Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel

Haibo Chen, *Huawei Central Software Institute and Shanghai Jiao Tong University;* Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu, *Huawei Central Software Institute*

https://www.usenix.org/conference/osdi24/presentation/chen-haibo

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel

Haibo Chen[1,2], Xie Miao[1], Ning Jia[1], Nan Wang[1], Yu Li[1], Nian Liu[1], Yutao Liu[1], Fei Wang[1], Qiang Huang[1], Kun Li[1], Hongyang Yang[1], Hui Wang[1], Jie Yin[1], Yu Peng[1], and Fengwei Xu[1]

[1]*Huawei Central Software Institute,* [2]*Shanghai Jiao Tong University*

Abstract

The virtues of security, reliability, and extensibility have made state-of-the-art microkernels prevalent in embedded and safety-critical scenarios. However, they face performance and compatibility issues when targeting more general scenarios, such as smartphones and smart vehicles.

This paper presents the design and implementation of HongMeng kernel (*HM*), a commercialized general-purpose microkernel that preserves most of the virtues of microkernels while addressing the above challenges. For the sake of commercial practicality, we design *HM* to be compatible with the Linux API and ABI to reuse its rich applications and driver ecosystems. To make it performant despite the constraints of compatibility and being general-purpose, we re-examine the traditional microkernel wisdom, including IPC, capability-based access control, and userspace paging, and retrofit them accordingly. Specifically, we argue that per-invocation IPC is not the only concern for performance, but IPC frequency, state double bookkeeping among OS services, and capabilities that hide kernel objects contribute to significant performance degradation. We mitigate them accordingly with a set of techniques, including differentiated isolation classes, flexible composition, policy-free kernel paging, and address-token-based access control.

HM consists of a minimal core kernel and a set of least-privileged OS services, and it can run complex frameworks like AOSP and OpenHarmony. *HM* has been deployed in production on tens of millions of devices in emerging scenarios, including smart routers, smart vehicles and smartphones, typically with improved performance and security over their Linux counterparts.

1 Introduction

Microkernels minimize functionality in the kernel and move components, such as file systems and device drivers, into well-isolated and least-privileged OS services, achieving better reliability, security, and extensibility than monolithic kernels such as Linux. Thanks to these virtues, state-of-the-art (SOTA) microkernels have been widely deployed in embedded and safety-critical scenarios [30, 52, 54].

On the other hand, while monolithic kernels like Linux dominate in general-purpose scenarios such as servers and the cloud, there are increasingly emerging scenarios such as smart vehicles and smartphones that require better security, reliability, and extensibility in addition to good performance, where Linux is less suitable. While being general, Linux evolves more towards servers and the cloud, making other scenarios less beneficial. For example, it took over 10 years for the preemptive-RT patch [1] to be partially merged, and its evolution is still out of the mainstream, let alone other domain-specific strategies [20, 21]. Moreover, it has been doomed to be difficult (if possible) for Linux to satisfy high-level industry certifications required for such scenarios [98, 113].

However, although microkernels have been extensively studied for decades [16, 28, 30, 46, 49, 52, 52–54, 64, 67, 73, 75, 76, 86], SOTA microkernels mainly target some specific domains, e.g., embedded and safety-critical ones. They usually use static resource partitioning and allocation, and lack general OS functionalities to run commercial off-the-shelf applications. Below, we summarize the major challenges in retrofitting a microkernel as a general OS kernel for such emerging scenarios.

Compatibility: POSIX subset-compliant is not enough. Rebuilding the entire software ecosystem is impractical. Therefore, SOTA microkernels, such as seL4 [67] and Zircon [46], achieve minimal POSIX subset compliance by providing custom libraries, e.g., musl-libc [47], that generate inter-process calls (IPC) to OS services. However, they face deployment issues [6, 116], e.g., not being binary compatible, and implementation challenges, e.g., `fork` and `poll`, in emerging scenarios. Moreover, they can hardly reuse device drivers with affordable engineering effort and uncompromised performance, which are crucial for production deployment.

Performance: IPC is not the only concern. Performance is the top priority in emerging scenarios, directly determining user experiences. While SOTA microkernels like seL4 [67] and recent architectural support [28, 49, 86] have achieved record-high IPC performance, we observe that they still cause non-trivial performance overhead because IPC frequency is significantly increased when microkernels go general (70x higher in smartphones than routers). Further, we observe equally severe performance issues caused by state double bookkeeping due to the multi-server design, which introduces additional performance overhead (2x slower than Linux) and memory footprint (35%). Moreover, capability-based access control, which hides frequently updated kernel objects behind capabilities, can cause significant overhead due to frequent invocations. For example, it causes page fault handling to be 3.4x slower than Linux.

We started the HongMeng kernel (*HM*) project over 7 years ago to re-examine and retrofit the microkernel into a general OS kernel for emerging scenarios. To be practical for production deployment, *HM* achieves full Linux API/ABI compatibility and is capable of reusing the Linux applications and driver ecosystems such that it can run complex frameworks like AOSP [42] and OpenHarmony [35] with rich peripherals. Despite the compatibility goal that may constrain its performance, *HM* still puts performance as its primary emphasis. Therefore, *HM* respects the design principles of microkernels but not to the extreme with careful compromises. Specifically, *HM* makes the following key design decisions.

Minimal microkernel with least-privileged and well-isolated OS services. HM retains the minimality principle by keeping only the necessary functionality in the core kernel, including thread scheduler, serial/timer drivers, and access control, and leaving all other components as isolated OS services (multi-server) outside the core kernel. In addition, *HM* adopts fine-grained access control to preserve the principle of least privilege for better security. As a result, *HM* inherits the security and reliability benefits of microkernels.

Maximizing compatibility by achieving Linux API/ABI-compliant and performant driver reuse. HM integrates existing software ecosystems by achieving full Linux API/ABI compatibility through ABI-compliant shim that identifies and redirects Linux syscalls to IPCs. Moreover, *HM* reuses unmodified Linux drivers via a driver container that provides Linux runtime atop *HM* with minor engineering effort, and eliminates critical path performance degradation by separating the control plane and the data plane with *twin drivers*.

Performance first by structural supports. HM prioritizes performance without violating the architectural principles of microkernels. Specifically, *HM* achieves flexible composition for hierarchically relaxing the isolation between trusted services to minimize IPC overhead, and coalesces tightly coupled services to minimize IPC frequency and eliminate state double bookkeeping in performance-demanding scenarios, while maintaining the ability to separate them in security-critical scenarios. *HM* also supplements capabilities with performant address token-based access control, facilitating efficient cooperation like policy-free kernel paging.

We have deployed *HM* on tens of millions of devices, including smart routers, smart vehicles, and smartphones, which provides not only better security and reliability but also better performance than their Linux counterparts. The critical components of *HM* are semi-formally verified [55] by formally specifying the design and using automated verification and verification-guided testing to validate the crucial security properties, such as free of integer and buffer overflow. *HM* has been certified with ASIL-D [61] (for safety) and CC EAL 6+ [62] (for security). In routers, *HM* allows 30% more client connections by reducing 30% system memory footprint. In vehicles, *HM* achieves a 60% faster boot time and a 60% lower cross-domain latency. In smartphones, *HM* achieves 17% shorter app startup time and 10% less frame drops.

2 The Case for a General Microkernel

2.1 Microkernel Review

A major hallmark of microkernels is the minimality principle [73, 76], which minimizes functionality in the core kernel and moves other functions to userspace services. SOTA microkernels also adopt capability-based fine-grained access control [46, 52, 67, 74] to preserve the least privilege principle. As a result, microkernels are inherently more secure, reliable, and extensible than monolithic kernels [12, 79].

However, although microkernels have been extensively studied for decades [16, 30, 52–54, 64, 67, 73, 75, 76, 122], SOTA microkernels primarily target specific domains, such as embedded and safety-critical systems. Examples include L4-embedded in Qualcomm cellular modem chips [30], QNX[1] in cars and embedded systems [54], and Zircon (kernel of Fuchsia) in smart speakers [46]. There has been little study on how microkernels could be extended as general OS kernels for emerging scenarios like smart vehicles and smartphones.

The industry adopted hybrid kernels such as Windows NT [88] and Apple XNU [4], which combine a core microkernel, e.g., Mach in XNU, with all other services (as a whole) in the kernel space, e.g., Executive in NT and BSD in XNU. Although hybrid kernels also minimize functionality in the core kernel, they do not inherit many advantages of microkernels. For example, OS services in hybrid kernels are not least privileged and not well isolated. Thus, any compromised or buggy OS services can corrupt the system [88], potentially causing severe consequences, such as corrupting user data.

2.2 Demand for a General Microkernel

Emerging scenarios like smart vehicles and smartphones demand rich peripherals and applications. For example, the industry standard of vehicles has evolved to require richer OS

[1]While QNX once supported tablets/phones [14] and ran AOSP apps via virtual machine, QNX discontinued this due to limited compatibility and performance [15, 110] and has fully transitioned to embedded markets [13].

functionalities [7]. Meanwhile, emerging scenarios also emphasize security and safety. For instance, vehicles require high reliability for passenger safety, and smartphones require enhanced security to protect sensitive data. We list the major differences from domain-specific scenarios below.

Software ecosystem. In domain-specific scenarios, applications are mostly *customized and source-available*. Thus, being POSIX-compliant is believed to be sufficient for application transplanting (not even true based on our deployment experiences). However, in emerging scenarios like smartphones, apps and libraries are typically distributed in *binary form*, and frameworks require *more than POSIX compliance* [6], which mandates Linux ABI compatibility.

Resource management. In domain-specific scenarios, there are only a few pre-determined applications, and the hardware resources are limited. Therefore, applications mostly *manage resources themselves*, and the kernel is primarily responsible for reserving resources. In emerging scenarios, however, competing applications require *coordinated resource management*. The kernel requires more fledged functionalities such as efficient resource management and fair allocation.

Performance. In domain-specific scenarios, microkernels prioritize security and strict resource (e.g., timing) isolation for mostly static applications, where performance is *not a primary concern*. In emerging scenarios, however, performance is also *a top priority*, which directly determines the user experience and, thereby, the widespread deployment of the kernel.

The call for integrating both rich software ecosystems and functionalities, as well as security and reliability, makes it difficult for existing OS to satisfy them simultaneously. One approach would be customizing a stock OS such as Linux for such scenarios, which is unfortunately very expensive to evolve with upstream (section 2.3). Previous work also proposes various architectures, including unikernel [65, 81, 102], multikernel [9], exokernel [31], and splitkernel [109]. However, they primarily target server scenarios with clear resource separation while lacking support for efficient and coordinated resource management required in emerging scenarios. Moreover, the synchronization overhead and complexity introduced by split states make it challenging to achieve compatibility.

Therefore, we believe it is worthwhile to explore another avenue of evolving the microkernel into a general OS kernel.

2.3 Issues with Linux

Linux has dominated the server and cloud markets and is increasingly penetrating other domains such as PC and embedded. However, it comes at the cost of compromised security, reliability, and performance, especially in emerging scenarios.

