Mobile Gaming on Personal Computers with Direct Android Emulation

Qifan Yang[1,2], Zhenhua Li[1✉], Yunhao Liu[1,3], Hai Long[2], Yuanchao Huang[2], Jiaming He[2], Tianyin Xu[4], Ennan Zhai[5]

[1]Tsinghua University [2]Tencent Co. Ltd. [3]Michigan State University [4]UIUC [5]Yale University

ABSTRACT

Playing Android games on Windows x86 PCs has gained enormous popularity in recent years, and the de facto solution is to use mobile emulators built with the AOVB (Android-x86 On VirtualBox) architecture. When playing heavy 3D Android games with AOVB, however, users often suffer unsatisfactory smoothness due to the considerable overhead of full virtualization. This paper presents DAOW, a game-oriented Android emulator implementing the idea of *direct Android emulation*, which eliminates the overhead of full virtualization by directly executing Android app binaries on top of x86-based Windows. Based on pragmatic, efficient instruction rewriting and syscall emulation, DAOW offers foreign Android binaries direct access to the domestic PC hardware through Windows kernel interfaces, achieving nearly native hardware performance. Moreover, it leverages graphics and security techniques to enhance user experiences and prevent cheating in gaming. As of late 2018, DAOW has been adopted by over 50 million PC users to run thousands of heavy 3D Android games. Compared with AOVB, DAOW improves the smoothness by 21% on average, decreases the game startup time by 48%, and reduces the memory usage by 22%.

ACM Reference Format:
Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. 2019. Mobile Gaming on Personal Computers with Direct Android Emulation. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19), October 21–25, 2019, Los Cabos, Mexico.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3300061.3300122

MobiCom '19, October 21–25, 2019, Los Cabos, Mexico
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6169-9/19/10…$15.00
https://doi.org/10.1145/3300061.3300122

1 INTRODUCTION

As one killer application of PCs and mobile devices, computer games make a billion-dollar business: as of 2018, the worldwide market is valued at 137.9 billion US dollars [52]. The evolution of computer games has driven a number of technical innovations in terms of both hardware (larger memories, faster CPUs, and graphics cards) and software (*e.g.,* multimedia support and OS kernel improvements) [12].

Along with the proliferation of mobile devices, mobile gaming has become the largest segment of the market: mobile games contribute to 51% of all game revenues in 2018 [52]. As a result, many game vendors prioritize implementing mobile games over their PC or console versions. Today, few mobile games have corresponding PC versions due to the tremendous efforts for porting mobile-based implementation onto PC platforms with different OSes and architectures. Even with tool support (*e.g.,* Unity [44] and Unreal [45]), the porting is non-trivial—existing tools provide neither correctness guarantee nor usability control.

The mobile-first game development creates high demands for supporting mobile games on PC platforms [34], driven by at least three motivations. First, some users may want to play games that only provide mobile versions, while not owning the required mobile devices. Second, the gaming experiences are generally better with PCs' large screen and high resolution. Third, PC-based gaming can deliver better control via the physical keyboard and accurate mouse control. As a matter of fact, there have been more than 70 competitors in the PC-based mobile game market [11].

The *de facto* solution for playing mobile games on PCs is often dependent on *mobile emulators*, such as Bluestacks [10], Genymotion [18], KoPlayer [27], Nox [9], and MEmu [31]. All these game-oriented emulators use a full virtualization architecture, known as AOVB (Android-x86 On VirtualBox)—running Android-x86 [5] on top of a VirtualBox [47] virtual machine (VM). Android-x86 is an x86 porting of the Android OS, and VirtualBox bridges Android-x86 (the guest OS) to the host OS (*e.g.,* Windows). Given that most Android games rely on native ARM libraries, Intel Houdini [4] is typically used for translating ARM instructions into x86 instructions at the binary level. The AOVB architecture gains popularity for its free, open-source nature, and most importantly being fully transparent to unmodified mobile game binaries.

While AOVB-based emulators can run most mobile games, they only provide desired gaming experiences for 2D games and less interactive 3D games. For *heavy* 3D games (cf. §4.1) like Vainglory [38] and PUBG Mobile [41], AOVB-based emulators lead to significantly degraded gaming experiences (measured by smoothness, cf. §3.1). Note that gaming is different from many other applications, in which millisecond-level stagnation can be detrimental to the overall experience.

We have built and maintained an AOVB-based emulator (referred to as AOVB-EMU), which has been used by more than 30 million users to run over 40,000 Android game apps. Our measurement of its user experiences shows that the performance bottleneck roots in the considerable overhead of full virtualization (§3.2). With the goal of supporting heavy mobile games, we apply a series of para-virtualization and hardware-assisted optimizations to AOVB-EMU (§3.3), including GPU acceleration for graphic processing, VirtIO [36] for increasing the bandwidth of rendering pipelines, and Intel VT [43]. While these optimizations substantially increase the smoothness when running heavy 3D games, they are insufficient to provide the desired experiences. To address this, we need to break the boundary of virtualization.

This paper presents DAOW [39] which, to the best of our knowledge, is the first and the only emulator that can provide the same level of smoothness for running heavy 3D Android games on Windows PCs, as being played natively on Android phones. This is accomplished based on the idea of *direct Android emulation*, which directly executes Android app binaries on top of x86-based Windows. More specifically, DAOW provides foreign Android binaries with direct access to the domestic PC hardware through Windows kernel interfaces, thus achieving nearly native hardware performance.

Direct Android emulation faces a number of challenges from the distinctions at the levels of ISA (ARM *vs.* x86), OS (Android *vs.* Windows), and device control (touch screen *vs.* physical keyboard and mouse). *First,* data structures and execution behavior of binaries are distinct between Android and Windows. Instruction-level rewriting can fix the distinction, but change the layout of the original binaries and complicate the implementation. *Second,* Android/Linux and Windows have different sets of system calls (syscalls). Translating Linux syscalls to Windows requires significant engineering efforts, as well as incurring large runtime overhead if not appropriately implemented. *Third,* there is an interaction gap between mobile and PC-based gaming. PC games use physical keyboards and mouses for inputs; mobile games define a variety of buttons in different contexts. Also, PCs' large screens could enlarge the subtle rendering issues of mobile games, causing uncomfortable aliasing effect.

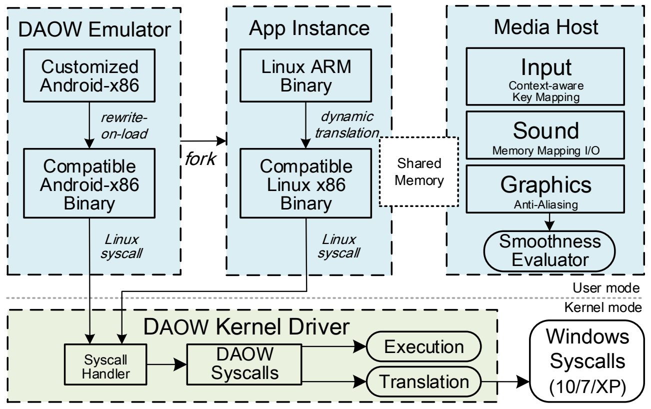


Figure 1: Architectural overview of DAOW.

