

Anti-Cheat Compatibility on Apple Silicon

Toward Closing the Anti-Cheat Gap on macOS

NZ-Linux (com.nzlinux)

Machine Learning and Cybersecurity Researcher

Yasmina Serrano (minasierra)

Research Assistant and Software Engineer

September 26, 2026

Abstract

Windows games protected by Easy Anti-Cheat (EAC) present a distinct compatibility problem on Apple Silicon Macs. Conventional Windows compatibility stacks can translate game APIs and instructions, but EAC also depends on the operating-system environment in which its client runs. The macOS Wine route does not provide EAC’s Linux/Proton client environment, while a Windows virtual machine follows the kernel-driver route and may be rejected as a virtual machine.

I describe a third route: run Valve’s ARM64 SteamOS in a lightweight Linux virtual machine on macOS, then execute the complete x86-64 Proton process tree through FEX-Emu. The EAC Linux client is x86-64-only and is injected into the game through Linux `ptrace`; consequently, the translator must reproduce the x86-64 Linux process-tracing, signal, memory, and `proc` behaviours expected by the client. My system combines Hypervisor.framework and libkrnl, a Venus-based Vulkan path to Metal, and a patched FEX build that aims for Linux behavioural fidelity rather than concealment or evasion.

An independent public proof of concept demonstrated VRChat on an M2; I tested a separate DirectX 11 title on an M4 Pro. Together, these demonstrations show successful EAC initialization and online play for two titles across two Apple systems. EAC remained unmodified and the VM and translator remained visible in both demonstrations. I also measured rendering performance, shader-pipeline compilation, and input latency. Cold Metal pipeline compilation is the dominant performance limitation: a persistent host-side cache reduces repeat costs, while my pipeline-library benchmark improves but does not eliminate first-use stalls. I present these findings, boundaries, ethical constraints, and open problems without claiming universal compatibility or publisher support.

Contents

1	Introduction	3
1.1	The compatibility gap	3
1.2	Purpose and contributions	3
1.3	Design principle: fidelity, not evasion	4
2	EAC Operating Modes and Platform Requirements	5
2.1	Windows mode and Linux/Proton mode	5
2.2	Requirements for an Apple Silicon implementation	6
2.3	Why existing macOS routes are insufficient	6
3	System Architecture: SteamOS in a macOS Virtual Machine	7
3.1	Layered design	7
3.2	VM-host goals and virtualization substrate	7
3.3	SteamOS image acquisition and disk construction	8
3.4	Kernel, initramfs, and read-only adaptation layer	8
3.5	CPU, page size, memory ordering, and storage	9
4	Graphics, Devices, and Host Integration	10
4.1	Vulkan forwarding through Venus	10
4.2	MoltenVK and KosmicKrisp	10
4.3	Display and presentation	11
4.4	Input and controller paths	11
4.5	Host-guest services, networking, audio, and sleep	12
4.6	Packaging and x86-64 user space	12
5	Kernel Fidelity and the EAC Injection Path	14
5.1	How the Proton EAC bootstrapper injects its client	14
5.2	Why ordinary host ptrace is not enough	15
5.3	FEX's x86-64 ptrace model	15
5.4	Self-modifying code	16
5.5	Signals, seccomp, and additional Linux semantics	16
6	Demonstrated Results and Failure Analysis	17
6.1	Demonstrations	17
6.2	Launcher error codes as diagnostic evidence	17
6.3	Known unsuccessful or untested cases	18
7	Performance and Pipeline Compilation	19
7.1	Throughput	19
7.2	Why cold graphics pipelines hitch	19
7.3	Pipeline-library feasibility benchmark	20
8	Input Latency	21

8.1	Latency path and measurement	21
8.2	Measured results	21
9	Security, Policy, and Ethics	23
9.1	What the anti-cheat can observe	23
9.2	Operational rules	23
9.3	Policy risk	23
10	Limitations and Threats to Validity	24
11	Open Problems and Future Work	25
11.1	Upstreaming emulator fixes	25
11.2	Reducing Metal pipeline stalls	25
11.3	Rosetta and native ARM64 EAC	25
11.4	Broader compatibility evaluation	26
12	Conclusion	27
A	Glossary	28
B	Reproduction Outline	29
	References	30

Chapter 1

Introduction

1.1 The compatibility gap

Apple Silicon systems provide substantial CPU and GPU capability, and macOS has mature technologies for running software built for other platforms. Rosetta 2 translates x86-64 applications, Wine and CrossOver implement Windows APIs, and graphics layers such as D3DMetal, DXVK, and MoltenVK translate rendering interfaces. For many games, this combination is sufficient to execute the game.

Online games protected by anti-cheat software impose a different requirement. Their security components inspect the process, system calls, kernel interfaces, and sometimes the machine’s virtualization status. Correctly translating the game is not enough if the anti-cheat has no client for that environment or cannot observe the operating-system behaviour it expects. Easy Anti-Cheat, BattlEye, and kernel-level systems therefore create a compatibility boundary that ordinary API translation does not solve.

1.2 Purpose and contributions

I evaluate a system that runs EAC-protected games through EAC’s Linux/Proton mode. My contributions are:

1. distinguish EAC’s Windows and Linux/Proton operating modes and derive the architecture required by the Linux client;
2. describe the host virtual machine, SteamOS image, guest kernel, Vulkan forwarding, and x86-64 user-space stack;
3. explain why `ptrace` fidelity and related Linux process semantics are central to the EAC launch path;
4. document my test, compare it with the public proof of concept, and present the failure codes, performance measurements, and measurement limits; and
5. set out security boundaries, ethical constraints, limitations, and avenues for future work.

The main engineering observation is that the anti-cheat’s Linux/Proton path depends on more than CPU instruction translation. It requires a coherent x86-64 Linux view for the full Proton process tree, including the EAC bootstrapper and the native Linux EAC client loaded into the game process.

1.3 Design principle: fidelity, not evasion

My system aims to make each layer behave more like the environment it claims to emulate. It does not modify, hook, replay, or replace EAC; spoof the machine’s hardware identity; or hide the hypervisor or translator. The anti-cheat can inspect the actual virtual machine and emulator. My results are consistent with reproducing the Linux process contract, not bypassing an anti-cheat check. This distinction is both a technical premise and an ethical boundary for the work.