Security and Reliability. Linux modules such as file system (FS) and device drivers cover about 80% of its 30 million line code base. They contribute to the majority of defects and vulnerabilities (90% of the total 1000 CVE [23] in the last 4 years) and significantly reduce reliability and security [19]. Additionally, about 80% of these CVEs are data leaks that can be avoided with proper isolation. Therefore, a long line of research [18,25,38,48,56,83,90–92,100,105,106,112,120,123] aims at isolating the kernel from the modules in a compartmentalized manner. However, the inherent tight coupling requires significant engineering effort and even rewriting [56, 90, 91]. Moreover, the instability of kernel module APIs and security patches force frequent upgrades, making them less practical for real-world deployments.

Generality vs. Specialization. While Linux targets general scenarios, recent patches and features witness that innovations are primarily driven by servers and the cloud, which even hamper the performance of other scenarios [89, 103]. Moreover, the growing diversity of devices with rich peripherals and varied scenarios call for specialized strategies to exploit the performance and energy efficiency headroom, such as allocating resources according to the quality of service [20, 21] or minimizing space usage [119]. However, such strategies require significant engineering effort to customize the kernel due to the inherent tight coupling of kernel modules. While there is much effort [58, 66, 78, 84, 93] to improve customizability, it is hard to integrate them into the mainstream kernel.

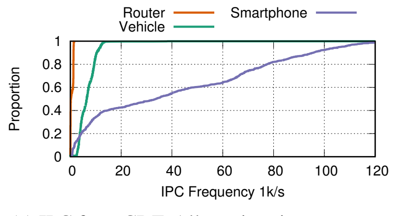
Customization vs. Evolution. Another issue is evolving the custom code. Synchronizing with upstream requires significant effort to reapply the changes, while not synchronizing may expose the system to security vulnerabilities. Years of production experience suggest that it is expensive due to the frequent changes in kernel internal APIs, and performance regressions require substantial effort to locate and even redesign the entire patches. This severely limits customizability in real-world deployments. Hence, a massive amount of products on the market are still running Linux 2.6 [50, 51, 117], which reached End-of-Life (EOL) 7 years ago [114] and has many known security vulnerabilities [24, 117].

3 Revisiting Microkernel for Going General

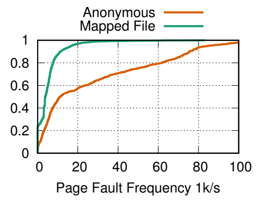
3.1 Microkernel at Scale

Deploying a microkernel in emerging scenarios poses challenges in both performance and compatibility. Figure 1 presents the observed characteristics of emerging scenarios from deploying *HM* in productions. For routers, we collected data directly from the production environment. For vehicles and phones, we replayed a typical usage (lasting 24 hours) derived from recorded massive amount of real-world executions at scale (anonymous and with user consent).

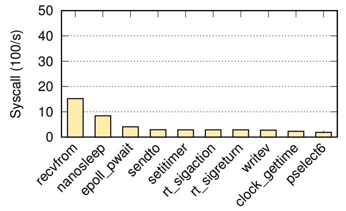
Observation 1: IPC frequency increases rapidly in emerging scenarios. Figure 1a shows the IPC frequency CDF in *HM* when configuring all OS services to be isolated in userspace. Smartphones (avg. 41k/s) and vehicles (7k/s) have a much higher IPC frequency than routers (0.6k/s, more similar to domain-specific scenarios). Figure 1b, 1e, and 1f illustrate it by showing the minor (i.e., not from disk/device)



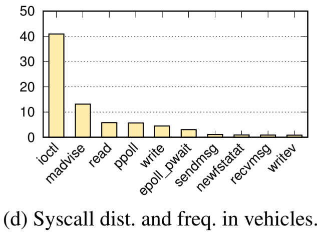
(a) IPC freq. CDF. All services in userspace.



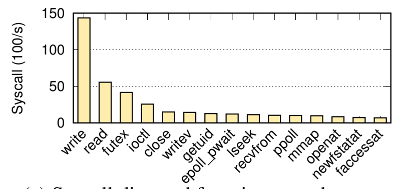
(b) Page fault freq. in phone.



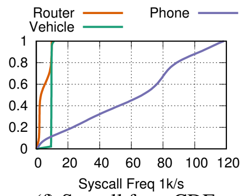
(c) Syscall dist. and freq. in routers.



(d) Syscall dist. and freq. in vehicles.



(e) Syscall dist. and freq. in smartphones.



(f) Syscall freq. CDF.

Figure 1: Characteristics of emerging scenarios obtained from the deployment of *HM* in tens of millions of devices. All OS services in *HM* are configured to be well-isolated in userspace.

page faults' frequency and the distribution and frequency of syscalls in phones. As shown in the figures, the high IPC frequency is not only caused by the higher syscall frequency (61k/s, 13x higher than routers), but also by invoking massive amounts of file operations (IPC to the FS), and triggering numerous page faults on memory-mapped files (5k/s), which requires an additional IPC roundtrip between the memory manager and the FS. Hence, we should not only optimize IPC performance but also minimize the IPC frequency.

Observation 2: Distributed multi-server causes state double bookkeeping. The minimality principle determines that there is no centralized repository for shared objects, such as the file descriptor (fd) and page caches, and distributes them in multiple places. However, as shown in Figure 1c-1e, applications in emerging scenarios frequently invoke functions like `poll` that rely on the centralized management of such states. Figure 2 further presents the CPU flame graph of application startup, which relies heavily on the performance of file mapping and is crucial to the user experience [45]. As marked in the figure, 16% of the time is spent on handling page cache misses, which introduces an additional IPC roundtrip and is 2x slower than Linux. Moreover, the double bookkeeping of page caches consumes an additional 50MB of memory on top of the 120MB base (FS+mem) in smartphones.

Observation 3: Capabilities inhibit efficient cooperation. Capabilities, which hide the kernel objects behind them, introduce significant performance overhead due to the frequent updating of some kernel objects (e.g., the page table)

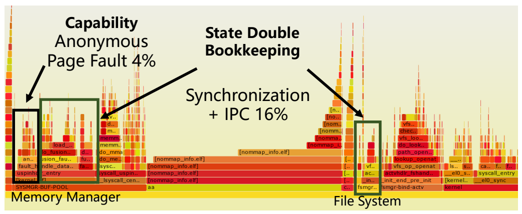


Figure 2: CPU flame graph of smartphone app startup in *HM*. Services coalescing and kernel paging are disabled.

managed outside the kernel and inhibit efficient cooperation between them. For example, this may cause the handling of anonymous page faults 3.4x slower than Linux, which frequently occurs in smartphones (avg. 27k/s, 80% of minor page faults in Figure 1b) and adds a non-trivial overhead to the app startup time (4% in Figure 2).

Observation 4: Eco-compatibility requires more than POSIX compliance. Many SOTA microkernels achieve a minimal subset of POSIX compliance by providing custom runtime libraries [47] that link directly to applications and generate IPC to OS services. However, it faces deployment issues of being not binary compatible and requiring a customized building environment. Moreover, since Linux uses file as a unified interface, which no longer exists in the microkernel, it is also challenging to implement efficient fd multiplexing like `poll` and vectored syscalls like `ioctl`, which are frequently used in emerging scenarios as shown in Figure 1c-1e.

Observation 5: Deployment in emerging scenarios requires efficient driver reuse. When deploying *HM* on smartphones, we observe a massive increase in the number of drivers required to function correctly. For routers, fewer than 20 drivers are required (primarily maintained in-house), which increases to more than 700 for vehicles and phones. Our estimates indicate that it would take more than 5,000 person-years to rewrite those drivers, and it takes time to get mature and keep evolving. Thus, reusing device drivers is a more reasonable option. However, previous work, including transplanting the runtime environment of drivers [3, 17, 32, 41, 118] and using virtual machines [72], faces compatibility, engineering effort, and performance challenges (discussed in section 5.2).

3.2 Overview of HongMeng

HM respects the core design principles of microkernels but not to the extreme, with careful compromises to address the performance and compatibility challenges in emerging scenarios. We summarize *HM*'s design decisions in Table 1 and list design principles below. Figure 3 shows *HM*'s overview.

Principle 1: Retain minimality. The security, reliability, and extensibility of microkernels derive from three fundamental architectural design principles, including separating policy and mechanism, decoupling and isolating OS services, and enforcing fine-grained access control. Hybrid kernels also enforce minimality through code decoupling but without proper isolation. Thus, it fails to inherit the major benefits of mi-

Table 1: Design decisions of HongMeng.

	SOTA Microkernels	**Hybrid Kernels**	**HongMeng's Design**
Minimality	Minimal Kernel	Code Decoupling	**Retained:** Minimal microkernel with isolated, least-privileged OS services.
IPC	IPC w/ Fastpath	Function Call	**Enhanced:** Synchronous RPC addresses resource alloc./exhaustion/acct. issues.
Isolation	Userspace Services	Coalesce w/ Kernel	**Flexibilized:** Differentiated isolation classes for tailored isolation and performance.
Composition	Static Multi-server	Static Single Server	**Flexibilized:** Flexible composition to accommodate diverse scenarios.
Access Control	Capability-based	Object Manager	**Extended:** Address tokens enable efficient kernel objects co-management.
Memory	Paging in Userspace	Paging in Kernel	**Enhanced:** Centralized management in a service with policy-free paging in kernel.
App Interfaces	POSIX-compliant	POSIX+BSD/Win	**Extended:** Linux API/ABI compatible via an ABI-compliant shim.
Device Driver	Transplanting/VM	Native Driver	**Enhanced:** Reusing Linux drivers efficiently via driver container with twin drivers.

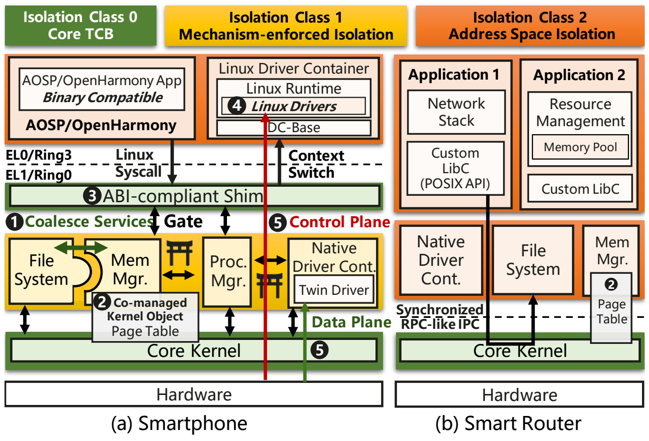


Figure 3: HongMeng overview. (a) and (b) show its composition in smartphones and routers. Different colors imply different isolation classes. ❶ coalesces coupled services. Address tokens enable kernel objects co-management ❷. ABI-compliant shim ❸ enables binary compatibility. Driver container ❹ reuses Linux drivers efficiently via data/control plane separation ❺.

crokernels. Therefore, while emphasizing compatibility and performance, *HM* respects the architectural design principles of microkernels.