We address these challenges with the following endeavors in the design and implementation of DAOW:

- *We take a data-driven, pragmatic approach to fulfill cost-efficient instruction rewriting and syscall emulation.* We comprehensively profile the instructions and syscalls used in a wide variety of Android game apps. Based on this, we reduce the many different types of instructions that need rewriting to only a few "patterns"; for each pattern, we utilize trampolines and write native Windows utility functions to minimize the changes in binary structures during instruction rewriting. Besides, we prioritize supporting the popular syscalls while treat the rarely used ones as exceptions; we also exploit the "common divisors" among the syscalls to greatly simplify the engineering efforts.
- *We make a number of optimizations in DAOW to improve its performance.* We enhance the efficiency of syscall emulation through extensive resource sharing, early preparation, and delayed execution. We also use shared memory for direct bulk data transfer between the app instances and the media component for real-time user interactions. In addition, we employ security approaches to prevent external cheating programs (*e.g.,* aimbot and speed hack on Windows) from modifying Android game app instances.
- *We leverage a series of graphics techniques to bridge the interaction gap between mobile and PC-based gaming.* We design an intelligent mapping technique which dynamically detects on-screen buttons and maps them to appropriate keys of the physical keyboards. Moreover, we design a progressive anti-aliasing method that assembles multiple existing techniques to smoothen rendering distortion and eliminate aliasing, without user-perceived overhead.

Figure 1 plots the system architecture that embodies our design of DAOW with three components: 1) Emulator, 2) Kernel Driver, and 3) Media Host. The Emulator `inits` a customized Android framework which is decoupled from the original Android-x86 distribution (by removing the built-in Linux kernel and the unused services), and rewrites its binaries while loading them into memory. The Emulator then forks a Windows process for running an Android game app, where ARM binaries are dynamically translated into x86 binaries. The Kernel Driver handles Linux syscalls via a series of DAOW syscalls (*i.e.,* our refined "common divisors" among Linux syscalls)—they are either directly executed or translated into Windows syscalls for execution. In addition, Media Host deals with user input, sound, and graphics issues, as well as measures the smoothness of the game.

DAOW is implemented in ~500K lines of C++ code. Since its first launch in Sep. 2017, it has been used by 50+ million users to run ~8000 heavy Android games on Windows PCs. Compared with AOVB-EMU, DAOW improves the smoothness by an average of 21%, from 0.76 ("rarely smooth") to 0.92 ("mostly smooth"), for millions of users when playing heavy 3D games. Also, it decreases the game startup time by 48% and the memory usage by 22% on average.

2 STATE OF THE ART

As listed in Table 1, we compare seven mainstream PC-based mobile gaming systems with large user bases, including our developed AOVB-EMU and DAOW. We focus on comparing five important features: 1) architecture, 2) accessibility, 3) syscall handling, 4) syscall coverage, and 5) media adaptation.

First, we study the basic architecture of these systems. Among them, Unity is the only one that generates a new PC game's program by compiling the original Android game's program. While this compilation approach earns the best performance, it sacrifices transparency to the game developers. On the contrary, Bluestacks and AOVB-EMU employ full virtualization based on the AOVB architecture, possessing fine transparency while bringing considerable overhead. Remix OS [25] runs Android-x86 on Linux PCs with specific drivers, so it does not require a hypervisor like VirtualBox. Different from Remix OS, Chromebooks use containers to host Android app instances on the Linux-based Chrome OS [13]. Neither Remix OS nor Chromebooks support Windows, the most popular PC operating system. As for Windows, a Linux subsystem called WSL is emulated in its version 10 distribution (64-bit) [32]; although WSL is not designed for mobile scenarios, some Android apps should be able to run atop the Linux subsystem in principle. Lastly, DAOW not only emulates Linux syscalls but also rewrites Android binaries, thus achieving direct Android emulation on Windows.

Second, we compare the accessibility of each system, which refers to the minimal effort users have to make before using it. We observe that systems like Remix OS, Microsoft WSL, and Android on Chrome OS have worse accessibility than the rest, because they either require users to install a specific OS (*e.g.,* Windows 10 64-bit) or enforce users to purchase specific equipments (*e.g.,* Chromebook). In comparison, Unity, Bluestacks, AOVB-EMU and DAOW only require users to install additional software packages, leading to better accessibility.

Next, we examine their syscall handling mechanisms which can be classified into three groups. Unity, Remix OS, and Android on Chrome OS are the first group that have no kernel-level compatibility problems by nature; thus, their syscalls are directly handled by the sole kernel taking full control of hardware, which can achieve the best efficiency. In comparison, AOVB-based systems take advantage of the hypervisor (VirtualBox) to handle all syscalls. The third group, including WSL and DAOW, handle Linux syscalls based on Windows kernel interfaces with specific strategies, which inevitably incurs extra runtime overhead. To address this shortcoming, DAOW utilizes the specially designed Kernel Driver and a series of optimizations for enhanced performance.

Fourth, we notice their syscall coverage is tightly related to their syscall handling and implementation manners. For instance, since WSL is a "clean room"[1] implementation of the Linux kernel's application binary interface (ABI), it passes 1466 out of 1904 Linux Test Project (LTP) test cases [29] [33]. Through comprehensive analysis (§4.5), we find that implementing 218 Linux syscalls is generally sufficient for DAOW to support nearly all games; similarly, Tsai *et al.* report that a common Ubuntu installation requires 224 syscalls [42].

Finally, we compare how these systems adapt to interaction gaps between PCs and mobile devices. There are mainly three approaches. The first approach, used by Unity and Android on Chrome OS, leaves the burden to app developers; however, only a few mobile games respond to PCs' keyboard events, so it is non-trivial for developers to support it. The second approach is to employ static key mapping predefined by users, *i.e.,* mapping every possible button and multi-touch gesture on the screen to a fixed key. It does not work well in heavy (*e.g.,* 3D FPS) games with complex user inputs. AOVB-EMU and DAOW use a new approach: we dynamically detect on-screen buttons and intelligently map them to a small number of user-friendly keys. We also detect aliasing and apply anti-aliasing techniques to all games. Note that we implement memory mapping I/O in DAOW rather than AOVB-EMU, as shared memory can hardly be achieved in AOVB due to the hindrance of full virtualization.

In general, the comparison shows that DAOW has the most practical architecture that balances performance and transparency, as well as the best accessibility and media adaptation. Although syscall handling and coverage of DAOW are slightly decreased compared with full virtualization, our

[1] "Clean room" means that WSL contains no code from the Linux kernel. In fact, WSL has a policy that its developers cannot even look at any of the Linux kernel source code [32]. This policy is also adopted by DAOW.

Table 1: Comparison of state-of-the-art PC-based mobile gaming systems.

System	Architecture	Accessibility	Syscall Handling	Syscall Coverage	Media Adaptation
Unity [44]	Program compilation	Software installation	Windows kernel	All	Developer coordination
Bluestacks [10]	AOVB	Software installation	Hypervisor, Dynamic translation	All	Static key mapping
AOVB-EMU	AOVB	Software installation	Hypervisor, Dynamic translation	All	Context-aware key mapping, Anti-aliasing
Remix OS [25]	Android-x86 with Linux PC drivers	Linux OS installation	Linux kernel, Dynamic translation	All	Static key mapping
Android on Chrome OS [13]	Container-hosted Android emulation on Linux	Chromebook	Linux kernel, Dynamic translation	All	Standard Android events
Microsoft WSL [32]	Linux subsystem emulation on Windows	Windows 10 64-bit	Windows kernel with pico-provider	1466/1900+ LTP tests	–
DAOW [39]	Direct Android emulation on Windows	Software installation	Rewriting on load, Dynamic translation, DAOW Kernel Driver	218/370+ syscalls	Context-aware key mapping, Memory mapping I/O, Anti-aliasing

experiences show that de-prioritizing the support for rarely-used Linux syscalls brings little negative effect in practice.

