Measuring and Optimizing the Performance of the Android Virtualization Framework

Hao-Jung Wei
National Taiwan University
r12922072@ntu.edu.tw

Leng-Kai Lin
National Taiwan University
r12922160@ntu.edu.tw

Chun-Yen Lin
National Taiwan University
r11922153@csie.ntu.edu.tw

Shih-Wei Li
National Taiwan University
shihwei@csie.ntu.edu.tw

ABSTRACT

Google recently released the Android Virtualization Framework (AVF) to support confidential computing for mobile devices. This work evaluated the performance of the Android AVF for the first time on Google Pixel devices. Our results show that pVMs exhibited significant performance overhead in I/O intensive workloads. We identified the overhead that stems from device drivers in the protected VMs performing bounce buffering. We thus extended Linux to optimize memory allocation and reduce lock contention to effectively improve the performance of pVMs by 20% in the best case compared to the baseline.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; • **General and reference → Measurement**; **Performance**.

KEYWORDS

Mobile Security, Measurement Study, Virtualization, Performance

ACM Reference Format:
Hao-Jung Wei, Leng-Kai Lin, Chun-Yen Lin, and Shih-Wei Li. 2024. Measuring and Optimizing the Performance of the Android Virtualization Framework. In *Proceedings of ACM SAC Conference (SAC'24).* ACM, New York, NY, USA, Article 4, 3 pages. https://doi.org/10.1145/3605098.3636097

1 INTRODUCTION

Google recently released the AVF [12] in Android 13. AVF incorporates a secure hypervisor to support virtual machine-based trusted execution environments (TEEs). The virtual machine-based TEEs, protected virtual machines (pVMs), are isolated from each other and the Android OS kernel. The AVF allows entire guest systems running legacy systems like Android to execute their code and computations in isolated pVMs. to run secure compilation, confidential machine learning, and key management services. As shown in Figure 1, the AVF builds on the Android Linux kernel and KVM to support virtualization. In contrast to KVM, the AVF incorporates a small TCB, the pKVM hypervisor, to protect pVMs against the untrusted host Android Linux OS. pKVM ensures that an attacker who controls the host OS cannot compromise the confidentiality and integrity of pVMs.

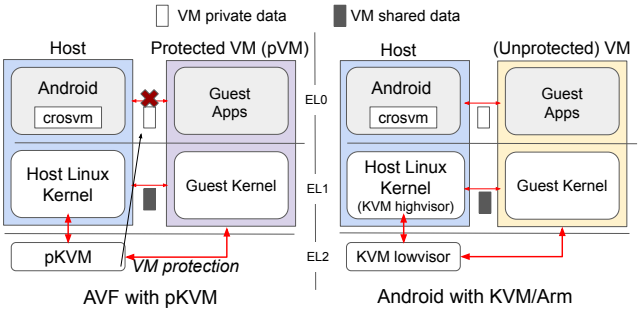

Figure 1: Android AVF Architecture and KVM/Arm

This work measured the cost of micro and application benchmarks running in pVMs on Google Pixel devices for the first time. Our results show that pVMs could suffer significant performance overhead. Most typically, we observed that pVMs suffered a considerable slowdown in I/O intensive workloads. Compared to regular unprotected VMs, block I/O performance is 25% worse and network performance of pVMs is 75% worse. We identified pKVM's memory protection mechanisms and bounce buffering cost for pVM's virtio use cases resulting in the performance overhead in pVMs. We optimized the SWIOTLB subsystem in the Linux kernel to accelerate bounce buffer allocation and reduce lock contention. Our evaluation shows that the SWIOTLB optimization improves the VM performance by over 20% in the best case compared to the baseline.