HM keeps only minimal and necessary functionality in the core kernel, including thread scheduler, serial and timer drivers, and access control. All other functionality is implemented in isolated OS services, such as process/memory manager, drivers, and FS. Moreover, *HM* adopts fine-grained access control to preserve the least privilege principle. Services can only access strictly restricted resources (kernel objects) necessary for functionality. As such, *HM* inherits the security, reliability, and extensibility of microkernels.

Retained: *Minimal microkernel with well-isolated and least-privileged OS services.*

Principle 2: Prioritize performance. The promising benefits of microkernels are compromised by architecture-inherent performance issues in emerging scenarios. Therefore, instead of enforcing uniform but overly strong isolation, *HM* provides structural support for assembling the system to satisfy both the performance and the security requirements. In particular, besides adopting an RPC-like fastpath that addresses the resource allocation/exhaustion/accounting issues (section 4.1), *HM* proposes differentiated isolation classes to reduce IPC overhead by relaxing the isolation between trusted OS services (section 4.2). *HM* also coalesces tightly coupled OS services (❶ in Figure 3) to minimize the IPC frequency in performance-demanding scenarios (section 4.3). Moreover, *HM* enables efficient kernel objects co-management (❷) by supplementing capabilities with address tokens (section 4.4), which facilitates policy-free in-kernel paging of anonymous memory (section 4.5).

Flexibilized: *Prioritize performance by providing structural supports for flexible assembly to adapt to diverse scenarios.*

Principle 3: Maximizing eco-compatibility. *HM* integrates with existing software ecosystems by achieving Linux ABI compliance through a shim (❸) that redirects all Linux syscalls to appropriate OS services and serves as a central repository to store and translate Linux abstractions (e.g., fd) to efficiently support functions like `poll` (section 5.1). Moreover, *HM* reuses unmodified Linux device drivers via driver container (❹), which provides the necessary runtime derived directly from the mainline Linux with minor engineering effort (section 5.2). *HM* further improves drivers' performance by exploiting control and data plane separation (❺).

Enhanced: *Maximize compatibility by achieving Linux API/ABI-compliant and performant driver reuse.*

***HM*'s Threat Model.** *HM* retains the architectural design principles of microkernels, thus maintaining a similar threat model to SOTA microkernels, which prevents malicious applications and OS services from accessing other's memory and ensures the confidentiality, integrity, and availability (CIA) properties of data, with the following differences.

First, since applications in emerging scenarios require centralized memory management for compatibility reasons (section 4.5), the memory manager (the only exception), including its coalesced services (only FS in phones on deployment), can inevitably access applications' address spaces. Besides, in safety-critical scenarios where memory is self-managed, *HM* does not create such a centralized memory manager.

Moreover, for the sake of performance, there are compromises on additional attack surfaces (section 4.2), different partitioning of failure domains (section 4.3), and additional data leakage possibilities on carefully selected objects (will not corrupt the kernel, section 4.4). The detailed security design will be discussed in the corresponding section.

4 Performance Design of HongMeng

4.1 Synchronous RPC-like IPC Fastpath

Microkernels use IPC to invoke OS services. A long line of research has proposed numerous optimizations to minimize IPC overhead. However, when applying them to emerging scenarios, we encountered several severe issues, either previously neglected or caused by changed assumptions. *HM* carefully addresses these issues, as summarized in Table 2.

Table 2: Comparison of IPC in *HM*.

	IPC Fastpath	Migration	HongMeng IPC
Bypass Scheduling	Yes	Yes	Yes
Reduced Switches	N/A	Registers	Reg./Address Space/Priv.
Resource Allocation	Pre-alloc	Pre-alloc	Pre-bind & Adaptive
Resource Exhaustion	Blocked	Blocked	Reserved for Reclaiming
Resource Accounting	Temporal	Temporal	Temp./Energy/Memory

Synchronous RPC or Asynchronous IPC. IPC typically assumes *symmetric endpoints* with the same execution model. Therefore, previous work suggests that asynchronous IPC can avoid serialization on multicore [30], allowing both endpoints to continue execution without blocking. However, in emerging scenarios, we observe that most IPCs are procedure calls, where the caller and callee can be clearly identified. Furthermore, OS services are mostly invoked passively rather than working continuously, and most subsequent operations of the application depend on the results of the procedure call. Therefore, synchronous Remote Procedure Call (RPC) is a more appropriate abstraction for service invocations.

HM adopts an RPC-like thread migration [33, 94] as the IPC fastpath for service invocations. When sending an IPC, the core kernel performs a direct switch (bypassing scheduling, similar to prior work [10, 30, 49, 67, 70]) and switches only the stack/instruction pointer (avoids switching other registers) as well as the protection domain. Specifically, *HM* requires OS services to register a handler function as the entry point and to prepare an execution stack pool. When an application invokes a service, the core kernel records the caller's stack/instruction pointer in an invocation stack and switches to the handler function with the prepared execution stack. On return, *HM* pops an entry from the invocation stack and switches to the caller. The IPC arguments are primarily passed through registers, with additional arguments passed through shared memory.

Performance gap. Although *HM* bypasses scheduling and avoids switching registers, it still faces non-trivial performance degradation due to privilege level/address space switching and cache/TLB pollution [9, 30, 49, 86] (accounts for 50% of total IPC cost). We further bridge this performance gap using differentiated isolation classes in section 4.2.

Resource Allocation. The *memory footprint of IPC has been largely neglected* by previous work. However, due to the extremely high IPC frequency and massive number of connections (>1k threads simultaneously) in emerging scenarios like smartphones, we find it essential to consider IPC's memory footprint in production, as it can cause severe problems such as out-of-memory (OOM) and even system hangs. Although each IPC connection in *HM* requires only an individual execution stack (rather than a full-fledged thread with all related data structures), its memory footprint is still non-trivial, given the massive amount of IPCs.

Previous work pre-allocates a thread/stack pool of a fixed size and binds it to connections. However, its size is hard to decide due to the diversity and dynamism of workloads, including the number of running threads and requirements for different OS services. A large pool would quickly drain the memory, while dynamic allocation on connection introduces runtime overhead on the critical path of IPC. We initially tried to prepare and bind stacks in each OS service for each thread on creation. However, we quickly realized that the problem still exists because some services are barely used by some threads (wasted), and there exist many IPC chains (to another OS service) that need another stack.

Therefore, *HM* strikes a sweet spot by *pre-binding stacks* in frequently-used OS services (e.g., process/memory manager and FS) for each thread while still maintaining a stack pool whose size is *adjusted adaptively* at runtime. When the remaining stacks fall below a threshold, the OS service will allocate more to reduce synchronous allocation. *HM* further reduces its memory footprint by reusing the same stack when calling the same service (e.g., ABA-like call) in an IPC chain.

Resource Exhaustion. IPC *can fail due to resource exhaustion*. Specifically, when the stack pool runs out while OOM occurs, OS services cannot allocate a new stack to process the IPC request. However, apps cannot handle such an error (not existing in a monolithic kernel). Therefore, such requests are queued (blocked) in SOTA microkernels, which may cause severe issues like circular wait and even system hangs.

An intuitive approach is to send another IPC to the memory manager to reclaim some memory synchronously. However, we find that under such a scenario (already OOM), the IPC to the memory manager is likely to fail again. Such a failure is likely to occur in emerging scenarios where workloads are non-deterministic and heavy loads occur frequently (e.g., opening multiple apps simultaneously).

HM mitigates this by reserving an individual stack pool. Once OOM occurs, the kernel will synchronously IPC to the memory manager using the pool for memory reclaim (repeatedly) until the user's IPC succeeds. Thus, applications' IPCs are guaranteed to be handled correctly.

Resource Accounting. IPC *assumes a different execution entity* when handling requests, thus attributing the consumed resource to OS services. However, since competing applications in emerging scenarios require a clear accounting of resources, the consumed resource should be precisely accounted to the caller app. Previous work achieves temporal isolation by inheriting the caller's scheduling context [70, 80]. However, emerging scenarios also require an accounting of both energy and memory consumption. Therefore, *HM* records the

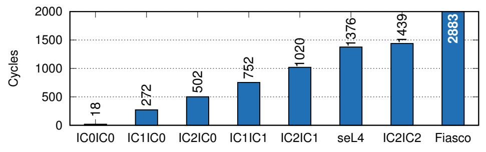


Figure 4: Round-trip IPC latency between ICx & ICy (`ICxICy`) in Raspberry Pi 4b. `IC0` includes the core kernel. `IC2` includes user apps. Zircon cannot run on Pi4b and is several times slower [49].

identity of the user app (root caller in the IPC chain), and attributes the consumed resources to it when handling IPC.

Decision: *Supplement async./sync. IPC with an RPC-like fastpath for invoking OS services while carefully addressing the resource allocation/exhaustion/accounting issues.*

4.2 Differentiated Isolation Classes

Isolation of OS Services. Placing all OS services in userspace may improve security, but it fails to meet performance requirements in emerging scenarios. We observe that *not all services require the same class of isolation*. In particular, mature, verified, and performance-critical OS services can be subjected to weaker isolation for optimal performance in practical deployments. Moreover, rapidly evolving services may frequently introduce bugs and vulnerabilities, thus requiring more robust isolation to prevent kernel corruption. OS services with large codebases and cumbersome features, such as drivers, require isolation to reduce the trusted computing base (TCB).

Therefore, *HM* adopts differentiated isolation classes (IC) to provide tailored isolation and performance for different OS services. Specifically, isolation classes *classify services and define the isolation between them*. Figure 4 shows the round-trip IPC latency between services at different isolation classes, compared to seL4 [67] and Fiasco.OC [69].

Isolation Class 0: Core TCB. IC0 applies to carefully verified, extremely performance-critical, trusted OS services, such as the ABI-compliant shim (the only IC0 service in deployment). No isolation is enforced between these services and the kernel. Therefore, IPCs are all indirect function calls.

IC0 Threat Model: IC0 is part of the core TCB, and any compromised IC0 services can arbitrarily read and modify others' memory. Therefore, placing services at IC0 should be carefully validated to avoid core kernel corruption.

Isolation Class 1: Mechanism-enforced Isolation. IC1 applies to performance-critical and validated OS services. Inspired by previous intra-kernel isolation approaches [11, 49, 59, 71, 112, 120], *HM* places these services in the kernel space and uses mechanisms to enforce isolation between services. Specifically, *HM* carefully divides the kernel address space into distinct domains and assigns each service a unique domain (IC0/core kernel also resides in a unique domain). *HM* uses ARM watchpoint [63] and Intel PKS [60] to prevent cross-domain memory access. Moreover, since IC1 services run in kernel space, they can execute privileged instructions. To prevent this, *HM* adopts binary-scanning and lightweight control-flow integrity (CFI, leveraging ARM pointer authentication (PA) [77]) to ensure services cannot execute illegal control flows that contain privileged instructions, and uses a secure monitor [49, 108] to guard the page table against code injection, which also traps any privileged instruction through *VM Exits* as a complement to CFI.

