

You Shall Not Pass into Ring-0! A User Privacy-Friendly Anti-Cheat Architecture for Personal Computers

Santosh Gokul Narayanan*

Nokia of America Corporation
Sunnyvale, USA
santosh.narayanan@nokia.com

Giovanni Paladino*

New York University
New York, USA
gpp9131@nyu.edu

Chuqi Zhang

National University of Singapore
Singapore, Singapore
chuqiz@u.nus.edu

Sangho Lee

Microsoft Research
Redmond, USA
sangho.lee@microsoft.com

Zhenkai Liang

National University of Singapore
Singapore, Singapore
liangzk@nus.edu.sg

Adil Ahmad[†]

Arizona State University
Tempe, USA
adil.ahmad@asu.edu

Abstract

Kernel-level anti-cheats are effective against malicious player behavior in competitive video games, but raise significant user privacy concerns regarding installing unverifiable components at privileged modes (i.e., ring-0 in x86). While existing research has focused on improving the effectiveness of anti-cheats, the user privacy concern has been largely ignored. TIRITH is an anti-cheat architecture that addresses this problem using two key ideas. First, instead of running video games within regular processes that players (as root admins) have control over, TIRITH executes video games in Protected Virtual Machines that naturally sandbox computations from untrusted admins. Second, to monitor user behavior outside the sandbox (e.g., see if they are running malicious drivers), TIRITH leverages a virtualization monitor that is trusted by both players and developers. Together, these ideas remove the need to run untrusted kernel-level anti-cheats, while providing the same level of protection compared to such solutions against a wide-range of common cheating mechanisms. The main challenge we face in implementing these ideas, however, is that the existing software stack for virtual machines is not designed to run video games and creates significant security and performance problems. We address these problems by proposing a security-focused Library OS kernel for games and an efficient graphics sharing pipeline for near-native rendering and display performance. In summary, without compromising on cheating behavior detection or performance, this work makes user privacy a first-class citizen in personal computers.

CCS Concepts

• Security and privacy → Virtualization and security.

*Both authors contributed equally to this work. Work completed while both authors were students at Arizona State University.

[†]Corresponding author.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

CCS '26, The Hague, Netherlands

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2871-6/2026/11

<https://doi.org/10.1145/3830454.3846577>

Keywords

Virtualization-based Security, Game Security and Privacy

ACM Reference Format:

Santosh Gokul Narayanan, Giovanni Paladino, Chuqi Zhang, Sangho Lee, Zhenkai Liang, and Adil Ahmad. 2026. You Shall Not Pass into Ring-0! A User Privacy-Friendly Anti-Cheat Architecture for Personal Computers. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*, November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3830454.3846577>

1 Introduction

With revenue exceeding \$195 billion in 2025 [56], online gaming is the largest entertainment sector in today's world. This revenue is driven by titles such as Fortnite, Apex Legends, CS2, and Valorant where players compete with each other for prestige, prizes, and even online followers through streaming platforms. Sadly, this aspect leads to cheating—professional teams, popular streamers, and casual players alike are routinely caught using cheats sold openly by vendors such as EngineOwning [30]. Such cheating behaviors disincentivize honest players, an aspect that has motivated game developers to invest heavily in countermeasures.

The prevalent countermeasure today is to deploy a class of solutions colloquially referred to as *kernel-level anti-cheats* by players (§2.1). As the name implies, such solutions install a privileged kernel driver on a player's computer, allowing them to monitor the system environment and prevent potentially malicious behavior (e.g., installing unsigned kernel modules and blocking debuggers from attaching to the game process). This driver typically operates in conjunction with a user-level component (running in the game process) to collect additional information and send it to the server for complete cheating behavior detection.

Unfortunately, kernel-level anti-cheats install untrusted (i.e., developer-controlled) software in privileged domain, leading to severe privacy risks for players: such anti-cheats have been found to collect telemetry far beyond gameplay, and in some cases ship spyware/rootkit-like behavior [29]. They also complicate security aspects for game developers. As modern kernels like Linux continue to grow larger and more complex, vulnerabilities in the kernel's codebase allows dishonest players to circumvent anti-cheat mechanisms (e.g., by tampering with anti-cheat modules after kernel compromise through a vulnerable signed driver [20]).

While there has been significant recent research on anti-cheat solutions [25, 37, 58, 71], it has focused on two orthogonal lines that do not adequately address the dual privacy-security risks introduced by kernel-level anti-cheats (§2.2). Specifically, the first line of research focuses on leveraging machine learning-based techniques to improve the effectiveness of detecting attacks like aimbots and wallhacks. The second line of research focuses on only improving the security of anti-cheat detectors by executing them within trusted execution environments (e.g., Intel SGX).

We present TIRITH¹, a privacy-friendly solution that obviates the need to run kernel-level anti-cheat components, while also strengthening anti-cheat security against kernel compromises. The solution leverages two key ideas:

- *Virtualization-Based Sandboxing*: Instead of running games in processes that users control, each game is isolated into a *Protected Virtual Machine* (PVM), a hardware-isolated guest environment provisioned by the OS vendor’s first-party trusted hypervisor and increasingly available on consumer machines [40, 54]. This provides natural two-way isolation of CPU and memory contexts between guest and host environments.
- *Split Responsibility Architecture*: A PVM must still share devices with the host. A player who is running a compromised platform (e.g., with unsigned drivers) can abuse this sharing to launch attacks against games. We split the responsibility of tracking whether the platform is trustworthy or not to the trusted and independent PVM hypervisor, instead of developer logic, and allow the developer to query this before starting game sessions.

To understand the feasibility of our approach, we ran a case-study on running video games using the state-of-the-art virtualization stack available on Linux (§5). We found two main problems. First, existing implementations all leverage a full-fledged OS like Linux within the virtualized environment, but this approach is both resource-intensive and raises security problems since Linux-based guest virtual machines require many interfaces to the outside world making it hard to reason about guest protection and host privacy. Second, all existing paravirtualization techniques to share the GPU with guest virtual machines incur prohibitive overheads running video games, since they require expensive coordination and slow message-passing between the guest and host contexts.

To address the first problem, we propose a *minimized game-centric library OS kernel* that hosts only the game and its developer’s anti-cheat module inside the PVM (§6.1). The kernel is extended with a virtual device subsystem exposing a shim DRM render node, an event device for input, a sound device, and a platform attestation device, so that existing games run unmodified. Game binaries and assets are loaded from a developer-signed manifest whose root hash is part of the boot measurement, while less latency-critical channels (network, sound, storage) attach via standard *virtio* interfaces. The LibOS also exposes hypercalls so the in-guest anti-cheat can collaboratively invoke trusted-monitor primitives, e.g., to attest the host’s driver-admission policy or query host platform measurements.

We address the second problem with a *native shared GEM* (*Graphics Execution Manager*) context (§6.2) that avoids the graphics command stream encode/decode and VM-exit costs of conventional

GPU paravirtualization. The host VMM allocates per-process GEM buffers from the host DRM render node, and the trusted hypervisor projects them into a pre-reserved region of guest physical memory under a write-combining cache policy. Inside the guest, the game and its OpenGL stack issue draw calls directly against these shared buffers with no frame copy or format conversion. Only the control plane (`ioctl` and graphics syscalls on the render node) is forwarded to the host to interact with the physical GPU, through an exception-less shared-memory channel. The host then displays the frames (which already live in a host-mapped GEM buffer) at native cost using a bridged windowing approach.

We implemented a TIRITH prototype for a Linux-based gaming environment using KVM as the type-1 hypervisor backend. The game-centric kernel is built on the Gramine Library OS [46, 75] and extended with the virtual device subsystem, manifest-based asset integrity, exception-less host RPC channel, and trusted-monitor hypercalls (platform measurement, driver-policy attestation, attestation digests) described above. The native shared-GEM rendering pipeline supports OpenGL 4.6 applications via the EGL specification, and the bridged display/input pipeline targets X11. Our prototype is available at <https://github.com/ASTERISC-Release/Tirith>.

Using our prototype, we conduct a structured security analysis (§7) of TIRITH from both sides of the trust boundary. We show that the player’s host privacy is preserved by hypervisor-based sandboxing together with a small, mediated set of guest-host interfaces, and that game security is upheld with the same anti-cheat primitives kernel-level anti-cheats rely on today — now enforced inside the guest where the developer maintains full control. We further illustrate the analysis against three real-world cheat families (wallhacks, aimbots/triggerbots, and map hacks), each defeated by a combination of in-guest and monitor-delegated primitives.

We also provide a detailed performance evaluation of TIRITH using benchmarks and real-world video games (§9). Compared with the state-of-the-art paravirtualized stack (VirGL and DRM-Native), TIRITH delivers near-native rendering throughput and input latency while reducing guest CPU and memory utilization, narrowing the gap to non-virtualized Linux to within a small constant factor across all three games. Concretely, our evaluation shows that TIRITH achieves average FPS and 1% lows within 4.1-5.2% and 16.5-18.2% of native respectively, which is 2.4-2.8× and 3-3.8× better than existing paravirtualization solutions. In conclusion, our work demonstrates that strong user privacy is achievable in modern computers, without compromising on cheat prevention or performance.

2 Motivation

2.1 Kernel-Level Anti-Cheats and Risks

Malicious players can attempt to interfere with game execution by modifying memory, injecting code, attaching debuggers, loading malicious drivers, or bypassing user-space enforcement mechanisms [26]. To address this problem, the current design of anti-cheats typically involves both a user-level component and a *privileged kernel component*. Their primitives are shown in Table 1. The user-level component is game-specific and deployed alongside the game client. It is responsible for tasks like collecting information from the game and sending it to servers for analysis (e.g., using machine learning to detect aimbots [25, 37, 42]). In contrast, the kernel

¹In J.R.R. Tolkien’s Sindarin language, Tirith means “guard,” “watch,” or “vigilance.”

Table 1: Typical protection primitives (P0 - P9) required by today's game protection software [3, 29, 33].

System-Level (<i>implemented by Kernel drivers</i>)
P0: Block unsigned/malicious kernel drivers from loading at boot.
P1: Block debuggers/inspectors that read or modify the game process.
P2: Block external devices from accessing memory or synthesizing input.
P3: Restrict runtime driver loading to signed and known-good drivers.
P4: Remotely attest platform integrity to the game server.
Game-Specific (<i>implemented by userspace component</i>)
P5: Scan game memory for known cheat signatures/suspicious writes.
P6: Obfuscate code and data to slow reverse engineering.
P7: Detect unauthorized modifications to game binary and asset files.
P8: Detect or block unauthorized DLL/module injection to game.
P9: Collect game runtime data (screenshots, player events).

component is installed as drivers or modules [1, 3, 16, 33, 37]. It offers *global system visibility and control*, allowing the anti-cheat to observe and regulate behaviors relevant to game integrity, including memory access, library injection, driver loading, debugging activity, and interactions with input and display paths. Table 1 illustrates the responsibilities of user and kernel anti-cheat components.

Famous examples of anti-cheat software that follow the aforementioned design include Riot Vanguard [33], Easy Anti-Cheat (EAC) [3], FACEIT [1], and BattlEye [16], colloquially referred to as *Kernel-Level Anti-Cheats* by users. These systems are highly-resistant to tampering, since software running at high privilege can better monitor its own integrity and resist attempts to disable or bypass its enforcement logic. For example, Vanguard integrates with secure boot and loads early during system startup to establish privileged protection and strengthen its integrity [33].

