PARAVIRT: Userland Containers for Mobile Systems

Isaac Ahlgren†, Yasin N. Silva†, Eric Chan-Tin†, George K. Thiruvathukal†, Kyu In Lee*
and Neil Klingensmith†,
*Department of Information Science Technology, University of Houston, Houston, Texas 77204
†Department of Computer Science, Loyola University Chicago, Chicago, Illinois 60660
Email: *klee48@central.uh.edu, †{iahlgren@, ysilva1@, chantin@cs., gkt@cs., neil@cs.}luc.edu

2025 IEEE Intelligent Mobile Computing (MobileCloud) | 979-8-3315-8917-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/MobileCloud66020.2025.00015

***Abstract*—Modern mobile operating systems like Android and iOS enhance security by isolating applications within specialized programming languages and runtime libraries. While this approach effectively secures apps, it severely restricts their portability and maintainability due to rapidly evolving and proprietary interfaces. In this paper, we introduce PARAVIRT, a novel userland containerization framework designed to securely isolate applications within a standard runtime environment without requiring root privileges. Our empirical analysis reveals that traditional userland containerization techniques, such as User Mode Linux (UML), incur significant performance penalties for I/O-intensive operations like networking and disk access due to inefficient system call handling. To overcome these limitations, PARAVIRT utilizes an optimized paravirtualization-inspired system call interface, significantly reducing overhead. Our benchmarks show that PARAVIRT improves I/O-bound workload performance by up to 2.5× compared to UML for disk write operations, achieves 4–5.9× greater network throughput, and delivers disk write speeds that closely approach those of commercial virtualization platforms like VMware Workstation. These results position PARAVIRT as a practical solution enabling cross-platform compatibility for mobile applications without sacrificing performance or security.**

***Index Terms*—Container virtualization, User-level virtualization, Container-based isolation, Mobile systems**

I. INTRODUCTION

Mobile operating systems face ongoing challenges in securely isolating untrusted applications, a critical requirement for safeguarding user data and device integrity. Popular platforms like Android and iOS address these security concerns by relying on specialized programming languages and proprietary runtime libraries, significantly constraining how developers build and maintain their applications. While this isolation approach effectively enhances security, it simultaneously impedes application portability and maintainability due to the frequent changes and non-standard nature of these interfaces. Consequently, application developers must constantly update their codebases, substantially increasing the risk of introducing bugs and vulnerabilities both in individual apps and the broader operating system.

Growing interest in live migration of applications between different types of devices, such as moving apps from mobile phones to desktop computers, highlights the urgent need for a universal and flexible runtime environment. Such environment would allow mobile applications to operate seamlessly across diverse platforms, significantly enhancing user experience and development efficiency [1], [2]. Achieving the vision of interchangeable mobile devices as originally proposed by Mark Weiser, which enables effortless transfer of applications and configurations, fundamentally requires addressing significant portability barriers [3]. Today, one major obstacle is the dependency of mobile software permissions management on proprietary, non-standard libraries, which makes cross-platform portability practically unattainable. Developers are therefore forced to rebuild applications from the ground up for each specific operating system, adding substantial overhead and complexity.

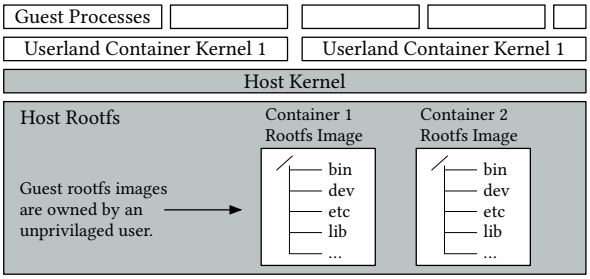

Fig. 1: Userland container architecture with privileged components in shaded gray. All guest components (kernel and rootfs) are owned by an unprivileged user.

In this paper, we present PARAVIRT, a userland containerization framework designed specifically to address these portability and security challenges. PARAVIRT isolates applications within a standardized runtime environment accessible without requiring root privileges, thereby significantly reducing security risks associated with elevated access. By translating guest system call semantics into compatible calls on the host platform, PARAVIRT provides guest applications with access to standard libraries and APIs, effectively insulating them from underlying system intricacies. A notable benefit of PARAVIRT is its capability to securely run containers launched by unprivileged users, making the solution both widely accessible and inherently safer on Linux and Android-based systems compared to traditional kernel-level containers.

