noise like improved cache hit rates, which is hard to avoid as prior works [23, 41, 80] show.

*2) Macro-benchmarks:* We performed two sets of evaluations to answer ***RQ3***. First, we demonstrate the system-wide impact of BULKHEAD by running a collection of Phoronix Test Suites [35]. Then, we zoom in on the performance overhead of a specific compartment. We select the vulnerable `ipv6` module as the target and test it on ApacheBench [36] so that we can draw a comparison with HAKC - one of the state-of-the-art kernel compartmentalization approaches.

The Phoronix suites provide a large number of system-wide tests, from which we select some representative ones to comprehensively characterize the performance of BULKHEAD. Our benchmarks are split into application tests and stress tests for specific subsystems. The application benchmarks include nginx (measures sustained requests/second, varying the number of concurrent connections); phpbench (tests the PHP interpreter); pybench (tests basic, low-level functions of Python); povray (3D ray tracing); gnupg (encryption time with GnuPG). Stress tests include dbench (measures disk performance via file system calls, varying the number of clients); postmark (transactions on 500 small files simultaneously); sysbench (performs CPU and memory tests). Table V shows the performance overhead compared to the vanilla kernel v6.1. Nginx incurs 3.57%-7.29% slowdown since the continuous network packet processing results in frequent compartment switches. All other benchmarks present negligible overhead and the average performance overhead with the `ipv6` module compartmentalized is 1.28%.

To further measure the overhead on the isolated `ipv6` module. We use ApacheBench to retrieve a 100KB, 1MB,

| Benchmarks | monitor | ipv6 | ipv6-nft | lkm-20 | lkm-160 |
|---|---|---|---|---|---|
| nginx-100 | 4.88 | 5.03 | 6.01 | 5.70 *(7)* | 7.29 *(19)* |
| nginx-200 | 4.47 | 4.55 | 5.54 | 5.38 *(7)* | 6.54 *(19)* |
| nginx-500 | 3.57 | 3.68 | 4.40 | 4.51 *(7)* | 5.74 *(19)* |
| phpbench | -0.24 | -0.12 | -0.44 | -0.28 *(7)* | 0.33 *(18)* |
| pybench | 0.35 | 0.17 | 0.43 | 0.52 *(7)* | 1.37 *(18)* |
| povray | 0.16 | 0.57 | 0.22 | 0.39 *(7)* | 0.2 *(17)* |
| gnupg | 0.10 | 0.01 | 0.35 | 0.08 *(7)* | 1.03 *(18)* |
| dbench-1 | 0.19 | 0.20 | 0.19 | 0.04 *(7)* | 0.47 *(19)* |
| dbench-48 | 0.52 | 1.05 | 1.73 | 3.74 *(7)* | 5.61 *(19)* |
| dbench-256 | 0.22 | 1.22 | 2.38 | 1.64 *(7)* | 2.11 *(19)* |
| postmark | 1.84 | 0.00 | 1.14 | 1.14 *(7)* | 0.39 *(18)* |
| sysbench-cpu | -0.05 | -0.03 | -0.04 | -0.01 *(7)* | 0.01 *(19)* |
| sysbench-mem | 0.02 | 0.26 | -0.53 | 0.53 *(7)* | 0.69 *(18)* |
| **Average** | **1.23** | **1.28** | **1.64** | **1.80** *(7)* | **2.44** *(18.46)* |

**TABLE V:** BULKHEAD performance overhead (in % over the vanilla kernel) on Phoronix Test Suites. The numbers in parentheses represent the number of compartments traversed for each benchmark.

| net | | | fs | usb |
|---|---|---|---|---|
| nfnetlink | nf_conntrack | ip_tables | overlay | typec |
| nf_defrag_ipv4 | nf_nat | ip6_tables | pstore_zone | tps6598x |
| nf_defrag_ipv6 | nf_tables | nft_compat | ramoops | xhci_pci_renesas |
| ipv6 | x_tables | nft_nat | pstore_blk | xhci_pci |