Sadly, **kernel-level anti-cheats introduce severe privacy risks for honest players, while also providing limited security against powerful dishonest players**. Players are asked to allow closed-source, developer-controlled components to execute with kernel privileges. With such privileges, anti-cheat software may access host data unrelated to gameplay, interfere with benign applications, and affect normal system behavior [63]. For example, Vanguard interposes UI windows to detect display-based cheat overlays, but this capability has also raised concerns because it enables full-screen screenshot capture [2]. Some widely deployed kernel-level anti-cheat systems have also been criticized as “rootkit-like” [29]. Such systems may conceal their presence and resist inspection or removal, while retaining broad capabilities for remote access and data collection [7]. On the security side, kernel anti-cheats can also be circumvented by abusing core kernel vulnerabilities [49, 78] or device drivers [20] to achieve full kernel compromise. Consider the Genshin Impact attack, where a signed vulnerable driver was used by a ransomware actor to bypass privileges and kill antivirus processes and services during a mass-ransomware deployment [69]. This attack shows that vulnerable drivers remain a realistic threat vector even against privileged software.

2.2 Scope of Existing Anti-Cheat Research

Prior work has focused on *improving the detection or prevention of cheating behavior*, but has not addressed the privacy risks introduced by kernel-level anti-cheat software itself. These solutions can be broadly grouped into the categories below.

The first category improves cheat detection or defense against specific cheat pipelines. These approaches collect richer evidence or directly target particular cheating behaviors. For example, AVM [37] makes execution tamper-evident and replayable, enabling suspicious game behavior to be audited against recorded execution history. Other systems use behavioral signals and machine learning to detect cheating patterns such as aimbots [13, 17, 35]. Invisibility Cloak [71] proactively perturbs rendered frames to make visual aimbots unreliable while preserving the normal visual experience for human players. Although these techniques improve cheat detection or defend against specific attacks, they do not address whether proprietary anti-cheat should hold privileged access on the host.

The second category uses trusted execution environments (TEEs) to enable lightweight countermeasures against particular cheating behaviors while protecting the defense logic itself from an untrusted host. For example, BlackMirror [58] uses Intel SGX to protect hidden game state and performs trusted visibility testing before rendering, thereby preventing wallhacks. BotScreen [25] places a pre-trained model inside an SGX enclave to detect aimbots on the client side. They show that trusted execution can harden selected portions of the game protection pipeline. However, they remain *protection-centric point solutions*: each targets a specific cheat surface, and user-space TEEs such as SGX do not replace the broader range of privileged enforcement mechanisms still relied upon by today's anti-cheat systems in practice. TZMon [41] relocates mobile game anti-cheat into the ARM TrustZone secure-world kernel, hardening it against a compromised Android OS. The anti-cheat itself, however, still holds privileges over the normal-world OS to inspect game state, retaining the same trust issue.

3 Goal and Approach

Our goal is to design a privacy-friendly anti-cheat architecture for Linux-based computers that does not rely on developer-controlled privileged components. We target Linux because kernel-level anti-cheats are notably absent, even as gaming on the platform continues to grow in popularity. Anti-cheat developers themselves argue this gap is not incidental: the Linux kernel restricts proprietary modules from tightly coupling with the kernel through GPL-only symbols, and any user can compile a custom kernel that hides cheats below an anti-cheat's view [28, 32]. This makes Linux precisely the right platform on which to rethink anti-cheat architecture from the ground up and treat host privacy as a first-class citizen. Nonetheless, the resulting design also ports to other platforms like Windows. Our approach (§3.1 - §3.2) is shown in Fig. 1.

3.1 Virtualization-Based Sandboxing

Games today are executed in regular user-level processes, even though their threat model is distinct from typical applications that execute in processes. For typical applications, the user (who has *root/admin* privileges on the computer) is assumed to be trusted. This assumption simply does not hold true for competitive games where the user is assumed to be actively trying to tamper with the game. Therefore, our first insight is that *the process abstraction is fundamentally unsuited to running competitive games*.

Based on the aforementioned insight, we propose to leverage an emerging abstraction in personal computers that naturally provides

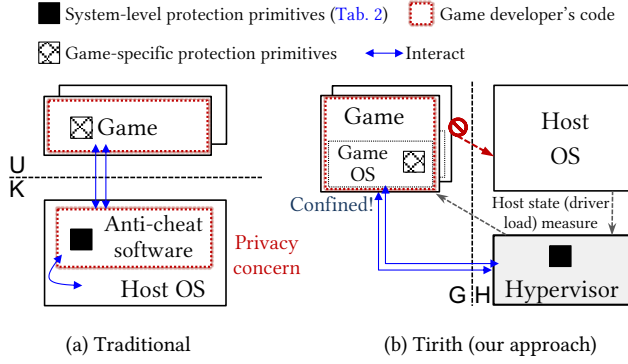


Figure 1: Overview of the VM-based sandboxing with split responsibility architecture (G: guest; H: host).

protection to computations against a root user, namely the *Protected Virtual Machine* (PVM). This abstraction is realized using a type-1 hypervisor that executes on the computer and isolates a guest virtual machine's context from an untrusted host. Modern commodity platforms increasingly implement a PVM abstraction layer, including Protected KVM on Linux/Android, and Virtualization-Based Security (VBS) Enclave on Windows [9, 40, 54].

The PVM abstraction provides several desirable properties required for our architecture. (i) *First-party and signed*: the hypervisor is provisioned by the OS vendor and launched as part of secure boot, requiring no trust in any third-party developer code. (ii) *Higher privileged than the kernel*: the hypervisor cannot be disabled or bypassed by software running on top, including the player acting as root. (iii) *Strong two-way guest isolation*: it hosts PVMs whose memory, CPU state, and devices are hardware-isolated from the host, so that neither the host nor the guest can inspect or tamper with the other. (iv) *Remote attestation*: Both the hypervisor and protected guest can be measured to produce attestation reports that a remote party (e.g., a game server) can verify.

Based on these properties, each game can be executed inside its own PVM guest to provide strong isolation from the host. In the guest, the game and the developer's anti-cheat logic run with kernel level privilege over the game environment, giving the developer full control over the guest for monitoring and enforcement. The same isolation works in the opposite direction. The developer's code is confined to PVM: it cannot read host memory, spy on unrelated applications or data, capture arbitrary screen contents, or persist on the machine after a session ends.

3.2 Split Responsibility Architecture

While Protected Virtual Machines (PVMs) provide isolation and sandboxing to computations running on the CPU, they cannot *alone* fully-protect computations like video games that leverage external devices like GPUs and input devices (e.g., keyboard). In particular, external devices are shared with the host using device-sharing primitives (discussed in §5), allowing an untrusted host to launch attacks. For example, a root user can install malicious drivers that manipulates the framebuffer for display, or tamper with

game binaries/assets on the host-backed guest filesystem to substitute modified resources. Therefore, *complete video game protection requires some degree of visibility and control over the host*.

Fortunately, we observe that the required host visibility and control can be achieved using the trusted PVM hypervisor [66]. Taking Windows VBS as an example, the hypervisor implements secure boot and measured launch establish a trusted boot chain. Moreover, the hypervisor governs driver installation and decides which kernel modules can load, as well as leverages the IOMMU and interrupt remapping mediate what devices can touch protected memory. It also supports remote attestation to expose the resulting state to a verifier. In fact, we are already seeing the hypervisor becoming an important cornerstone of anti-cheat protection. For instance, the developers of Call of Duty: Black Ops 7 already incorporate VBS driver verification into their anti-cheat pipeline to strengthen protection against host-resident cheats [6].

This motivates TIRITH to partition the responsibility of anti-cheat enforcement along the host and guest boundaries (Table 1).

- **System-level enforcement**, including boot integrity, driver verification, anti-debugging, device isolation, and remote attestation, are generic platform security features that PVM hypervisors enforce. The developer simply attests to these features when a player wants to start a new game session. While the game session is in-play, the hypervisor ensures that the attested state is not modified (e.g., prevents new driver loading).
- **Game-specific enforcement**, including memory scanning, binary hardening, asset and module integrity, and game data collection, are enabled by user-space anti-cheat components that developers already ship alongside today's game clients (§2.1). These components can operate directly within the protected guest environment alongside the game binary.

3.3 Deployment Assumptions

The fragmentation of the Linux ecosystem poses a remote attestation challenge because users and distribution maintainers reserve the right to customize trusted components and kernels. Therefore, our immediate deployment target are mature distributions (e.g., Ubuntu, SteamOS) and controlled platforms (e.g., Steam Deck, Steam Box). In these targets, maintainers can actively track the platform status like Microsoft does for Windows setups [53]. Gaining enough traction, we believe the Linux community will standardize remote attestation requirements. A precedent for this can be found in prior standardization work for UEFI Secure Boot [22].

4 System and Threat Model

The game player has root (or admin) privileges on their computer and the technical ability to launch software-based attacks against video games (e.g., wallhacks via memory tampering). The player does not trust game-developer code, which may attempt to disclose host data, while the developer does not trust the player.

Both parties trust the OS vendor (e.g., the Linux distribution provider) to ship the platform. From the game developer's perspective, the host kernel is initially benign but, since the player has administrative control, may be manipulated at runtime (e.g., by installing malicious drivers). This assumption is exactly the

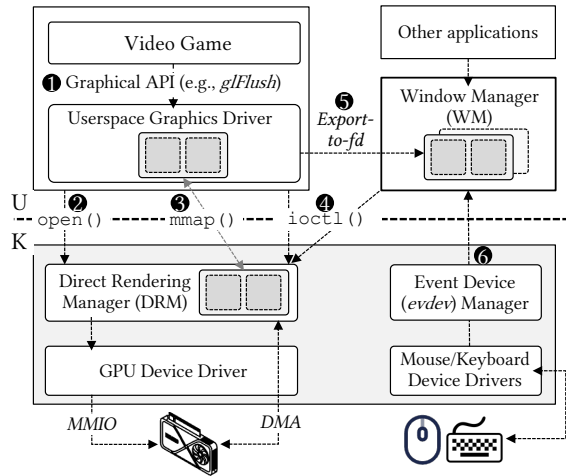


Figure 2: Native graphics rendering and display pipeline.

one modern OSES already adopt: Windows VBS and Android Protected KVM are deployed precisely because the OS no longer trusts the kernel to be vulnerability-free and instead relies on hardware-assisted isolation and integrity hardening. Within our architecture, the hypervisor is the trusted computing base (TCB) on which both parties rely for isolation, integrity, and mediation across the host and game environments. The hypervisor is launched before the host OS through secure boot [8], becomes permanently resident, extends its measurements into the TPM, alongside the host’s initial state and security configuration. All parties can query the TPM to attest that the system is running atop a trusted hypervisor.

Out-of-scope. Consistent with current kernel-level anti-cheats, we exclude hardware-based attacks that rely on the interposition of display or input devices [65]. These attacks are generally expensive and error-prone. We do not consider memory-based attacks (e.g., control-flow hijacking through ROP/BROP) that allow attackers to compromise the kernel without installing malicious drivers. Finally, we also consider micro-architectural defects [55] and side-channels [44, 52] as out-of-scope.

5 Running Video Games under Virtualization

Our proposed approach (§3) requires video games to execute within virtualized environments. This section investigates the suitability of the current virtualization stack for this approach. It begins by providing preliminary knowledge related to virtualization and video game requirements, and then discusses our findings based on a real-world case-study of running games in virtual machines.

5.1 Preliminary Knowledge

Modern computers can leverage hardware support [39] to virtualize CPUs at near-native speeds, but share devices between a guest and host using slower software techniques [5]. In terms of video games, the most latency-sensitive device is the Graphics Processing Unit (GPU). The remaining paragraphs explain the pipeline (in Linux-based systems) through which games access the GPU in native (non-virtualized) and virtualized environments.