Chapter 2

EAC Operating Modes and Platform Requirements

2.1 Windows mode and Linux/Proton mode

EAC operates differently on Windows and Linux/Proton. The Windows configuration uses a signed kernel driver together with a user-mode client. In the Linux/Proton configuration described here, the EAC runtime is user-mode-only: a native Linux ELF client runs alongside a Windows game launched through Wine/Proton. Valve distributes the Proton EasyAntiCheat Runtime as Steam application 1826330. The Windows bootstrapper (`start_protected_game.exe`) and the game's EAC calls are bridged to that Linux client.

Table 2.1: The two EAC operating modes.

Property	Windows mode	Linux/Proton mode
Privilege	Signed kernel driver and user-mode client	User mode only
Client	Windows PE, x86-64	Native Linux ELF supplied through the Proton EAC runtime
Game integration	Windows game calls the Windows client	Wine/Proton bridges the Windows game and bootstrapper to the Linux client
Enablement	Default Windows configuration	Developer opt-in on a per-title basis
Architectures	x86-64 in the relevant configuration	x86 and x86-64; no ARM64 client in the described runtime
VM behaviour	VM checks may refuse a guest environment	Tolerates environments supported by the Linux/Proton path; title policy still applies

The Linux client is available as x86-64 code, and Epic Online Services documentation lists the anti-cheat client interface as unsupported on Linux ARM64 [10, 1]. This platform constraint is

central: the EAC client is loaded into the game process, so translating only the game’s executable is insufficient.

2.2 Requirements for an Apple Silicon implementation

The operating-mode distinction implies four requirements:

1. **A Linux environment.** EAC’s Linux/Proton mode is the route without a kernel driver. A Linux kernel must therefore run in a virtualized guest on macOS.
2. **An x86-64 process tree.** The EAC client is an x86-64 ELF shared object loaded into the game. Wine, its preloader, the bootstrapper, and the game must use compatible x86-64 process state; native ARM64 Wine cannot load that x86-64 client into its process.
3. **Linux-faithful translation.** FEX must present the x86-64 Linux system-call and process model, including `ptrace`, signals, seccomp-related behaviour, and `proc` observations.
4. **Accelerated graphics.** Real-time games require hardware-accelerated 3D in the guest; software rendering is not an adequate target for the described workload.

2.3 Why existing macOS routes are insufficient

I compare common approaches as follows:

Table 2.2: Established routes and the constraints that prevent this EAC flow.

Route	Translation	3D path	Limitation for EAC
Wine, CrossOver, or Game Porting Toolkit on macOS	Rosetta 2	D3DMetal	There is no applicable EAC client environment: no Windows kernel for Windows mode and no Linux environment for Linux mode.
Windows 11 on ARM in a VM	Windows Prism	Paravirtualized DirectX	Uses the Windows kernel-driver mode in a virtual machine; EAC’s VM check can reject this route.
Linux VM with Rosetta for Linux	Rosetta	No guest 3D acceleration in the described Virtualization.framework configuration	In addition to the graphics limitation, Rosetta does not support <code>ptrace</code> for translated binaries [12].
Linux VM with native ARM64 Proton	FEX for game code only	Venus	The x86-64 EAC client cannot be loaded into the ARM64 Wine process; the observed result was code 206.
Linux VM with all-x86-64 Proton under patched FEX	FEX for the entire process tree	Venus forwarded to Metal	This is the architecture I evaluate.

The final route does not make EAC’s Linux mode operate as Windows mode. Instead, it runs the Linux mode inside an ARM64 Linux guest while translating the complete x86-64 user-space stack.

Chapter 3

System Architecture: SteamOS in a macOS Virtual Machine

3.1 Layered design

The architecture has two principal halves. A native macOS application manages the virtual machine and presents its display, input, audio, and host–guest services. Inside the guest, Valve’s ARM64 SteamOS runs native system components and hosts an emulated x86-64 Linux user space. At the deepest application layer, Steam Linux Runtime launches x86-64 Proton; Proton starts the Windows game and the EAC bootstrapper; FEX translates all of those x86-64 processes to ARM64.

The graphics path follows a separate, parallel chain: the Windows game’s Direct3D calls pass through DXVK or VKD3D-Proton to Vulkan, the guest Venus driver forwards Vulkan commands through virtio-gpu, and the host replays them through MoltenVK or KosmicKrisp onto Metal. This permits the game-facing software stack to remain largely unchanged while the host performs the final graphics translation.

3.2 VM-host goals and virtualization substrate

My VM host runs Valve’s ARM64 SteamOS image rather than a repackaged root filesystem. It uses Hypervisor.framework through libkrn from a normal macOS application with the `hypervisor` entitlement; I use neither a kernel extension nor Apple Virtualization.framework [3]. Its graphics objective is Vulkan-level acceleration, exposing a modern feature set to Proton’s DXVK/VKD3D layers. Its user-experience objective is to make SteamOS operate as a Mac application, with one window, native Mac input, CoreAudio, clipboard integration, suspend/resume, and host time-zone settings.

3.3 SteamOS image acquisition and disk construction

I construct the SteamOS disk from Valve’s update infrastructure rather than distributing the OS image. The process is:

1. **Discover an update.** Query Valve’s update metadata for the ARM64 variant and selected branch (stable or release candidate) to obtain the current RAUC update bundle and content-addressed chunk store.
2. **Verify authenticity.** Download the approximately 2 MB RAUC bundle and verify its CMS signature with Security.framework against Valve’s pinned certificate authority, rather than the system trust store. A built-in squashfs reader extracts the signed manifest and root-filesystem chunk index; the compatibility string (`steamos-aarch64`), version, and slot size are checked. The root-filesystem chunk index uses the `.caibx` format.
3. **Reconstruct the root filesystem.** Use a pinned build of `desync`, a casync implementation, to assemble the approximately 10 GB root filesystem from about 4.4 GB of compressed chunks fetched in parallel. Downloads can be stopped and resumed.
4. **Create the disk layout.** Write a sparse disk image with a GPT layout matching Valve’s partitions: `ESP`, `efi-A/B`, `rootfs-A/B`, `var-A/B`, and `home`. Hash the root filesystem while writing, compare its SHA-256 to the signed manifest, and populate both root slots.
5. **Provision first boot.** Attach a temporary cpio payload containing the build ID, partition UUIDs, a SHA-512 crypt password hash, and machine ID. The `initramfs` formats remaining partitions, makes the B-slot filesystem ID unique, writes SteamOS boot configuration, and then removes the payload.