IPC between IC1 services (or to IC0) will enter a gate in the core kernel that performs a minimal context switch (switch instruction and stack pointers, w/o address space switching and scheduling) and configures the hardware to switch domains (take only a few cycles). Such a gate cannot be bypassed since domain switches require privileged instructions. Therefore, the IPC overhead is significantly reduced. As shown in Figure 4, it reduces the IPC latency between IC1 services by 50% compared to userspace services (IC2IC2).

IC1 Threat Model: IC1's threat model differs from other multi-server microkernels by assuming the correctness, soundness, and security of the applied isolation mechanism, which does expose some additional attack surfaces. For example, there are new attacks on ARM PA emerged recently [22]. Besides that, IC1 shares the same threat model, which prohibits any compromised service from reading/writing the core kernel's memory (and other OS services') and executing privileged instructions.

Isolation Class 2: Address Space Isolation. IC2 applies to non-performance-critical services or those containing third-party code (e.g., Linux drivers), enforced by address space and privilege isolation. IPC between IC2 services in *HM* (IC2IC2) is slightly slower than in seL4, mainly due to fine-grained locking, which is essential for scaling to multi-core processes under real-world loads.

IC2 Threat Model: IC2 shares exactly the same threat model as other multi-server microkernels.

Although IC1 significantly reduces the IPC overhead, it also introduces additional attack surfaces and has resource limitations (e.g., 16 domains in Intel PKS, 4 domains in ARM Watchpoint). Therefore, only performance-critical and validated OS services are placed at IC1. In addition, *HM* can quickly move all services back to IC2 if new attacks emerge. We further discuss deployment experiences on configuring isolation classes in section 4.3. Moreover, IC0/1 does not imply coupling to the kernel. The isolation classes allow for configurable isolation decisions during deployment rather than an isolation assumption during development. Different scenarios use different configurations, as shown in Figure 3.

Decision: *Not all OS services require the same class of isolation. Adopt differentiated isolation classes to relax isolation between trusted services for improved performance.*

4.3 Flexible Composition

Partitioning of OS Services. Although intuitively, OS services should be well-decoupled, e.g., FS and memory manager, we observe that *OS services are asymmetric* since some functionalities require close cooperation between specific services. For example, the FS is not the only entrance to accessing a file. POSIX supports file mapping that reads files through the memory manager, and it frequently appears on the critical path and significantly affects the user experience.

The isolation classes enforce the same isolation between same-class OS services. Therefore, without further structural support, *HM* still faces performance degradation compared to the monolithic kernel. First, the frequently invoked IPCs between tightly coupled services still cause noticeable overhead (20% in page fault handling for memory-mapped files) even in IC1 (kernel space). Moreover, double bookkeeping of shared states, such as page caches, introduces significant memory footprint and synchronization overhead. Finally, there is no global view of page caches to guide resource recycling (e.g., Least Recently Used, LRU).

To bridge the performance gap, *HM* adopts a configurable approach that allows coalescing tightly coupled OS services in performance-demanding scenarios, trading off isolation for better performance, while retaining the ability to separate them in safety-critical scenarios. When coalesced, no isolation is enforced, and IPCs between two services become function calls, while others remain as they are (well-isolated).

Coalescing also enables efficient co-management of page caches. Instead of maintaining them in both the FS and the memory manager, they can be co-managed when coalesced. It eliminates double bookkeeping and synchronization overhead and provides a global view for efficient recycling. To retain the ability to separate them, we provide a mechanism to automatically convert accesses of shared page caches into IPCs when separated. However, it will introduce non-trivial overhead. Therefore, in deployment, we implement both versions (sep./shr.) manually and enable them accordingly.

Performance. As shown in Table 3, when coalescing the FS with the memory manager, replacing the IPC reduces the latency of handling page faults caused by page cache misses by 20% (*Sep. Cache*). It can be further reduced by 30% (*Shr. Cache*) and achieves similar performance with Linux (5.10, detailed in section 6.2) by co-managing the shared page caches. Coalescing also speeds up the write throughput of *tmpfs* by 40%. Moreover, the memory footprint of coalesced services is reduced by 37% (FS+memory) in smartphones.

Security. The coalesced services are in a single failure domain, whose threat model (as a whole) remains the same as the isolation class in which it resides. Therefore, any failed or compromised service can only corrupt its coalesced services, which is also the primary compromise for performance. Hence, service coalescing should be carefully evaluated. In practice, due to the extremely high frequency of file operations in smartphones (Figure 1e), their performance targets can only be achieved by coalescing the FS with the memory manager. However, the security is still improved (isolated from other services) compared with monolithic kernels.

Deployment Experiences. Together with the differentiated isolation classes, *HM* enables flexible composition, allowing the key components to be assembled flexibly (user-space or kernel-space, separated or coalesced), enabling exploration of tradeoffs between isolation and performance according to scenarios' requirements, and the ability to scale from routers to smartphones with the same code base. The evolution of *HM* witnesses such explorations. Initially, all services were isolated at IC2. To meet the performance goal, we carefully assemble the system to retain most security properties by preserving the following rules.

Table 3: Performance improved by coalescing the FS service and the memory manager in the big core of Kirin9000 [57].

	Separated	**Coalesced**	**Linux**
Page Fault (Cycles)	7092	5290 (Sep. Cache) 3785 (Shr. Cache)	3432
Tmpfs Write (MB/s)	1492	2067	2133
Memory Footprint (MB)	190	120	N/A

Table 4: Address tokens support most operations of capabilities and allow direct access, except restricting fine-grained operation and chain revocation.

	Capabilities	**Address Tokens**
Token	CSlot id	Mapped Address
Access	Delegate to Kernel	Direct(RW)/`writev`(RO)
Ownership	Caps in CNode	Mapped Pages
Grant	Move to CNode	Map Page to VSpace
Revoke	Remove from CNode	Unmap Page
Chain Revoke	Support	No support
Security	Monitor all operations	Restrictions on mapped Obj.

First, due to the additional attack surfaces, IC1 services cannot contain any third-party code. Thus, although some drivers are also performance-critical, we kept them at IC2 and sped up via control/data plane separation (section 5.2). Second, service coalescing, especially with the memory manager, undeniably weakens isolation and security (though still improved compared with monolithic kernels). Therefore, we leave it configurable and only enable it on phones. Moreover, IC0 not only increases the core TCB but also has strict memory limitations and non-blocking requirements. Thus, in practice, *HM* only places the ABI shim (which can be opted out) in IC0. Section 6.1 details the configurations.

Decision: *OS services are asymmetric. Coalesce tightly coupled OS services and flexibly assemble the system to meet diverse requirements in various scenarios.*

4.4 Address Token-based Access Control

SOTA microkernels make all kernel objects explicit and subject to capability-based access control [30] to preserve the principle of least privileged, which is primarily implemented in a partitioned fashion that keeps a token (typically a slot ID) in userspace representing the permission to access a kernel object. However, we encountered severe performance issues when deploying it in emerging scenarios.

Clear relationship but slow access. Although capabilities are effective in describing the *external relationships* of kernel objects, i.e., the authorization chain, *accessing their internal* *contents* requires sending the token with the operations to the core kernel, which introduces non-trivial performance overhead due to privilege switches and accesses to multiple metadata tables. Kernel objects are hidden behind the capabilities and are only accessible by the core kernel. However, due to the minimality principle, the content of some kernel objects (e.g., page tables) should be frequently updated by OS services outside the core kernel, for which partitioned capabilities are no longer efficient.

Some microkernels speed up access by mapping specific objects to userspace. However, they can only be applied to few limited objects (e.g., memory objects [30, 67], part of the thread control block [3, 9, 76], and kernel interface page [76]) for security and lack the ability to synchronize data correctly, which inhibits the cooperation between the kernel and OS services. To address these issues, *HM* proposes a generalized address token-based approach that can be applied to a broader range of objects, enabling efficient co-management.

Specifically, as shown in Figure 5, each kernel object is placed on a unique physical page in *HM*. Granting a kernel object to an OS service requires mapping such a page to its address space (❶). Thus, the mapped address serves as the token to access the kernel object directly from the hardware without involving the kernel (unwillingly). Kernel objects can be granted (mapped) as read-only (RO) or read-write (RW). OS services can read RO kernel objects without kernel involvement. To update them, a new syscall, `writev`, should be used, passing the target address with the updated value, and the core kernel will verify permissions by referring to the kernel object's metadata (❷). For RW kernel objects, once granted, can be updated by OS services without kernel involvement (❸). Moreover, for objects smaller than a page with the same property (permission) and a similar life cycle, *HM* batches these objects into a single page upon allocation, allowing them to be granted and revoked collectively.

Functionality. Address tokens support most operations of capabilities, as shown in Table 4 (compared to seL4 [107], Zircon has similar functionality [37]), with two exceptions. First, address tokens cannot restrict fine-grained operations once granted, which weakens security and exposes additional attack surfaces. Besides, capabilities store the detailed relationship, allowing chain revocation, which address tokens do not support due to implicit ownership. Nevertheless, address tokens are only used by selected co-managed kernel objects. The attack surfaces are carefully mitigated (discuss below). Moreover, due to the centralized resource management, kernel objects have specific owners (will not be granted to others). Thus, chain revocation is rarely used.

Security. Once an address token is granted to an OS service, the kernel cannot monitor the subsequent operations. *HM* mitigates this by restricting the objects mapped to userspace (enforced by static analysis). Only kernel objects that exclusively contain the values of certain variables in kernel-preserved structures (pointers are not allowed to prevent the time-to-check to time-to-use attack) are mapped RW (e.g., PCache in section 4.5), ensuring they will not corrupt the kernel with incorrect or inconsistent data. The rest of the kernel's internal states (e.g., pointers and reference counters) can only be mapped RO or not granted at all to prevent it from being corrupted. *HM* further applies a sanity check when reading from RW objects. It does introduce some attack surfaces by leaking kernel-internal information, which can be mitigated by hardware encryption like ARM PA.

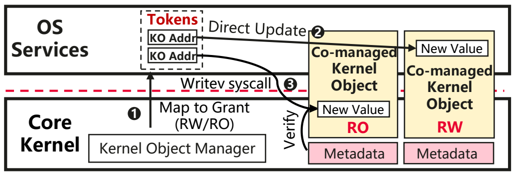


Figure 5: Address token-based access control in *HM*. ❶ Map kernel object's page to grant. ❷ Direct access to RW objects. ❸ Use `writev` to update RO objects, verified by the kernel.

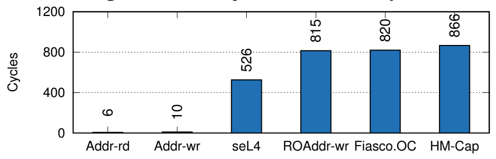


Figure 6: Latency of accessing kernel objects on Raspberry Pi 4b. *Addr-rd/wr* represent address tokens in *HM*. *ROAddr-wr* represents writing to read-only objects in *HM*.