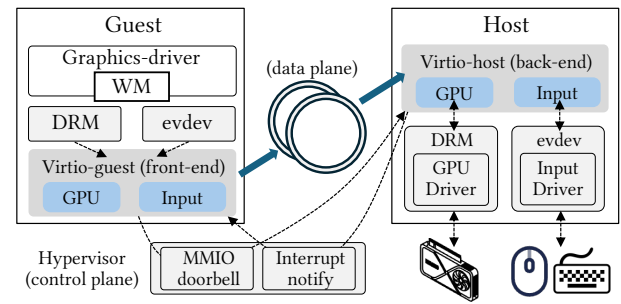


Figure 3: Overview of virtio-based device virtualization.

Native graphics processing pipeline. Video games coordinate with the GPU device to render frames (i.e., generate images from game state or 3D models) and ask the Window Manager (WM) to display these frames (Fig. 2).

Graphics rendering via user-space and kernel drivers. The game issues render (or draw) calls through a standardized graphics API, such as OpenGL or Vulkan on Linux (❶). These APIs are handled by a user-space graphics driver (or library) that is included within the game process. The library interacts with the GPU through the kernel's *Direct Rendering Manager* (DRM) subsystem [73]. Like the user-space driver, the role of DRM is to maintain a standardized interface to send data and commands to underlying GPU device driver (and hardware). It exposes a virtual render node (e.g., `/dev/dri/128`) corresponding to a GPU device that the user-space driver opens for access (❷). The user-space driver then interacts with DRM along two complementary pathways:

- *Data pathway via memory-mapped GEMs (③)*: DRM allocates Graphics Execution Manager (GEM) objects to hold rendering data such as framebuffers, textures, and vertices; the user-space driver `mmap`s them into its own address space and writes directly, avoiding per-frame memory copies.
- *Command pathway via system calls (④)*: With data staged in GEMs, the driver submits commands and synchronization primitives through `ioctl`; DRM forwards them to the GPU device driver, which programs the GPU over MMIO while bulk data moves through Direct Memory Access (DMA).

Graphics display via the window manager. Display (to the screen) is handled by a system-wide window manager daemon, such as an X11 server or a Wayland compositor [77]. Once a frame has been rendered, DRM’s `export-to-fd` capability hands the GEM handle directly to the WM, which asks the GPU driver to scan the frame out without data copies between different userspace contexts (5) [77]. The WM is a higher-privileged user-space daemon that composes surfaces from all running applications at their respective resolutions and scaling. Note that the window manager also facilitates player input (e.g., mouse and keyboard). The player input events happen and enter the kernel through their device drivers, are normalized into a uniform stream by the Event Device (`evdev`) subsystem, and are forwarded by the WM to the focused application (the game) using inter-process communication (6) [50, 77].

Virtualized graphics processing pipeline. The most efficient way for guest to access devices today is to leverage *paravirtualization*, an approach where the host provides an emulated device interface through which the guest sends and receives data to/from the device. The default interface in Linux is called *virtio*, which supports shared ring buffer message passing and a doorbell mechanism (using MMIO) to notify the guest and host.

It is important to note that paravirtualization techniques like *virtio* are unsafe in general trusted execution environments (e.g., Confidential VMs like Intel TDX [4] running in cloud machines) because an administrator can read or tamper with the communication (e.g., by installing a malicious kernel module). In our architecture, a hypervisor implemented by the trusted OS vendor allows attestation and runtime enforcement of the host’s integrity state (§3.2). Therefore, after attestation, paravirtualization is suitable in our setting since the attested and protected host kernel will not allow an administrator to hook onto VM-device communication.

Fig. 3 illustrates paravirtualization for GPUs in Linux. The *virtio-gpu* device allows sharing of graphics data to the host. A thin guest-side driver surfaces a virtual GPU device to the game’s userspace graphics stack and the kernel’s DRM subsystem to receive data payloads in batches. Every batch crosses the guest/host boundary along two coupled planes. The *control-plane* world switches through the hypervisor (an MMIO write in the guest triggers a VM exit, the hypervisor dispatches to the host back-end to handle the commands, and an interrupt injection returns the result). The commands are followed by a *data-plane* copy through the shared ring carrying the per-batch graphical command payload. The device offers two ways to carry payloads using a higher-level GL/Vulkan command stream (so-called VirGL) or a lower-level vendor DRM stream (so-called DRM native context):

- **VirGL [74]:** The guest encodes each graphical API call (e.g., OpenGL) into a portable, vendor-independent command stream and sends it to the host; the host decodes the stream and re-issues the calls against the real GPU’s user-space driver. The encode/decode step makes VirGL portable, but adds a per-call translation cost on both sides of the boundary.
- **DRM native context [36]:** Instead of encoding/decoding graphical API calls, the game’s user-space graphics driver is connected directly to the host’s DRM subsystem. This allows the application to issue the same per-vendor `ioctl`s and GEM operations it would on bare metal, while *virtio-gpu* acts as a thin transport for those calls (e.g., copying data between GEM).

5.2 Case-Study and Findings

We conducted a case study with the state-of-the-art virtualization stack for gaming in Linux systems. Concretely, we ran *muvm* [15], a lightweight gaming-focused VM that runs a cut-down version of Arch Linux on the optimized *libkrnl* virtual machine manager [27]. *muvm* leverages hardware acceleration through KVM and supports paravirtualization (§5.1), both VirGL and DRM native context. Inside the guest, we ran three popular open-source Linux games and compared their performance to execution on the host. Refer to §9 for details about the hardware and games.

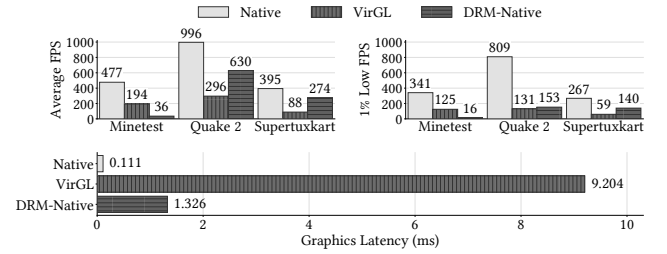


Figure 4: The top graphs show performance for existing graphics solutions across three games. The bottom graph shows graphics pipeline latency in SuperTuxKart at 1080p.

Even before starting a game, we made the observation that this virtual machine setup significantly increased resource utilization compared to native execution. Linux, by default, is an inherently multi-tasking OS implementation that spins-up multiple background services, system daemons, and auxiliary processes alongside the game task. Many of these tasks are not required in our setting, but it is challenging to disable them entirely from the kernel. The result of this was that in one game the CPU utilization of the system went by 4.9×, while the system also consumed approximately 600MiB of additional memory for running the same game. The memory impact, while small for dedicated gaming machines, would be significant for low-resource computers (e.g., laptops).

More concerning than the resource utilization aspect, is the fact that full kernel implementations like Linux are complex and huge, making it challenging to reason about isolation between the guest and host environments. Specifically, recent work [46] has analyzed the interfaces between lightweight Linux guest kernels [12] and the hypervisor/VMMs, estimating that around 911 such pathways for communication exist. Validating each of these pathways to ensure host privacy as well as guest protection is a non-trivial task.

Observation-1: Full-fledged OS kernels raise both resource utilization and isolation challenges in the gaming context.

We next examined graphical performance using the end-to-end *frames-per-second (fps)*, the standard metric for real-time gameplay, within the guest environment. Fig. 4 (top graphs) reports the average and 1%-lows fps inside the guest. We noticed that the games produced 1.4-13.3× and 1.9-21.3× fewer frames, respectively, against native execution, well below what high-performance games today tolerate. This overhead was felt for both VirGL and DRM native context (shown as DRM-Native in the figure).

To localize the cause of this performance drop compared to native execution, we measured the roundtrip time on each graphics command submitted to DRM (at `ioctl`) and on each frame submitted to the WM (at IPC) in both the host and guest environments. Fig. 4 (bottom) illustrates the average graphical pipeline latency for one game, and we note that both paravirtualization solutions incurred at least an order of magnitude overhead in terms of their graphics pipelines. In the worst-case, VirGL’s graphical pipeline was up to 82.9× slower than native execution. Note that this latency is amortized by CPU computation time (e.g., executing game logic

and preparing frames to be rendered), which is roughly the same inside the guest and host due to efficient hardware CPU virtualization, and that is why the impact on *fps* is marginally lower.

The increased graphical pipeline latency is not surprising looking at the guest-host crossing of *virtio-gpu* (§5.1) paid on every batch job submission. Recall that this incurs an MMIO-triggered guest exit, a hypervisor dispatch to the host backend, a shared-ring copy of the per-batch payload, and an interrupt-injection return. VirGL additionally adds a per-call encode on the guest and a matching decode on the host. DRM native context removes the translation step (and does improve performance in most scenarios compared to VirGL) but leaves the per-batch world switch intact.

Observation-2: *Virtio-based GPU paravirtualization pathways impose excessive cost in intensive video game scenarios.*

6 TIRITH Design

TIRITH is a privacy-friendly gaming architecture that is built on our proposed approach of virtualization-based sandboxing and split responsibility architecture (§3). To address the limitations of the current virtualization software stack in terms of supporting our approach (previous section), this section introduces two new design ideas: (a) a Library OS kernel design that is lightweight and minimized in terms of its outside interfaces specifically to support Linux games (§6.1) and (b) a novel graphics sharing pipeline for our proposed kernel design that does not rely on slow paravirtualization interfaces provided by existing solutions (§6.2).

6.1 Minimized Game-Centric LibOS Kernel

A Library Operating System (LibOS) kernel provides a specialized execution environment within guest virtual machines to support a single computation. Like traditional kernels, LibOS kernels contain several subsystems that ensure smooth execution of the application inside the guest including system call handlers, scheduling, memory management, socket handling, and device drivers. Additionally, modern LibOS kernels provide wide support for the POSIX system call abstraction, which is critical for running unmodified Linux applications. Famous examples of LibOS kernels include Gramine [46] and UniKraft [45].

There are two problems in leveraging existing LibOS implementations for TIRITH. First, existing LibOSs are designed for cloud computations and they lack support for devices like graphics and input. A naive approach would be to port the infrastructure for these devices from Linux kernels, but this is complex given the differences between Linux and LibOSs, as well as potentially unsafe (i.e., opens unnecessary attack surfaces). Second, for the same reason as above, existing implementations are neither designed for game delivery nor coordination with the (trusted) hypervisor for split game protection. This section describes how we address both problems. Fig. 5 illustrates the overall LibOS architecture.

Game-specific device interfaces and drivers. Recall that Linux games interact with critical devices using the virtual device subsystem (§5.1), thus TIRITH implements a virtual device subsystem that supports backward-compatibility of existing games. Also, TIRITH

provides other paravirtualized device interfaces. Both the subsystem and interfaces are designed to maintain only a small set of controlled, outside interfaces (refer to §7 for details).

① *Virtual Device Subsystem:* TIRITH’s device subsystem implements four devices, including a virtual Direct Rendering Manager (DRM) for graphics, event device for input, sound device, and a platform device for host state attestation. The main role of these subsystems is to intercept system calls from user space and redirect them either to internal TIRITH-LibOS components (e.g., helper libraries ②) or the host virtual machine manager. Since communication to these devices is latency-sensitive, TIRITH implements an efficient exception-less message-passing interface with the host. We explain this message-passing interface and the function of the graphics and input subsystems in the next section (§6.2).

The platform device facilitates coordination between the hypervisor and the guest code to establish and maintain the trustworthiness of the host using three steps. First, it allows the guest to query platform state and measurement related to boot events (e.g., stored in the computer’s TPM). Second, it allows the guest to obtain a signed attestation digest that can be reported to the remote game server for verification. Third, it allows the game developer to determine the currently-loaded kernel drivers and ask the hypervisor to prevent further loading of drivers until the game is terminated. Each of these aspects are enabled by calling the trusted hypervisor (using hypercalls). Note that game developers can add more features to implement more robust game protection in the future.