Before downloading, users must accept Valve’s SteamOS image licence, which limits use to personal purposes and prohibits redistribution. I therefore reconstruct the image from Valve’s servers rather than distributing it with the application.

3.4 Kernel, initramfs, and read-only adaptation layer

`libkrun` boots directly from a kernel image without firmware, UEFI, or GRUB. The host supplies its own kernel, `initramfs`, and adaptation layer while keeping Valve’s SteamOS root filesystem unmodified.

I build the guest around Linux 7.2.9, with drivers built in and no modules. The kernel uses 4 KB pages for x86 semantics; Apple TSO support for x86-style memory ordering; 16 KB alignment for GPU blob memory to match the host page size; and `virtio`, `uhid`, `hidraw`, and `hid-playstation` support for virtual devices and controller recreation.

The `initramfs` takes over responsibilities normally handled by platform firmware and a bootloader. It implements SteamOS A/B slot selection with an attempt counter and rollback, mounts the selected read-only root filesystem, configures overlays for `etc` and `usr`, provisions the machine on first boot, applies host settings such as SSH enablement, time zone, and Steam client selection,

repairs filesystems after crashes, and applies compatibility adjustments. One cited example is leaving procs mounted in a way that permits Flatpak sandboxes to start.

A read-only EROFS adaptation image overlays `usr` to make the headset-oriented SteamOS image operate as a desktop console in a virtual machine. It starts gamescope on the virtual DRM display, masks services for absent headset hardware (including SteamVR), replaces VR desktop mode with KDE Plasma nested inside gamescope, translates SteamOS sleep into a host-side VM pause, adds the guest agent, provides an on-screen QR-code sign-in mode because headset Bluetooth pairing is unavailable, and shortens shutdown time-outs.

Because the root filesystem remains unchanged, SteamOS’s RAUC/atomupd updater continues to install into the inactive slot and can roll back. I verified this behaviour with a real update and rollback [17].

3.5 CPU, page size, memory ordering, and storage

Hypervisor.framework vCPUs execute ARM64 guest code natively. The host assigns appropriate quality-of-service classes to vCPU and GPU threads so macOS can schedule them on performance cores. A libkrnl patch masks Apple’s SME feature on M4 chips because the guest kernel would otherwise attempt to use it.

Page size is a correctness constraint. Apple Silicon macOS uses 16 KB pages, whereas x86 software assumes 4 KB granularity for `mmap`, single-page `mprotect`, guard pages, and JIT write protection. FEX can expose the expected 4 KB behaviour only when the kernel below it uses 4 KB pages. I build the guest kernel with `CONFIG_ARM64_4K_PAGES`; page size is a guest-kernel property, not a hypervisor setting.

x86 provides total store ordering (TSO); ARM64 is more weakly ordered. A translator can compensate with software barriers, but those barriers are costly in multithreaded games. Apple’s cores provide a hardware TSO mode, enabled for guest processes through `prctl(PR_SET_MEM_MODEL, TSO)`. My guest-kernel log confirms that Apple’s TSO support is available inside the Hypervisor.framework guest. This assists both performance and correctness of concurrent application and anti-cheat code.

The guest disk is a sparse file on APFS, HFS+, or exFAT and is attached with `virtio-blk`. The VM process takes an exclusive `flock` to prevent two VM instances from mounting the same home directory concurrently. An 86 GB image initially occupies approximately 9 GB on APFS.

Chapter 4

Graphics, Devices, and Host Integration

4.1 Vulkan forwarding through Venus

The game’s graphics path is Direct3D 11/12 through DXVK or VKD3D-Proton, then Vulkan and SPIR-V in the guest. Mesa’s Venus driver serializes Vulkan calls into shared-memory rings [4, 14]; virtio-gpu transports the commands and blob resources to host-side virglrenderer, which replays the commands through a Vulkan implementation backed by Metal. MoltenVK translates SPIR-V to Metal Shading Language through SPIRV-Cross; KosmicKrisp translates through NIR to Metal. Apple’s Metal compiler then produces GPU-specific code.

Venus forwards the Vulkan API itself, rather than a lower-level game-specific command stream. Guest software therefore sees a Vulkan device with a feature set tied to the host driver. My guest exposes Vulkan 1.4 and identifies the device using the host GPU, for example, “Virtio-GPU Venus (Apple M4 Pro).”

Guest-visible, host-mapped buffers and images use virtio-gpu blob resources. A blob is backed by a host allocation exposed through POSIX shared memory and mapped into guest physical memory, allowing vertex data, uniforms, and uploads to be shared without copying. My libkrun modifications add the `VIRTIO_GPU_F_BLOB_ALIGNMENT` feature at 16 KB, shared-memory blob mapping, Venus fence signalling, and `SET_SCANOUT_BLOB` for direct display scanout. I also raise the VM’s file-descriptor limit because each mapped allocation holds several descriptors; without this adjustment, games lost their GPU context after a few dozen allocations.

Two guest Venus providers are installed: an aarch64 build for native applications such as gamescope, Steam UI, Zink, and ARM64 Proton, and an x86-64/i386 build used as FEX’s graphics provider for translated applications.

4.2 MoltenVK and KosmicKrisp

The host can use either MoltenVK or KosmicKrisp. MoltenVK is a mature Vulkan-on-Metal implementation based on UTM’s fork with geometry-shader support. KosmicKrisp is LunarG’s Mesa driver for Metal 4 and macOS 26 or later [5, 6]. My game-specific work includes:

- **MoltenVK:** geometry-shader emulation for DXVK (including strides, instancing, adjacency, and fans), transform feedback and queries, `depth_clip_enable`, null descriptors, robustness fixes, descriptor-array limits, fragment-output type conversion, deferred resource release, and query availability on copy.
- **KosmicKrisp:** geometry shaders, transform feedback, `fillModeNonSolid` for DXVK, shared counter heaps for queries, and sparse binding using Metal 4 placement-sparse resources for the VKD3D-Proton D3D12 feature-level 12_0 path.