Apart from the abovementioned production systems, our work also benefits from research prototypes of OS virtualization, such as *Cells* [15], *Cider* [6] and *Drawbridge* [22]. In order to run multiple virtual Android phones on top of a physical Android phone, *Cells* uses device namespaces to provide isolation and efficient hardware resource multiplexing. Somewhat similarly, in order to run Android OS and app instances on top of a Windows PC, DAOW introduces kernel spaces (via the Kernel Driver) and the Media Host to multiplex the PC hardware. Besides, although *Cider* targets native execution of iOS apps on Android (rather than PC-based mobile execution), it also offers foreign binaries domestic access to underlying hardware and software in pursuit of native performance. Moreover, to run POSIX-based applications on Windows PCs, *Drawbridge* implements core POSIX APIs on Windows by leveraging a series of "embassy" interfaces; further, to practically support rich-media Android games, DAOW efficiently emulates necessary Linux syscalls on Windows by leveraging a series of intermediate syscalls.

3 UNDERSTANDING AOVB

Since its first launch in Dec. 2015, AOVB-EMU has attracted 30M+ users playing tens of thousands of Android games on Windows PCs per month. The basic implementation of AOVB-EMU follows the AOVB architecture: running the vanilla Android-x86 (version 4.4.4) on a PC-based VirtualBox (version 5.1.10) VM, coupled with Intel Houdini for dynamic binary translation to support the native ARM libraries.

As shown in Figure 2, Android-x86 and the Android app instance run natively on CPU Ring-3. Unlike running on Ring-0 in native Android, the Linux kernel in VirtualBox is reconfigured to Ring-1 by VirtualBox (which is also adopted by Xen [7]). When the Linux kernel executes a privileged instruction, it traps and the VMMRC kernel driver steps in

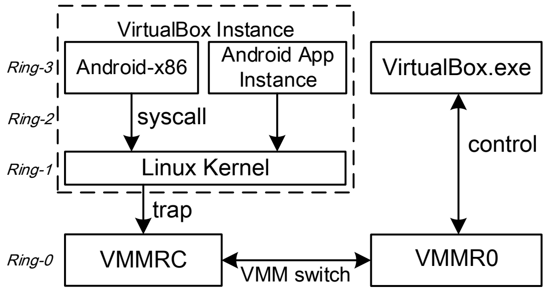


Figure 2: Architectural overview of the basic implementation of AOVB-EMU. Here VMMRC is VirtualBox Raw Context and VMMR0 is VirtualBox Host Context Ring-0.

to handle the fault or external interrupts. It makes a VMM switch to the VMMR0 kernel driver for privileged resource emulation such as clock interrupt, physical memory allocation, and device emulation. User input and virtual display are also emulated by VMMR0 and ridged to `VirtualBox.exe`.

In this section, we first devise a novel metric for quantifying the smoothness of mobile game emulation, and then present AOVB-EMU's bottleneck and optimizations.

3.1 Quantifying Smoothness

Smoothness is the primary measure of gaming experience. There are several metrics used to quantify smoothness, *e.g.*, Dune [19] and TinyDancer [53] use *skipped frame ratio* reported by `Choreographer` (an Android system component often used by normal apps but not games) to measure smoothness, while 3DMark Benchmark [17] uses *frame rates* as the metric. There are also proposals for taking the variation of frame rates into account [51]. We find that existing metrics each capture one important aspect of Android gaming smoothness; however, there are more practical issues to consider for a comprehensive evaluation.

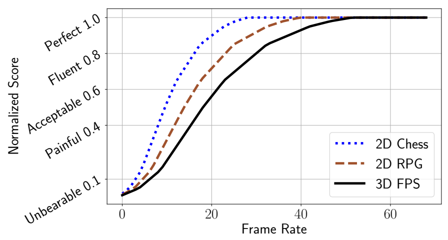


Figure 3: Normalized frame-rate score (*i.e.,* user-perceived smoothness when there is no fluctuation of frame rates) depends on both frame rate and game genre.

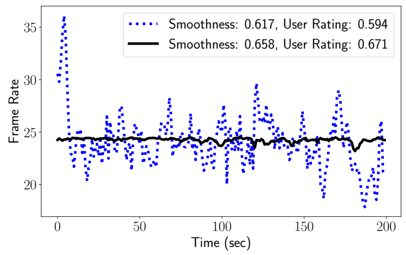


Figure 4: For a 3D FPS game, while the two curves have the same (quite low) average frame rate, their user-perceived smoothnesses are different.

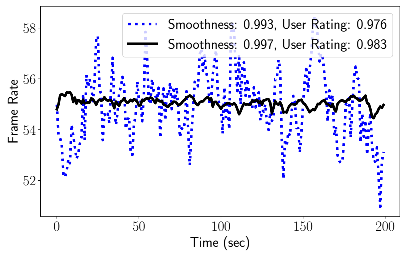


Figure 5: For a 3D FPS game, when the two curves have the same (quite high) average frame rate, their user-perceived smoothnesses are similar.

Seeking for a comprehensive smoothness metric, we invited more than 100 users to report their ratings of *perceived smoothness* when playing a variety of representative Android games (covering all genres). The rating scale is quite coarse-grained to calibrate users' perceptions: unbearable (0.1), painful (0.4), acceptable (0.6), fluent (0.8), and perfect (1.0). We have three insights from the collected data. *First*, the relationship between frame rate and smoothness is not only non-linear but also game-genre dependent as illustrated in Figure 3. *Second*, when frame rates are not high enough, smoothness is also influenced by the fluctuation of frame rates, as depicted in Figure 4. *Third*, when frame rates are high enough, smoothness is seldom affected by the fluctuation of frame rates, as shown in Figure 5.

Driven by the above insights, we devise a fine-grained smoothness metric. The smoothness of a game execution at the t-th second is defined as:

$$Smoothness_t = N_t * (1 - Penalty(N_{t-1}, N_t)). \tag{1}$$

N_t is the normalized frame-rate score lying between 0 and 1.0, which is calculated as

$$N_t = f(FR_t, Genre_{game}), \tag{2}$$

where FR_t is the frame rate, $Genre_{game}$ is the genre of the game (*e.g.,* 2D Chess, 3D RPG, and 3D FPS), and f is the normalization function demonstrated in Figure 3.

$Penalty(N_{t-1}, N_t)$ denotes the penalty caused by the fluctuation of frame rates in two consecutive seconds:

$$Penalty(N_{t-1}, N_t) = \begin{cases} \frac{N_{t-1}-N_t}{N_{t-1}}, & N_{t-1} > N_t, \\ 0, & N_{t-1} \le N_t, \end{cases} \tag{3}$$

which indicates that a decrease of frame rates leads to a penalty (inversely proportional to the frame rate in the $(t-1)$-th second) but an increase does not. When the frame rates stay high, the penalty would be little to zero, which complies with users' perceptions.

Our experiences interacting with users show that the devised metric approximates their perception of smoothness, as demonstrated in Figure 4 and Figure 5.

3.2 Bottleneck

When heavy Android 3D games are played on PCs, the results in Figure 6 show that AOVB-EMU bears extremely poor smoothness ($\le$ 0.12 on average). By carefully examining each procedure happening in the AOVB architecture, we find the issue is mainly attributed to the overhead of VMM switch—as demonstrated in Figure 2, when the Linux kernel is accessing privileged resources, the hypervisor steps in and makes a VMM switch to VMMR0 for privileged resource emulation, consuming 2.5× of the native process switching time. Furthermore, when running heavy Android games, we notice VMM switches frequently happen for CPU interrupts (50%), I/O (22% and 13% for read and write respectively), and inner timer (10%). This concludes that the performance bottleneck of AOVB stems from full virtualization.