③ *Paravirtualized Device Interfaces:* While our LibOS implements new solutions to access GPUs and input devices, it leverages *virtio* to share other (less latency-critical) devices with the host. Specifically, games require access to (a) the network to send/receive information from game servers, (b) the sound card for audio playback and microphone capture, and (c) storage to retrieve files and game assets. The LibOS relies on *virtio-vsock* and *virtio-snd* to support network and sound, respectively. The virtual socket (*vsock*) interface is optimized for direct socket interactions (i.e., it does not require expensive operations like *virtio-gl*), and it avoids an implementation of the complex network stack inside the LibOS.

For storage, TIRITH supports two interfaces, namely *virtio-blk* and *virtio-fs* [11, 64]. The former is a high-performance interface that allows loading entire virtual machine images (as block devices) into the guest environment. TIRITH leverages this interface to efficiently load the initial game image (next heading). The latter interface allows reading and writing to files on the host. It is used at runtime to read additional files if needed, or write-back data to the host (e.g., log files). All files that are read from the device are integrity-checked through the LibOS’s in-memory file system using a signed manifest securely provided at runtime (③). We explain game deployment and loading in the next heading.

Game deployment and runtime protection workflow. This section explains the end-to-end event workflow for executing games with TIRITH in Protected Virtual Machines (PVMs). To support the deployment of video games, TIRITH provides a template guest image (which we call a *Zygote*) to developers. In addition to previously-described LibOS kernel components, this image contains core helper libraries responsible for implementing efficient graphics and input pipelines (described in the next sections).

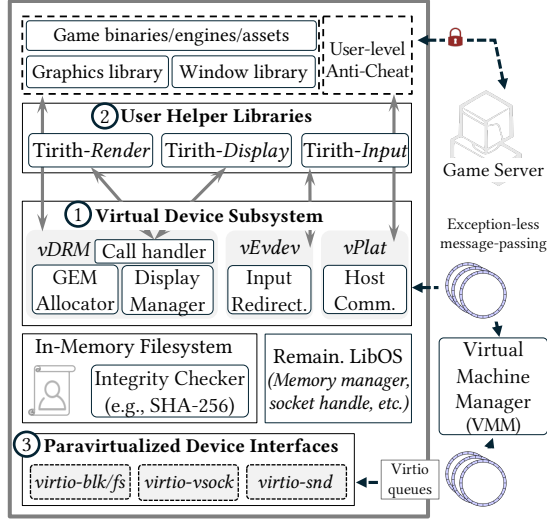


Figure 5: TIRITH LibOS architecture and components.

During offline preparation, the game developer creates a cryptographically signed *manifest*, a *Zygote*, and a game (storage) drive for their particular game. The storage drive contains the game’s primary assets (e.g., game binary and libraries) and its user-level anti-cheat component (§2.1). The manifest contains integrity hash measurements of the game drive and other file assets that will be loaded into the virtual machine at runtime. It is signed using the developer’s private key. At this stage, the game developer also installs their public key into their game’s *Zygote*. This allows loading of correctly-signed game manifests, once the *Zygote* executes inside the PVM. The signed manifest, *Zygote*, and game drive are sent to the user’s computer (e.g., through standard game delivery platforms like Steam or Epic Game Library).

To start a game, the host invokes the trusted hypervisor to instantiate a PVM using the provided *Zygote* image. The hypervisor measures the *Zygote* at launch time, records these measurements (e.g., in its protected memory regions or a TPM), and enforces strict isolation between the PVM and the host environment. After boot, the *Zygote*’s LibOS first reads the signed manifest (through *virtio-fs*) and verifies it using the installed key. If the signature passes, it loads the game drive and other files mentioned in the manifest and verifies its integrity using the provided hash. At this stage, if any file is found to be corrupted the PVM stops execution, since this signals a potentially-malicious host or some other corruption. Note that game asset integrity-checking is also a common required step by today’s anti-cheat software [31].

After the game environment is set-up inside the PVM, the game connects over the network (through *virtio-vsock*) to authenticate itself with the game server. Specifically, using TIRITH’s platform device, the game obtains the platform’s measurement (recorded during secure boot within the host TPM), as well as the measurement pertaining to the initially-loaded *Zygote* image. These results are embedded into an attestation digest and sent to the game server. Game developers can remotely attest the PVM by validating the digest measurements, thereby confirming that the game is executing

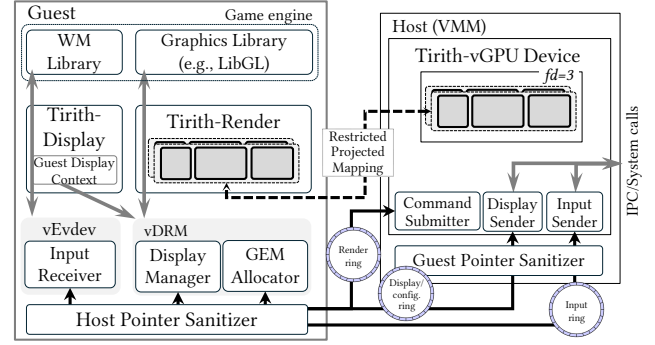


Figure 6: Illustration of Shared GEM context graphics.

atop a TIRITH-enabled hypervisor with an unmodified *Zygote* and intact protection mechanisms. Only after successful attestation is the game allowed to start a competitive session.

6.2 Shared GEM Context Graphics Pipeline

TIRITH implements an efficient graphics pipeline that video games use during execution to avoid the expensive data-plane and control-plane operations of *virtio-gpu* (§5.2). The key insight for this pipeline is that instead of sending data frames to the host, we can securely map the host’s graphics data-containing structures, namely the Graphics Execution Manager (GEM) objects into the guest and allow it to operate directly on these objects. This shared mapping is safe, because TIRITH’s anti-cheat architecture can protect the confidentiality and integrity of GEM data regions (§5.1). In particular, the hypervisor’s attestation establishes the trustworthiness of the PVM and host (as described in the previous section), which in turn prevents users from accessing game-related GEMs.

With the aforementioned approach, only command submission must be redirected to the host, which can be efficiently achieved given the small size of control packets and TIRITH’s exception-less message passing, as well as restricted address space projection. Another observation we make is that, with our new pipeline, rendered frames are already available within the host virtual machine manager (i.e., within the GEM objects). Therefore, the rendered frames can be *directly* displayed by instantiating a window on the host and thus achieve native display performance (Fig. 6).

Cross-Context GEM allocation and mapping. While GEMs belonging to the host DRM interface and GPU cannot be allocated by the guest, we observe that they can (a) first be mapped into the host virtual machine manager (VMM)’s address space and then (b) be projected into the guest’s allocated physical regions. Recall that GEM objects are unique per-process (§5.1), and thus there is no security or privacy problem in projecting the VMM’s GEM objects into the guest. Moreover, GEM objects only contain *static* data (i.e., no internal nested pointers), thus only projecting the GEM regions is sufficient. The paragraphs below explain how TIRITH achieves the aforementioned mapping and projection.

During boot, TIRITH reserves a small pre-defined region of its allocated physical memory to hold GEM objects. In our experiments, a region of 128MiB was mostly sufficient for our evaluated applications. In the scenario where the pre-allocated guest region

for GEMs is exhausted, TIRITH leverages the following workflow. The game's LibOS kernel first reserves another free physical region. Then, it notifies the trusted hypervisor of the newly-reserved region's parameters (i.e., address and size) through a secure hypercall. This allows the hypervisor to track that this region is free and can be re-allocated for GEMs if requested (see below).

When the game executes, the userspace graphics library (e.g., OpenGL) requests creation of GEM objects to TIRITH's *vDRM*. These calls are forwarded to the host VMM (using the channel described in the next heading). The VMM asks the host DRM to allocate GEM objects as required into its address space. The physical frames containing these objects must be mapped into the guest, but the host cannot directly change a PVM's physical regions for security purposes. Specifically, the Extended Page Tables (EPT) that control guest mappings are controlled by the trusted hypervisor. Therefore, the VMM sends the physical address of the objects to the guest *vDRM*, which first validates these mappings (i.e., they do not belong to a pre-existing guest region). If validation succeeds, it requests the trusted hypervisor to update mappings.

Note that GEM objects contain VRAM-resident buffers (i.e., memory regions directly shared between the GPU and CPU). Once these objects are mapped into the guest's address space using the workflow described in the previous paragraph, we must ensure that writes made to these regions by the guest remain visible to the GPU. In non-virtualized settings, the kernel automatically handles this by setting a *Write-Combining* (WC) policy in page tables related to GEM pages. This avoids storing writes to GEM regions in the CPU cache using efficient batching, ensuring cache-coherency across CPU-GPU contexts. By default, our remapped GEM objects, however, are marked cache-coherent only within the guest context. This means that the guest's write operations on GEM objects may be cached by the CPU, and not be directly visible on the host (and GPU) at command submission time. And, this is critical for VRAM resident buffers, where signaling mechanisms goes over GEMs. To address this problem, TIRITH ensures *Consistent Caching Parameters* for the GEM reserved region. Specifically, TIRITH modifies guest page tables after GEM remapping and marks the GEM-related physical pages as Write-Combine (WC). Note that this policy is only required for the GEM reserved region.

Control redirection for render command submission. With the VMM's graphical data-plane projected directly into the guest, TIRITH must ensure that control commands are appropriately forwarded to the host for submission. Recall that this includes system calls like `ioctl` for rendering (§5.1). Additionally, `ioctl` is also used for CPU-GPU synchronization in Linux. In particular, applications use these calls to both allocate kernel-backed fences (`DRM_IOCTL_SYNCOBJ_CREATE` parameter) and signal the GPU (e.g., `DRM_SYNCOBJ_WAIT`) using the DRM as an intermediary.

A naive way would be to leverage a *virtio* interface to redirect control commands, but it incurs significant latency that is particularly harmful for frequent synchronization operations. TIRITH avoids this by implementing an exception-less message-passing interface [14, 68]. This interface leverages shared memory between two communicating contexts and background threads that are

quickly woken-up on message arrival. Specifically, TIRITH's interface contains two shared regions, including a `COMM` and `DATA` region, and uses CPU notifiers to wake-up threads. Our evaluation indicates that our implementation is efficient even under a small number of available system threads.

While other control-plane commands are self-contained, `ioctl` packets raise a complication because they may internally contain pointers to other in-memory data structures in guest memory. For example, take a given `ioctl` which takes as an argument, a reference to a struct that contains various other objects which may be located on the stack, heap, or other memory segments. One way to address this problem is to copy/translate pointers across contexts [38], but this would incur extra costs and reliability problems.

To address the aforementioned complication, TIRITH implements *Read-Only Identity-Mapping* of the guest's data regions into the host virtual machine manager. Specifically, the guest's data regions are mapped at the same virtual address in the VMM, allowing both the guest and host to reference points using the same address, removing the complexity and wasted computation of translating object references at runtime. Note that this process is simplified in a LibOS context, because (a) LibOS kernels have a flat address space where the virtual and physical address is identical and (b) the heap, global, and data regions are determined during boot [46]. TIRITH achieves identity-mapping by initially reserving a part of the VMM's address space before the PVM launches, and then requesting the trusted hypervisor to map the physical frames of the guest to the reserved space.

Sharing direct pointers across untrusted contexts can result in confused deputy attacks (e.g., the guest passes a malicious pointer that belongs to the host's memory to steal data [24]). TIRITH addresses this problem by implementing *Pointer Sanitization* at both the guest and host entry-points by checking that pointer addresses are valid (i.e., guest pointer only points to guest regions and vice versa) before relaying any command to internal components.