**TABLE VI:** The 20 compartmentalized LKMs from the 3 most vulnerable subsystems.

and 10MB file 1000 times from a local Apache server through an ipv6 address. The `nf_tables` module implements a packet filtering mechanism within the kernel. Each experiment is repeated 10 times to avoid randomness. The results normalized to the vanilla kernel are listed in Figure 9, and the metric of transfer rate reveals a similar trend to requests/second. We also show the overhead reported in HAKC for comparison. The overhead of most settings is around 2%, which is much better than HAKC. Even with more compartments traversed, the performance does not show a significant degradation. Specifically, the *lkm-20* setting executes 4 compartments, while the *lkm-160* setting executes 14 compartments. In contrast, HAKC only evaluated the `ipv6` and `nf_tables` modules.

*3) Scalability:* One of the major drawbacks of the previous work is the significant degradation of performance as the number of isolated domains increases. In particular, HAKC suffers from a linear growth of 14%-19% per compartment involved in overhead, which makes it impractical. To indicate the effectiveness of locality-aware two-level compartmentalization and answer ***RQ4***, we perform two sets of experiments.

*lkm-20* selects LKMs listed in Table VI from the 3 most vulnerable subsystems according to the vulnerability analysis in §II-A. There are 12 LKMs from `net`, 4 LKMs from `fs`, and 4 LKMs from `usb`. The `net` LKMs belong to one address space, while other LKMs belong to the other. *lkm-160* extensively isolate all LKMs supported by the experimental machine with `localmodconfig`, 160 compartments in total. We partition these LKMs into different address spaces according to module dependencies, specify compartmentalization policies, and comprehensively isolate them with BULKHEAD. If a module depends on more than 12 modules, the dependent modules will occupy more than one address space and BULKHEAD will perform address space switching on demand. The evaluation results in Table IV, Table V, and Figure 9 show that the increase

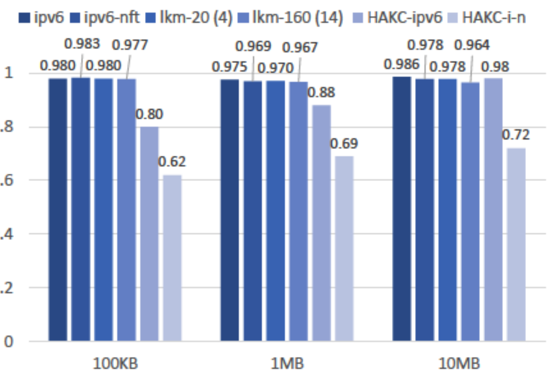


**Fig. 9:** BULKHEAD performance overhead normalized to the vanilla kernel when transferring various sized payloads on ApacheBench (requests/sec), compared with the overhead of HAKC [2].

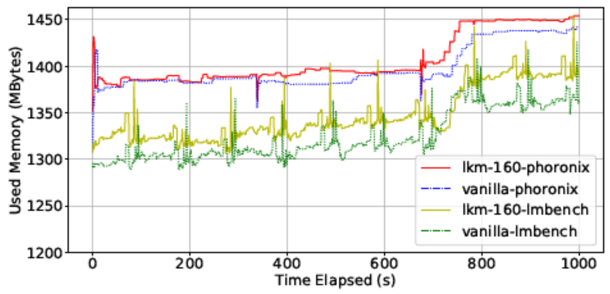


**Fig. 10:** Memory usage of BULKHEAD when running LMbench and Phoronix with *lkm-160* and the vanilla kernel.

in the number of compartments and address spaces does not significantly increase overhead, benefiting from our lightweight switch gates and locality-aware design.

Since not all compartments present are executed, we recorded the number of compartments actually traversed for each benchmark to reflect scalability faithfully. The results reveal several findings. First, each benchmark involves only a few compartments, which also corroborates our locality-aware design. Take Phoronix test suites for example, the *lkm-20* setting executes 7 compartments, and the *lkm-160* setting executes 18.46 compartments on average. Nevertheless, we have to prepare a large number of compartments for complex workloads so as to confine all possible vulnerabilities. Second, the presence of compartments not involved does not affect performance. Third, looking at Table IV and Table V by rows, the increase in the number of traversed compartments only slightly adds to the overhead. Lastly, the vast majority of compartment transitions are monitor entries/exits for privileged instruction or critical objects (e.g., PT) updates, which explains the slight overhead caused by other traversed compartments. For LMbench, 99.99% of *lkm-20* transitions and 99.75% of *lkm-160* transitions are monitor entries/exits, while for Phoronix, the percentages are 99.82% of *lkm-20* transitions and 93.40% of *lkm-160* transitions. Overall, with two-level compartmentalization, we can support unlimited compartments without sacrificing performance.