Unlike conventional containerization technologies, which require elevated privileges or complex configurations, PARAVIRT's userland containers operate entirely within the user-space, inheriting only the privileges of the launching user. Host administrators retain clear and straightforward control over critical resources like the filesystem and I/O, using familiar administrative tools. Although traditional containers provide powerful privilege management tools, their inherent

979-8-3315-8917-2/25/$31.00 ©2025 IEEE
DOI 10.1109/MobileCloud66020.2025.00015

complexity frequently leads to misconfigurations and operational errors, creating unintended security vulnerabilities or facilitating container escapes [4], [5], [6], [7].

Userland containerization operates by inserting a specially modified kernel that runs entirely as a standard user-mode process between the host kernel and guest applications, as illustrated in Fig. 1. This intermediate kernel acts as a secure barrier, intercepting and emulating all system calls and signals made by guest processes, similar to how type-2 hypervisors handle privileged guest operations. Historically, the primary drawback of userland virtualization solutions has been their significant performance degradation in I/O-intensive workloads. These performance limitations have notably hindered their broader adoption despite clear security advantages. PARAVIRT directly tackles these performance concerns through optimized approaches inspired by paravirtualization, allowing userland containers to achieve performance levels comparable to traditional virtualization solutions while maintaining their robust security posture.

The primary contributions of this paper include:

- An in-depth empirical analysis comparing the performance of userland containerization against traditional virtualization technologies, clearly identifying key bottlenecks impacting I/O-intensive tasks.
- The development and evaluation of PARAVIRT, an optimized userland kernel that leverages a paravirtualization-inspired system call interface. This approach delivers significant performance improvements, achieving up to 2.5× faster disk I/O performance and 4–5.9× greater network throughput compared to User Mode Linux (UML), while maintaining disk write speeds comparable to commercial virtualization platforms such as VMware Workstation.

The remainder of this paper is structured as follows: Section II provides an empirical analysis of current userland containerization techniques through a detailed case study of UML. Section III introduces the architecture and optimization strategies implemented in PARAVIRT. Section IV offers comprehensive performance evaluations of PARAVIRT. Section V discusses implications and future research directions, and Section VI reviews relevant prior research in this field.

II. A CASE STUDY: USER MODE LINUX (UML)

User Mode Linux (UML) is a Linux kernel variant designed to run entirely in user mode. Initially developed in the late 1990s primarily as a debugging tool for kernel developers [8], UML allowed new kernel features to be tested without the need for frequent reboots—a task now predominantly handled by hypervisors. In our study, we repurpose UML as a userland virtualization platform positioned between the host kernel and guest processes. To achieve this, UML rewrites driver and kernel components typically executed with privileged mode access, redirecting their system calls to the host Linux kernel. Consequently, filesystem interactions, I/O operations, and other syscalls are effectively passed through UML's guest kernel to the host. Crucially, all virtualization requests from

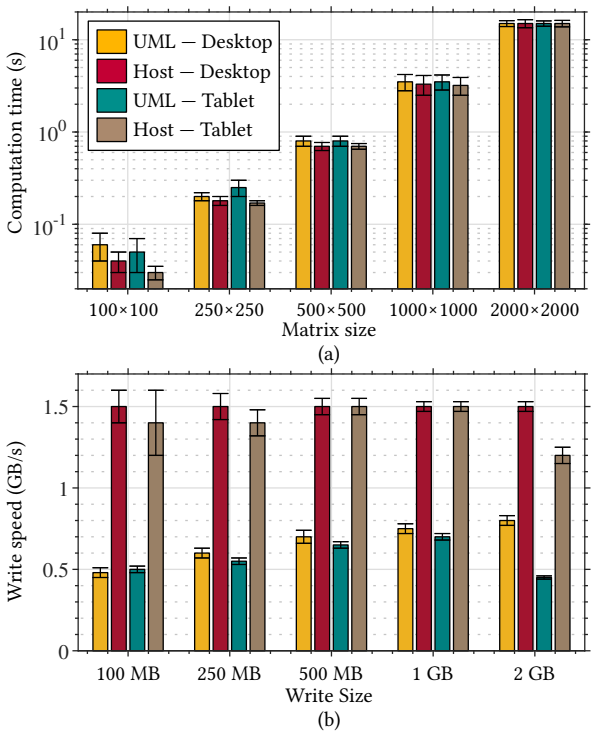

Fig. 2: Performance comparison of UML vs. host execution on (a) Matrix multiplication and (b) I/O operations.

the guest to the host occur via standard Linux system calls without privilege escalation.