Figure 7 depicts how context switch works between two threads of an app in (a) Android-x86 native execution on a Linux PC, and (b) the basic implementation of AOVB-EMU on a Windows PC. In (a), the first app thread directly wakes the second thread up and puts itself to sleep (*i.e.,* switch on a CPU core) by invoking a Linux syscall, costing merely 0.4 μs on average. In (b), there are two extra steps (trap and VMM switch) involved, bringing an additional time cost of 3.4 μs. Hence in this case, virtualization significantly increases the time cost of context switch by 9×. Such context switches happen frequently in heavy Android games, where 20+ active threads bring over 10K switches per second in a single process on average. If VT (Intel Virtualization Technology[43] or AMD Virtualization[1]) is not turned on, the only one available virtual CPU core on VirtualBox would bring extra overhead on the context switching latency, because parallel context switches need to be performed by a single CPU core.

3.3 Optimizations

To address the performance bottleneck of AOVB-EMU, we mainly make the following optimizations [2, 3]:

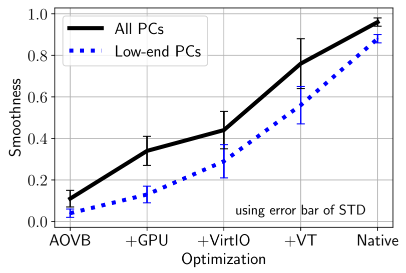


Figure 6: Running smoothness of heavy 3D games for AOVB-EMU after various optimizations are applied, on all PCs and low-end PCs respectively.

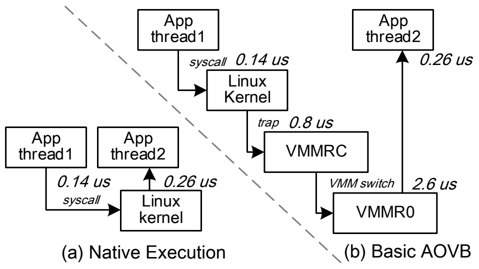


Figure 7: Context switching between two threads of an app in (a) Android-x86 native execution and (b) the basic implementation of AOVB-EMU.

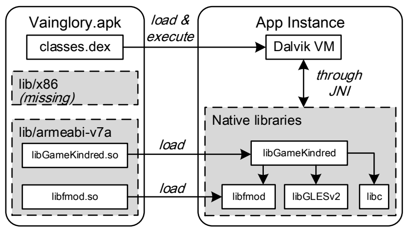


Figure 8: Runtime overview of an Android game.

- *GPU acceleration*: VirtualBox does not provide 3D acceleration for Android [48] but GPUs can be used to accelerate graphics processing. To fully exploit PCs' capabilities, all OpenGL instructions in Android are intercepted, encoded, and transferred to the GPU driver for executions.
- *Adopting VirtIO*: As a virtual I/O interface, VirtIO is used to increase the throughput of the rendering data pipeline between Android and Media Host. It constructs a ring buffer for efficient data transmission, and needs the collaboration between the Linux kernel and VirtualBox hypervisor.
- *Enabling VT*: We instruct our users to turn on VT via BIOS configuration to leverage hardware-assisted virtualization support. Eventually, 57% of AOVB-EMU users have enabled VT. For AOVB-EMU with VT, we enable additional acceleration techniques of VirtualBox such as Nested paging and VPIDs [49] which greatly reduce the overhead of VM exits, page table accesses, and context switching.

We measure the smoothness improvements of the above optimizations respectively. Figure 6 shows the results. First, when GPU acceleration is applied, the smoothness is greatly increased by nearly 2×. After VirtIO is adopted and VT is enabled, the eventual average smoothness of AOVB-EMU reaches 0.76 (acceptable) on all PCs and 0.57 (frequent stagnation) on low-end PCs. Both values (0.76 and 0.57) are lower than the frequent level ($\geq$ 0.8) and the satisfactory level ($\geq$ 0.9). In comparison, when we run heavy Android 3D games on Linux PCs with Android-x86 installed (referred to as "Native" in Figure 6 since VirtualBox is not needed), the average smoothness is 0.95 on all the experimented PCs and 0.89 on low-end PCs. In summary, even with the integration of all the optimizations, we still fail to make full virtualization based solution achieve desired smoothness in supporting heavy games. This drives our exploration in direct Android emulation and DAOW.

4 DAOW: DESIGN & IMPLEMENTATION

DAOW embodies the idea of *direct Android emulation on Windows*. To achieve this, we address significant differences at the levels of OS (Android/Linux *vs.* Windows), architecture (ARM *vs.* x86), and device (mobile devices *vs.* PCs). Figure 1 depicts all the building blocks of DAOW. In this section, based on static and dynamic profiling of a wide variety of Android games, we design and implement the key enabling mechanism(s) of each building block, in particular how they address the aforementioned multi-level differences *effectively* and *efficiently*.

4.1 Profiling Android Games

Similar as other Android apps, a game app mainly consists of four types of files in the APK: a platform-independent Dalvik executable (`.dex`), native ARM libraries (`.so`) and occasionally x86 libraries (for portability onto x86-based mobile devices), manifests, and resources. As exemplified in Figure 8, `Vainglory.apk` (a popular 3D game on mobile platform only) contains a dex file (6.6 MB) which can be executed by a Dalvik JVM, native libraries `libGameKindred.so` (24 MB) and `libfmod.so` (1.6 MB) for ARM-v7 platforms, a manifest file (0.5 MB) specifying the app's metadata, and a variety of resource files (1 GB) including images, audio, videos, and 3D models. Since there are no native x86 libraries, this game cannot run on x86 platforms without translation.

The two native libraries are loaded into memory and bridged to Java bytecode through the Java Native Interface (JNI) at runtime. `libGameKindred` relies on other shared libraries such as `libfmod` for audio processing and `libGLESv2` for graphic processing. The Java bytecode dispatches Android events into native libraries to convey user operations. Native libraries interact with the kernel though Application Binary Interface (ABI) to maintain the game loop.

Static profiling. AOVB-EMU and DAOW systems are associated with a major Android game market (abbreviated as

Market-G [40]) that hosts nearly 500,000 games. To understand the static characteristics of Android games, especially the native libraries and instructions, we scan the binaries (included in the native libraries and/or Java bytecode) of each game app and obtain the following key findings:

- *98.2% games uses native libraries to improve efficiency.* Games that do not use native libraries are mostly Word and Puzzle games that are insensitive to execution speed. We observe that a mobile game uses seven native libraries on average (varying among games) . Therefore, to support Android games on PCs, we would need an Android environment (*e.g.,* Android-x86) to execute Android's native libraries.
- *Native x86 libraries are not often provided.* All the games provide native ARM libraries while only 27.4% provide native x86 libraries. Therefore, to support ARM-based Android games on x86 PCs, we have to translate ARM instructions to x86 instructions using Intel Houdini [4].
- *Among all (~800) types of existing x86 instructions [24], only 30% are actually used by games.* Hence, at most 240 types of instructions may need rewriting for binary compatibility.