## C. Memory Overhead

As mentioned in §VI, BULKHEAD groups shared objects based on privilege, and makes them page-aligned to avoid over-sharing between compartments. For ***RQ5***, we measure the memory overhead caused by page alignment. Figure 10 presents the memory usage when running LMbench and Phoronix with *lkm-160* and the vanilla kernel. On average, the memory overhead is 1.66% for LMbench and 0.63% for Phoronix, which is negligible for modern systems.

## VIII. Discussion

*PKS Granularity.* BULKHEAD leverages PKS for compartmentalization, which only supports page-size granularity. To prevent privilege escalation caused by over-sharing and imprecise access control, we group shared objects into different privilege classes according to the source-target compartments and make all compartment memory page-aligned. We argue that the page granularity is already considered fine-grained for monolithic kernel compartmentalization and is comparable to other PT/EPT-based efforts [18, 37, 38]. Furthermore, BULKHEAD allows developers to select specific objects for additional bound checking during data transfer. Combining SFI on the basis of PKS-based protection will realize finer isolation granularity at the cost of performance.

*Policy Generation.* As a fundamental compartmentalization mechanism, BULKHEAD can support developer-defined policy flexibly. The LKM isolation use case deploys a relatively simple policy based on boundary analysis, shared data analysis, and security check analysis. More complex policies may further improve kernel security, but it is still an open problem to generate compartmentalization policy automatically. Basically, exploring the optimal policy can be reduced to the Partition Problem, which is NP-Complete [81]. Some attempts have been made on bare-metal systems [82, 83]. However, scaling the complex static analysis to a huge kernel like Linux is extremely hard. $\mu$SCOPE [3] proposed another way based on dynamic analysis but cannot guarantee the soundness of the result. Combining static and dynamic analysis for policy generation is promising [84], and we expect type-based dependence analysis [78] can provide a practical solution, which is one of our ongoing works.

*Performance Optimization.* Though BULKHEAD shows acceptable overhead even for multiple kernel compartments. Its performance can be further optimized through some engineering efforts. One possible option is to eliminate redundant checks during compartment communication. For example, we can create a fast path for switches that occurs frequently in a short term once the two compartments pass the first check and establish trust for communication temporarily.

## IX. Related Work

*Microkernels.* As opposed to the monolithic architecture, microkernels [6–9] maintain only the core functionality in the kernel space to minimize the attack surface. The reduced codebase facilitates formal verification [7]. More recently, some efforts even explored the development of microkernels in safe languages, such as RedLeaf [9] written in Rust. However, a cost coming with these security benefits is its low performance, especially the IPC overhead between isolated components [8]. Besides, the shift from traditional monolithic kernels to microkernels requires complete system redesign, while BULKHEAD enhances the monolithic kernel security with minor engineering effort through compartmentalization.

*SFI-based Approaches.* Software fault isolation (SFI) [10] inserts security checks during compilation time to regulate all accesses within specific domain boundaries. Following this basic idea, BGI [12] manages an access control list to isolate Windows drivers. LXFI [13] further enforces kernel API integrity based on programmers' annotations. Nevertheless, the heavy checks impose a significant performance overhead. What's worse, as the number of domains increases and cross-domain interactions become more frequent, the performance of SFI-based approaches degrades sharply [85]. For example, BGI induces over 30% throughput loss for isolating many memory blocks. Benefiting from PKS-based permission validation, BULKHEAD avoids the significant overhead caused by software checks.