For my DirectX 11 test title on an M4 Pro, MoltenVK measured approximately 140 FPS in the menu and 130–135 FPS in a busy scene; KosmicKrisp measured approximately 95 FPS in the menu and 75–97 FPS in a busy scene. These results are workload-specific: KosmicKrisp performed better on a D3D12 title on an M1 Max.

For robustness, if the host driver cannot build a pipeline, virglrenderer substitutes a placeholder and drops its draws rather than passing a null handle that could crash the game. The Venus device identity includes a hash of the host driver’s pipeline-cache UUID, so a host-driver update invalidates guest-side shader caches instead of reusing incompatible ones. Guest OpenGL, including Xwayland glamor, uses Zink (OpenGL over Vulkan) and follows the same route to Metal.

4.3 Display and presentation

Gamescope renders the final composited frame into a blob used as the virtio-gpu scanout. The host presents it through a Metal layer and can optionally use MetalFX spatial upscaling to the Mac panel’s Retina resolution, allowing the guest to render fewer pixels.

The host adjusts the virtual display when the application window or screen changes, using a libkrn display-resize patch. It generates an EDID whose physical dimensions match the Mac display to keep guest DPI stable. Refresh rate can follow the Mac display, including a 120 Hz ProMotion panel or a 144 Hz external display. The host monitors virtio-gpu and Venus activity; if no GPU commands arrive for two seconds, it can display a shader-compilation status message. If the guest-agent heartbeat also stops, it reports that the guest is unresponsive.

4.4 Input and controller paths

Keyboard and mouse use virtio-input. Because gamescope does not accept absolute pointer coordinates, the host converts Mac cursor movement into unaccelerated relative deltas. The guest cursor then tracks precisely, and the host can hide the Mac cursor over the game image. When the guest agent reports that a game has focus, the first click captures the mouse for mouse-look; returning to Steam releases it.

Gamepads are read through Apple’s GameController framework and forwarded over a virtio-console port, rather than emulated as a fixed virtio device. A guest root service creates a corresponding `uinput` device: an Xbox 360 controller or a DualSense/DualShock 4 with the appropriate vendor and product IDs. The service implements kernel `ff-memless` force-feedback rules and returns rumble levels to the host for GameController haptics.

The DualSense can also use raw HID passthrough. The host reads it with IOHIDManager; the guest recreates it through `dev/uhid` using the actual report descriptor, IDs, and bus type. The guest's `hid-playstation` driver and Steam's HIDAPI driver then see a DualSense with touchpad, gyro, lightbar, adaptive triggers, and mute LED. Output and feature reports travel back to the physical controller.

4.5 Host-guest services, networking, audio, and sleep

A Rust guest agent and the macOS application communicate over named virtio-console ports. The `progress` port reports boot/shutdown stages, disk provisioning, Steam updates, heartbeats, and focus events (`focusgame<appid>`, `focussteamWxH`, or `focusdesktop`). The `pad` port carries controller state, rumble, and raw HID reports. The `clock` port corrects wall time after a VM pause; `sleep` converts SteamOS sleep into host VM pause/resume; and `clipboard` carries text and PNG images between macOS and guest X11/Wayland clipboards, with protection for password-manager items. These services integrate the guest with the host without modifying Steam.

Networking uses virtio-net connected to goproxy, a user-mode NAT and DHCP service; the guest address is 192.168.127.2. Game and anti-cheat backend connections use ordinary macOS sockets, without a network extension or bridged networking. Optional forwards provide localhost-only SSH and Steam LAN Remote Play through a discovery relay.

Audio passes through virtio-snd to CoreAudio, with speaker layouts up to 7.1.4 mapped to the Mac output. I measured built-in-speaker latency at about 65 ms.

On host suspend, the application pauses vCPUs and guest audio; the VM consumes no CPU while the Mac sleeps. libkrun shifts the guest virtual timer so monotonic time does not include the pause, avoiding watchdog timeouts. On resume, the agent advances wall time over the clock port. SteamOS's own suspend-to-idle is unsuitable in a VM without a wake source, so the adaptation layer replaces `systemd-suspend` with a host pause request. A click or button press resumes the VM. The guest time zone and clock format follow macOS settings at boot.

4.6 Packaging and x86-64 user space

The system is packaged as one macOS application containing the VM host, frameworks for libkrun, virglrenderer, MoltenVK, KosmicKrisp, and libepoxy, goproxy and desync helpers, the guest kernel and initramfs, the adaptation image, and Valve's CA certificate. The application is signed with the hypervisor and disable-library-validation entitlements, and the release build is notarized. Metal's on-disk shader cache is keyed to the application bundle identifier.

FEX-Emu loads x86-64 ELF programs, JIT-compiles instructions to ARM64, and translates system calls to ARM64 Linux calls [2]. Those processes need x86-64 libraries, including an x86-64 Venus provider. Valve distributes FEX as a Steam compatibility tool on ARM64 SteamOS, allowing the chain:

FEXcompatibilitytool $\longrightarrow$ SteamLinuxRuntime $\longrightarrow$ Proton $\longrightarrow$ game

FEX runs the entire process tree: Proton scripts, Wine, the game, EAC bootstrapper, and injected EAC client. It is the only component that requires the fidelity patches described in Chapter 5. I install the patched build as a separate compatibility tool so Steam updates do not overwrite it. Steam Linux Runtime provides a versioned x86-64 library environment. Stock Proton settings `PROTON_EAC_RUNTIME` and `EAC_LAUNCHERDIR` point Proton to the EAC runtime and bootstrapper data directory. Non-Steam stores can use the same chain as a wrapper; I used the open-source **legendary** client as an example [13], registered in Steam as a non-Steam game so gamescope presents the window.