Android Virtualization Framework. Figure 1 shows the architecture of the AVF. The AVF leverages Arm VE features to run the pKVM hypervisor in Hyp mode (EL2). pKVM deprivileges the host Linux and Android frameworks respectively to EL1 and EL0. The AVF integrates crosvm to provide VM management and virtual devices. pKVM protects the integrity and confidentiality of pVM's CPU and memory against the host Linux kernel, Android OS, or any other pVMs while relying on the host kernel to provide resource allocation, scheduling, and I/Os. As shown in Figure 1, VM exits are routed to pKVM in EL2. If an exit requires the host kernel's functionality, pKVM context switches the VM's CPU states to the host. pKVM manages Arm stage 2 page tables for each pVM and the host kernel to enforce memory access control. pKVM unmaps

SAC'24, April 8 –April 12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04
https://doi.org/10.1145/3605098.3636097

pVMs' private memory from the host's stage 2 page tables. PKVM tracks the owner and sharing states of each physical page and stores the states in the software bits of page table entries (PTEs). For the page that a given PTE maps to, the state defines the access type that an entity that uses the PTE has for that page. pKVM exposes hypercalls to pVMs to share and unshare pages with the host; and to the host to transfer page ownership.

pKVM supports virtio and relies on crosvm to implement virtio block, network, and serial devices. The mainline KVM supports *zero-copying* virtio. In AVF, the virtio back-end in the host Linux has no access to VM memory. It relies on pVMs to employ bounce buffering, restricting the virtio devices to access buffers allocated from a shared pool. The driver in pVM leverages Linux's SWIOTLB to manage the shared pool to allocate and deallocate bounce buffers. The virtio-backend accesses bounce buffers to get the pVM's output I/O payloads or provide input payloads to the pVM.

2 PERFORMANCE EVALUATION

We measure the performance of protected VMs running on the AVF with pKVM enabled and unprotected VMs (or KVM VMs) running on KVM/Arm without protection. We measured the performance of hypervisor and VM micro-level operations, application workloads, and multi-vm performance of unprotected and KVM VMs.

Experimental Setup. The experiments were conducted on a Google Pixel 6 Pro and Google Pixel 7 Pro device with 8 cores and 12 GB RAM. For both devices, we ran the stock Android kernel android-13.0.0_r0.59 raviole kernel. Each VM instance was spawned with 2 virtual CPUs (vCPUs) and 4GB RAM for hypervisor microbenchmarks. Each VM instance was spawned with 8 vCPUs and 4GB RAM for application benchmarks.All VMs used virtio block devices with `O_DIRECT` option to disable caching. All VMs used standard virtio network configuration over the host's TUN/TAP interface. We ran the client in the VM and the server on the host for network-related benchmarks. Each VM ran Ubuntu 20.04. For all VMs, we used the same guest Linux kernel source as Microdroid from the branch `pkvm` of the Android-KVM kernel repository using the default kernel config. For all cases, we used crosvm android-13.0.0_r30. We configured crosvm to disable loading a pVM firmware.

Microbenchmarks. We first measured the performance of low-level hypervisor operations on the AVF. We ported KVM unit test into the guest Linux kernel driver and ran it in protected and unprotected VMs on the AVF. Table 1 shows the results in CPU cycles.

Microbenchmark	Pixel 6 Pro		Pixel 7 Pro	
	VM	pVM	VM	pVM
Hypercall	1,497	1,641	1,443	1,680
I/O Trap Kernel	1,763	2,099	1,736	2,003
I/O Trap User	4,465	5,386	4,846	5,722
Virtual IPI	3,844	3,967	3,842	3,935
Share Hypercall	NA	2,307	NA	70,688
Unshare Hypercall	NA	3,678	NA	72,119

Table 1: Microbenchmark Measurements (cycles)

pVMs perform worse than the unprotected VMs in NULL hypercall, I/O Trap Kernel and I/O Trap User due to stage 2 translation and the extra CPU context switching efforts in pKVM. I/O Trap User resulted in higher overhead because it requires a context switch to crosvm. An IPI send includes two VM exits due to IPI send and receive to the KVM host. Consequently, the IPI cost is roughly 2X of the I/O Kernel. Share/Unshare Hypercall causes much higher overhead than the null Hypercall. To handle these hypercalls, pKVM traverses the caller pVM's and host's stage 2 page tables to look for the PTE that maps the shared page. pKVM then updates the page state in the PTEs to specify memory sharing/unsharing. On Pixel 7, when handling the hypercall, pKVM performs extra management of the stage 2 memory protection unit (S2MPU), which protects an I/O device's memory access, which resulted in a much higher overhead than Pixel 6.