*PT Switching and Virtualization-based Isolation.* Since the page table (PT) is a critical structure in charge of address translation and permission validation, PT switching is another well-explored mechanism for isolation. There are several works using different PTs to construct different memory views for userspace applications, such as lwC [86] and SMV [87]. While representative studies for kernel isolation are Nooks [88], SIDE [89], and SKEE [38], all suffer from significant overhead caused by updating PT and flushing TLB. With the development of virtualization technology, researchers further introduced the hypervisor to facilitate EPT switching and enhance security [14–19]. In contrast, BULKHEAD does not need the additional privilege layer as TCB, instead a lightweight in-kernel monitor enforces comprehensive security invariants.

*Hardware-based Isolation.* Besides Intel PKS, researchers have explored various hardware features for isolation [2, 20–22, 90, 91]. Nested Kernel [20] utilizes the WP (Write-Protect bit) mechanism of the x86-64 hardware for intra-kernel privilege separation. Hilps [21] leverages the TxSZ mechanism of AArch64 to build a secure kernel domain by dynamic virtual address range adjustment. However, both approaches only support isolation between two domains. DIKernel [22] enforces isolation between the core kernel and extensions with the ARM Memory Domain Access Control mechanism, which has been deprecated recently. Memory Tagging Extension (MTE) and Pointer Authentication (PA) are recent features on ARM, which have been used for in-process compartmentalization [91] and kernel compartmentalization [2] but show significant performance overhead. Works like Mondrix [92], CODOMs [93], CHERI [39], and SecureCells [40] facilitate compartmentalization through newly designed hardware architecture. Compared to these efforts, BULKHEAD is based on the real available commodity hardware feature, which is more compatible and practical. Moreover, it breaks the hardware limitation of PKS to support unlimited compartments.

*Other System Compartmentalization.* Compartmentalization has been explored as a principled defense against the myriad possible faults in software other than the monolithic kernel. Wedge [94] splits complex applications into fine-grained, least-privilege compartments. SOAAP [95] allows programmers to reason about application compartmentalization using source code annotations. Glamdring [96] uses annotations and static analysis to partition applications for SGX. In addition to user-space applications, there is also a series of works on compartmentalizing embedded systems [82, 83, 97, 98] and libOS [99–101]. For instance, ACES [82] enforces developer-specified policies with the MPU hardware feature for embedded system compartmentalization. EC [83] further provides a comprehensive and automatic compartmentalization toolchain for Real-Time Operating Systems (RTOSs) and bare-metal firmware. FlexOS [99] specializes in the isolation strategy of a libOS at compilation/deployment time instead of design time, and enforces strategies with multiple hardware and software protection mechanisms. In contrast, BULKHEAD targets the monolithic kernel, which is more challenging due to its large codebase and privileged environment.

## X. Conclusion

In this paper, we present BULKHEAD, a secure, scalable, and efficient kernel compartmentalization approach using a novel application of PKS. It guarantees bi-directional isolation between compartments and introduces a lightweight in-kernel monitor to enforce security-critical invariants, especially compartment interface integrity against confused deputy attacks. Besides, a locality-aware two-level scheme can provide unlimited compartments for scalability. We implement a prototype system for automated LKM isolation and extensive evaluations show that BULKHEAD incurs negligible performance overhead on real-world applications.

## Acknowledgment

We sincerely appreciate the anonymous reviewers for their insightful comments. Authors from Nanjing University were supported in part by NSFC under Grant Nos. 61772266, 61431008. Yinggang Guo was also partly supported by a scholarship from the Graduate School of Nanjing University and by the funding from the University of Minnesota. Weiheng Bai and Kangjie Lu were supported in part by NSF awards CNS-1815621, CNS1931208, CNS2045478, CNS-2106771, and CNS-2154989. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

## References

[1] SecurityScorecard. (2024, Apr.) Linux kernel vulnerabilities. [Online]. Available: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

[2] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakc," in *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, vol. 22, 2022, pp. 1–17. [Online]. Available: https://doi.org/10.14722/ndss.2022.24026

[3] 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, "$\mu$scope: A methodology for analyzing least-privilege compartmentalization in large software artifacts," in *24th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 296–311. [Online]. Available: https://doi.org/10.1145/3471621.3471839