Direct display with bridged windowing. Due to shared GEM objects, rendered frames are already available within the host context. To avoid cost for displaying these frames, TIRITH creates a window on the host that fully mimics the game's intended operations (e.g., resolution, etc.) and directly displays the frames on this window. Specifically, TIRITH's display library intercepts all windowing-related calls from the game engine and presents to it a virtual display context inside the LibOS environment. For each of these commands, it coordinates with the host VMM to perform these operations as required (explained below).

There are three main classes of windowing-related calls: (a) configuration calls which change display options like create windows and set resolutions (e.g., `XCreateDisplay` for X11 windows), (b) display calls that send frames to be displayed and synchronize their execution, and (c) input calls that receive player interactions from keyboard, mouse, and other devices. The first two classes of calls must be sent from the guest to the host, while the third class must be sent from the host to guest.

TIRITH leverages two exception-less message-passing interfaces to handle these calls (i.e., one for the guest-to-host channel and one for the host-to-guest channel). In particular, all input-receiving calls (e.g., `XSelectInput`) are intercepted at the host library and

relayed to the guest using shared memory. Note that since input latency is critical for many video games, TIRITH creates a dedicated channel (with its own background threads) just for input redirection to minimize latency (refer §9.1 for details). The remaining calls are intercepted at the guest and relayed to the host using the same exception-less memory-based communication channel established in the previous section.

Like graphical rendering commands, window-related calls also contain nested pointers. Recall that guest data regions are already shadowed in the host's address space (previous heading), thus guest-to-host calls and embedded pointers are automatically handled. For host-to-guest pointers, TIRITH creates a reserved heap region within the host VMM, which is similarly shadowed into the guest's address space. The host leverages a custom memory allocator on this region and receives input commands from the host on this region. This allows *zero-copy* message-passing both ways. As we explained previously, pointers are always checked when passing between guest-host contexts to prevent attacks.

7 Security Analysis

This section analyzes TIRITH's design and architecture to show that it provides both player host privacy and an equivalent level of game anti-cheat protection (as kernel-level anti-cheat). Note that TIRITH's virtualization-based security monitor is the first component loaded during system boot and serves as the root of trust.

Newly introduced interfaces. Beyond the monitor itself, all interaction between pVM and the host traverses a small, auditable set of interfaces, summarized in Table 2. Following the discipline of recent confidential-VM frontends [46], we keep this surface deliberately narrow: each interface has a fixed direction, a single mediator, and carries only a well-typed payload. Compared to native Gramine-VM [46], which already exposes virtio-blk/-fs for storage and virtio-vsock for network, TIRITH introduces four new interfaces (I1–I4) for graphics, windowing, input, and host attestation. The two subsections that follow analyze the host-privacy and game-security guarantees.

7.1 Host Privacy Protection

The game developer's anti-cheat executes with full kernel privileges inside the guest. A malicious developer could attempt to extract data from the host by:

- *Directly reading host memory of processes or devices or issuing DMA from a guest-controllable device;*

The guest executes within a Protected VM (PVM) whose CPU state and memory are isolated from the host by the hypervisor's Extended Page Tables, and whose DMA-capable devices are constrained by the IOMMU and interrupt-remapping. Specifically, the guest cannot leverage the CPU to read or write to any memory region that is not shared with it. Moreover, the guest does not have direct access to any physical device to launch DMA attacks; all of its devices are paravirtualized and hence mediated (next heading).

- *Abusing the in-band interfaces of Table 2 to leak host state or perform unsanctioned host actions.*

Each interface in Table 2 is constrained by its mediation checks so that it cannot leak unrelated host state or coerce unsanctioned

host actions. In particular, the guest can try to leak host data using the message-passing interfaces (I1) (e.g., Boomerang [24]). The host-side sanitizer bounds-checks every pointer against the guest's allocated regions to block these attacks. For the bridged windowing calls (I2), the bridge maintains an allow-list of lifecycle API (create/resize/destroy/swap) operating on game's own window: for instance, the bridge refuses any cross-window or top-level overlay operation that would let the game to draw on or read from other host windows and break user privacy. For the shared GEM regions (I3), the trusted hypervisor projects only host-allocated GEM objects into a pre-reserved guest physical region; the guest cannot enlarge this mapping or redirect it to other host regions. Last, for the trusted-monitor hypercall (I4), the hypervisor exposes only an allowlisted set of platform-measurement, driver-enumeration, policy-tightening, and attestation-digest queries; it returns measurement results rather than raw host state and never executes developer-supplied code on the host.

7.2 Game Security Enforcement

The player retains administrative privileges on the host, but remains constrained by the trusted hypervisor from gaining malicious access to game state. A malicious player can try to do this by:

- *Reading or modifying the guest's memory or devices using host privilege (e.g., debuggers, drivers, or DMA devices);*

Like in the opposite direction, the hypervisor's memory isolation prevents the host from mapping the guest's pages, debugging across the boundary, or DMAing into guest memory.

- *Tampering with the host kernel and modifying important kernel functionality (e.g., using malicious drivers);*

The hypervisor also measures the initial state of the host kernel (i.e., during secure boot) and tracks which drivers are being executed on the kernel. Therefore, if the user changes system configurations or installs malicious drivers, it will be reported (§6.1).

- *Abusing file system access to load malicious files and network interfaces to tamper with network packets;*

Each file loaded into the guest is integrity-checked using an integrity hash contained within the signed manifest provided by the developer, catching any tampering attempts. Furthermore, network communication is end-to-end encrypted between the guest and the game server using remote attestation and TLS connections.

- *Booting without the trusted hypervisor;*

Remote attestation of the TPM-anchored boot measurement lets the game server detect a non-virtualization boot at session start.

7.3 Cheat Prevention Case Studies

TIRITH preserves every enforcement category of today's kernel-level anti-cheats (Table 1). The system-level enforcement is delegated to the trusted hypervisor and attested by the game server at session start. The game-specific enforcement is, as in today's anti-cheats, already implemented as components packaged with the game client (§2.1); in TIRITH it ships unchanged into the guest, where the developer holds full authority. To further illustrate this,

Table 2: Interfaces newly introduced by TIRITH between PVM and the host. “G”: PVM; “H”: host.

#	Interface	Carries	Potential Attacks	Mediation and Checks
I1	Exception-less message-passing (G→H, §6.2)	Forwarded <code>ioctl</code> / graphics syscalls	Boomerang-style pointer args reading host memory [24]	Host wrapper bounds-checks pointers against G’s identity-mapped data region
I2	Bridged windowing calls (G→H, §6.2)	Lifecycle calls (create / resize / destroy / swap)	Screenshot of unrelated host windows [2]; cross-window overlays	Allowlisted lifecycle API on G’s own window; no read-back call; no cross-window overlay
I3	Shared GEM data plane (H→G, §6.2)	Per-process host-allocated GPU buffers	Projecting attacker-chosen host pages into the guest	Hypervisor projects only host-allocated GEMs into a pre-reserved region; not enlargeable
I4	Trusted-monitor hypercalls (G→H, §6.1)	Attestation digests, host driver verification, policy enforcement	Excessive host introspection or unsanctioned host actions	Allowlisted query/policy-tightening set; returns signed measurement digests, not raw host state; no host-side code exec

we have designed systematic PoC tests that simulate attack behavior and show how TIRITH enforces protection against three well-known classes of cheats attempted on competitive games.

Wallhacks [47]. These attacks allow players to shoot across walls, and are perpetrated by intercepting the game’s OpenGL library (e.g., DLL injection) to tamper with depth-testing operations. With TIRITH, the library executes inside the protected guest which cannot be debugged (C3). Only developer-specified modules and libraries can be loaded into the guest (enforced through the manifest), and these are verified through integrity checks before being enabled. Our PoC test tried to install a custom `.so` file to intercept graphic library calls (OpenGL) through `LD_PRELOAD`. The LibOS flagged the `.so` file and terminated the guest.

Aimbots and triggerbots [43]. Memory-based aimbots reverse-engineer in-memory player coordinates from outside the game to compute aim, or install a host-side hook that injects synthetic mouse/keyboard events. Out-of-VM memory access is blocked by TIRITH; in-guest memory inspection by an injected module is further rejected by the LibOS’ FS/module verification. Our PoC test to simulate this cheat attempts to reverse-engineer in-memory player coordinates from an attack process through `process_vm_readv`. It was unable to read isolated VM memory and thus the attack fails.

It is worth noting that cheaters could leverage visual models (e.g., object-detection on display) and implement visual-aimbots via synthetic inputs. Like kernel-level solutions, TIRITH does not directly address this attack, rather it retains the information that anti-cheats rely on for visual aimbot prevention. Contemporary anti-cheat systems primarily thwart visual aimbots by detecting unapproved third-party hardware/drivers and applying data-driven heuristic or machine learning analysis to player event trajectories [70]. TIRITH preserves both protection mechanisms. The hypervisor continuously monitors the host to detect and report unauthorized hardware or driver interfaces to the game server. Furthermore, runtime game data collection, including in-game event logging and screen capture mechanisms, functions without modification inside the PVM, ensuring developers retain full telemetry to analyze player behavior. **Map hacks [23, 43].** RTS/RPG map hacks lift the fog of war by either correlating in-memory state with network packets or by sniffing the packets through a malicious kernel module on the host network stack (e.g., *PUBG-Radar*). The host’s driver-admission policy refuses unsigned modules (can be verified by the trusted hypercalls), foreclosing the standard malicious-driver vector. Moreover, host network stack would only see packets that are end-to-end encrypted between the game server and the guest. Our PoC test for

simulating this cheat relies on having an unsigned kernel module installed at the host to hook host network stack. Our hypervisor detected and flagged this module.

8 Implementation

We built a TIRITH prototype for AMD x86 systems running Linux with the KVM hypervisor backend. We could not use protected KVM (*pKVM*) [40], which is the new hypervisor backend for supporting Protected VMs, because it is currently under-development [51] and being ported from the Android (ARM) kernel [9]. Benchmarks indicate that *pKVM* incurs a small performance penalty compared to KVM, and mainly during initial boot [59]. Prior work has similarly emulated PVM infrastructure [34], given emerging deployment of *pKVM*. The rest of this section describes our implementation of the guest LibOS kernel and host virtual machine manager (VMM).

Our LibOS kernel implementation is derived from the industry-deployed Gramine [46, 75] (6.2k LoC changes). Other LibOS-like implementations (e.g., UniKraft [45]) can be leveraged in the future. Gramine is already POSIX-compatible and supports over 170 system calls. We added 13 new system calls (e.g., `mremap`) to achieve compatibility with Simple DirectMedia Layer (SDL) library. SDL is used by many Linux native games, therefore our compatibility with SDL minimizes the work needed to support similar games.

To support our efficient graphics pipeline (§6.2), we wrote two helper libraries that are loaded into the LibOS during boot. The rendering library currently supports the Open Graphics Library (OpenGL) standard (compliant with version 4.6). It intercepts OpenGL context creation (`glCreateContext`) and translates them to the newer EGL backend [72] that is designed to render without display integration. Another library is responsible for direct display in our pipeline by intercepting userspace relevant OpenGL APIs (e.g., `SwapBuffers`). It also manages input redirection.

We built the host-side implementation using QEMU v10.1 as our VMM. We built our *virtual GPU (vGPU)* as a new paravirtualized device. This listener handles all the commands passed from the guest. It serves syscalls requests from the guest and executes them via a lockless mechanism to ensure performance in multi-threaded game environments by spawning multiple listener threads as required. In addition to syscalls commands, the listener module serves requests from the guest to execute display related calls, (e.g., `xcb_present`, `xcb_resize`). We also updated the QEMU memory backend to request projected address space mapping (§6.2) from the hypervisor when the guest starts. Our total implementation for the QEMU-side was approximately 1k LoC.