When launching containers, UML uses the `fork()` syscall to create its init process, resulting in all processes within a UML container appearing as descendants of the UML kernel process from the host system's perspective. To isolate guest subprocesses effectively, UML leverages the `ptrace()` syscall, allowing one user-mode process (the userland kernel) to inspect and control another's execution. Using `ptrace()`, UML intercepts system calls, manages memory and register states, and handles signals and other low-level operations.

Every component within UML operates entirely in userland, inheriting privileges from the host's unprivileged user that initiated the container launch. Privilege escalation is unnecessary for two primary reasons. First, the container's kernel is executed as a standard process on the host, running alongside other unprivileged user processes. Second, the container's root filesystem (rootfs) image exists as a regular file on the host, owned by an unprivileged user, ensuring privileges within the container remain local to its own image. Importantly, these rootfs images are never mounted onto the host's filesystem, thus preserving clear separation of privileges.

A. Performance Study of UML Containers

To empirically assess the performance impact of user-mode containerization, we conducted benchmarks evaluating both compute-bound and I/O-bound workloads. For compute-bound tasks, we performed matrix multiplications involving randomly

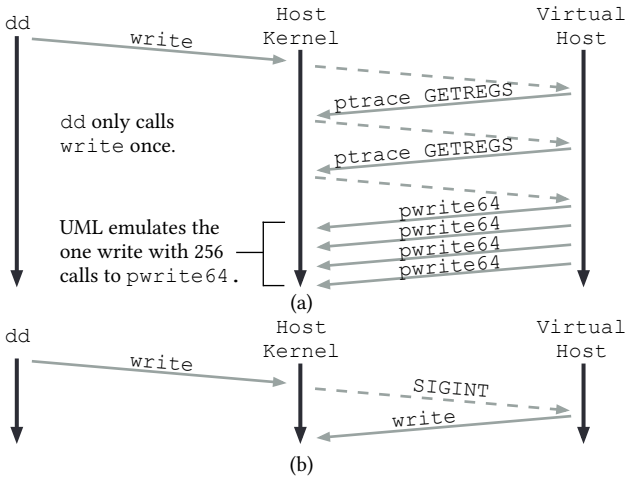

Fig. 3: Comparison of syscall handling: (a) UML's inefficient multiple-syscall approach and (b) PARAVIRT's signal-based method.

generated square matrices. For I/O-bound benchmarks, we employed the `dd` utility (copying bytes from `/dev/zero` to disk) and `iperf` (measuring network throughput). Each benchmark was executed 500 times both directly on the host system and within a UML container environment. These tests were performed on two hardware setups: a desktop (AMD Ryzen 7 2700X CPU @ 3.7 GHz with 32 GB RAM, Ubuntu 22.04) and a tablet device (Google Pixel Slate with Intel Core i5-8200Y CPU @ 1.30 GHz, 8 GB RAM, ChromeOS).

As shown in Fig. 2(a), the computational performance for matrix multiplication within UML is nearly identical to native host execution across all tested matrix dimensions (from 100×100 to 2000×2000). This result indicates negligible virtualization overhead for CPU-bound tasks within UML. This indicates that CPU-bound workloads perform efficiently within userland containers. In contrast, Fig. 2(b) highlights a substantial performance disparity for disk I/O operations. The `dd` command measured disk write throughput for varying write sizes (100 MB to 2 GB). While native host execution consistently achieved approximately 1.5 GB/s, UML containers exhibited significantly lower throughput, ranging between 0.5–0.8 GB/s. This discrepancy represents a reduction of about 60–70%. This significant I/O bottleneck persists across all write sizes and on both hardware platforms, highlighting a fundamental limitation in userland containerization that PARAVIRT aims to address.

B. Analysis of UML Performance Bottlenecks

Our benchmarking identified several critical performance bottlenecks contributing to UML's substantial I/O performance degradation:

Syscall Amplification: When guest processes issue syscalls, UML intercepts and emulates them, resulting in substantial overhead. A single guest syscall often triggers numerous additional syscalls within UML, a phenomenon we refer to as "syscall amplification." For instance, a single `write()` syscall to write 1 MB data expands into approximately 256 `pwrite64()` syscalls in UML, each accompanied by additional `ptrace()` operations, as depicted in Fig. 3(a). Table I quantifies syscall amplification further, highlighting the substantial syscall overhead incurred by UML compared to other containerization techniques. UML issued approximately 42,138 syscalls for a single 64 MB disk write operation, while Docker and native host execution required only 217 and 202 syscalls, respectively. This clearly demonstrates UML's inefficiency and its disproportionate syscall overhead.