Chapter 5

Kernel Fidelity and the EAC Injection Path

5.1 How the Proton EAC bootstrapper injects its client

The public proof of concept *eac-arm64-emulation-poc* by StonkGuy (MIT-licensed) recorded the launcher's raw system calls on native x86-64 Linux and compared them with FEX. I credit that project for its emulator patches and analysis [1]. The injection sequence is:

1. The Wine bootstrapper forks. The child calls `prctl` with `PR_SET_DUMPABLE` to mark itself non-dumpable, requests tracing with `PTRACE_TRACEME`, and stops with `SIGSTOP`.
2. The parent becomes the tracer, sets `PTRACE_O_TRACESYSGOOD` and `PTRACE_O_TRACEEXEC`, then single-steps the child from syscall to syscall with `PTRACE_SYSCALL`. It reads x86-64 registers using `PTRACE_GETREGSET` and `NT_PRSTATUS`.
3. The child creates a `memfd`, writes the approximately 6.8 MB x86-64 EAC shared object into it, and executes the game's Wine preloader.
4. At the `PTRACE_EVENT_EXEC` stop, the tracer reads `proc/<pid>/exe` and `PTRACE_GETSIGINFO`. Once the main executable is mapped, it reads the ELF header using `PEEKDATA`, inserts an `int3` breakpoint at `_start` using `POKEDATA`, catches the ensuing `SIGTRAP`, rewinds `rip`, and restores the original bytes.
5. The tracer uses roughly 300 `PEEKDATA` operations to walk dynamic-linker and `libc` structures and locate a `dlopen`-like routine. It places the string `proc/self/fd/3` and an intentionally invalid return address on the child's stack, and changes registers to enter the routine. The deliberate return fault raises `SIGSEGV`, returning control to the tracer after the client has been loaded from the `memfd`. The tracer then calls the client's initialization function, restores registers, and detaches.

The sequence takes approximately 530 `ptrace` operations and about 1.2 seconds on the native system measured by the proof-of-concept project.

5.2 Why ordinary host ptrace is not enough

Under FEX, tracer and tracee are both ARM64 host processes executing translated x86-64 programs. If the x86-64 tracer directly uses host `ptrace`, it receives correct information about the ARM64 FEX processes, but not the guest x86-64 processes it expects:

- syscall stops expose FEX’s ARM64 syscalls rather than x86-64 syscall numbers and `orig_rax`;
- register requests return ARM64 `user_pt_regs` rather than x86-64 registers such as `rip`, `rsp`, and `rax`;
- an `int3` trap belongs to FEX’s JIT rather than the guest instruction stream;
- `proc/<pid>/exe` identifies the FEX binary rather than the guest preloader; and
- writes through `POKEDATA` may leave FEX executing stale translated code, while the non-dumpable state complicates ordinary host-side sharing.

Stock FEX provided only limited passthrough support. In my launch test, it returned code 210, “Unexpected error. (#1).”

5.3 FEX’s x86-64 ptrace model

The proof of concept adds `ptrace` emulation in FEX’s Linux syscall layer. A shared state block for each traced process holds an x86-64 register image, pending-stop record, dirty mask, code-flush range, and exec path. A `PTTRACE_TRACEME` child publishes the block in a shared file-backed page under `dev/shm`, allowing the tracer to access state despite the child’s non-dumpable flag. After `execve`, FEX locates the new state block through `proc/<pid>/maps` at a known offset and uses host-side reads and writes.

FEX’s syscall dispatcher synthesizes x86-64 syscall-entry and exit stops when requested. At entry, the tracer sees the x86-64 syscall number in `orig_rax` and `-ENOSYS` in `rax`; at exit it sees the actual result. The tracee parks with a host `SIGSTOP`, and the tracer’s `wait4` wrapper rewrites the stop to `SIGTRAP|0x80`, matching `TRACESYSGOOD`. The tracer can change the syscall or its result as on Linux.

Guest `int3` and `SIGSEGV` events become guest signal-delivery stops with x86-64 register state; the `rip` value after `int3` matches Linux. Register changes are applied when the tracee resumes, and the signal can be suppressed, enabling the launcher’s intentional fault-and-resume sequence. The implementation covers `GETREGS/SETREGS`, `GETREGSET/SETREGSET`, `PEEKUSER/POKEUSER`, `PEEKDATA/POKEDATA`, `SETOPTIONS`, `GETSIGINFO`, `CONT`, `SYSCALL`, and `DETACH`. Writes to guest code invalidate the relevant JIT cache range. For traced, exec’d processes, `proc/<pid>/exe` reports the guest executable. A tracee does not park without an attached host tracer, and forked children do not inherit tracing.

The PoC provides a freestanding x86-64 `ptrace-inject` test. It reports `RESULT:PASS` on patched FEX; stock FEX hangs after the test’s “PASS: fork” stage.

5.4 Self-modifying code

FEX normally write-protects pages from which it has translated code. A write triggers a fault, invalidates translated code, and retries the write. I observed that EAC frequently modifies pages that also contain executed code, such as inline-hook trampolines or data adjacent to code. Repeated faults and retranslations prevented initialization from finishing.

The corresponding patch counts faults per page. After a default threshold of 32, the page remains writable and FEX emits a per-instruction checksum check for translated code from that page. The approach preserves correctness while stopping the repeated fault/invalidation cycle.

5.5 Signals, seccomp, and additional Linux semantics

Wine uses signals for thread suspension and exception handling, while anti-cheat code can inspect process state. The PoC therefore addresses kernel-contract mismatches beyond `ptrace`:

- Signal-mask patch 0007 reproduces Linux behaviour: handlers run with the interrupted mask, `sa_mask`, and current signal blocked; `rt_sigreturn` restores the interrupted mask. The previous FEX behaviour lost wake-ups in Wine’s server protocol and caused network timeouts in approximately every second session.
- Register-handling patch 0008 ensures that a handler entered while its thread is in a syscall retains register changes made by the handler.
- Seccomp patch 0010 ensures that a `SECCOMP_RET_TRAP SIGSYS` reports the instruction after the syscall, preventing Wine’s syscall-trapping filter from re-executing it indefinitely.
- Code-invalidation patch 0004 makes `mprotect` on data pages with approximately 200 threads about 22 times cheaper, avoiding minute-long stalls.
- Patches 0011–0026 cover `arch_prctl` CPUID faulting, hardware debug registers, `Seccomp:` and `TracerPid:` in `proc/<pid>/status`, unknown regsets, `restart_syscall`, signal-frame fields (EFLAGS, CS/SS selectors, and XSAVE magic), alternate-stack exit, and a single-step trap after `IRETQ`.

While testing seccomp, I found that `FEX_NEEDSSECCOMP=1` incorrectly duplicated file descriptors before path-based `execve`, breaking new processes. I fixed this by limiting duplication to descriptor-based execs while preserving binfmt compatibility and descriptor-copy requirements. My setup does not require seccomp emulation.

```
if (IsFDExec && (!IsBinfmtCompatible || NeedsFDCopy))
```

Diagnostic patches 0003, 0005, 0006, 0009, and 0024 add statistics, sampling, and thread snapshots. EAC flags external `ptrace` debuggers and `perf` as “Debugger detected,” so I avoid them on protected processes. My sessions suggest EAC accepted the visible VM once FEX reproduced Linux behaviour; FEX mappings, virtio devices, and VM hardware remain visible.