Synchronization. There are two approaches to sharing data between OS services and the kernel leveraging address tokens. First, OS services and the kernel can exchange messages asynchronously (message-passing). For example, PCache in section 4.5 sends pre-allocated pages to the kernel for future kernel paging. *HM* uses a lock-free ring buffer to synchronize the data correctly. Besides, OS services can apply in-place updates to the objects (e.g., VSpace in section 4.5, which stores the memory layout) that the kernel may read concurrently. *HM* adopts fine-grained locking to ensure correctness. However, it may block the kernel when the service is preempted while executing critical sections. Therefore, the kernel can only use the `trylock` operation on RW-mapped objects. If it fails, *HM* will redirect to the OS services (slow path) to finish the procedure (e.g., paging in section 4.5).

Performance. Figure 6 compares the latency of accessing kernel objects after applying address tokens. The reading and writing (to RW) latencies are significantly reduced compared with capability-based approaches. However, the latency of writing RO objects is slower than seL4 on RPi4b, mainly due to the use of `AT` instruction on ARM to translate the address and check the permissions, which is slow on RPi4b (yet optimized in the advanced smartphone chips).

Usage Scenario. For security concerns (see above), address tokens are OS-internal abstractions that supplement the capabilities for efficient co-management with OS services. Specifically, besides enabling direct updates to kernel objects managed by services, it allows them to read internal states (e.g., poll list in section 5.1) without kernel involvement, sim-

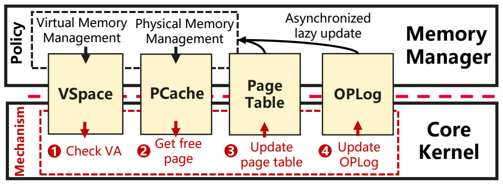


Figure 7: Policy-free kernel paging in *HM*. On page fault, the kernel checks the address ❶ and, if anonymous, ❷/❸ maps a pre-allocated page, and ❹ records an OPLog.

ilar to virtual dynamic shared objects (vDSO) in Linux [82]. It also allows handling performance-critical events (e.g., page faults in section 4.5) entirely in the kernel without violating the minimality principle by making policy-driven decisions in advance (by services) and communicating with the core kernel through co-managed objects.

Decision: *Capabilities that hide kernel objects behind interpose kernel (unwillingly) on the data plane. Supplement with address tokens for efficient co-management.*

4.5 Policy-free Kernel Paging

Centralized Management vs. Distributed Pager. Some SOTA microkernels (e.g., seL4) delegate memory management to applications with individual custom pagers. However, since competing applications in emerging scenarios require coordinated and centralized management, we found it difficult to implement certain features efficiently that require a global view of memory with decentralized pagers, such as the control group (cgroup) and memory recycling. Therefore, *HM* manages the memory through a centralized memory manager. For minimality, the memory manager is outside the core kernel, which manages the physical and virtual memory and handles page faults for all applications and OS services.

Slow userspace paging. We observe a significant performance degradation in performance-critical scenarios (e.g., app startup in Figure 2) due to the slow paging procedure of anonymous memory (e.g., stack/heap), which occurs frequently in smartphones, as shown in Figure 1b. The degradation is primarily due to the extra round-trip from the kernel to the pager. Specifically, after throwing a page fault exception, the kernel issues an IPC to the pager, which checks the address and assigns a new page, then returns to the kernel to update the page table before finally returning to the application. Such a round-trip is inevitable because page fault handling involves a policy of deciding whether and which physical page to map, which should be kept out of the kernel [27], while the exceptions are handled inside the kernel, and the page table is hidden behind a capability in the kernel.

To improve the performance of handling page faults of anonymous memory, *HM* makes policy-driven decisions in advance, and leaves a policy-free page fault handling mechanism in the core kernel. Thus, it eliminates the extra IPC round-trip on the critical path. Specifically, the memory manager provides the address range of anonymous memory along

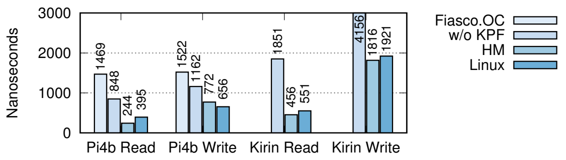


Figure 8: Page fault latency of private anonymous memory. Read is optimized with zero page. seL4 is not included since it does not support demand paging by default.

with several pre-allocated physical pages. As shown in Figure 7, if the page fault is triggered within the range (❶), the core kernel can map it directly to a pre-allocated physical page (❷ and ❸), and record an operation log (OPLog, ❹), which the memory manager will use to asynchronously update its internal states (e.g., the counter of the mapped anonymous pages). Otherwise, if the address is outside the specified range (not performance-critical) or the pre-allocated pages are exhausted, the core kernel will make an IPC to the memory manager. The involved kernel objects are co-managed by the memory manager via address tokens, including the page table, the operation log, the VSpace, which records the layout of virtual memory space for identifying anonymous memory, and the PCache, which stores the pre-allocated pages.

Compromises. By making policy-driven decisions in advance, the policies (whether/which to map) are still kept outside the core kernel. The only compromised ability is to change the policy after being pre-allocated to the PCache, which reduces flexibility. PCache also introduces some additional memory footprints. However, since PCache can be periodically replenished (off the critical path), its size remains relatively small, making these tradeoffs acceptable.

Performance. Figure 8 shows the reduced latency of kernel paging (KPF) in *HM*. *HM* reduces read/write latency by 72%/33% on Pi4b and 75%/55% on Kirin9000 (little core), making it even slightly shorter than Linux (6.1 on Pi4b and 5.10 on Kirin9000). seL4 is not included since it requires a custom pager and does not support demand paging by default. The round-trip (to the pager) of fault handling takes about 140ns on Pi4b (measured using sel4bench), which makes it significantly slower than Linux.

Decision: *Enable policy-free kernel paging by preempting policy-driven decisions.*

5 Compatibility Design of HongMeng

5.1 Linux ABI Compatibility

Deploying in emerging scenarios requires Linux ABI compatibility, which poses challenges in multi-server microkernels. K42 [68] achieves Linux ABI compatibility through trap reflection, which redirects syscalls back to the k42 library loaded into the application's address space. However, it introduces significant performance overhead due to the additional roundtrip to the kernel [2] and also faces implementation challenges [29, 111] due to keeping states in userspace. FreeBSD [36] and Windows (WSL1 [87]) also achieve partial Linux ABI compatibility through syscall emulation. However, since all their OS services reside in kernel space, the emulation layer can map abstractions like fd directly to their internal states and efficiently support functions like `fork` and `poll`, which is challenging in multi-server microkernels.

Syscall Redirection. *HM* achieves Linux ABI compatibility by placing an ABI-compliant shim in IC0 (kernel space), which redirects Linux syscalls into IPCs towards appropriate OS services (identified by syscall number, with native syscalls bypassing the shim), as illustrated in Figure 3a. In addition, the shim is optional. In scenarios where applications are predominantly custom, *HM* replaces the shim with POSIX-compliant libraries, as shown in Figure 3b.

Centralized States. Apart from binary compatibility, microkernels no longer have a central repository for global states, such as the file descriptor (fd) table, making functions like fd multiplexing (i.e., polling) and syscalls like `fork`[2] challenging to implement. Specifically, the fd table is usually kept in the application's address space (only contains credentials verified by OS services). Thus, fd multiplexing requires mapping all waiting fd to a notification primitive and sending it to all related services. Moreover, syscalls like `fork` have to correctly assemble such distributed states in the userspace. It introduces significant complexity and performance overhead, primarily due to passing states from parent to child and the additional page faults caused by updating these copy-on-write states [8, 29]. Therefore, SOTA microkernels, including seL4, Fiasco, and Zircon, do not support `fork`, while `fork` in K42 is known to have severe performance issues [29, 111].

Therefore, the ABI-compliant shim in *HM* also serves as a central repository for global states like the fd table, enabling efficient implementation of both fd multiplexing like `poll` and syscalls like `fork`. Specifically, the shim maintains the fd table, which maps fd to credentials (used by OS services to identify the user). Therefore, implementing `poll` only requires maintaining a poll list within the shim, co-managed with OS services via address tokens. It also avoids copying the fd table in userspace when executing `fork`.

Vectored Syscalls. Although most of the syscall translations are achieved solely in the ABI-compliant shim, there are vectored syscalls [116] (e.g., `ioctl/fcntl`) that extend system APIs and allow custom extensions (for drivers/modules) via the file abstraction. *HM* redirects and handles them in the FS service (e.g., invoking driver containers in section 5.2).

Deployment Experiences. *HM* passes all the tests in the AOSP compatibility and vendor test suite (CTS/VTS [43, 44]), which examines both the kernel functionalities and driver behavior. Although most binaries can run out of the box, we observe that some apps rely on unstable/undocumented Linux behavior and fail to run on *HM*. For example, an application that depends on a specific `epoll` return order [95] fails to run on *HM* (it also fails with different Linux versions).

[2]While there have been arguments that `fork` should be deprecated [8], popular frameworks like AOSP/OpenHarmony still use `fork`.

Decision: *Achieve Linux binary compatibility through ABI-compliant shim.*

5.2 Driver Container

Linux undeniably has the richest device driver ecosystem. Further, some drivers are not source-available, which makes porting challenging. Therefore, reusing Linux drivers is essential for widespread deployment.

Challenging practical and performant driver reuse. Previous work, including both transplantation [3, 17, 32, 41, 118] and VM-based methods [72], face challenges in *achieving high compatibility, reasonable engineering effort, and uncompromised performance simultaneously*. In particular, transplanting the runtime environment requires re-implementing all kernel APIs (KAPIs) used by drivers. Since some drivers use a large number of KAPIs, some of which are even constantly evolving, this approach faces challenges of compatibility and affordable engineering effort. In addition, reused drivers (with large untrusted code base) should be enforced with strict address space isolation for better security and to avoid license contamination [34], which also degrades performance. Reusing drivers through a virtual machine can achieve better compatibility with less human effort. However, it introduces issues including memory double management that causes extra memory footprint (crucial in memory-constrained scenarios like smartphones) and thread double scheduling that degrades performance due to the frequent use of asynchronous notifications in drivers.

HM reuses Linux drivers (Figure 9) through a driver container, which strikes to find a sweet spot between compatibility, engineering effort, and critical-path performance.

Compatibility. Inspired by LKL [101], UML [26], and SawMill [39], the Linux Driver Container (LDC) provides all necessary Linux KAPIs by reusing the Linux code base as a userspace runtime, allowing existing Linux drivers to run without modification. The main difference with LKL/UML/SawMill is that LDC reuses the driver rather than components like the file system and network stack. Thus, drivers should be able to access the hardware devices directly rather than redirecting to host drivers. Further, the runtime relies on *HM* for resource management. Therefore, all related functionalities, like the thread scheduler, are removed.