TABLE I: Comparison of the overhead of PARAVIRT to existing containerization techniques when performing a 64 MB disk write.

PLATFORM	Number of syscalls	Overhead
UML	42,138	41,936
Docker	217	15
Host	202	-
Paravirt: Shared memory	423	221
Paravirt: Sockets	665	463

Inefficient `ptrace()` Interface: The Linux `ptrace()` syscall, while powerful, inherently limits data transfers between tracer and tracee to small 8-byte chunks, forcing UML to perform multiple, expensive ptrace() calls for large buffers, further exacerbating syscall amplification.

Lack of Kernel Support for Userland Virtualization: The Linux kernel lacks built-in support for userland virtualization. While many guest syscalls could potentially proceed without mediation after initial validation (such as read/write operations on already-opened files), the `ptrace()` interface does not allow selective tracing of syscalls. This all-or-nothing approach prevents performance optimizations that would otherwise be possible with more targeted intervention.

These performance bottlenecks identified in our UML case study serve as the foundation for our optimized userland container design, which we describe in the next section.

III. EFFICIENT USERLAND CONTAINER: PARAVIRT

Based on the performance limitations observed in UML, we designed and implemented PARAVIRT, an optimized userland virtualization approach aimed at mitigating critical performance bottlenecks, particularly syscall amplification and inefficiencies introduced by the `ptrace()` interface. Crucially, all optimizations detailed in this section are implemented entirely within userland, requiring no modifications to the host kernel, thus preserving the unprivileged and flexible nature of our solution.

Unlike UML, which relies exclusively on the `ptrace()` interface for syscall emulation, PARAVIRT adopts an explicit message-passing mechanism inspired by traditional paravirtualization techniques. Instead of emulating each low-level

instruction, this approach facilitates efficient, direct communication channels between guest processes and the host environment. Consequently, PARAVIRT significantly reduces overhead by streamlining syscall interception and handling, directly addressing the performance bottlenecks identified previously.

A. Reducing Syscall Amplification

PARAVIRT addresses syscall amplification by employing two distinct and complementary strategies: *shared memory* and *socket-based* communication. Both methods share the fundamental goal of creating efficient transmission channels for syscall arguments between guest processes and the userland kernel. These direct communication channels eliminate the multiple intermediate syscalls characteristic of `ptrace()`-based interception.

Implementing such syscall emulation requires modification of guest processes. This can be achieved either at compile-time by altering the source code or at runtime through dynamic syscall trapping and opcode replacement. Compile-time modifications involve manually changing the guest application code, substituting conventional syscalls with specialized message-passing function calls. Alternatively, runtime modifications dynamically replace syscall opcodes with function calls during execution. Although both approaches have been prototyped and validated, our detailed performance evaluations focus primarily on the compile-time modification approach due to its simplicity and clarity in demonstrating the underlying concepts.

1) Shared Memory: The shared memory method leverages a shared memory page to efficiently communicate syscall arguments between guest processes and the PARAVIRT userland kernel. This approach replaces standard syscall instructions, typically invoking `ptrace()`, with an interrupt instruction (int 3) that generates a SIGINT signal. This signal promptly notifies the PARAVIRT kernel of incoming syscall requests, dramatically reducing the number of necessary intermediate calls (illustrated in Fig. 3(b)). The primary advantage of shared memory communication is its ability to minimize redundant data copying, making it particularly efficient for syscalls involving large data transfers. However, shared memory communication introduces potential security considerations, as processes share read/write access to memory regions [9]. These risks can be effectively mitigated through rigorous data validation and error checking within the shared memory space.

2) Sockets: The socket-based communication strategy utilizes a socket interface to transfer syscall arguments from guest processes to the PARAVIRT kernel. A key implementation challenge is that sockets themselves rely on syscalls, necessitating selective syscall interception to prevent recursive trapping. Our current evaluation methodology temporarily disables `ptrace()` monitoring to assess socket-based communication performance independently, simplifying the initial experimental setup. This simplified approach helps isolate and clearly quantify the inherent performance characteristics of the socket-based mechanism. A complete, production-level solution would require sophisticated syscall filtering or kernel-level adjustments to manage recursive syscall interception effectively.