Chapter 6

Demonstrated Results and Failure Analysis

6.1 Demonstrations

I compare my results with a public proof of concept:

Table 6.1: EAC demonstrations.

	Public proof of concept	My setup
Title	VRChat, Unity, Steam	DirectX 11 online title, Unreal Engine, Epic Games Store, launched through legendary
Host	Apple M2 MacBook Air, 16 GB; Fedora Asahi (muvn) and macOS 27 with a libkrn guest	Apple M4 Pro, 24 GB, macOS 27.0.1
Guest	Valve ARM64 SteamOS with 4 KB pages	Same platform; Linux 7.2.9 kernel and Mesa 26.3 Venus
Translator	FEX-2610 with patches 0001–0026	Same patch set built in the guest; Valve FEX-2607 used as control
EAC outcome	Code 301 in three sessions, “AntiCheat Session Begin: Success,” and world entry	Code 301; client initialized and registered, anti-cheat enabled, online login, matchmaking, and match loading
Concealment	None; optional DMI “realism” script was not needed	None

Together, these demonstrations cover two titles across two Apple chips and two storefronts, using the same EAC launch mechanism. Individual titles retain their own configuration and server-side policy, so success with one title does not guarantee support for another.

6.2 Launcher error codes as diagnostic evidence

EAC writes a numeric result to the Proton log for each launch. I used the progression as a debugging method:

Table 6.2: Observed launch outcomes under different configurations.

Configuration	Result	Interpretation
ARM64 Proton with game code under FEX	206	Failure to load the anti-cheat module; attributed to the x86-64 client/ARM64 Wine ISA mismatch.
x86-64 Proton under stock FEX	210	Unexpected error; attributed to unsupported injection through <code>ptrace</code> .
x86-64 Proton with seccomp emulation and an <code>execve</code> fix	210	Client starts, reads process maps/status, then exits after about 0.4 seconds. This initially resembled deliberate emulator detection, but the cause was a Linux-behaviour defect.
x86-64 Proton under kernel-faithful FEX	301	EAC successfully loaded in-game.

The methodological lesson is cautious: an anti-cheat failure that resembles environment detection may be an emulator-fidelity defect, so I test that possibility before attributing intent. In this case, the same visible environment succeeded after I corrected the emulator’s Linux behaviour.

6.3 Known unsuccessful or untested cases

Replacing graphics DLLs in the game directory, including asynchronous-shader DXVK builds, caused an immediate crash after EAC loaded in one test. An EAC integrity check is the most likely explanation, so I do not attempt to work around it. Titles that require kernel-mode anti-cheat without a Linux/Proton client are outside this method. THE FINALS, which uses Theia anti-tamper and Denuvo Anti-Cheat, stopped in its own anti-tamper system in my test, including on stock Proton; this is not an EAC or FEX failure. Titles whose developers have not enabled EAC Linux/Proton support remain unavailable. BattlEye’s analogous Proton path is plausible but untested.

Chapter 7

Performance and Pipeline Compilation

7.1 Throughput

On my M4 Pro configuration, the DirectX 11 test title was GPU-bound rather than CPU-bound. At 60 FPS, the game thread used about 24 percent of one core and the render thread about 16 percent. My FEX microbenchmarks for atomics and cold translation were comparable to native Linux Asahi measurements, making the GPU path, rather than CPU translation, the limiting factor.

Table 7.1: Measured performance at 2560 by 1440 and 144 Hz.

Vulkan-on-Metal driver	Menu	Busy scene / typical frame
KosmicKrisp	About 95 FPS	75–97 FPS; 13.5 ms typical frame
MoltenVK	About 140 FPS	130–135 FPS; 7.0 ms typical frame

Display configuration materially affected the result. An external 1440p display in macOS HiDPI mode with a 5120 by 2880 backing store kept the GPU about 45 percent busy even with the game closed, compared with approximately 10 percent in 1:1 mode. MetalFX upscaling to a backing store larger than the panel could show added display latency (12.5 ms rather than about 6 ms) and reduced frame rate.

7.2 Why cold graphics pipelines hitch

A Vulkan pipeline binds shader stages to render state. DXVK creates pipelines for combinations used by the game; the graphics driver must produce GPU-executable code before drawing with a new pipeline. In my stack, the path is:

`DXBC → DXVK → SPIR-V → Venus → MoltenVK → MSL → AIR → GPU binary`

The Metal front-end and GPU back-end compilation occur synchronously on the Mac while the guest waits for a pipeline handle. On native Linux GPUs, DXVK can use `VK_EXT_graphics_pipeline_`

library (GPL) to compile shader stages during shader creation and link at draw time. I checked with `vulkaninfo` that neither MoltenVK nor KosmicKrisp exposed GPL or `VK_KHR_pipeline_library` through Venus in my guest; therefore DXVK compiled at draw time.

My first level load took approximately 40–52 seconds, mostly waiting on GPU compilation, while a repeat load took about five seconds. One online load exceeded the server’s patience and the client was dropped before the error appeared after loading. In the first four minutes of a match with unseen content, I observed about ten freezes lasting 0.4–1 second, followed by one 0.4-second hitch in the next four minutes on the same content.

The compatible mitigation is Metal’s persistent host-side pipeline cache, stored outside the guest and invisible to EAC. It survives game and VM restarts. A scripted visit to all levels warmed a cache of 332 MB for 93 levels; content appearing only in other players’ sessions could still hitch on first use. In contrast, asynchronous pipeline builds such as DXVK async or gplasync render with a fallback while compilation completes [8, 9]. Offline testing reduced cold level loads from 28–72 seconds to 9–10 seconds, but requires replacing game-directory DLLs, which EAC appears to reject.