HM creates another device manager that manages both the Linux and the native driver containers (where the native drivers reside). Besides initializing driver containers, it registers entries (❶ in Figure 9) in the virtual file system (VFS) so that driver invocations through VFS (e.g., `ioctl` ❷) can be correctly redirected to the appropriate driver container (❸).

Using the LDC, *HM* has successfully reused over 700 device drivers from Linux, including all the needed ones for smartphones and vehicles to function correctly, such as camera, display, audio, NPU/GPU, and storage. Though most drivers can directly run out of the box, several exceptions exist. Since the LDC runs in userspace (IC2), drivers that use privileged instructions (e.g., `smc`) will trigger faults. They require binary rewrites or manual porting (for those frequently using privileged instructions, e.g., GIC) in the core kernel.

Engineering Effort. The Linux runtime in the LDC is derived directly from mainline Linux, with minor modifications to redirect several functionalities to the driver container base (DC-base in Figure 9) for proper execution. Therefore, the required engineering effort is minor. Specifically, we provide a virtual architecture and redirect the kthread/memory interfaces to *HM*. To make the drivers work correctly, DC-base creates a virtual timer and a virtual IRQ chip to provide the interrupt request and reserves a linear mapped space for functions like `virt_to_phys`. Compared to the VM-based method, which introduces double memory management and double thread scheduling, the driver container avoids these issues by redirecting and managing them in *HM*.

In practice, supporting long-term support (LTS) kernel distributions is sufficient for reusing most drivers (currently, *HM* supports 4.4, 4.19, and 5.10). In addition, since the Linux interfaces associated with the DC-base are relatively stable, only minor modifications are required to upgrade the Linux runtime. Upgrading from 4.19 to 5.10 requires less than 100 changes to the DC-base, most of which are minor modifications to the procedure names, arguments, and structures.

Critical Path Performance. The LDC is placed in IC2 (userspace) to preserve security (drivers have large untrusted code bases) and avoid license contamination. However, it introduces non-trivial overhead in driver-critical scenarios, such as app startup and camera. Therefore, *HM* applies control plane and data plane separation by creating a twin driver in the native driver container that handles I/O IRQs on the performance critical path (❹ in Figure 9). The twin driver rewrites the data handling procedure and can thus be enforced with weaker isolation (placed at IC1 in kernel space), resulting in significantly better performance. The control planes, which contain cumbersome procedures like init/suspend/resume, remain in the LDC (❺).

The twinned drivers synchronize the states (usually a variable) via IPC. Since the control plane is handled entirely in the LDC, the twin driver does not modify the states (I/O errors are redirected to the LDC). On initialization, the LDC passes device information to the native one to create the twin driver. When handling non-I/O IRQs and errors, the LDC synchronizes the updated states back to the twin driver. Unlike the transparent integration solely in LDC, which results in poor performance, the twin driver requires additional engineering effort to split and redirect interrupts and synchronize states. Therefore, the twin driver is used only for performance-critical drivers like the Universal Flash Storage (UFS) driver (others are integrated transparently w/o modification).

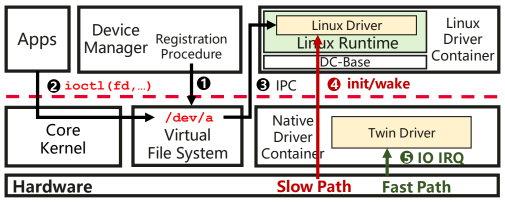
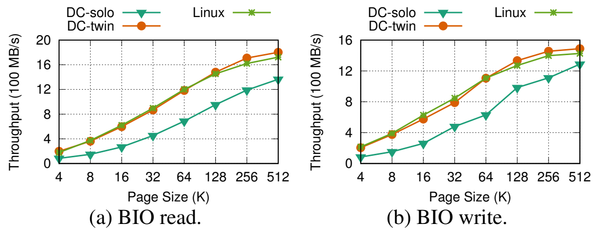


Figure 9: Drivers in *HM*. The device manager creates file nodes in the VFS ❶. VFS redirects invocations ❷ to drivers ❸. *HM* improves performance by separating the control ❹/data ❺ plane.



(a) BIO read. (b) BIO write.

Figure 10: Block I/O throughput on Kirin9000. DC-twin applies data and control plane separation, while DC-solo does not.

Figure 10 shows the improved throughput in the UFS Block I/O benchmark. In the experiment, I/O requests are issued directly from the driver. DC-twin (applied data/control plane separation) achieves a similar throughput to Linux 5.10 and outperforms DC-solo (w/o separation) by 140% at 4K size.

Security. The LDC is almost a normal userspace (IC2) OS service in *HM*, with an additional ability to create a linear mapped space whose range is strictly restricted to its allocated memory (by only setting the present bit on the allocated pages). Thus, it shares the same threat model as IC2 OS services and userspace drivers in other microkernels. In addition, *HM* uses SMMU [5] to prevent DMA attacks, with its driver residing in an isolated native driver container.

Decision: *Reuse Linux device drivers efficiently through driver containers with control/data plane separation.*

6 HongMeng in the Wild

6.1 Implementation and Deployment

The core kernel of *HM* is implemented primarily in a confined subset of C, consisting of 90k lines of code (LoC), which includes the basic functionalities. All other OS services are decoupled and can be deployed individually, totaling over 1 million LoC. The *HM*'s build system can assemble the OS services based on detailed configurations specified for various scenarios, such as placing OS services in different isolation classes or coalescing some OS services.

HM has been deployed in tens of millions of devices in various emerging scenarios, which share the same code base but with different configurations. In safety-critical scenarios, such as smart vehicles (dashboard and entertainment system) and the trusted execution environment (TEE) of smartphones, security and strict isolation are prioritized over performance. In addition, applications are mostly customized

Table 5: LMbench results.

Benchmark Commands[1]	Unit	Linux	*HM*	Norm.[2]
`lat_unix -P 1`	μs	10.23	10.39	0.98
`lat_tcp -m 16`	μs	21.22	17.19	1.23
`lat_tcp -m 16K`	μs	24.54	18.9	1.29
`lat_tcp -m 1K (Same Core)`	μs	21.21	17.19	1.23
`lat_tcp -m 1K (Cross core)`	μs	37.96	25.66	1.47
`lat_udp -m 16`	μs	17.83	19.48	0.92
`lat_udp -m 16K`	μs	23.63	22.02	1.07
`lat_udp -m 1K (Same Core)`	μs	18.04	19.55	0.92
`lat_udp -m 1K (Cross core)`	μs	34.17	26.84	1.27
`bw_tcp -m 10M`	MB/s	1812	3109	1.71
`bw_unix`	MB/s	7124	8478	1.19
`bw_mem 256m bcopy`	MB/s	17696	17202	1.02
`bw_mem 512m frd`	MB/s	14514	14593	0.99
`bw_mem 256m fcp`	MB/s	17492	15867	0.91
`bw_mem 512m fwr`	MB/s	34771	35318	1.01
`bw_file_rd 512M io_only`	MB/s	8976	9396	1.04
`bw_mmap_rd 512M mmap_only`	MB/s	26073	27520	1.05
`lat_mmap 512m`	μs	3315	3628	0.91
`lat_pagefault`	μs	0.83	0.78	1.06
`lat_ctx -s 16 8`	μs	4.53	3.41	1.32
`bw_pipe`	MB/s	3808	4127	1.08
`lat_pipe`	μs	9.00	7.88	1.14
`lat_proc exec`	μs	336	1305	0.26
`lat_proc fork`	μs	323	1280	0.25
`lat_proc shell`	μs	2269	4778	0.47
`lat_clone (create thread)`	μs	28.6	54.3	0.52

[1] Argument "`-P 1`" is omitted.
[2] *Norm.* shows the normalized performance. For throughput, use *HM*/Linux, for latency, use Linux/*HM*. The more the better.

and source-available. Therefore, *HM* places all OS services in IC2 (userspace) and exposes the POSIX API to applications through libraries. Moreover, *HM* achieves fault tolerance by introducing a driver micro-reboot in the TEE. Drivers in the TEE can be considered stateless since only re-initialization is required to recover a corrupted driver. With micro-reboot, the TEE can recover from driver corruption within hundreds of milliseconds, whereas a complete system reboot is required with a monolithic kernel. Fault tolerance for a broader range of scenarios (e.g., stateful OS services in rich-OS) requires additional efforts to store states and preserve their consistency, which we leave for future work.

In performance-demanding scenarios like smartphones, *HM* places the performance-critical OS services in IC1 (kernel space), including the process manager, memory manager, FS, and native driver container, and coalesces FS with the memory manager. The Linux driver container and other non-performance-critical OS services, such as CPU frequency governor and power manager, remain in IC2 (userspace).

6.2 Performance

We present the end-to-end performance comparison between *HM* and Linux in emerging scenarios, including smartphones (using Kirin9000 SoC [57]), smart vehicles, and smart routers, which existing microkernels fail to support. The compared Linux 5.10 counterparts are already highly optimized (used in prior products) rather than vanilla versions.

LMbench. We evaluate the basic OS functionalities using LMbench [85] on Kirin9000. Table 5 shows the results related to OS architecture. Compared to Linux (5.10), the context switching `lat_ctx` (32%) and networking (avg. 21%) are faster on *HM*, mainly due to the simplified handling procedure compared to Linux [89,96]. Memory operations perform similarly to Linux. Although `fork` still performs worse than Linux in the microbenchmark, we observe that the major overhead of `fork` in the real-world load comes from copying virtual memory areas (VMAs). It can be accelerated through parallelism, which reduces its overhead from 150ms to 60ms (in typical apps, close to 30ms in Linux). `Clone` (creating thread) is also 1x slower than Linux, mainly due to the additional IPCs between multiple OS services (especially the driver container in IC2) and the core kernel.

Geekbench. Figure 11c presents the normalized single-core results of the CPU-intensive Geekbench 5.3.2 [99]. By assembling the system to prioritize performance, *HM* achieves similar performance with Linux, with minor differences due to the different CPU frequency altering strategies.

Application Cold Startup Time. App startup time is critical to the user experience, stressing multiple OS services (e.g., reading from flash memory and creating threads) with extensive IPCs. Figure 11a shows the startup time of the top 30 AOSP apps on *HM*. The framework/app versions are the same on Linux and *HM*. As analyzed in section 3.1, the major overhead of microkernel in such scenarios comes from state double bookkeeping and slow paging, which *HM* eliminates. Therefore, the startup time is even 17% shorter (geometric mean) than Linux, mainly due to the lighter loads (see below) and the custom scheduling strategies.