Table I provides empirical evidence of syscall reduction achieved by PARAVIRT using two strategies. Using the socket-based method, the number of syscalls required for a 64 MB disk write operation is reduced to 665, a substantial improvement compared to UML's 42,138 syscalls. The shared memory approach further enhances efficiency, requiring only 423 syscalls, clearly demonstrating the substantial effectiveness of our paravirtualization-inspired optimizations in reducing syscall amplification. Both communication strategies implemented in PARAVIRT significantly reduce syscall overhead compared to traditional UML-based containerization, resulting in enhanced performance approaching that of established virtualization solutions like VMware Workstation. In Section IV, we present comprehensive benchmarks that validate these performance improvements across various workloads.

IV. EVALUATION

In this section, we present a thorough evaluation of the performance of our optimized userland kernel, PARAVIRT, comparing its effectiveness to UML, VMware Workstation, and Docker. The evaluation specifically targets I/O-intensive tasks, as these operations typically suffer the most significant performance degradation in virtualization environments, especially within UML.

A. Network Performance Analysis

Networking operations pose particular challenges for virtualization environments due to their inherently I/O-intensive nature and the overhead associated with emulating privileged operations. We employed `iperf`, a standard network benchmarking tool, to measure TCP throughput between processes. To isolate virtualization overhead effectively, we configured iperf to run its server component on the local host and executed the client either natively or within a container environment. This local setup minimizes uncertainties introduced by external network interfaces and ensures that measured latency and throughput differences are attributable primarily to the virtualization platform itself. For UML, network connectivity was established through SLiRP, a helper program that forwards TCP packets between the UML guest kernel and the host kernel without requiring privileged access.

Table II presents our network throughput results, highlighting significant performance improvements achieved by PARAVIRT. Utilizing the shared memory paravirtualization approach, PARAVIRT achieved throughput of approximately 7,799 MB/s, comparable to native host performance (7,600 MB/s). This performance represents a substantial improvement of approximately 5.86× over UML (1,330 MB/s) and approximately 5.06× over VMware Workstation (1,540 MB/s). The socket-based paravirtualization strategy of PARAVIRT also delivered notable improvements, achieving 5,277 MB/s, outperforming UML by approximately 3.97× and VMware Workstation by approximately 3.43×. These substantial performance gains result primarily from

TABLE II: TCP throughput as measured by `iperf` between a pair of processes running on our desktop platform.

PLATFORM	Throughput (MB/s)	Standard deviation
UML	1,330	23
Docker	4,288	299
VMWare	1,540	75
Host	7,600	1,014
Paravirt: Shared memory	7,799	1,018
Paravirt: Sockets	5,277	488

PARAVIRT's optimized syscall handling, significantly reducing context switches and syscall amplification compared to UML and VMware. UML exhibited notably reduced TCP throughput compared to all other tested platforms. This performance gap is primarily attributed to the syscall amplification introduced by UML's use of SLiRP, leading to excessive syscall overhead and context switching.

B. Disk Writes

We next evaluated the disk write performance of PARAVIRT against UML, VMware Workstation, Docker, and native host execution. Our benchmark utilized the Linux utility `dd`, repeatedly performing disk write operations 500 times for various data sizes (ranging from 100 MB to 2 GB) to ensure robust and consistent results. To minimize filesystem caching effects, each test wrote data to uniquely named files, ensuring subsequent write operations bypassed host-level caching.

The benchmark results, depicted in Fig. 4, clearly demonstrate PARAVIRT's effectiveness in mitigating I/O overhead across different write sizes, including the critical 2 GB size, which we use for detailed performance comparisons. Native Linux host execution provided a baseline performance of approximately 1.4 GB/s, indicative of optimal throughput without virtualization overhead. Docker, leveraging direct kernel-level integration, nearly matched native execution with approximately 1.38 GB/s. VMware Workstation, operating as a type-II hypervisor and thus incurring hardware emulation overhead, achieved approximately 1.2 GB/s throughput, an acceptable performance considering its extensive virtualization capabilities.

UML exhibited substantially lower throughput, averaging around 0.43 GB/s due to severe syscall amplification and inefficient syscall handling. In contrast, PARAVIRT's shared memory strategy delivered throughput of approximately 1.08 GB/s, representing a 2.5× improvement over UML at the 2 GB write size. The socket-based strategy of PARAVIRT also improved significantly over UML, achieving about 0.91 GB/s, approximately a 2.1× improvement at the same write size.