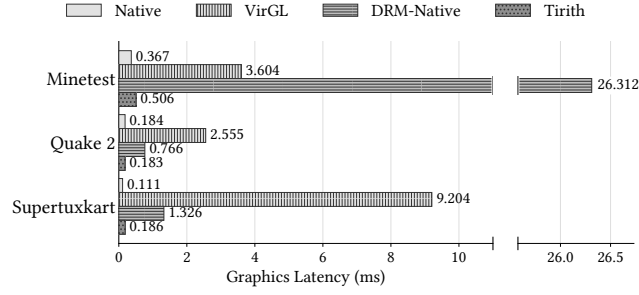


Figure 7: Graphics pipeline latency at 1080p compared across all environments and games.

9 Performance Evaluation

Setup. We ran all our experiments on a desktop machine with AMD Ryzen 5600X, 32 GB DDR4 RAM, and an AMD ATI Radeon RX 6950 XT with 16GiB GDDR6. The system runs Arch Linux with kernel version 6.19.11. All game virtual machines (including those running TIRITH and other baselines) were assigned 1 vCPU and 8GiB of memory. A single vCPU was used due to a current limitation in the Gramine LibOS for AMD CPUs: the kernel is originally designed for Intel CPUs and leverages Intel-specific x2APIC virtualization features. On AMD CPUs, it reverts to the older xAPIC virtualization which significantly degrades performance for all multi-threaded programs, including ones that do not use graphics. Our experiments on the graphics pipeline (outside the guest) and results from prior work [57, 68] show that techniques like exception-less message-passing that TIRITH leverages scale nicely across threads.

Comparison. We compared TIRITH against four main baseline solutions. First, we implemented an enhanced bare-metal solution (**Native**), which leverages TIRITH’s EGL backend (§8). We found that this solution had better performance than default game backends in *all* scenarios. Second, we ran the games in a raw bare-metal configuration (**Unmodified Native**), to show the performance benefits TIRITH gains from the aforementioned EGL backend. Third, we ran the default Linux graphical configuration for setting up paravirtualized graphics (**VirGL**) [74]. Fourth, we ran the experimental concurrent work that represents the cutting-edge of graphics virtualization performance (**Native-DRM**) [36]. VirGL and DRM-Native were executed on the state-of-the-art *muvm*. For running our games, we leveraged 720p, 1080p, and 1440p resolutions.

9.1 Micro-Benchmarks

Initialization and Memory Cost. TIRITH incurs a set of initial costs while starting a game session. TIRITH must set-up address space shadowing and exception-less message-passing in QEMU (2.9s), initialize the LibOS kernel and paravirtualized devices (56ms), as well as create rendering contexts (i.e., set up framebuffers, map GEM objects) and initial window manager contexts (16ms) (§6.2). Across these initialization steps, TIRITH only took an average of 3.1s. The major cost incurred was for address space shadowing while setting up QEMU, since it requires memory remapping. These costs are only one-time and amortized over long-running game sessions. In addition, TIRITH adds minor additional memory overhead

for various data structures maintained in the LibOS. In our measurements, we found an average of 4.5-12.6MiB memory overhead across games, which is negligible.

Exception-Less Message Passing Cost. At runtime, TIRITH’s graphics engine depends heavily on the exception-less message passing rings, therefore we measured the round-trip cost for passing an empty message. This was measured across 100M messages and only took an average of 254ns per-message. This lightweight interface is primarily because message-passing occurs over shared memory and leverages background threads that are woken-up.

Additional Input latency. Input latency is critical for competitive games that have high frame-rates. Precisely measuring the entire end-to-end latency (e.g., from a physical mouse click to its visible effect on screen) requires external instrumentation such as a high-speed camera, which we currently lack. However, we evaluated input latency in TIRITH using virtual mouse clicks for both a game and a custom benchmark. The clicks are generated by a virtual mouse created through Linux `uinput` and traverse the normal compositor and X11 input path.

First, we ran Quake 2 and measured the delay from a virtual mouse click into its window until the game processes the input by triggering an attack. Across 1,000 clicks, TIRITH increases average end-to-end input latency from 584μs to 609μs, resulting in an overhead of 25μs. Of this, 20μs comes from delivering the input event to QEMU, encoding it, and forwarding it to Gramine through shared memory. The remaining 5μs comes from delivering the event to Quake 2. Once delivered to the game, processing the event introduces little to no measurable overhead.

Second, we evaluated the input path with a complementary GLX application that continuously polls for input. Across 1,000 clicks, TIRITH increases average end-to-end input latency from 105μs to 178μs, resulting in an overhead of 73μs. Nearly all of this overhead comes from delivering the input event to QEMU, encoding it, and forwarding it to Gramine LibOS through shared memory, while subsequent delivery to the application and event processing introduce little to no measurable overhead. By continuously polling, the experiment exposes virtualized input latency that could otherwise be masked by a game’s less frequent input polling.

9.2 Real-World Video Game Performance

Game Selection and Benchmarks. In terms of game selection, we opted to evaluate all environments on a suite of open-source games with OpenGL rendering back-ends and one closed-source game. This allows for a deeper understanding of the current performance limitations of TIRITH and existing solutions in well known and highly auditable systems.

We selected four games to evaluate the performance of TIRITH against existing paravirtualization solutions. First, we chose Minetest, a voxel-based, randomly generated open world game, a style of game notorious for stressing systems on high settings. Second, we chose Quake 2, a classic first-person shooter which demands low latency feedback for the player to make fast-paced combat decisions. Third, we selected Supertuxkart which is a single or multi-player kart racing game. Finally, we selected Team Fortress 2 which is a competitive online first-person shooter that leverages a modern anti-cheat system called VAC [70] which is used in popular

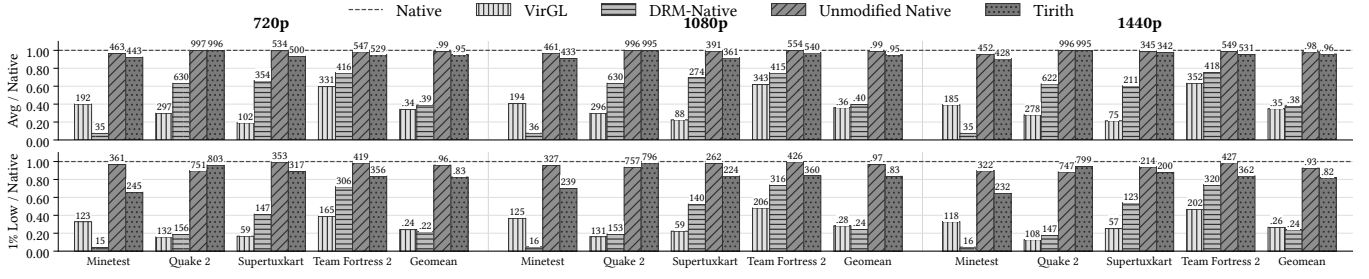


Figure 8: Performance relative to Native and Unmodified Native shown in terms of average FPS and 1% lows. 1% lows reflect frame-time consistency and the severity of FPS dips. Geomean bar labels show relative values; other bar labels show raw FPS.

competitive games like Dota 2 and Counter-Strike 2. We disabled VAC to avoid the system being flagged.

For the purposes of selecting benchmarks for our selection of games, Quake 2 and Supertuxkart offer built-in benchmarking features in which either the computer plays the game (Supertuxkart), or the game plays back recorded gameplay sequences (Quake 2). For Minetest, a constant seed and spawn point were chosen for the world, in which the player was spawned in and sat still. For Team Fortress 2, the player was spawned in a consistent spot on a single map and sat still while the environment changed.

The games were ran on at or near maximum graphics settings for at least 5 minutes, with the data presented being the average of all data at the 5 minute mark. This amount of time allows the games to finish loading assets and in the case of minetest, finish generating local world data. Such concurrent loading operations run alongside rendering, and allowing time for their completion and maximizing graphics settings emphasizes rendering performance.

Results and Breakdown Analysis. In terms of gaming performance, as shown in Fig. 8 TIRITH outperformed all existing paravirtualization solutions, achieving only 4.1-5.2% geometric mean overhead compared to **Native** in average FPS which is 2.4-2.8× better than other solutions. This positive trend continues in 1% lows, in which TIRITH sees 16.5-18.2% overhead compared to **Native**, which is 3-3.8× better compared to existing work.

Starting with Supertuxkart and Quake 2, we see that TIRITH sees 1.3-7.4× better average FPS and 1% lows than prior works, incurring only 0.1-8.5% overhead compared to **Native**. Even in the worst case across these games, a user may experience only a minor loss of 11.1-16.3% overhead in 1% lows when compared to **Native**. TIRITH showcases similar performance and viability when utilized to run the competitive and modern shooter Team Fortress 2. With average FPS and 1% lows that incur only 2.7-4.9% and 15.8-16.5% overhead compared to **Native** respectively. This results in achieving a 1.1-2.2× better performance than existing work.

Particularly noteworthy are the Minetest results. TIRITH makes a solid showing with average FPS and 1% lows, only 7.7 – 9.4% and 30.1-35.4% overhead compared to **Native** respectively. This results in achieving a staggering result of 1.9-16.4× better performance than prior solutions. Another observation is that DRM-Native performs very poorly (4-7.5% of **Native**), when compared with its prior results, it is a clear outlier that requires further investigation.

Analyzing **Native** versus **Unmodified Native**, we see worthwhile gains of 1.1-1.8% and 3.5-8.1% in average FPS and 1% lows

respectively. This increased performance is attributed to the EGL backend changes described in §8. As a result, we see TIRITH close the gap even further to **Unmodified Native**—bringing the performance overhead to only 2.4-4.1% and 11.5-13.8% for average FPS and 1% lows, respectively, for our tested games.

Additional analysis was done on *Graphics Latency* (Fig. 7). We define *Graphics Latency* as graphics pipeline overhead including CPU overhead from copying data and sending encoded data across a paravirtualized interface (§5.1). TIRITH only sees a maximum increase of 0.14ms over **Native**, other solutions lag behind substantially at a maximum latency increase of 25ms. The excellent performance of TIRITH is due to the zero-copy nature of the Native Shared GEM Context (§6.2), which minimizes CPU overhead in the graphics pipeline incurred by existing solutions.

9.3 Key Takeaways

- TIRITH achieves average FPS and 1% lows within 4.1-5.2% and 16.5-18.2% of **Native** and 2.4-4.1% and 11.5-13.8% of **Unmodified Native**.
- TIRITH performs 2.4-2.8× and 3-3.8× better than existing paravirtualization solutions in average FPS and 1% lows.

Note that existing anti-cheat solutions already incur performance overhead within the range of 5-10% and even large amounts of system instability as reported by users across many games [21, 60–62]. In that regards, TIRITH’s overhead is similar to kernel-level anti-cheats, while offering strong user privacy as outlined in §7.1. Moreover, TIRITH provides equivalent protections against cheating mechanisms as compared to kernel-level solutions. Therefore, TIRITH as an end-to-end anti-cheat solution, is a compelling approach to address the gap between user privacy and cheat prevention.

10 Related Work

TIRITH is part of a broader line of work on isolating code that runs on a machine whose root user is untrusted, without granting that code unrestricted host privilege. This problem has been approached over the years with progressively coarser isolation primitives, and we organize the relevant prior work along that trend.