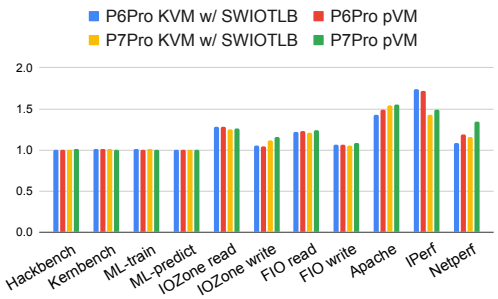

Figure 2: Results of Application Benchmarks

Application Benchmarks. Figure 2 shows the overhead of application benchmarks running on pVMs. The results are normalized against the performance of the unprotected KVM VM using zero-copying virtio to show pVM's overhead compared to the most optimized virtio use cases. We also show the results of KVM VMs using SWIOTLB in Figure 2 for comparison. 1 means the benchmark performs on par with KVM VMs using zero-copying virtio. pVMs show low overhead in hackbench [5] and kernbench [6], and ML predict [14]. These workloads are CPU and memory-intensive and do not require much hypervisor intervention. In contrast, pKVM performs significantly worse than KVM/Arm for I/O-related workloads. Using SWIOTLB resulted in much poorer performance in I/O workloads [1–4, 13] due to bounce buffering.

3 THE PVM OPTIMIZATION

SWIOTLB allocates and deallocates from the buffer pool for each I/O request at runtime. The buffer allocation and deallocation could take O(n), which n is the size of the memory pool. Further, SWIOTLB uses a single spinlock to protect concurrent metadata access. Frequent access to the metadata could lead to lock contention and incurs performance slowdown. We extended Linux's SWIOTLB subsystem to support efficient I/O by (1) optimizing bounce buffer allocation, and (2) reducing lock contention. We denote the optimized SWIOTLB subsystem as ***SWIOTLB-opt***.

We layered a bounce buffer cache above the existing SWIOTLB memory pool. Each bounce buffer cache consists of free lists, which each include several pre-allocated bounce buffers of the same size. Our current implementation adds a total of 13 free lists. Each of the lists includes buffers with a size equal to 2 to the `n` order of bytes, with `n` ranges between 0 to 12. We use ring buffer metadata for each free list to track the allocation status in a constant time.

We ported the patch for multi-lock [15] from the v6.0-rc1 kernel to the v5.15-based pKVM Linux. We split each per-CPU SWIOTLB area into multiple free lists with different sizes of SWIOTLB buffer

chunks. We associate each pairing of (area, buffer size) with a unique spinlock. The locking scheme includes per-CPU and per-buffer locks. Each CPU first allocates buffers from its free lists. If the buffer cache can satisfy the requested size, it searches through the free lists for free buffers. If its free list is empty, the CPU then tries to allocate buffers from the free lists of other CPUs. If the free lists of all CPUs are exhausted, the CPU then allocates buffers from the shared bounce buffer pool. When a bounce buffer is freed, SWIOTLB-opt adds it back to either the free lists or the memory pool, depending on where it was obtained.

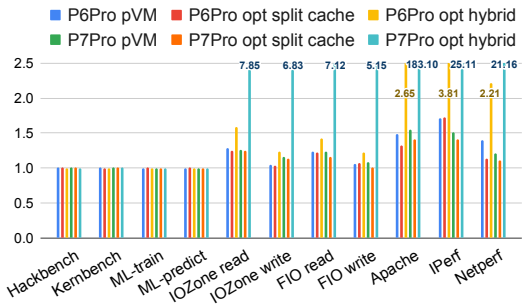

Figure 3: Results of pVM Optimization

Figure 3 presents the performance results of : **opt split-cache**: a pVM running a guest Linux with per-CPU SWIOTLB bounce buffer cache and pool with SWIOTLB-opt; and ***opt hybrid***: a pVM running a guest Linux with SWIOTLB-opt and runtime page sharing via the share and unshare hypercalls for page-aligned data buffers to support zero-copying. We normalized the results to unprotected VMs with virtio as the results presented in Figure 2.