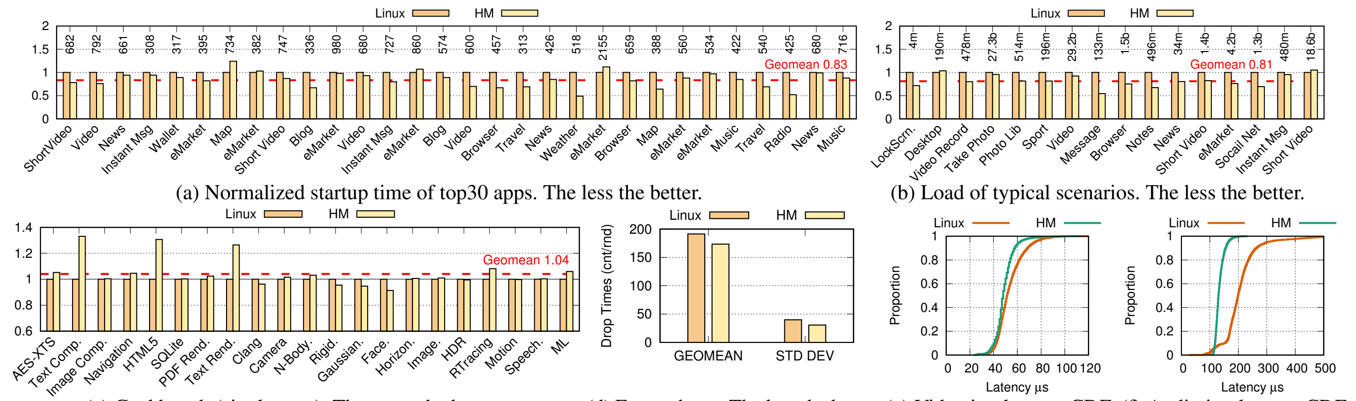
Application Loads. Figure 11b presents the loads in a period in different scenarios. The loads (number of executed instructions) are collected using `perf`, which includes the executed instructions in OS services (or in Linux kernel). The load on *HM* is 19% lighter (geometric mean) than on Linux. The proposed techniques in *HM* significantly reduce the overhead of minimality and fine-grained access control. Lighter loads also enable *HM* to achieve better performance and energy efficiency than Linux.

We further present improvements using custom strategies in *HM*, which are challenging to apply in Linux (section 2.3).

Frame Drops. Figure 11d shows the frame drop times (crucial for user experience) in 20 rounds of running the typical usage model in section 3.1. Due to the lighter load and a *custom QoS-guided scheduling* in *HM*, frame drops are 10% less and 20% stabler than Linux.

Interrupt Latencies. Figure 11e and 11f show the latency CDF of the related interrupts when playing video and audio, which are essential for user experience. *HM* reduces their latencies by 10% (video) and 65% (audio) by using a *custom experience-first strategy* that executes all the handling procedures at once, which is handled in another additional interrupt (due to lazy disable of ARM GIC [40]) in Linux.

Experiences in smart routers and smart vehicles. In smart routers, *HM* reduces OS memory footprint by 30%, allowing 30% more client connections. In smart vehicles, *HM* reduces system cold boot time from 1.5s (Linux) to 0.6s (critical for user experience, e.g., enabling 360-degree surround view) and reduces cross-domain (dashboard and entertain-



(a) Normalized startup time of top30 apps. The less the better. (b) Load of typical scenarios. The less the better.

(c) Geekbench (single core). The more the better. (d) Frame drops. The less the better. (e) Video int. latency CDF. (f) Audio int. latency CDF.

Figure 11: Performance of *HM* compared with optimized Linux 5.10 on Kirin9000. (a), (b) and (c) normalized the result for comparison. Labels in (a) show the startup time in milliseconds on *HM*. Labels in (b) show the executed instructions on *HM*.

ment) communication latency from 250μs to 100μs.

7 Lessons and Experiences

Compatible first, then nativize gradually. Compatibility is a crucial first step for commercial deployment. First, a product typically prefers a unified code base for various platforms for cost-efficiency. Second, some third-party apps/drivers are distributed in binaries. Moreover, even aiming to rebuild a new software ecosystem, many essential libraries still require Linux compatibility. Therefore, only by being compatible at first can a new OS be widely deployed and have a chance to evolve towards native interfaces for improved performance.

Specification alone is insufficient. Examine compatibility via large-scale testing. Achieving full compatibility is difficult (if possible), primarily due to Hyrum's Law [121], which reveals that all observable system behaviors will be depended on. Therefore, rather than satisfying certain specifications, we examine compatibility through large-scale testing, which is necessary to uncover hidden compatibility issues.

Deploy first, then optimize continuously. A microkernel is hard to satisfy all performance goals initially and requires full-system optimizations (e.g., framework, even hardware). Without deployment, promoting cooperation among multiple teams for such optimizations is difficult. Moreover, production deployments require time to test reliability. Therefore, deployments should commence early, starting on a small scale.

Use automated verification as much as possible. We found that complete formal verification (using interactive theorem proving) is unsustainable due to the rapid growth in code size and functionalities. Hence, we resort to semi-formal verification of critical components and use automated verification and verification-guided testing to enhance the code quality.

Amplification of hardware failures/bugs due to the scale effects. We found that some low-probability hardware faults or bugs are relatively likely to occur when deployed at scale, significantly affecting user experience and potentially becoming fatal in safety-critical scenarios. *HM* mitigates these issues by isolating critical drivers in different LDCs, restarting stateless drivers in TEE, and creating watchdogs for monitoring. *HM*, as a microkernel, also provides opportunities to address these issues through architectural design in future work.

Big kernel lock is not scalable in emerging scenarios. While it is argued that a big kernel lock is sufficiently scalable for a microkernel [97], mainly due to the short duration of most syscalls, we found that it still faces scalability issues on phones. First, phones have a high syscall frequency (61k/s, Figure 1f), causing *significant contentions*. Moreover, emerging scenarios demand some *complex functionality with long durations*. Examples include `poll`, which requires synchronizing a large number of states within the shim (IC0), and energy-aware scheduling [115], which involves frequent and costly calculations of power consumption for each scheduling decision due to the short-running nature of threads on phones.

8 Conclusion and Future Work

HongMeng is a commercialized general-purpose microkernel that retains microkernel principles while providing structural supports to address compatibility and performance challenges in emerging scenarios. It also facilitates future exploration of microkernels' benefits in production. For instance, its flexibility offers opportunities to accommodate the increasing hardware heterogeneity that Linux fails to address [104], and to achieve fault tolerance for improving availability.

Acknowledgements

We thank our shepherd, Timothy Roscoe, and the anonymous reviewers for the insightful comments. We also thank colleagues in the Huawei OS Kernel Lab and Shanghai Jiao Tong University for their helpful work and support. Haibo Chen is also partly supported by National Natural Science Foundation of China (No. 61925206 and 62132014).

References

[1] Real time Linux. https://wiki.linuxfoundation.org/realtime/start. Accessed 16 April 2024.

[2] J. Appavoo, M. Auslander, M. Butrico, D. M. da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Experience with K42, an Open-Source, Linux-Compatible, Scalable Operating-System Kernel. *IBM Syst. J.*, 44(2):427–440, jan 2005.

[3] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, R Wisniewski, and Jimi Xenidis. Utilizing Linux kernel components in K42. Technical report, Technical report, IBM Watson Research, 2002.

[4] Apple. XNU Project. https://github.com/apple-oss-distributions/xnu. Accessed 16 April 2024.

[5] ARM. System MMU Support. https://developer.arm.com/Architectures/System%20MMU%20Support. Accessed 16 April 2024.

[6] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[7] AUTOSAR. Adaptive Platform. https://www.autosar.org/standards/adaptive-platform. Accessed 16 April 2024.

[8] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A Fork() in the Road. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 14–22, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.

[10] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, page 102–113, New York, NY, USA, 1989. Association for Computing Machinery.

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. *SIGOPS Oper. Syst. Rev.*, 29(5):267–283, dec 1995.

[12] Simon Biggs, Damon Lee, and Gernot Heiser. The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[13] BlackBerry. BlackBerry QNX: Real-Time OS and Software for Embedded Systems. https://blackberry.qnx.com/en. Accessed 16 April 2024.

[14] BlackBerry. Meet the Power Behind the BlackBerry Tablet OS. https://www.qnx.com/company/announcements/blackberry_tablet_os.html. Accessed 16 April 2024.

[15] BlackBerry. BlackBerry OS End of Life. https://www.blackberry.com/us/en/support/devices/end-of-life, 2020.

[16] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1–19. USENIX Association, 2020.

[17] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[18] Anton Burtsev, Vikram Narayanan, Yongzhe Huang, Kaiming Huang, Gang Tan, and Trent Jaeger. Evolving Operating System Kernels Towards Secure Kernel-Driver Interfaces. In Malte Schwarzkopf, Andrew Baumann, and Natacha Crooks, editors, *Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS 2023, Providence, RI, USA, June 22-24, 2023*, pages 166–173. ACM, 2023.

[19] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In Haibo Chen, Zheng Zhang, Sue Moon, and Yuanyuan Zhou, editors, *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, page 5. ACM, 2011.

[20] Yonghun Choi, Seonghoon Park, and Hojung Cha. Graphics-Aware Power Governing for Mobile Devices. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, page 469–481, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Yonghun Choi, Seonghoon Park, Seunghyeok Jeon, Rhan Ha, and Hojung Cha. Optimizing Energy Consumption of Mobile Games. *IEEE Transactions on Mobile Computing*, 21(10):3744–3756, 2022.

[22] CVE. CVE-2021-30769. https://nvd.nist.gov/vuln/detail/CVE-2021-30769. Accessed 16 April 2024.

[23] CVE. CVE Records. https://cve.mitre.org/. Accessed 16 April 2024.

[24] CVEdetails. Linux Kernel 2.6 Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/version_id-410986/Linux-Linux-Kernel-2.6.html. Accessed 16 April 2024.

[25] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram 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*, ASPLOS '15, page 191–206, New York, NY, USA, 2015. Association for Computing Machinery.

[26] Jeff Dike. User-mode Linux. In *5th Annual Linux Showcase & Conference (ALS 01)*, Oakland, CA, November 2001. USENIX Association.

[27] Björn Döbel. Memory, IPC, and L4Re. https://os.inf.tu-dresden.de/~doebel/downloads/02-MemoryAndIPC.pdf, 2012.

[28] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. XPC: architectural support for secure and efficient cross process call. In Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman, editors, *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 671–684. ACM, 2019.

[29] David Edelsohn. Providing a Linux API on the Scalable K42 Kernel. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*, San Antonio, TX, June 2003. USENIX Association.

[30] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 133–150. ACM, 2013.

[31] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In Michael B. Jones, editor, *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 251–266. ACM, 1995.

[32] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, 1997.

[33] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to A Migrating Thread Model. In *USENIX Winter 1994 Technical Conference (USENIX Winter 1994 Technical Conference)*, San Francisco, CA, January 1994. USENIX Association.

[34] Free Software Foundation. Frequently Asked Questions about the GNU Licenses. https://www.gnu.org/licenses/gpl-faq.en.html#MereAggregation. Accessed 16 April 2024.

[35] OpenAtom Foundation. OpenHarmony Project. https://gitee.com/openharmony/docs/blob/master/en/OpenHarmony-Overview.md. Accessed 16 April 2024.