[4] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1451869

[5] H. Lefeuvre, V.-A. Bădoiu, Y. Chen, F. Huici, N. Dautenhahn, and P. Olivier, "Assessing the impact of interface vulnerabilities in compartmentalized software," in *Proceedings 2023 Network and Distributed System Security Symposium.* Internet Society, 2023. [Online]. Available: https://doi.org/10.14722%2Fndss.2023.24117

[6] J. Liedtke, "On micro-kernel construction," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 237–250. [Online]. Available: https://doi.org/10.1145/224056.224075

[7] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[8] J. Gu, X. Wu, W. Li, N. Liu, Z. Mi, Y. Xia, and H. Chen, "Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 401–417. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/gu

[9] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev, "RedLeaf: Isolation and communication in a safe operating system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 21–39. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[10] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, p. 203–216, Dec. 1993. [Online]. Available: https://doi.org/10.1145/173668.168635

[11] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. USA: USENIX Association, 2006, p. 75–88. [Online]. Available: https://www.usenix.org/conference/osdi-06/xfi-software-guards-system-address-spaces

[12] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 45–58. [Online]. Available: https://doi.org/10.1145/1629575.1629581

[13] 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*, ser. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 115–128. [Online]. Available: https://doi.org/10.1145/2043556.2043568

[14] X. Xiong, D. Tian, P. Liu *et al.*, "Practical protection of kernel integrity for commodity os from untrusted extensions." in *NDSS*, vol. 11, 2011. [Online]. Available: https://www.ndss-symposium.org/ndss2011/practical-protection-of-kernel-integrity-for-commodity-os-from-untrusted-extensions/

[15] R. Nikolaev and G. Back, "Virtuos: An operating system with kernel virtualization," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 116–132. [Online]. Available: https://doi.org/10.1145/2517349.2522719

[16] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou, "SecPod: a framework for virtualization-based security systems," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 347–360. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/wang-xiaoguang

[17] 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)*. Renton, WA: USENIX Association, Jul. 2019, pp. 269–284. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/narayanan

[18] 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*, ser. VEE '20. New York, NY, USA: Association for Computing Machinery, 2020, p.

157–171. [Online]. Available: https://doi.org/10.1145/3381052.3381328

[19] 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)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 613–631. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/huang-yongzhe

[20] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 191–206. [Online]. Available: https://doi.org/10.1145/2694344.2694386

[21] Y. Cho, D. Kwon, H. Yi, and Y. Paek, "Dynamic virtual address range adjustment for intra-level privilege separation on arm," in *NDSS*, 2017. [Online]. Available: https://doi.org/10.14722/NDSS.2017.23024

[22] V. J. Manès, D. Jang, C. Ryu, and B. B. Kang, "Domain isolated kernel: A lightweight sandbox for untrusted kernel extensions," *Computers & Security*, vol. 74, pp. 130–143, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404818300282

[23] S. Gravani, M. Hedayati, J. Criswell, and M. L. Scott, "Fast intra-kernel isolation and security with iskios," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 119–134. [Online]. Available: https://doi.org/10.1145/3471621.3471849

[24] Google. (2024, Apr.) syzbot reported kernel bugs. [Online]. Available: https://syzkaller.appspot.com/

[25] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "MiniBox: A Two-Way sandbox for x86 native code," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 409–420. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_yanlin

[26] Y. Chen, J. Li, G. Xu, Y. Zhou, Z. Wang, C. Wang, and K. Ren, "SGXLock: Towards efficiently establishing mutual distrust between host application and enclave for SGX," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 4129–4146. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-yuan

[27] C. Castes, A. Ghosn, N. S. Kalani, Y. Qian, M. Kogias, M. Payer, and E. Bugnion, "Creating trust by abolishing hierarchies," in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, ser. HOTOS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 231–238. [Online]. Available: https://doi.org/10.1145/3593856.3595900

[28] Y. Chien, V.-A. Bădoiu, Y. Yang, Y. Huo, K. Kaoudis, H. Lefeuvre, P. Olivier, and N. Dautenhahn, "Civscope: Analyzing potential memory corruption bugs in compartment interfaces," in *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*, ser. KISV '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 33–40. [Online]. Available: https://doi.org/10.1145/3625275.3625399