Dynamic profiling . Among the 500,000 games hosted in Market-G, the top 40,000 receive almost all (>99.9%) of the popularity in a certain period of time (*e.g.,* one year). Thus, to unravel the general characteristics of Android games at runtime, we study the top-40K games by collecting their execution traces from around 1/5 of AOVB-EMU clients (all of which belong to volunteer users with informed consent) during Jan. 2017. The traces were limited to one month of measurements, and are fully decoupled from any user identifiers or personally identifiable information. From the traces we have the following key observations:

- *All system calls are not created equal.* Counting all the 40K games, only 200 (out of 370+ in total) syscalls are invoked at least once at the runtime. The most frequently invoked syscalls are `gettimeofday`, `read`, `write`, `futex`, and so forth. This allows us to prioritize supporting the popular syscalls and treat the rarely used syscalls as exceptions.
- *Games do not use all the system services.* Nearly 1/3 of Android services are never accessed by games, *e.g.,* in the Android-x86 version 4.4.4 used by AOVB-EMU, 32 out of 102 services are never accessed (refer to §4.2 for the details). This enables us to customize the vanilla Android-x86 to be lightweight yet still adequate for running Android games.
- *Rendering instructions are the major overhead.* The main computation of running an Android game comes from the invocation of OpenGL rendering instructions to display each graphic frame. As shown in Figure 9, there is a clear boundary between heavy 3D games (*e.g.,* FPS, Racing, Sports, RPG and ACT) and simple 3D games (*e.g.,* Card, Puzzle and Word). On average, the former invoke 2000+ rendering instructions per frame while the latter invoke 1000-.

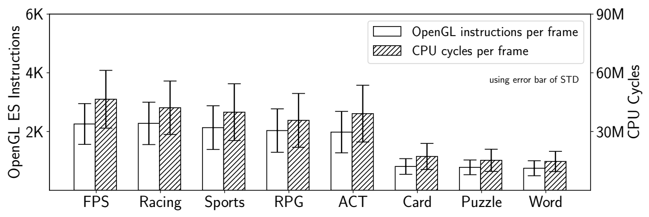


Figure 9: Different genres of 3D Android games invoke significantly different numbers of OpenGL rendering instructions to display a graphic frame. Meanwhile, their used CPU cycles per frame are tightly relevant.

4.2 Android-x86 Customization

As discussed in §4.1, in the original full-fledged Android-x86 version 4.4.4 system we employ, nearly 1/3 (32 out of 102) of Android services are never accessed by Android games. Services such as printing, NFC, and infrared (sensors[50, 55]) are often used by non-game apps rather than game apps, and thus can be removed. Moreover, the built-in Linux kernel of Android-x86 is removed since its role is taken over by our developed DAOW Kernel Driver.

Apart from the 32 unused services, 11 other services (*e.g.,* Bluetooth, WiFi, smartphone battery, and vibrator) are *neutralized* by their hardware or software alternatives in PCs. In detail, three kinds of neutralizations are implemented in DAOW. *First*, since WiFi and Bluetooth hardware modules are commonly seen in almost all of today's PCs, they are reused to serve the Android games run in DAOW with certain limitations (*e.g.,* the Android games are not allowed to configure or control the two hardware modules). *Second*, for a desktop PC which does not have a battery, we simulate the battery charging state. *Third*, because vibrators are rarely used by PCs, we programmatically shake the emulated display window of an Android game to mimic the required vibrations.

Besides removing and neutralizing 43 services, we enhance the performance of several services in Android-x86 that are tightly related to users' gaming experiences, such as input, audio, and graphics services. The enhancements are implemented in Media Host and the details will be presented in §4.7. Note that the enhancements should not incur additional native binaries and Linux syscalls. With all above efforts, the runtime memory footprint of Android-x86 is considerably reduced from 1.2 GB to 700 MB.

4.3 Rewriting Binaries on Loading

DAOW Emulator uses `init` to load the customized Android-x86. During the loading, the instructions of Android-x86 have to be rewritten for compatibility since Android-x86 is based on the Linux kernel but the instructions will be executed on Windows. Besides, when an app uses native x86 libraries, the included x86 instructions also need rewriting. Specially, rewrite-on-load deals with two types of distinctions between Linux and Windows: 1) different data structures, such as the binary format and process layout; 2) different runtime behavior, such as the syscalls and register usages.

The rewriting takes three steps as illustrated in Figure 10: 1) capturing instruction-level incompatibility "patterns," 2) transmuting instructions using trampolines, and c) supporting the functionality of instructions by composing native Windows utility functions. One key design decision is to generate Windows-compatible instructions with minimum changes in the binary structures. Otherwise, excessive disassembling and reconstructing operations are required to insert rewritten instructions into the original binary, which would substantially increase the rewriting overhead and complicate the implementation. Currently, rewriting a 30-MB Android-x86 binary requires less than 120 μs on an average Windows PC, and the total rewriting time of the 67 Android-x86 services (containing 155 binaries) is less than 580 *ms*.

As discussed in §4.1, there are less than 240 types of x86 instructions actually used by Android game apps. After carefully examining these instructions, we find only half of them need rewriting for binary compatibility; more importantly, multiple types of instructions can be rewritten with one pattern while some type of instructions has to be rewritten with several patterns. In the end, DAOW captures ten instruction-level incompatibility patterns. Among these patterns, two patterns occur the most frequently: the `int 0x80` interruptions, and the usage of particular segment registers.

The first pattern (*Pattern A* in Figure 10, which profiles the invocation of the `$number`-th syscall) stems from the fact that Android-x86 often uses `int 0x80` to make syscalls while Windows programs use `int 0x21` or `sysenter`. When it is captured, the incompatible instructions are rewritten by using the dynamically-generated in-situ *Trampoline A*. In detail, this fixed-size (typically 5 byte) trampoline is used to keep the original structure of the corresponding Android-x86 binary. Then, *Trampoline A* passes the execution flow to the corresponding "helper" function for realizing the `$number`-th syscall. This native Windows utility function adjusts the data organization, and makes the corresponding Linux syscall to DAOW Kernel Driver. The control is transferred back to Android-x86 code once the syscall is complete.

The second pattern (*Pattern B* in Figure 10) uses two "undefined" segment registers `gs` and `fs` [24]. 32-bit Linux x86

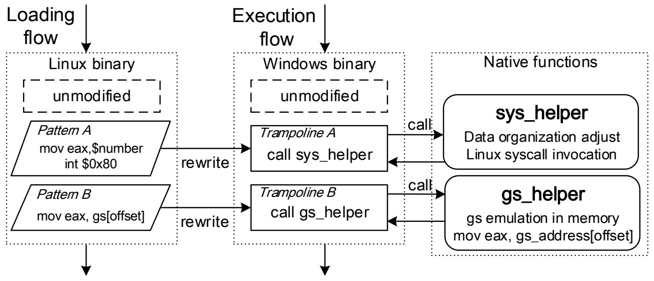


Figure 10: Rewriting Android-x86 binaries on loading (them into memory) by capturing incompatibility patterns, leveraging trampolines, and composing native Windows utility functions as effective "helpers."

binaries use `gs` to access the thread-local storage (TLS) while Windows x86 binaries use `fs` for a similar purpose; however, when rewriting Linux binaries we cannot simply replace a `gs` in Linux with an `fs` in 64-bit Windows, because `gs` and `fs` are not accessible for user-space processes in 64-bit Windows. As a result, *Trampoline B* is employed to call the `gs` helper function, which first emulates the `gs` in memory and then moves the desired data pointed by the `gs` to `eax`.

4.4 Dynamic Binary Translation

As we observe in §4.1, all Android games provide ARM libraries while only 27.4% provide a complete set of corresponding x86 libraries. As a consequence, when running Android games on Windows PCs (almost all of which are using x86 CPUs), in most cases we have to *dynamically* translate ARM instructions to x86 instructions. We use Intel Houdini to do the translation as it has the best performance (*i.e.,* the translation only incurs ~30% performance degradation according to our observations) and compatibility compared to the others [8, 21, 37]. Houdini provides a set of Linux x86 executables and auxiliary Android ARM libraries (such as `libc.so` and `libGLESv2.so`). To incorporate the functionality of Houdini, we modify the built-in Dalvik VM of Android-x86 to make it go through Houdini. Thereby, when the Dalvik VM detects native ARM libraries in an Android app instance, it invokes corresponding Houdini functions to load and translate target ARM instructions.