7.3 Pipeline-library feasibility benchmark

I evaluated whether GPL in the host Vulkan-on-Metal driver could move compilation to load time without modifying the game installation. I captured Metal shaders generated by MoltenVK for the test title and Steam UI using `MVK_CONFIG_SHADER_DUMP_DIR`: 261 vertex/fragment pipelines, of which 250 were usable. The descriptor type was `MTLRenderPipelineDescriptor`. Each reconstruction included vertex attributes, colour/depth formats, dual-source blending, and function constants specialized as MoltenVK does. Each run used a fresh empty Metal cache through a private `MTLSetShaderCachePath`; no pipeline failed.

Table 7.2: Metal pipeline compile measurements (250 usable pipelines).

Draw-time path	Median	p90	Max.	Over 16.7 ms	Total
Cold today: compile both MSL libraries and pipeline state	58.2 ms	83 ms	355 ms	239	14.7 s
GPL-style: precompile libraries, create pipeline state at draw	14.9 ms	24 ms	55 ms	80	3.8 s; 9.9 s moved to loading
Warm Metal cache	3.2 ms	–	39 ms	1	0.9 s

I draw three conclusions. First, a GPL-style path could reduce first-time hitches by roughly four times (approximately seven frames to two at 120 Hz), but pipeline-state creation still invokes a GPU-specific Metal back-end compile. Second, for cached content the warm Metal cache is faster than the tested GPL-style path, at about 3 ms rather than 15 ms. Third, the benchmark is optimistic: a production implementation would need to produce per-stage MSL without full pipeline state such as attachment formats, sample count, and vertex layout, while DXVK expects fast linking. Metal compilation remains a platform constraint, and cache prewarming is the most effective current anti-cheat-compatible mitigation; GPL is a moderate, risky improvement rather than a complete fix.

Chapter 8

Input Latency

8.1 Latency path and measurement

The path from physical input to visible response includes macOS HID, the host application, virtio input or USB HID passthrough, the guest kernel, compositor, Wine, game simulation, render thread, DXVK, Venus, Metal, host presentation (and optional MetalFX), and the display. The two extra hops relative to native play are host-to-guest input and guest-frame-to-host presentation.

The host application's probe measures key-to-presented-frame latency without a camera. It takes the next presented guest frame as the baseline for a screen region, injects a key event, and records the time until more than 3 percent of the region's 64-by-64 sample points change by more than 40/255. This includes host and guest input, simulation, rendering, Venus/Metal, and host presentation. It excludes display scan-out and panel response.

8.2 Measured results

My measurements show little latency from virtualization when input handling stays off busy threads and frames are not over-processed. The largest measured term was game-internal pipelining. Wireless controllers can dominate: a Bluetooth DualSense delivered approximately 63 reports per second to macOS, or one every 16 ms, exceeding the virtualization overhead.

Table 8.1: Input and presentation measurements.

Measurement	Before	After	Change or interpretation
Key to presented frame, median, $n = 20$	36.7 ms	27.7 ms	Disabled the game's one-frame thread-lag option; render thread no longer trails the game thread, with no FPS loss.
Guest frame to display	12.5 ms	About 6 ms	Skip upscaling when the panel has no more physical pixels than the guest frame.
Gamepad report to guest, worst case	p99 1–4 ms; maximum 10 ms	p99 0.1 ms; maximum 0.3 ms	HID work moved to a dedicated high-priority host thread; newest report replaces rather than being dropped behind an unsent report.
DXVK latency controls & – & No measurable gain & A maximum frame latency of one and latency sleep did not improve the measured result.			

Chapter 9

Security, Policy, and Ethics

9.1 What the anti-cheat can observe

My setup leaves the anti-cheat-visible environment intact:

- game and anti-cheat files remain stock;
- no external debugger is attached to the protected process;
- `proc`, `ptrace`, and signal behaviour are made Linux-compatible rather than concealed;
- the hypervisor and virtual hardware are visible, with no DMI/SMBIOS spoofing; and
- FEX and its mappings remain visible rather than being hidden.

My observations show EAC accepting a visible VM and translator, not a configuration made acceptable by masking either one.

9.2 Operational rules

I follow these constraints: do not modify, hook, shim, or replace EAC; do not use null-client or offline overrides for online play; do not hide the hypervisor or spoof hardware identity; accept anti-cheat rejection of modified game files rather than probing for a workaround; and do not attach debuggers or profilers that use `ptrace` to a protected process.

9.3 Policy risk

Publishers and Epic do not support virtual machines or emulation in this context. Current acceptance reflects the present EAC implementation and each title's policy; either may change. I assess account risk as low but non-zero. My technical results should not be read as authorization, official support, or a guarantee that a title will continue to permit this configuration.

Chapter 10

Limitations and Threats to Validity

The evidence is limited in several ways:

- **Small sample.** The evidence summarized here covers two titles, two Mac models, one macOS major version (27), and short observation periods. It provides no long-term enforcement data.
- **Latency scope.** The latency figures end at the presented frame rather than measuring photons at the panel. Each configuration used 20 probes in one scene.
- **Pipeline benchmark.** Pipeline descriptors were reconstructed from shader interfaces rather than captured from live driver state. A private Metal API was used to isolate caches.
- **Version sensitivity.** Steam updates may change Proton, Steam Linux Runtime, or the EAC runtime. Vulkan-on-Metal driver updates change the Venus device identity and invalidate pipeline caches.
- **Resource requirements.** A full guest needs substantial memory and storage. The cited PoC used an 8 GB guest minimum on 16 GB hosts with memory tuning, and disk use reaches tens of gigabytes.
- **Title-specific policy.** Each game has its own EAC configuration and server-side rules; a successful launch in one title does not establish compatibility or permission in another.

I conducted the measurements on the hardware and software configurations stated here. I have not independently reproduced the public proof-of-concept results; interpret all values within their stated sample sizes and measurement methods.

Chapter 11

Open Problems and Future Work

11.1 Upstreaming emulator fixes

Submitting the kernel-fidelity changes upstream to FEX-Emu would make the work easier to maintain. I view them as general Linux-correctness improvements rather than anti-cheat-specific modifications. At the time of writing, the cited proof-of-concept project had not submitted the patches upstream.

11.2 Reducing Metal pipeline stalls

Potential avenues include implementing GPL support in MoltenVK or KosmicKrisp, distributing pre-built pipeline caches for specific titles at the Metal-binary level (analogous to Steam’s Fossilize precaching), and background compilation driven by recorded pipeline lists. Each would aim to warm the host cache without modifying guest game files. The benchmark in Chapter 7 bounds the possible GPL benefit and indicates that Metal’s back-end compile would remain.