[29] A. Burtsev, V. Narayanan, Y. Huang, K. Huang, G. Tan, and T. Jaeger, "Evolving operating system kernels towards secure kernel-driver interfaces," in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, ser. HOTOS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 166–173. [Online]. Available: https://doi.org/10.1145/3593856.3595914

[30] Intel. (2023, Dec.) Intel 64 and ia-32 architectures software developer manuals. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[31] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on PKU-based memory isolation systems," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1409–1426. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[32] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, "You shall not (by)pass! practical, secure, and fast pku-based sandboxing," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 266–282. [Online]. Available: https://doi.org/10.1145/3492321.3519560

[33] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing syscalls for PKU-based memory isolation systems," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 936–952. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

[34] J. Edge. (2013, Oct.) Kernel address space layout randomization. [Online]. Available: https://lwn.net/Articles/569635/

[35] P. Media, "Phoronix test suites: Open-Source, Automated Benchmarking," https://www.phoronix-test-suite.com/, 2023.

[36] T. A. S. Foundation, "Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html, 2023.

[37] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xmp: Selective memory protection for kernel and user space," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 563–577. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9152671

[38] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "Skee: A lightweight secure kernel-level execution environment for arm," in *NDSS*, vol. 16, 2016, pp. 21–24. [Online]. Available: https://doi.org/10.14722/NDSS.2016.23009

[39] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 20–37. [Online]. Available: https://doi.org/10.1109/SP.2015.9

[40] A. Bhattacharyya, F. Hofhammer, Y. Li, S. Gupta, A. Sánchez Marín, B. Falsafi, and M. Payer, "Securecells: A secure compartmentalized architecture," in *44th IEEE Symposium on Security and Privacy*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 2921–2939. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00125

[41] L. Maar, M. Schwarzl, F. Rauscher, D. Gruss, and S. Mangard, "Dope: Domain protection enforcement with pks," in *Proceedings of the 39th Annual Computer Security Applications Conference*, ser. ACSAC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 662–676. [Online]. Available: https://doi.org/10.1145/3627106.3627113

[42] 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)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1357–1374. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/li-shih-wei

[43] Z. Mi, D. Li, H. Chen, B. Zang, and H. Guan, "(mostly) exitless VM protection from untrusted hypervisor through disaggregated nested virtualization," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1695–1712. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/mi

[44] J. Park, S. Kang, S. Lee, T. Kim, J. Park, Y. Kwon, and J. Huh, "Hardware hardened sandbox enclaves for trusted serverless computing," *ACM Trans. Archit. Code Optim.*, nov 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3632954

[45] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1963–1976. [Online]. Available: https://doi.org/10.1145/3548606.3560585

[46] J. Criswell, N. Dautenhahn, and V. Adve, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 292–307. [Online]. Available: https://doi.org/10.1109/SP.2014.26

[47] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 969–986. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7546545

[48] J. Gionta, W. Enck, and P. Larsen, "Preventing kernel code-reuse attacks through disclosure resistant code diversification,"

in *2016 IEEE Conference on Communications and Network Security (CNS)*, 2016, pp. 189–197. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7860485

[49] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172. [Online]. Available: https://ieeexplore.ieee.org/servlet/opac?punumber=7093633

[50] V. van Rijn and J. S. Rellermeyer, "A fresh look at the architecture and performance of contemporary isolation platforms," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 323–335. [Online]. Available: https://doi.org/10.1145/3464298.3493404

[51] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[52] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor:{Intra-Process} isolation for {High-Throughput} data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 489–504. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/hedayati

[53] H. Kuzuno and T. Yamauchi, "Kdpm: Kernel data protection mechanism using a memory protection key," in *Advances in Information and Computer Security*. Cham: Springer International Publishing, 2022, pp. 66–84. [Online]. Available: https://doi.org/10.1007/978-3-031-15255-9_4