4.5 Emulating Linux Syscalls

DAOW Kernel Driver is responsible for emulating Linux syscalls on Windows. We find that it is inefficient to emulate each syscall independently[28], because syscalls use shared kernel resources. Therefore, we make great efforts to inspect and exploit the common divisors among these syscalls, especially for syscalls with highly related functions.

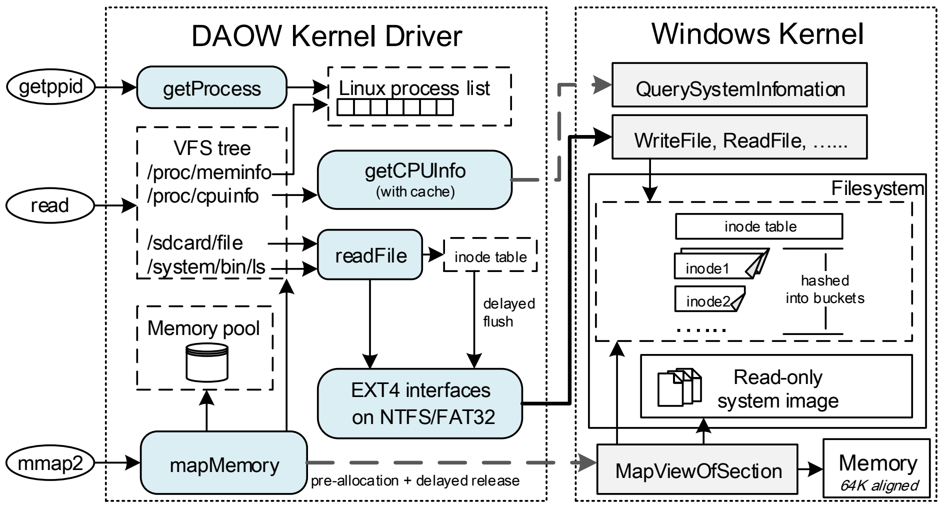


Figure 11: Efficiently emulating Linux syscalls in DAOW Kernel Driver by exploiting the common divisors among the syscalls and a common set of utilities, as well as early preparation and delayed execution.

As discussed in §4.1, the 40K Android games use less than 200 Linux syscalls (out of 370+) at runtime. This indicates that for directly emulating Android games on Windows, we can de-prioritize the support for 170+ Linux syscalls that are rarely used. In fact, the current DAOW Kernel Driver supports 218 Linux syscalls, including the less than 200 syscalls actually used by Android games and the additional 18+ syscalls used for debugging/logging. For the remaining ones, in case they are invoked (the occurrence is smaller than 0.007% for daily active instances), DAOW returns the `LINUX_ENOSYS` (*i.e.,* function not implemented) exception to the game app, and then watches whether the exception will cause essential problems to the game app. By customizing the Android-x86 `SystemServer`, once an app hangs or crashes after an unimplemented syscall is invoked, the exception message will be automatically reported to us. According to our collected reports, crashes and hangs happen with a probability of 24%. Therefore, the eventual occurrence of essential problems caused by unimplemented syscalls is negligible ($0.007\% \times 24\% = 0.002\%$), and we can always support more syscalls if really needed.

We classify the supported 218 Linux syscalls into eight groups: process, file, filesystem, memory, IPC, system, network, and user. Different groups have different design for the emulation, while share a common set of DAOW syscalls (*i.e.,* the "common divisors") and utilities. Some DAOW syscalls can be directly executed inside the Kernel Driver while other DAOW syscalls have to be translated into Windows syscalls. Below we describe our emulation principles and insights using a typical example where three Linux syscalls (`getppid`, `read`, and `mmap2`) are emulated in DAOW Kernel Driver.

As shown in Figure 11, the first Linux syscall `getppid` means to return the process ID of the parent of a calling process. Although it is possible to query the Windows kernel for the parent process's ID, the result may not comply with Linux specifications. Hence, we emulate `getppid` by composing the DAOW syscall `getProcess` and maintaining a Linux process list. With these efforts, we can return a correct result without employing any Windows syscalls.

The second Linux syscall `read` retrieves not only regular files but also pseudo files maintained by the kernel, such as the system information files under `/proc`. For the former (reading a regular file), DAOW Kernel Driver has to deal with the differences between Linux and Windows in naming restrictions and file attributes. More specifically, we need to emulate EXT4 interfaces on NTFS/FAT32 (abbreviated as "EXT4Windows"), which treat read-only and writable files differently. For the sake of efficiency, read-only files (*e.g.,* `/system/bin/ls`) are pre-packed into a binary image with inner files aligned in 4K blocks, and frequently-used real-only files are cached in batch. On the other side, writable files are named by the inode number; the inode table is frequently queried and synchronously updated in memory, but asynchronously flushed to the disk (or says "delayed flush"). For the latter (*e.g.,* reading a pseudo file `/proc/cpuinfo`), the CPU information is pre-queried from the Windows kernel and cached early when DAOW Kernel Driver starts up.

The third Linux syscall `mmap2` is mainly responsible for allocating memory or mapping files into memory. When it is emulated on Windows, an instant obstacle lies in the distinct memory alignment granularities between Linux (4K) and Windows (64K). To address this, a simple but space-consuming method is handling every memory-related syscalls with the 64K alignment; in contrast, our devised DAOW syscall `mapMemory` maintains a memory pool to resolve such inconsistency coupled with early allocation and delayed release. If the required memory block can be satisfied by the memory pool, a memory block is immediately returned to the user application without disturbing the Windows kernel. Otherwise, `mapMemory` has to be translated to corresponding Windows syscalls. A larger memory block will be allocated and the required memory block will be returned; the remainder is put into the memory pool for later use.

4.6 Security Defenses

As DAOW does not use full virtualization, it has to take care security concerns due to weaker resource isolation [14, 54]. First, DAOW must prevent malicious Android apps from attacking Windows programs. In our experiences, we have never observed Android apps' attacking Windows programs—even malicious Android apps do not have the motivations to attack Windows PCs. Still, DAOW prevent users from running any malicious apps. This is achieved by checking apps' fingerprints (*i.e.,* its MD5 hash code) when loading apps from the APKs, with the help of Market-G (refer to §4.1) which hosts almost all the popular and official Android games and provides their security labels. If an app fails to pass the checking, DAOW will explicitly notify the users of the potential risk.

Second, DAOW has to prevent Windows-based malware from attacking Android apps. Such attacks, in practice, are observed[2] and have a clear motivation—cheating in gaming. To address this, we build a series of security defenses to prevent external cheating programs (*e.g.,* aimbot and speed hack on Windows) from modifying Android game app instances. Specifically, we notice that most cheating programs are granted with user privileges, and only a few of them possess kernel privileges. Cheating programs with only user-level privileges can be easily defended. Since the DAOW Kernel Driver works in the kernel mode, it can easily detect and then block the cheating programs' access/tampering attempts on Android game app instances.

If cheating programs also work in the kernel mode, DAOW is able to detect most of them, but not all. When a kernel-mode cheating program is known to the community and we have understood its key characteristics or fingerprint, DAOW can detect it by leveraging such features. Nonetheless, DAOW is not able to directly prevent it from accessing or tampering Android game app instances. Instead, when such an attack is detected, DAOW would explicitly prompt a window to notify the user and ask the user to block the cheating program. If the user is unwilling to block the cheating program (implying that the user is likely to be a cheater/attacker), DAOW would shut down itself and report the situation to our security center. After manual checking, our security center may either warn the user or ban the user from running DAOW.