The first trend used user-space *trusted execution environments* (TEEs), most notably Intel SGX. Haven [18] demonstrated how to shield unmodified Windows applications inside an enclave. Graphene-SGX [76], Panoply [67], and Scone [14] extended the same approach to Linux applications and containers. Ryoan [19]

built a distributed sandbox for untrusted computation on secret data. These designs share a common shape: the protected workload runs in a user-space enclave that is isolated from a compromised host kernel. However, they cannot themselves provide kernel-level functionality such as driver admission or anti-debugging, and so cannot replace a privileged kernel anti-cheat on the host.

A more recent trend on consumer machines moves the isolation boundary up to whole-VM virtualization. Microsoft Virtualization-Based Security (VBS) Enclaves [10] use the Hyper-V hypervisor to isolate sensitive workloads from a host Windows kernel, and Android Protected KVM [9, 40] provisions Protected VMs on consumer phones with hardware-enforced two-way isolation. TwinVisor [48] achieves the same property on ARM by leveraging TrustZone.

11 Conclusion

We presented TIRITH, a virtualization-based architecture for private and secure PC gaming that replaces today's kernel-level anti-cheats with a two-way Protected VM. By splitting responsibility between the host's first-party virtualization monitor and the game developer's code, TIRITH preserves the full set of enforcement primitives kernel anti-cheats provide today while removing the host-side privacy concern. With the native shared-GEM rendering and bridged split-windowing pipelines, TIRITH delivers near-native graphics and display performance. Our evaluations on real games and cheating mechanisms show that TIRITH can achieve strong security guarantees with modest performance overheads.

Acknowledgements

We thank our shepherd and reviewers for their insightful comments. This work was partly supported by the U.S. National Science Foundation (NSF) under Cooperative Agreement No. 2543639.

References

- [1] Anti-Cheat - FACEIT.com. <https://www.faceit.com/en/anti-cheat>.
- [2] ANTI-CHEAT EXPERT: ALL YOUR PIXELS ARE BELONG TO US. https://invlp.dev/post/ace_screenshots/.
- [3] Easy Anti-Cheat. <https://www.easy.ac/en-US>.
- [4] Intel trusted domain extensions. <https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-final9-17.pdf>.
- [5] Is a Virtual Machine Good for Gaming? A Comprehensive Guide. <https://cyfuture.cloud/kb/gaming/is-a-virtual-machine-good-for-gaming#:~:text=Virtual%20machines%20provide%20an%20isolated,impacting%20your%20primary%20operating%20system>.
- [6] Microsoft's updated anti-cheating measures go far beyond Secure Boot, including 'Remote Attestation' for online system verification. <https://www.pcgamer.com/software/microsofts-updated-anti-cheating-measures-go-far-beyond-secure-boot-including-remote-attestation-for-online-system-verification>.
- [7] RED SHELL Spyware - "Holy Potatoes! We're in Space?!" integrated and removed it after complaints. <https://www.reddit.com/r/Steam/comments/8pud8b/comment/e0e6uy1/>.
- [8] System Guard: How a hardware-based root of trust helps protect Windows. <https://learn.microsoft.com/en-us/windows/security/hardware-security/how-hardware-based-root-of-trust-helps-protect-windows>.
- [9] Virtual Machine as a core Android Primitive. <https://android-developers.googleblog.com/2023/12/virtual-machines-as-core-android-primitive.html?m=1>.
- [10] Virtualization-based security (VBS) enclaves. <https://learn.microsoft.com/en-us/windows/win32/trusted-execution/vbs-enclaves>.
- [11] virtio-fs: A Shared File System for Virtual Machines. <https://virtio-fs.gitlab.io/>, 2019.
- [12] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434. USENIX Association, 2020.
- [13] Hashem Alayed, Fotos Frangoudes, and Clifford Neuman. Behavioral-based cheating detection in online first person shooters using machine learning techniques. In *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8, 2013.
- [14] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumar, Dan O'Keeffe, Mark Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.
- [15] Asahi Linux Project. mvm: Micro-VMs for Linux desktop applications. <https://github.com/AsahiLinux/mvm>, 2026. Accessed: 2026-04-29.
- [16] BattlEye Innovations. About: BattlEye. <https://www.battleye.com/about/>, 2024. Accessed: 2024-05-28.
- [17] Erick Bauman and Zhiqiang Lin. A case for protecting computer games with sgx. In *Proceedings of the 1st Workshop on System Software for Trusted Execution, SysTEX '16*, New York, NY, USA, 2016. Association for Computing Machinery.
- [18] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. 33(3), August 2015.
- [19] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. 33(3), August 2015.
- [20] Bitdefender. What is Bring Your Own Vulnerable Driver (BYOVD)? <https://techzone.bitdefender.com/en/tech-explainers/what-is-bring-your-own-vulnerable-driver--byovd-.html>, 2026.
- [21] Blizzard Entertainment Community. New anticheat is the cause of widespread fps issues. <https://us.forums.blizzard.com/en/overwatch/t/new-anticheat-is-the-cause-of-widespread-fps-issues/1010084>, March 2026. Technical discussion on FPS degradation and input latency following anti-cheat updates. Accessed: 2026-04-29.
- [22] Tim Burke. Red Hat Blog – UEFI Secure Boot. <https://www.redhat.com/en/blog/uefi-secure-boot>, June 2012.
- [23] Elie Bursztein, Mike Hamburg, Jocelyn Lagarenne, and Dan Boneh. Openconflict: Preventing real time map hacks in online games. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP)*, pages 506–520. IEEE, 2011.
- [24] Yung Ryn Choe, Aravind Machiry, Eric D Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, 2017.
- [25] Minyeop Choi, Gihyuk Ko, and Sang Kil Cha. Botscreen: trust everybody, but cut the aimbots yourself. In *Proceedings of the 32nd USENIX Conference on Security Symposium, SEC '23*, USA, 2023. USENIX Association.
- [26] Sam Collins, Alex Pouloupoulos, Marius Muench, and Tom Chothia. Anti-cheat: Attacks and the effectiveness of client-side defences. In *Proceedings of the 2024 Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*, 2024.
- [27] Containers Project. libkrun: A dynamic library providing Virtualization-based process isolation capabilities. <https://github.com/containers/libkrun>, 2026. Accessed: 2026-04-25.
- [28] Adam Conway. Proton is so good these days, I wish I could make the switch to Linux. <https://www.xda-developers.com/proton-linux-wish-switch/>, May 2025. XDA Developers. Accessed: 2026-04-29.
- [29] Christoph Dörner and Lukas Daniel Klausner. If it looks like a rootkit and deceives like a rootkit: A critical examination of kernel-level anti-cheat systems. In *Proceedings of the 19th International Conference on Availability, Reliability and Security, ARES 2024*, 2024.
- [30] EngineOwning. Insights regarding the recent detection / ban wave. <https://www.engineowning.ru/threads/insights-regarding-the-recent-detection-ban-wave.115740/>, 2024. Technical forum post discussing anti-cheat detection and kernel-level visibility. Accessed: 2026-04-29.
- [31] Epic Games. Using the anti-cheat interfaces. <https://dev.epicgames.com/docs/epic-online-services/trust-and-safety/anti-cheat-interfaces/using-anti-cheat#anti-cheat-integrity-tool-configuration>.
- [32] Fedora Project Community. Kernel-level anticheat and Linux: How it works. <https://discussion.fedoraproject.org/t/kernel-level-anticheat-and-linux-how-it-works/162491>, 2026. Technical community discussion regarding kernel integrity and game security. Accessed: 2026-04-29.
- [33] Riot Games. What is vanguard? <https://support.valorant.riotgames.com/hc/en-us/articles/360046160933-What-is-Vanguard>, January 2024. Accessed: 2024-05-21.
- [34] Varun Gandhi, Sarbartha Banerjee, Aniket Agrawal, Adil Ahmad, Sangho Lee, and Marcus Peinado. Rethinking system audit architectures for high event coverage and synchronous log availability. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 391–408. USENIX Association, 2023.
- [35] GDC. Robocalypse now: Using deep learning to combat cheating in counter-strike: Global offensive. <https://www.youtube.com/watch?v=kTiP0zKF9bc>, 2020. Accessed: 2024-05-28.

- [36] Gervais, R. and Viperman, K. VirtIO-GPU DRM Native Context. https://indico.freedesktop.org/event/2/contributions/53/attachments/76/121/XDC2022_%20virtgpu%20drm%20native%20context.pdf, 2022.
- [37] Andreas Haebler, Paarijat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.
- [38] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, Carlsbad, CA, July 2022. USENIX Association.
- [39] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide, 2016.
- [40] Intel Jason Chen. Supporting TEE on x86 Client Platforms with pKVM. https://www.youtube.com/watch?v=EP9ps_h-WeI.
- [41] Sanghoon Jeon and Huy Kang Kim. Tzmon: Improving mobile game security with arm trustzone. *Comput. Secur.*, 2021.
- [42] Sukrit Kalra, Rishabh Sanghi, and Mohan Dhawan. Blockchain-based real-time cheat prevention and robustness for multi-player online games. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, 2018.
- [43] Anssi Kanervisto, Tomi H. Kinnunen, and Ville Hautamäki. Gan-aimbots: Using machine learning for cheating in first person shooters. *IEEE Transactions on Games*, 15:566–579, 2022.
- [44] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: exploiting speculative execution. *Commun. ACM*, 63(7):93–101, June 2020.
- [45] Simon Kuenzer, Vlad-Andrei Badoiu, Hugo Lefevre, Sharan Santhanam, Alexander Jung, Gauthier Gain, Cyril Soldani, Costin Lupu, Stefan Teodorescu, Costi Raducanu, Cristian Banu, Laurent Mathy, Razvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*. ACM, 2021.
- [46] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. Gramine-tdx: A lightweight os kernel for confidential vms. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, CCS '24. Association for Computing Machinery, 2024.
- [47] Peter Laurens, Richard F. Paige, Phillip J. Brooke, and Howard Chivers. A Novel Approach to the Detection of Cheating in Multiplayer Online Games. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*. IEEE Computer Society, 2007.
- [48] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. Twin-Visor: Hardware-isolated Confidential Virtual Machines for ARM. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.
- [49] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.
- [50] Linux Kernel Development Community. The Linux Input Subsystem. <https://www.kernel.org/doc/html/latest/input/input.html>, 2026. Accessed: 2026-04-29.
- [51] Linux Kernel Organization. Protected KVM (pKVM) on Arm64. <https://www.kernel.org/doc/html/latest/virt/kvm/arm/pkvm.html>, 2026. Official Linux Kernel Documentation. Accessed: 2026-04-29.
- [52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: reading kernel memory from user space. *Commun. ACM*, 63(6):46–56, May 2020.
- [53] Microsoft. Microsoft Learn - HealthAttestation CSP. <https://learn.microsoft.com/en-us/windows/client-management/mdm/healthattestation-csp>, July 2025.
- [54] Microsoft Corporation. Virtualization-based Security (VBS) | Microsoft Learn. <https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs>, 2024. Details the use of the Windows hypervisor to create an isolated root of trust. Accessed: 2026-04-29.
- [55] Kit Murdoch, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.
- [56] Newzoo. Year in review: 2025 to date. <https://newzoo.com/resources/blog/year-in-review-2025-to-date>, 2025. Accessed: 2026-04-29.
- [57] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys '17*, page 238–253. Association for Computing Machinery, 2017.
- [58] Seonghyun Park, Adil Ahmad, and Byoungyoung Lee. Blackmirror: Preventing wallhacks in 3d online fps games. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 987–1000, New York, NY, USA, 2020. Association for Computing Machinery.
- [59] Quentin Perret. Protected KVM on Arm64: A Technical Deep Dive. KVM Forum 2022. Available at: <https://www.youtube.com/watch?v=9npebeVFbFw>, 2022. Technical lecture on the architecture of Protected KVM and EL2 deprive. Accessed: 2026-04-29.
- [60] Reddit r/FACEITcom. FACEIT AC cause huge fps drop. https://www.reddit.com/r/FACEITcom/comments/1oni98o/faceit_ac_cause_huge_fps_drop/, August 2024. User discussion describing major FPS drops associated with FACEIT Anti-Cheat. Accessed: 2026-04-29.
- [61] Reddit r/riotgames. LoL Vanguard impacted my performance a lot. https://www.reddit.com/r/riotgames/comments/1doh7iz/loL_vanguard_impacted_my_performance_a_lot/, June 2024. User discussion reporting significant game performance impact after Vanguard. Accessed: 2026-04-29.
- [62] Reddit r/techsupport. Blue Screen When Launching Rust / Easy Anti-Cheat. https://www.reddit.com/r/techsupport/comments/1jjvp00/blue_screen_when_launching_rust_easy_anticheat/, March 2025. User support thread describing blue-screen crashes when launching Rust as Easy Anti-Cheat starts. Accessed: 2026-04-29.
- [63] Riot Games. Riot vanguard faq - league of legends. <https://support-leagueoflegends.riotgames.com/hc/en-us/articles/24169857932435-Riot-Vanguard-FAQ-League-of-Legends>, 2024. Accessed: 2024-05-31.
- [64] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.
- [65] s4dbird. How kernel anti-cheats work. <https://s4dbird.github.io/posts/how-kernel-anti-cheats-work/#ai-powered-cheats>, 2023.
- [66] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. *SIGOPS Oper. Syst. Rev.*, 41(6):335–350, October 2007.
- [67] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.
- [68] Livio Soares and Michael Stumm. Flexsc: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.
- [69] Ryan Soliven and Hitomi Kimura. Ransomware actor abuses Genshin Impact anti-cheat driver to kill antivirus. https://www.trendmicro.com/en_us/research/22/h/ransomware-actor-abuses-genshin-impact-anti-cheat-driver-to-kill-antivirus.html, August 2022.
- [70] Steam Support. Steam support: Valve anti-cheat (vac) system. <https://help.steampowered.com/en/faqs/view/571A-97DA-70E9-FF74>, 2024. Accessed: 2024-05-28.
- [71] Chenxin Sun, Kai Ye, Liangcai Su, Jiayi Zhang, and Chenxiong Qian. Invisibility cloak: Proactive defense against visual game cheating. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.
- [72] The Khronos Group. EGL Native Platform Interface. <https://www.khronos.org/egl/>, 2024. Standardized interface between Khronos rendering APIs and the underlying native platform window system.
- [73] The Linux Kernel Development Community. Dm internals. <https://www.kernel.org/doc/html/v4.16/gpu/drm-internals.html>, 2018.
- [74] The Mesa 3D Graphics Library. VirGL: Virtual 3D GPU for QEMU. <https://docs.mesa3d.org/drivers/virgl.html>, 2026. Accessed: 2026-04-29.
- [75] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, April 2014.
- [76] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, June 2017.
- [77] Wayland Project. The Wayland Protocol: Architecture. <https://wayland.freedesktop.org/docs/book/Architecture.html>, 2026.
- [78] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupe, Yan Shoshitaishvili, and Tiffany Bao. Retspill: Igniting user-controlled data to burn away linux kernel protections. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