[54] H. Lu, S. Wang, Y. Wu, W. He, and F. Zhang, "MOAT: Towards safe BPF kernel extension," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1153–1170. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/lu-hongyi

[55] H. LI, J.-Y. GU, Y.-B. XIA, B.-Y. ZANG, and H.-B. CHEN, "Memory isolation mechanism of ebpf based on pks hardware feature," *Journal of Software*, vol. 34, no. 12, p. 5921, 2023. [Online]. Available: https://www.jos.org.cn/josen/article/abstract/6762

[56] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: Architecture for tamper-evident and tamper-resistant processing," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 2003, pp. 357–368. [Online]. Available: https://doi.org/10.1145/782814.782838

[57] R. Wilkins and B. Richardson, "Uefi secure boot in modern computer security solutions," in *UEFI forum*, 2013, pp. 1–10. [Online]. Available: https://api.semanticscholar.org/CorpusID:14326971

[58] S. F. Yitbarek, M. T. Aga, R. Das, and T. Austin, "Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 313–324. [Online]. Available: https://doi.org/10.1109/HPCA.2017.10

[59] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019. [Online]. Available: https://doi.org/10.1109/TCAD.2019.2915318

[60] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19. [Online]. Available: https://doi.org/10.1109/SP.2019.00002

[61] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USA: USENIX Association, 2018, p. 973–990. [Online]. Available: https://doi.org/10.1145/3357033

[62] S. Fan, Z. Hua, Y. Xia, H. Chen, and B. Zang, "Isa-grid: Architecture of fine-grained privilege control for instructions and registers," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589050

[63] C. Wu, M. Xie, Z. Wang, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, M. Yang, and T. Li, "Dancing with wolves: An intra-process isolation technique with privileged hardware," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 3, pp. 1959–1978, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9760152

[64] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "Kernel protection against just-in-time code reuse," *ACM Trans. Priv. Secur.*, vol. 22, no. 1, jan 2019. [Online]. Available: https://doi.org/10.1145/3277592

[65] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, mar 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133377

[66] J. Li, X. Tong, F. Zhang, and J. Ma, "Fine-cfi: Fine-grained control-flow integrity for operating system kernels," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 6, pp. 1535–1550, 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8269390

[67] 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*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1867–1881. [Online]. Available: https://doi.org/10.1145/3319535.3354244

[68] M. Bauer, I. Grishchenko, and C. Rossow, "Typro: Forward cfi for c-style indirect function calls using type propagation," in *Proceedings of the 38th Annual Computer Security Applications Conference*, ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 346–360. [Online]. Available: https://doi.org/10.1145/3564625.3564627

[69] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys – efficient In-Process isolation for RISC-V and x86," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1677–1694. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[70] Y. Xu, C. Ye, Y. Solihin, and X. Shen, "Hardware-based domain virtualization for intra-process isolation of persistent memory objects," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 680–692. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00062

[71] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 241–254. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon

[72] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "EPK: Scalable and efficient memory protection keys," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 609–624. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/gu-jinyu

[73] Z. Yuan, S. Hong, R. Chang, Y. Zhou, W. Shen, and K. Ren, "Vdom: Fast and unlimited virtual domains on multiple architectures," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 905–919. [Online]. Available: https://doi.org/10.1145/3575693.3575735

[74] Y. Guo, Z. Wang, B. Zhong, and Q. Zeng, "Formal modeling and security analysis for intra-level privilege separation," in *Proceedings of the 38th Annual Computer Security Applications Conference*, ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 88–101. [Online]. Available: https://doi.org/10.1145/3564625.3567984

[75] K. Lu, A. Pakki, and Q. Wu, "Detecting Missing-Check bugs via semantic- and Context-Aware criticalness and constraints inferences," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1769–1786. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lu

[76] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 265–266. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[77] J.-J. Bai, T. Li, K. Lu, and S.-M. Hu, "Static detection of unsafe DMA accesses in device drivers," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1629–1645. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/bai

[78] K. Lu, "Practical program modularization with type-based dependence analysis," in *2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 1610–1624. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00092