[2]Such attacks happen thousands of times per day according to our statistics.

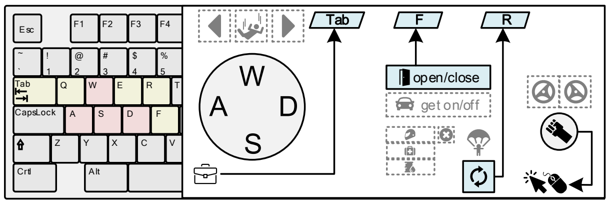


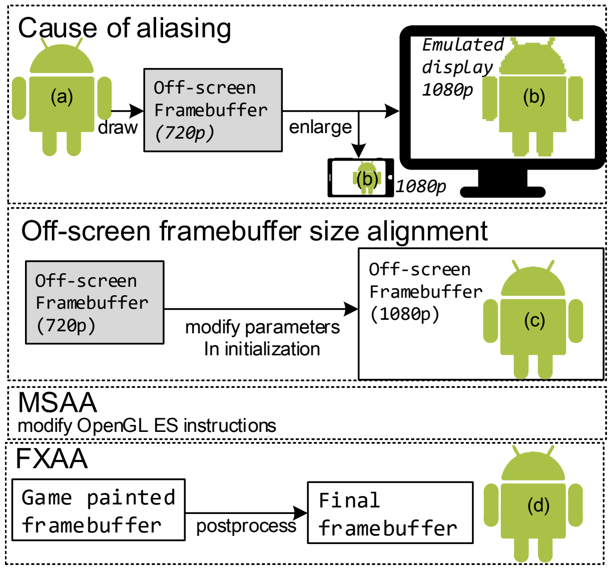
Figure 12: Context-aware key mapping for an FPS game. Mobile phones' on-screen buttons are dynamically detected and mapped onto easy-to-reach PC keys, and the "Attack" button is mapped to left-click of the mouse. Dashed buttons are temporarily disabled in the specific context.

4.7 Gaming Support

We build two gaming support—context-aware key mapping and progressive anti-aliasing—to fill the interaction gaps between mobile devices and PC platforms.

Context-aware key mapping. When users play Android games on mobile devices, on-screen finger touch is the major user input method; in contrast, almost all PC games use standard keyboards and mouses as the input. To bridge this gap, a simple and widely used solution is to statically map every possible button or gesture to a fixed key. If the number of possible buttons and gestures is small (typically < 20), this solution works fine. Otherwise, hard-to-reach "cold" keys have to be employed, thus impairing users' gaming experience. The essential drawback of static mapping lies in that it cannot capture the dynamics of button configurations under different contexts (game scenarios). As shown in Figure 12, round buttons are fixed while square buttons change their locations and visibility with the context.

In order to improve users' gaming experience, DAOW employs a context-aware key mapping method. When loading a game, DAOW recognizes the textures of all predefined buttons, which are then used to track them on the screen. During the gaming process, we capture on-screen buttons and their positions by inspecting the OpenGL drawing instructions. Using the inspection results, we dynamically map each on-screen button to an available key based on heuristic rules and manifests. As demonstrated in Figure 12, the mapping prefers the keys in the vicinity of the four direction keys "W, A, S, D", such as "F, R, Q, E" and "Tab", so as to make the



Cause of aliasing
(a)
draw
Off-screen Framebuffer (720p)
enlarge
Emulated display 1080p
(b)
(b)
1080p
Off-screen framebuffer size alignment
Off-screen Framebuffer (720p)
modify parameters In initialization
Off-screen Framebuffer (1080p)
(c)
MSAA
modify OpenGL ES instructions
FXAA
Game painted framebuffer
postprocess
Final framebuffer
(d)

Figure 13: The cause of aliasing in graphics and our utilized three anti-aliasing techniques.

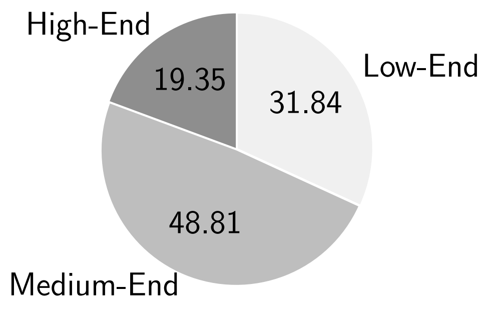


Figure 14: Distribution of involved users' PCs.

buttons easier for users to reach. When the context changes, the mapping will also change to reuse available keys and resolve possible conflicts, so that only a few easy-to-reach "hot" keys are needed in most cases. In general, our method brings negligible overhead to graphics rendering but great convenience to PC users.

Progressive Anti-Aliasing. The screens of PCs are much larger than those of mobile devices, which could exaggerate the subtle, unnoticeable rendering problems of mobile games to a noticeable extent. Thus, when emulating mobile games on PCs we need to make various graphic adaptations. Among all graphics problems, aliasing is the most commonly seen and can often make users feel uncomfortable. As illustrated in Figure 13, the root cause of aliasing lies in the up-conversion from an inadequate-sized off-screen framebuffer to a large emulated screen (*i.e.,* from 720p to 1080p). In comparison, although aliasing also happens on mobile devices with 1080p screens, the relatively small screen size (implying more pixels per inch than PC display) makes it less noticeable to users.

Fundamentally solving the aliasing problem requires source code-level modification to Android games, which is obviously impossible for a general-purpose emulator like DAOW. Hence, we adopt a transparent anti-aliasing method which detects the aliasing phenomenon by constantly checking the size and smoothness of the off-screen framebuffer. If the smoothness is not affected, we would *progressively* apply three existing anti-aliasing techniques by rewriting or adding OpenGL instructions at runtime [20, 23]. First, we attempt to apply off-screen framebuffer size alignment since it can usually make the greatest improvement. This technique, however, is not compatible to all games. Thus, our second choice is multi-sample anti-aliasing (MSAA [26]), which can make effective improvement with moderate GPU overhead and is compatible to most games. Finally, we apply fast approximate anti-aliasing (FXAA [30]) which has some basic effect with minor overhead and is compatible to all games. Otherwise (if the smoothness is affected), we do not apply anti-aliasing techniques to preserve the smoothness.

5 EVALUATION

This section evaluates the major performance and overhead of DAOW for emulating heavy Android games, in comparison to AOVB-EMU, using extensive real-world data collected from our users and micro-benchmark results of various key operations in the emulation. We also compare the performance of Bluestacks with black-box benchmark results.

5.1 Methodology

To practically evaluate the effectiveness of direct Android emulation on Windows, we collect DAOW users' performance and overhead reports every time they run an Android game (as long as they are connected to the Internet), as well as their PC hardware configurations (shown in Figure 14). The performance includes fine-grained running smoothness in each second (refer to §3.1) and the startup time of an Android game. The overhead includes the average memory, CPU and GPU usages during a whole running process of an Android game, as well as the app coverage. As a comparison, we also collect AOVB-EMU users' reports in the same manner during the same period of time for a fair comparison. All the reports are collected with informed consent of opt-in users, and are fully decoupled from personally identifiable information.

Since its launch in Sep. 2017, DAOW has been used by 50M+ users to run ~30K Android games, among which ~8K are heavy 3D games. In comparison, since its launch in Dec. 2015, AOVB-EMU has been used by over 30M users to run

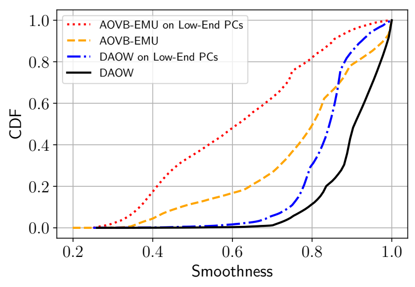


Figure 15: Running smoothness of DAOW and AOVB-EMU.