A Artifact Appendix

A.1 Abstract

TIRITH's source code is publicly available, and its execution environment can be fully reproduced. In this appendix, we describe the system specifications, hardware/software dependencies, containerized build procedures, host proxy setup, and evaluation workflows to reproduce the gaming and graphics benchmark results presented in the paper.

A.2 Description and Requirements

A.2.1 How to Access. We maintain a public GitHub repository for artifact evaluation and reproducibility:

- **GitHub Repository:** <https://github.com/ASTERISC-Release/Tirith>
- **Zenodo Repository:** <https://zenodo.org/records/22836974>

A.2.2 Hardware Dependencies. TIRITH was evaluated and verified on the following hardware configuration:

- **Host CPU:** AMD x86-64 processor (e.g., AMD Ryzen 5 5600X with 12 threads) supporting AMD-V / SVM virtualization extensions.
- **Host GPU:** Discrete AMD GPU (e.g., AMD Radeon RX 6950 XT) with Linux DRM/KMS support for OpenGL 4.6. *(Note: Non-AMD CPUs or non-discrete AMD GPUs are untested)*.
- **RAM & Storage:** 32 GB RAM and at least 50 GB available SSD storage.

A.2.3 Software Dependencies.

- **Host OS:** Linux (Ubuntu 20.04+, 22.04+, 24.04+ or Arch Linux, Linux Kernel 6.19+).
- **Container Engine:** Podman and podman-compose (podman.socket enabled).
- **Host Display Server:** X11 or XWayland display server (with xhost access control utility).
- **Container Toolchain Components:**
 - Modified QEMU VMM (qemu) with paravirtualized vGPU listener.
 - Extended Gramine LibOS (gramine-tdx).
 - Modified Mesa3D graphics driver stack (mesa3d).
 - OpenGL to EGL translation library (egl-redirect).
- **Steam and account setup:**
 - Steam must be installed through your package manager.
 - You must have a steam account with the free copy of Team Fortress 2 claimed.

A.2.4 Benchmarks & Workloads.

- **Graphics Benchmark:** glxgears-with-lows for frame rates and low-level graphics pipeline testing.
- **Game Workloads:** Linux-native OpenGL games evaluated in Section 9: Quake 2, Supertuxkart, Minetest, and Team Fortress 2.

A.3 Set-up & Build Instructions

A.3.1 Container Build and Toolchain Compilation. 1. **Build and enter the Podman container:**

```
human@asterisc:~/Tirith$ cd docker && ./
build.sh
human@asterisc:~/Tirith/docker$ ./exec.sh
```

2. Build components inside the container shell:

```
# Build QEMU VMM
tirith@container:~$ cd ~/Tirith/qemu && ./
qemu-install.sh

# Build Gramine LibOS
tirith@container:~$ cd ~/Tirith/gramine-tdx
&& ./gramine-install.sh

# Build Mesa3D
tirith@container:~$ cd ~/Tirith/mesa3d &&
./mesa-install.sh

# Build OpenGL-EGL Translation Library
tirith@container:~$ cd ~/Tirith/
egl-redirect && make
```

A.3.2 System Settings & Host Window Forwarding Setup. Before running game benchmarks, start the host window proxy script in a separate terminal outside the container:

```
human@asterisc:~/Tirith$ cd scripts/
human@asterisc:~/Tirith/scripts$ xhost +
human@asterisc:~/Tirith/scripts$ ./
host-proxy-xwayland.sh
# Use ./host-proxy.sh if using native X11
```

Note: This proxy process must remain running on the host for the duration of the evaluation.

A.3.3 Steam startup & Host steam Forwarding Setup. Before running game benchmarks, start Steam on the host and start the host steam proxy script in a separate terminal outside the container:

```
# Start Steam on host with app launcher of
your choice and login before next step
.
human@asterisc:~/Tirith$ cd scripts/
human@asterisc:~/Tirith/scripts$ ./
host-steam-proxy.sh
```

Note: This proxy process and Steam must remain running on the host for the duration of the evaluation of Team Fortress 2.

A.4 Evaluation Workflow

A.4.1 Major Claims.

- **(C1):** TIRITH's containerized build environment and host window forwarding proxy can be reliably deployed.

- **(C2):** TIRITH supports Linux-native OpenGL games (Quake 2, Supertuxkart, Minetest and TF2) with low rendering overhead and consistent FPS presentation (Experiments E1–E5).

A.4.2 Experiments. We leverage our benchmarks to evaluate TIRITH across the following experiments:

Experiment (E1): Glxgears Graphics Benchmark [EST 5 mins]. Build and execute the glxgears benchmark within the container shell:

```
tirith@container:~$ cd ~/Tirith/gramine-tdx
/CI-Examples/graphics/
glxgears-with-lows
tirith@container:~/glxgears-with-lows$
make
tirith@container:~/glxgears-with-lows$
gramine-vm glxgears
```

Results: The game renders on the host display while FPS performance data is logged to the terminal every minute.

Experiment (E2): Quake 2 Game Workload [EST 10–20 mins]. Build and execute Quake 2 under TIRITH:

```
tirith@container:~$ cd ~/Tirith/gramine-tdx
/CI-Examples/graphics/quake2
tirith@container:~/quake2$ ./setup-quake.
sh

# Initial setup: run native binary once to
maximize graphics
tirith@container:~/quake2$ ./run.sh 0 1
# In game: Press <ESC>, set graphics to
maximum, quit with <ESC> -> <Y>

# Run in Tirith
tirith@container:~/quake2$ make
tirith@container:~/quake2$ gramine-vm
quake2
```

Results: The game renders on the host display while FPS performance data is logged to the terminal every minute.

Experiment (E3): Supertuxkart Game Workload [EST 10–20 mins].

Install and execute Supertuxkart under TIRITH:

```
tirith@container:~$ cd ~/Tirith/gramine-tdx
/CI-Examples/graphics/supertuxkart
tirith@container:~/supertuxkart$ ./
install-stx.sh

# Initial setup: run native binary to
adjust settings
tirith@container:~/supertuxkart$ ./run.sh
0 1
# In game: Click wrench icon, set graphics
level to max, set FPS cap to 1000

# Run in Tirith
tirith@container:~/supertuxkart$ make
tirith@container:~/supertuxkart$
gramine-vm supertuxkart
```

Results: The game renders on the host display, while FPS performance data is logged to the terminal only in benchmark mode at the end of the run.

Experiment (E4): Minetest Game Workload [EST 10–20 mins].

Setup and execute Minetest under TIRITH:

```
tirith@container:~$ cd ~/Tirith/gramine-tdx
/CI-Examples/graphics/minetest
tirith@container:~/minetest$ ./
setup-minetest.sh

# Initial setup: run native binary once
tirith@container:~/minetest$ ./run.sh 0 1
# In game: Create a new map, head to
settings to maximize graphics settings
, then quit

# Run in Tirith
tirith@container:~/minetest$ make
tirith@container:~/minetest$ gramine-vm
minetest

# In game: Click "New" -> "Create" -> "Play
Game"
```

Results: The game renders on the host display while FPS performance data is logged to the terminal every minute.

Experiment (E5): Team Fortress 2 [EST 30–60 mins].

Setup and execute Team Fortress 2 under TIRITH:

```
tirith@container:~$ cd ~/Tirith/gramine-tdx
/CI-Examples/graphics/tf2
tirith@container:.../tf2$ ./setup-tf2.sh
# This next step requires that
  STEAM_ACCOUNT is a username for a
  steam account that has claimed the
  free Team Fortress 2 game on steam.
tirith@container:.../tf2$ ./setup-tf2.sh
  STEAM_ACCOUNT
# You will be prompted for the password
  associated with STEAM_ACCOUNT and 2FA
  confirmation through the steam mobile
  app if enabled.
tirith@container:.../tf2$ ./run-vm.sh
```

Results: The game renders on the host display while FPS performance data is logged to the terminal every minute (the game

may take some time to start the game server and join a team [3–5 minutes]).

A.5 General Notes & Troubleshooting

- (1) Ensure the proxy script (`host-proxy-xwayland.sh` or `host-proxy.sh`) remains running on the host throughout all evaluations. If a game hangs during the startup phase, restart this proxy script on the host.
- (2) FPS metrics are automatically logged to the terminal every minute for most games (and at the end of the run for Super-tuxkart).

B Generative AI Usage

Large Language Models (LLMs) were used for copy-edit purposes to improve the clarity and grammar of our paper writing. Additionally, we used LLMs to help debug our implementation. All usage of LLMs was reviewed by the paper authors.