[36] FreeBSD. Linux emulation in FreeBSD. https://docs.freebsd.org/en/articles/linux-emulation/. Accessed 16 April 2024.

[37] Google Fuchsia. Zircon Handles. https://fuchsia.dev/fuchsia-src/concepts/kernel/handles. Accessed 16 April 2024.

[38] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of microdrivers. In Susan J. Eggers and James R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 168–178. ACM, 2008.

[39] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th Workshop on ACM SIGOPS European*

Workshop: Beyond the PC: New Challenges for the Operating System, EW 9, page 109–114, New York, NY, USA, 2000. Association for Computing Machinery.

[40] Thomas Gleixner and Ingo Molnar. Linux generic IRQ handling. `https://www.kernel.org/doc/html/v4.18/core-api/genericirq.html`, 2010.

[41] Shantanu Goel and Dan Duchamp. Linux Device Driver Emulation in Mach. In *USENIX Annual Technical Conference*, pages 65–74, 1996.

[42] Google. Android Open Source Project. `https://source.android.com/`. Accessed 16 April 2024.

[43] Google. AOSP Compatibility Test Suite. `https://source.android.com/docs/compatibility/cts`. Accessed 16 April 2024.

[44] Google. AOSP Vendor Test Suite (VTS) and infrastructure. `https://source.android.com/docs/core/tests/vts`. Accessed 16 April 2024.

[45] Google. App startup time. `https://developer.android.com/topic/performance/vitals/launch-time`. Accessed 16 April 2024.

[46] Google. Fuchsia Zircon Kernel. `https://fuchsia.dev/fuchsia-src/concepts/kernel?hl=en`. Accessed 16 April 2024.

[47] Google. Fuchsia's libc. `https://fuchsia.dev/fuchsia-src/development/languages/c-cpp/libc?hl=en`. Accessed 16 April 2024.

[48] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. Fast Intra-Kernel Isolation and Security with IskiOS. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, page 119–134, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 401–417. USENIX Association, 2020.

[50] Nikolai Hampton. The working dead: The security risks of outdated Linux kernels. `https://www2.computerworld.com.au/article/615338/working-dead-security-risk-dated-linux-kernels/`, 2017.

[51] Nikolai Hampton and Patryk Szewczyk. A survey and method for analysing soho router firmware currency. 2015.

[52] Gernot Heiser and Ben Leslie. The OKL4 microvisor: convergence point of microkernels and hypervisors. In Chandramohan A. Thekkath, Ramakrishna Kotla, and Lidong Zhou, editors, *Proceedings of the 1st ACM SIGCOMM Asia-Pacific Workshop on Systems, ApSys 2010, New Delhi, India, August 30, 2010*, pages 19–24. ACM, 2010.

[53] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A Highly Reliable, Self-Repairing Operating System. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, jul 2006.

[54] Dan Hildebrand. An Architectural Overview of QNX. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[55] Hans J Holberg and Udo Brockmeyer. ISO 26262 compliant verification of functional requirements in the model-based software development process. In *White paper. Embedded World Exhibition and Conference*, 2011.

[56] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating Device Driver Isolation. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 613–631. USENIX Association, 2022.

[57] Huawei. Kirin9000. `https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-9000`. Accessed 16 April 2024.

[58] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 588–604. ACM, 2021.

[59] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, apr 2007.

[60] R Intel. Architecture instruction set extensions and future features programming reference. https://www.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf, 2021.

[61] ISO. Road vehicles Functional safety. https://www.iso.org/obp/ui/#iso:std:iso:26262:-3:ed-1:v1:en. Accessed 16 April 2024.

[62] ISO. ISO/IEC 15408-1:2022: Information security, cybersecurity and privacy protection - Evaluation criteria for IT security. https://ccsp.alukos.com/standards/iso-iec-15408-1-2022/, 2022.

[63] Jinsoo Jang and Brent ByungHoon Kang. In-process Memory Isolation Using Hardware Watchpoint. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 32. ACM, 2019.

[64] Robert Kaiser and Stephan Wagner. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, volume 50, 2007.

[65] Antti Kantee et al. Flexible operating system internals: the design and implementation of the anykernel and rump kernels. 2012.

[66] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[67] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009.

[68] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, et al. K42: building a complete operating system. *ACM SIGOPS Operating Systems Review*, 40(4):133–145, 2006.

[69] l4re. L4Re Operating System Framework. https://l4re.org/. Accessed 16 April 2024.

[70] Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, page 93–102, New York, NY, USA, 2012. Association for Computing Machinery.

[71] Hugo Lefeuvre, Vlad-Andrei Badoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: towards flexible OS isolation. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 467–482. ACM, 2022.

[72] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *OSDI*, pages 17–30, 2004.

[73] Roy Levin, Ellis S. Cohen, William M. Corwin, Fred J. Pollack, and William A. Wulf. Policy/Mechanism Separation in HYDRA. In James C. Browne and Juan Rodriguez-Rosell, editors, *Proceedings of the Fifth Symposium on Operating System Principles, SOSP 1975, The University of Texas at Austin, Austin, Texas, USA, November 19-21, 1975*, pages 132–140. ACM, 1975.

[74] Henry M Levy. *Capability-based computer systems*. Digital Press, 2014.

[75] Jochen Liedtke. Improving IPC by Kernel Design. In Andrew P. Black and Barbara Liskov, editors, *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, pages 175–188. ACM, 1993.

[76] Jochen Liedtke. On micro-Kernel Construction. In Michael B. Jones, editor, *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 237–250. ACM, 1995.

[77] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA, August 2019. USENIX Association.

[78] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and Performance in a Filesystem Semi-Microkernel. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 819–835. ACM, 2021.

[79] Elton Lum. Study Confirms That Microkernel Is Inherently More Secure. https://blogs.blackberry.com/en/2020/09/study-confirms-that-microkernel-is-inherently-more-secure, 2020.

[80] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[81] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 461–472. ACM, 2013.

[82] Linux manual page. VDSO: Virtual Dynamic Shared Object. https://man7.org/linux/man-pages/man7/vdso.7.html. Accessed 16 April 2024.

[83] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with API integrity and multi-principal modules. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 115–128. ACM, 2011.

[84] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a Microkernel Approach to Host Networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.

[85] Larry W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996*, pages 279–294. USENIX Association, 1996.

[86] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[87] Microsoft. Windows Subsystem for Linux Documentation. https://learn.microsoft.com/en-us/windows/wsl/. Accessed 16 April 2024.

[88] Microsoft. MS Windows NT Kernel-mode User and GDI White Paper. https://learn.microsoft.com/en-us/previous-versions/cc750820(v=technet.10), 2014.

[89] Till Miemietz, Maksym Planeta, and Viktor Laurin Reusch. New Mechanism for Fast System Calls. *arXiv preprint arXiv:2112.10106*, 2021.

[90] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scotty Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 269–284. USENIX Association, 2019.

[91] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and VM functions. In Santosh Nagarakatte, Andrew Baumann, and Baris Kasikci, editors, *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, pages 157–171. ACM, 2020.

[92] Ruslan Nikolaev and Godmar Back. VirtuOS: an operating system with kernel virtualization. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 116–132. ACM, 2013.

[93] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[94] Gabriel Parmer. The case for thread migration: Predictable ipc in a customizable and reliable os. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2010)*, page 91, 2010.

[95] Roman Penyaev. epoll: make sure all elements in ready list are in FIFO order. https://patchwork.kernel.org/project/linux-fsdevel/patch/20181212110357.25656-2-rpenyaev@suse.de, 2018.

[96] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4), nov 2015.

[97] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. For a microkernel, a big lock is fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[98] Mark Pitchford. Using Linux with critical applications: Like mixing oil and water? https://www.embedded.com/using-linux-with-critical-applications-like-mixing-oil-and-water/, 2021.

[99] Primate Labs Inc. Geekbench 5 CPU Workloads. https://www.geekbench.com/doc/geekbench5-cpu-workloads.pdf, 2019.

[100] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 563–577. IEEE, 2020.

[101] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. LKL: The Linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333. IEEE, 2010.

[102] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. Unikernel Linux (UKL). In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 590–605, New York, NY, USA, 2023. Association for Computing Machinery.

[103] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An Analysis of Performance Evolution of Linux's Core Operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 554–569, New York, NY, USA, 2019. Association for Computing Machinery.

[104] Timothy Roscoe. It's Time for Operating Systems to Rediscover Hardware. USENIX Association, July 2021.

[105] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: taming device drivers. In Wolfgang Schröder-Preikschat, John Wilkes, and Rebecca Isaacs, editors, *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pages 275–288. ACM, 2009.

[106] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 73–86. ACM, 2009.

[107] seL4. seL4 capabilities. https://docs.sel4.systems/Tutorials/capabilities.html. Accessed 16 April 2024.

[108] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 335–350. ACM, 2007.

[109] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[110] Anand Lal Shimpi. The BlackBerry PlayBook Review. https://www.anandtech.com/show/4266/blackberry-playbook-review/14, 2011.

[111] Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, Amos Waterland, David Tam, and Andrew Baumann. K42: An Infrastructure for Operating System Research. *SIGOPS Oper. Syst. Rev.*, 40(2):34–42, apr 2006.

[112] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. *SIGOPS Oper. Syst. Rev.*, 37(5):207–222, oct 2003.

[113] Sysgo. Open Source and ASIL D Certification – possible? https://www.sysgo.com/blog/article/open-source-and-asil-d-certification-possible, 2018.

[114] Willy Tarreau. Linux 2.6.32.71 (EOL). https://lwn.net/Articles/679874/, 2016.

[115] The Linux Kernel documentation. Energy Aware Scheduling. https://docs.kernel.org/scheduler/sched-energy.html, 2019.

[116] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern Linux API usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[117] Johannes vom Dorp and René Helmke. Home Router Security Report 2022. https://www.fkie.fraunhofer.de/content/dam/fkie/de/documents/2022-11-28_HRSR_2022.pdf, 2022.

[118] Hannes Weisbach, Björn Döbel, and Adam Lackorzynski. Generic user-level PCI drivers. In *Proceedings of the 13th Real-Time Linux Workshop.*, 2011.

[119] Embedded Linux Wiki. Embedded Linux System Size. https://elinux.org/System_Size. Accessed 16 April 2024.

[120] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 31–44, New York, NY, USA, 2005. Association for Computing Machinery.

[121] Hyrum Wright. Hyrum's Law. https://www.hyrumslaw.com/, 2012.

[122] Fangnuo Wu, Mingkai Dong, Gequan Mo, and Haibo Chen. TreeSLS: A Whole-System Persistent Microkernel with Tree-Structured State Checkpoint on NVM. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 1–16, New York, NY, USA, 2023. Association for Computing Machinery.

[123] Feng Zhou, Jeremy Condit, Zachary R. Anderson, Ilya Bagrak, Robert Ennals, Matthew Harren, George C. Necula, and Eric A. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation OSDI'06, November 6-8, Seattle, WA, USA*, pages 45–60. USENIX Association, 2006.