[79] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294. [Online]. Available: https://www.usenix.org/conference/usenix-1996-annual-technical-conference/lmbench-portable-tools-performance-analysis

[80] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida, "Sok: Benchmarking flaws in systems security," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 310–325. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8806739

[81] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell, "Tetris is hard, even to approximate," in *Computing and Combinatorics*, T. Warnow and B. Zhu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 351–363. [Online]. Available: https://erikdemaine.org/papers/Tetris_TR2002/

[82] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic compartments for embedded systems," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 65–82. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/clements

[83] A. Khan, D. Xu, and D. J. Tian, "Ec: Embedded systems compartmentalization via intra-kernel isolation," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2990–3007. [Online]. Available: https://ieeexplore.ieee.org/document/10179285

[84] C. P. Ortega, "Flexc: Flexible compartmentalization through automatic policy generation," Master's thesis, Massachusetts Institute of Technology, 2022. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/144506/Ortega-cortegap-meng-eecs-2022-thesis.pdf?sequence=1

[85] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10. USA: USENIX Association, 2010, p. 1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity10/adapting-software-fault-isolation-contemporary-cpu-architectures

[86] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel, "Light-Weight contexts: An OS abstraction for safety and performance," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 49–64. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[87] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer, "Enforcing least privilege memory views for multithreaded applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 393–405. [Online]. Available: https://doi.org/10.1145/2976749.2978327

[88] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, p. 207–222, oct 2003. [Online]. Available: https://doi.org/10.1145/1165389.945466

[89] Y. Sun and T.-c. Chiueh, "Side: Isolated and efficient execution of unmodified device drivers," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12. [Online]. Available: https://doi.org/10.1109/DSN.2013.6575348

[90] J. Seo, J. You, Y. Cho, Y. Cho, D. Kwon, and Y. Paek, "Sfitag: Efficient software fault isolation with memory tagging for arm kernel extensions," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 469–480. [Online]. Available: https://doi.org/10.1145/3579856.3590341

[91] K. Dinh Duy, K. Cho, T. Noh, and H. Lee, "Capacity: Cryptographically-enforced in-process capabilities for modern arm architectures," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 874–888. [Online]. Available: https://doi.org/10.1145/3576915.3623079

[92] E. Witchel, J. Rhee, and K. Asanović, "Mondrix: Memory isolation for linux using mondriaan memory protection," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 31–44. [Online]. Available: https://doi.org/10.1145/1095810.1095814

[93] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "Codoms: Protecting software with code-centric memory domains," in *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ser. ISCA '14. IEEE Press, 2014, p. 469–480. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2678373.2665741

[94] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08. USA: USENIX Association, 2008, p. 309–322. [Online]. Available: https://www.usenix.org/conference/nsdi-08/wedge-splitting-applications-reduced-privilege-compartments

[95] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, "Clean application compartmentalization with soaap," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1016–1031. [Online]. Available: https://doi.org/10.1145/2810103.2813611

[96] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic application partitioning for intel SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 285–298. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind

[97] X. Zhou, J. Li, W. Zhang, Y. Zhou, W. Shen, and K. Ren, "Opec: operation-based security isolation for bare-metal embedded systems," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 317–333. [Online]. Available: https://doi.org/10.1145/3492321.3519573

[98] A. Khan, D. Xu, and D. J. Tian, "Low-cost privilege separation with compile time compartmentalization for embedded systems," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 3008–3025. [Online]. Available: https://doi.org/10.1109/SP46215.2023.10179388

[99] H. Lefeuvre, V.-A. Bădoiu, A. Jung, S. L. Teodorescu, S. Rauch, F. Huici, C. Raiciu, and P. Olivier, "Flexos: Towards flexible os isolation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 467–482. [Online]. Available: https://doi.org/10.1145/3503222.3507759

[100] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020, pp. 143–156. [Online]. Available: https://doi.org/10.1145/3381052.3381326

[101] V. A. Sartakov, L. Vilanova, and P. Pietzuch, "Cubicleos: a library os with software componentisation for practical isolation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 546–558. [Online]. Available: https://doi.org/10.1145/3445814.3446731