The marked performance degradation observed in UML primarily arises from high syscall amplification and excessive context switching, particularly costly on modern x86 architectures. PARAVIRT successfully mitigates these issues by bypassing the inefficient `ptrace()` interface, substantially reducing the overhead associated with syscall handling. Both communication strategies employed by PARAVIRT demonstrated near-parity performance with VMware Workstation, clearly highlighting the practical advantages and efficiency of our paravirtualization-inspired optimization strategies. In summary, the evaluation results presented here clearly demonstrate PARAVIRT's capability to significantly enhance userland container performance, particularly for I/O-intensive operations, positioning it as a competitive alternative to established virtualization solutions.

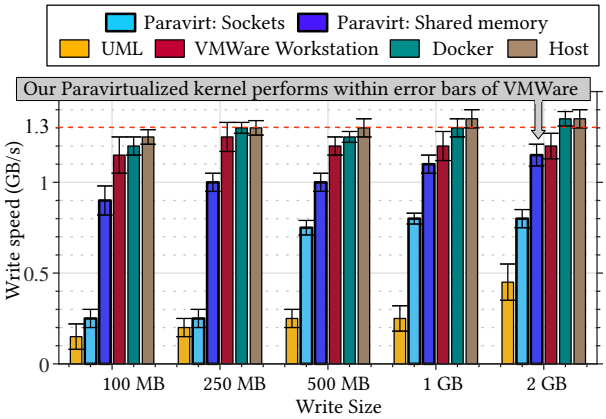

Fig. 4: Disk write throughput comparison showing PARAVIRT (outlined bars) achieving near VMware-level performance across multiple write sizes.

V. DISCUSSION & FUTURE WORK

Through our empirical evaluations, we have demonstrated that PARAVIRT significantly reduces syscall amplification and the associated overhead, achieving performance metrics that approach or even surpass widely adopted virtualization technologies like VMware Workstation.

Despite these promising results, certain aspects of our implementation currently require manual modifications to guest binaries at compile-time, which could limit ease of adoption and scalability. Moving forward, one critical area for improvement involves automating these modifications at runtime, enabling broader and simpler integration without altering the guest applications' source code. We are actively exploring techniques to dynamically intercept syscall instructions at runtime, replacing them with optimized communication channels seamlessly. Specifically, we aim to prototype methods where the userland kernel dynamically replaces syscall opcodes with interrupt instructions (such as int 3), redirecting subsequent syscalls through the efficient paravirtualized paths developed in PARAVIRT.

Security implications of userland virtualization also warrant further exploration. While the shared memory method provides optimal performance, it introduces potential security considerations due to its reliance on shared read/write memory spaces. Comprehensive threat modeling and rigorous security evaluations of the communication channels are essential to ensure robust isolation comparable to traditional virtualization methods.

Finally, the promising results demonstrated by PARAVIRT encourage investigation into broader applicability within heterogeneous device ecosystems, including scenarios involving seamless application migration between mobile and desktop environments. Evaluating PARAVIRT's performance and security characteristics in these diverse contexts will be critical to fully realizing the potential benefits of our approach.

VI. RELATED WORK

Numerous efforts have explored virtualization and containerization strategies for enhancing security and performance across platforms. TaintDroid and other work monitors sensitive data flows on Android in real-time, supporting privacy enforcement but not application portability [10], [11]. Lightweight hypervisors like Hermes improve real-time responsiveness in mobile and IoT systems through task isolation [12], [13]. Nezha enables multi-user OS-level virtualization on mobile devices [14], while VMware's Mobile Virtualization Platform and Condroid offer stronger isolation at the cost of requiring privileged system access [15], [16].

Userland-based tools such as UML and PRoot offer unprivileged virtualization by emulating kernel functionality in user space [17]. However, their reliance on `ptrace()`-based syscall handling introduces significant performance penalties for I/O-bound tasks. INSANE overlays kernel-bypass paths (XDP, DPDK, RDMA) on user-space containers, driving network-stack overhead down to the nanosecond scale while preserving portability [18]. In cloud-native contexts, systems like Falco and audit-based detection frameworks identify container anomalies and escape attempts via runtime syscall monitoring and eBPF tracing [19], [20]. These solutions enhance runtime security but are reactive and do not address userland container efficiency.