11.3 Rosetta and native ARM64 EAC

Hardware-accelerated 3D for Linux guests on Apple’s Virtualization.framework could make Rosetta an alternative to FEX for translation. However, Rosetta’s lack of `ptrace` support for translated binaries would still block the EAC injection flow; FEX’s fidelity work would remain necessary unless that limitation changed [12].

A native ARM64 Linux EAC client offers a possible longer-term path. The cited Epic Online Services material documents Linux ARM64 support, but I have not verified whether the EAC client interface follows. If it does, the x86-64 translation requirement could disappear and native ARM64 Proton might become viable with lower translation overhead [10].

11.4 Broader compatibility evaluation

Further work includes systematic testing of more EAC and BattlEye Proton titles, documenting title-specific outcomes, and publishing a compatibility list. Such a list would need to distinguish technical launch results from developer policy and official support.

Chapter 12

Conclusion

In my M4 Pro test, I ran an EAC-protected Windows game on Apple Silicon macOS through EAC’s user-mode Linux/Proton mode rather than its Windows kernel-driver mode. An ARM64 SteamOS guest supplies Linux; Venus forwards Vulkan rendering to Metal; and FEX runs the entire x86-64 Proton process tree. The critical constraint is that the Linux EAC client is x86-64 and is injected using `ptrace`, making Linux-faithful process tracing and signal semantics essential.

My patched FEX build reproduces the relevant x86-64 register state, syscall stops, signals, code writes, and `proc` observations. My test achieved successful EAC initialization and online play on an M4 Pro; the public proof of concept separately demonstrated the same outcome in VRChat on an M2 [1]. These results remain title- and version-specific, unsupported by publishers, and limited by the small sample.

The most important remaining performance cost is first-use graphics compilation on Metal. A persistent host-side pipeline cache substantially reduces repeat stalls, while a GPL-style path may shorten cold compiles but cannot remove Metal’s pipeline-state back-end work. The larger contribution of the work is thus not a universal compatibility guarantee, but an architectural demonstration: when an anti-cheat provides a Linux user-mode mode, a faithful translation of its expected operating-system contract can matter more than instruction translation alone.

Appendix A

Glossary

EAC

Easy Anti-Cheat, an Epic Games anti-cheat product.

Proton

Valve's distribution of Wine with DXVK/VKD3D-Proton and Steam integration.

FEX-Emu

Open-source x86/x86-64 to ARM64 user-mode binary translator for Linux.

libkrnl

Minimal virtual-machine monitor library using KVM on Linux and Hypervisor.framework on macOS.

Venus

virtio-gpu context type that forwards Vulkan API calls from guest to host.

MoltenVK / KosmicKrisp

Vulkan implementations on top of Apple's Metal.

DXVK

Direct3D 9/10/11 to Vulkan translation layer.

GPL

`VK_EXT_graphics_pipeline_library`, which permits pipeline stages to be compiled separately and linked later.

TSO

Total store ordering, the x86 memory-ordering model discussed here.

SMC

Self-modifying code.

ptrace

Linux process-tracing interface used by debuggers and by the EAC launcher for client injection.

Appendix B

Reproduction Outline

I provide the following high-level reproduction outline as documentation, not as a guarantee that software versions or individual game policies will behave identically.

1. Set up a libkrun-based VM host running Valve ARM64 SteamOS with Venus forwarding; I recommend MoltenVK for the workload I tested.
2. In the guest, install x86-64 Proton, Steam Linux Runtime 4.0, Proton EasyAntiCheat Runtime, and Steam's FEX compatibility tool.
3. Build FEX-2610 with PoC patches 0001–0026 inside the guest and install it as a separate compatibility-tool copy to avoid Steam updates overwriting it.
4. Launch through FEX, Steam Linux Runtime, and Proton, using stock settings `PROTON_EAC_RUNTIME`, `EAC_LAUNCHERDIR`, and `PROTON_USE_XALIA=0`. Other storefronts can use the same wrapper chain; I used `legendary` as an example.
5. Check Proton's log for `Launcherfinishedwith:301`, the successful EAC initialization result I observed.
6. Before online play, visit the game's content once to warm the host Metal pipeline cache and reduce first-use compilation stalls.

References

- [1] StonkGuy, *eac-arm64-emulation-poc*: FEX-Emu patches, tests, and analysis for EAC on ARM64, including README.md, docs/how-it-works.md, docs/macos-port.md, and docs/platform-viability.md. <https://github.com/StonkGuy/eac-arm64-emulation-poc>
- [2] FEX-Emu project. <https://github.com/FEX-Emu/FEX>
- [3] libkrun project. <https://github.com/containers/libkrun>
- [4] Mesa, “Venus driver.” <https://docs.mesa3d.org/drivers/venus.html>
- [5] Khronos Group, MoltenVK project. <https://github.com/KhronosGroup/MoltenVK>
- [6] LunarG, “KosmicKrisp: A Vulkan on Metal Mesa 3D Graphics Driver.” <https://www.lunarg.com/a-vulkan-on-metal-mesa-3d-graphics-driver/>
- [7] Khronos Group, VK_EXT_graphics_pipeline_library specification. https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_graphics_pipeline_library.html
- [8] DXVK project. <https://github.com/doitsujin/dxvk>
- [9] dxvk-gplasync project. <https://gitlab.com/Ph42oN/dxvk-gplasync>
- [10] Epic Online Services, Anti-Cheat documentation on platform support for the client interface, as cited by [1] in its platform-viability documentation.
- [11] VRChat, “Using VRChat in a Virtual Machine,” as cited by [1] for EAC virtual-machine checks.
- [12] “Rosetta and ptrace of translated binaries,” cited by [1]. https://abe.seclab-bonn.de/2026/posts/gdb_macs_2/
- [13] Legendary, open-source Epic Games Store client. <https://github.com/derrod/legendary>
- [14] virglrenderer project. <https://gitlab.freedesktop.org/virgl/virglrenderer>
- [15] gypvproxy (gvisor-tap-vsock) project. <https://github.com/containers/gvisor-tap-vsock>
- [16] desync, casync implementation. <https://github.com/folbricht/desync>
- [17] RAUC update framework. <https://rauc.io>
- [18] Valve, gamescope project. <https://github.com/ValveSoftware/gamescope>