With the SWIOTLB optimization, network benchmarks on pVMs performed between 8% to 22% better than unmodified SWIOTLB, and block I/O benchmarks could perform 7% better in the best case than unmodified SWIOTLB (on FIO on Pixel 7 Pro). The improvement was more significant in benchmarks that used smaller data buffers, such as Apache and Netperf. Block I/O-related workloads, including IOZone and FIO, benefit less because they use larger data buffers that cause SWIOTLB cache misses, so the allocations fall back to SWIOTLB's default allocator. Enabling zero-copying I/O resulted in a significant performance slowdown. We identified that frequent invocations of the share/unshare hypercalls shadowed the performance gain from the SWIOTLB optimization. As shown earlier, the share and unshare hypercalls exhibited high overhead. For each page-aligned data buffer, pVMs make a pair of share/unshare hypercalls to grant the untrusted host kernel access to the buffer, then revoke its access when the transaction completes. For ***opt hybrid***, applications running on Pixel 7 Pro performed much worse due to its higher share/unshare hypercall costs .

4 RELATED WORK

Earlier Arm-based confidential VMs [8–11] supported virtio through either bounce buffering or zero-copying. Our optimizations and evaluations are beneficial to these VMs. Bifrost [7] enhances the network I/O performance of protected VMs on AMD SEV and Intel TDX. Bifrost assumes VMs employ an end-to-end approach. It modifies Linux's TLS support to eliminate redundant buffer bouncing. Bitfrost also optimizes the virtio backends to utilize residual CPU resources for batching packet processing. Our proposed optimization improves the SWIOTLB bounce buffering subsystem used by general I/O workloads and are not limited to TLS-enabled network workloads. Further, our SWIOTLB optimization applies to Bifrost for performance enhancement of the x86-based protected VMs.

5 CONCLUSION AND FUTURE WORK

This work measured the performance of the Android AVF on Google Pixel devices. We uncovered a significant slowdown in pVMs running I/O intensive workloads and optimized Linux's SWIOTLB to improve performance. To further improve performance, pKVM can be extended to avoid page table walks during the share and unshare hypercalls. pKVM could allocate external metadata to track ownership and sharing status for each physical page. pVMs could potentially leverage zero-copying to optimize I/O performance. Equipping mobile devices with device passthrough capabilities, such as SRIOV and direct interrupt delivery could improve I/O performance. Finally, reducing pVM's exit cost could optimize performance. Google has recently attempted to extend pKVM for supporting VHE in the AVF [16] to reduce the context switching overhead.

ACKNOWLEDGEMENT

This research was supported in part by the National Science and Technology Council of Taiwan under research grants 111-2628-E-002-022- and 112-2628-E-002-027-, and MediaTek Inc. research grant 111HT912004.

REFERENCES

[1] 2023. *Apache benchmarking tools.* https://httpd.apache.org/docs/2.4/programs/ab.html
[2] Jens Axboe. 2006. *FIO.* https://fio.readthedocs.io/en/latest/fio_doc.html
[3] Dugan et al. 2014. *Iperf3.* https://iperf.fr/
[4] HP Networking Performance Team et al. 2015. *Netperf.* https://hewlettpackard.github.io/netperf/
[5] Russell et al. 2008. *Hackbench.* https://manpages.ubuntu.com/manpages/bionic/man8/hackbench.8.html
[6] Con Kolivas. 2009. *Kernbench.* https://github.com/linux-test-project/ltp/tree/master/utils/benchmark/kernbench-0.42
[7] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. 2023. Bifrost: Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines. In *2023 USENIX Annual Technical Conference (USENIX ATC 23).*
[8] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. 2021. TwinVisor: Hardware-Isolated Confidential Virtual Machines for ARM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21).* 17 pages.
[9] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP 2021).*
[10] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *30th USENIX Security Symposium (USENIX Security 21).* 3953–3970.
[11] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and Verification of the Arm Confidential Compute Architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22).*
[12] Google LLC. 2022. Android Virtualization Framework (AVF) overview. (Oct 2022).
[13] William D. Norcott and gDon Capps. 1991. *IOZone.* https://www.iozone.org/
[14] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.
[15] Tianyu Lan. 2022. swiotlb: split up the global swiotlb lock. http://git.infradead.org/users/hch/dma-mapping.git/commit/20347fca71a387a3751f7bb270062616ddc5317a.
[16] Marc Zyngier. 2022. Now You See Me, Now You Don't: Splitting pKVM Into Discrete, Mutually Exclusive Address Spaces. *KVM Forum* (2022).