In contrast, PARAVIRT proactively eliminates syscall overhead through optimized communication channels, achieving performance near that of kernel-level virtualization while retaining the safety and accessibility of user-space execution.

VII. CONCLUSION

This paper introduced PARAVIRT, a high-performance userland container framework that addresses the severe I/O inefficiencies of conventional user-mode virtualization like UML. By emulating system calls through shared memory or socket-based channels instead of `ptrace()`, PARAVIRT dramatically reduces syscall amplification and context switching overhead in I/O-intensive environments. Our evaluation using `dd` and `iperf` confirms that PARAVIRT achieves 2.5× better disk I/O and up to 5.9× greater network throughput than UML, with disk performance closely comparable to VMware Workstation. Crucially, PARAVIRT operates entirely in userland and requires no root privileges, making it a practical and secure solution for sandboxing mobile applications in environments where traditional containers are often infeasible or unavailable.

ACKNOWLEDGEMENTS

Kyu In Lee was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. RS-2024-00463802). Isaac Ahlgren, Yasin N Silva, Eric Chan-Tin, George K Thiruvathukal and Neil Klingensmith were supported by National Centers of Academic Excellence in Cybersecurity (NCAE) H98230-22-1-0306. Klingensmith and Thiruvathukal were also supported by National Science Foundation(NSF) 2107020.

REFERENCES

[1] A. Bapat, J. Shastri, X. Wang, A. Sundarasamy, and B. Ravindran, "Dapper: A lightweight and extensible framework for live program state rewriting," in *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, 2024.

[2] I. Apple, "Use airplay to stream video or mirror the screen of your iphone or ipad," https://support.apple.com/en-us/102661, 2025.

[3] M. Weiser, "The computer for the 21st century," *ACM Mobile Computing and Communications Review (MC2R)*, 1999.

[4] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, "Houdini's escape: Breaking the resource rein of linux control groups," in *Proceedings of the 2019 ACM Conference on Computer and Communications Security (CCS)*, 2019.

[5] National Institute of Standards and Technology, "CVE-2015-3629," https://nvd.nist.gov/vuln/detail/CVE-2015-3629, 2015.

[6] ——, "CVE-2015-3629," https://nvd.nist.gov/vuln/detail/CVE-2015-3629, 2015.

[7] ——, "CVE-2015-3631," https://nvd.nist.gov/vuln/detail/CVE-2015-3631, 2015.

[8] J. Dike, "User-mode linux," in *Proceedings of the 5th Annual Linux Showcase & Conference (ALS)*, 2001.

[9] National Institute of Standards and Technology, "CVE-2016-5195," https://nvd.nist.gov/vuln/detail/CVE-2016-5195, 2016.

[10] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, 2014.

[11] A. Razeen, V. Pistol, A. Meijer, and L. P. Cox, "Better performance through thread-local emulation," in *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2016.

[12] N. Klingensmith and S. Banerjee, "Hermes: A real time hypervisor for mobile and IoT systems," in *Proceedings of the 19th International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2018.

[13] ——, "Using virtualized task isolation to improve responsiveness in mobile and IoT software," in *Proceedings of the International Conference on Internet of Things Design and Implementation (IoTDI)*, 2019.

[14] B. Yang, S. Yan, S. Liu, Z. Long, J. Yu, H. Zhang, Y. Yao, R. Xu, F. Feng, and J. Wu, "Nezha: Mobile os virtualization framework for multiple clients on single computing platform," in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2019.

[15] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis, "The vmware mobile virtualization platform: is that a hypervisor in your pocket?" *ACM SIGOPS Operating Systems Review*, vol. 44, 2010.

[16] L. Xu, G. Li, C. Li, W. Sun, W. Chen, and Z. Wang, "Condroid: A container-based virtualization solution adapted for android devices," in *Proceedings of the 2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MOBILECLOUD)*, 2015.

[17] "Proot," https://proot-me.github.io.

[18] L. Rosa, A. Garbugli, A. Corradi, and P. Bellavista, "Insane: A unified middleware for qos-aware network acceleration in edge cloud computing," in *Proceedings of the 24th International Middleware Conference (MIDDLEWARE)*, 2023.

[19] F. D. Pierro, "Falco 0.32.0," https://falco.org/blog/falco-0-32-0/.

[20] N. R. Ivánkó, "Detecting a container escape with cilium and ebpf," https://isovalent.com/blog/post/2021-11-container-escape/.