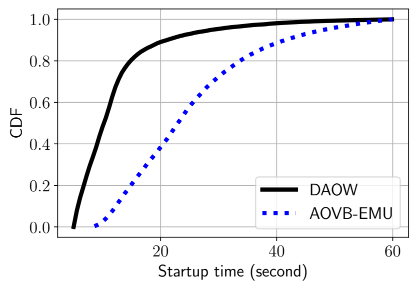


Figure 16: Android games' startup time on DAOW and AOVB-EMU.

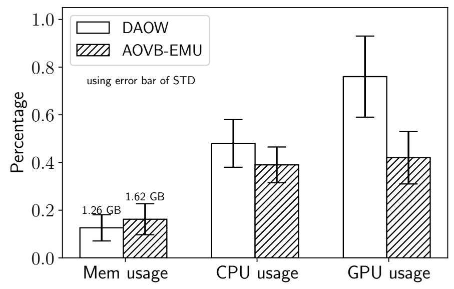


Figure 17: Avg memory, CPU, and GPU usages of DAOW and AOVB-EMU.

40K+ Android games, which fully cover the 8K heavy 3D games. Because our research targets the emulation of heavy 3D games, below we focus on the collected results with regard to heavy 3D games from both systems.

As for Bluestacks, another mainstream AOVB-based emulator owning 250M+ users [35], we are unable to collect its user data at scale. Also, since we are unable to reverse engineer its client, we resort to small-scale black-box benchmarks to approximately evaluate the performance of Bluestacks.

5.2 User-Reported Results

Running smoothness. Figure 15 profiles the running smoothness of both systems on all PCs and low-end PCs, respectively. In general, we observe that DAOW achieves satisfactory ($\geq$ 0.9) smoothness in 60% cases and fluent ($\geq$ 0.8) smoothness in 90% cases. The average smoothness reaches 0.918 and the median is as high as 0.923. In comparison, AOVB-EMU only achieves satisfactory smoothness in 20% cases and fluent smoothness in 50% cases. The average smoothness is 0.76 and the median is 0.79, implying that most users cannot smoothly play heavy games with AOVB-EMU.

When heavy games are emulated on low-end PCs, DAOW can achieve an average smoothness of 0.83 (not satisfactory but still fluent), while the average smoothness of AOVB-EMU sharply falls to 0.57, meaning that users have to suffer from frequent stagnations. On the other hand, better hardware can compensate the overhead of full virtualization to a certain extent: 20% of AOVB-EMU users experience a high smoothness ($>$ 0.9) when running heavy games, and the vast majority of them actually possess high-end PCs. In general, compared to AOVB with manifold optimizations (refer to §3.3), direct Android emulation on Windows essentially improves the smoothness by an average of 21% (from 0.76 to 0.918).

Game startup time. Figure 16 quantifies the startup time of heavy 3D games. We find that both systems can always start up a game within one minute, which is basically acceptable to users. On average, the startup time is 13 seconds with DAOW and 25 seconds with AOVB-EMU, thus achieving an obvious (48%) decrease. This is mainly owing to our adequate utilization of shared memory and memory pool, as well as our efficient emulation of Linux syscalls on Windows.

Memory, CPU, and GPU usages. As depicted in Figure 17, the average memory usage of DAOW (1.26 GB) is 22% smaller than that of AOVB-EMU (1.62 GB). This is because DAOW takes advantage of Windows file mappings and caches to enable fine-grained memory allocation (exemplified in §4.5); in contrast, the separated and complete Linux kernel in AOVB-EMU consumes more memory and often does not return the allocated memory to Windows in time (owing to full virtualization).

On the other hand, we notice that both the CPU and GPU usages of DAOW are higher than those of AOVB-EMU (by 8% for CPU and 34% for GPU). This is the result of DAOW's abandoning full virtualization and having direct access to a PC's hardware—more adequate utilizations of the CPU and GPU bring essentially higher smoothness.

App coverage. Thanks to its using the full-fledged environment of Android-x86 and VirtualBox's full virtualization (§3), AOVB-EMU supports 95% of Android games. For the unsupported games, 26% of them intentionally prevent AOVB-EMU from running them in emulation [46]; 22% are ascribed to technical bugs in Houdini's dynamic binary translation and Android-x86; and the remaining 52% are banned by the game developers through Market-G due to the fairness concerns when they are played with PC keyboards, mouses (rather than finger touches) or emulated sensors such as GPS.

In comparison, although DAOW essentially improves the smoothness, it slightly decreases the app coverage from 95% to 92%. This limitation is mainly attributed to two reasons. First, rarely-used incompatible CPU instructions are not completely handled by our rewriting (§4.3) and dynamic translation (§4.4). Specially, some instructions are not handled for security concerns to ensure the stability of Windows. Second,

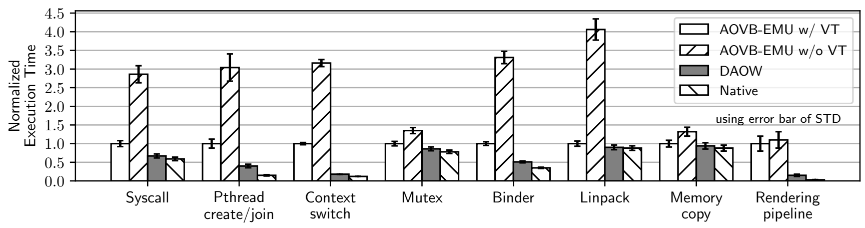


Figure 18: Micro-benchmark results of various key operations in AOVB-EMU, DAOW, and native Android-x86.

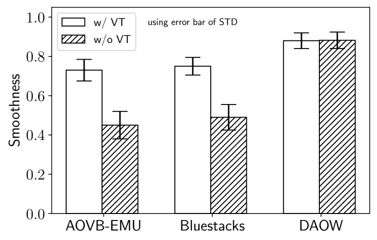


Figure 19: Avg smoothness of AOVB-EMU, Bluestacks, and DAOW.

as mentioned in §4.5, we are not emulating all Linux syscalls in DAOW Kernel Driver to avoid cost-inefficient engineering efforts. In a nutshell, we trade little decrease of compatibility for large increase of smoothness in developing DAOW[3].

5.3 Micro-Benchmark Results

We conduct a series of micro-benchmarks with AOVB-EMU (with or without VT), DAOW, and native Android-x86 execution (abbreviated as Native), on a common PC with a 4-core Intel i5-3470 CPU @3.2GHz, an integrated graphics card, and 4-GB DDR3 RAM. As shown in Figure 18, for each benchmark, we divide the execution time by that of AOVB-EMU with VT for normalization.

First, we examine the execution time of all the syscalls invoked during a play of Vainglory (a typical heavy 3D game). Due to our special design of DAOW Kernel Driver, the overall syscall time of DAOW is 32% shorter than AOVB-EMU with VT while 9% longer than Native. This is a fundamental reason why the performance of DAOW is essentially better than that of AOVB-EMU while close to that of Native.

Second, we perform other kernel-space benchmarks such as threading (Pthread), context switch, synchronization (Mutex), and interprocess communication (Binder). For Pthread creat/join and context switch, DAOW reduces 60% and 80% execution time compared to AOVB-EMU with VT, respectively. In essence, a heavy Android game typically creates hundreds of threads and maintains 20+ active threads; they are scheduled on the 4 CPU cores, incurring ~10000 context switches per second. Therefore, intensive threading brings much more overhead to AOVB-EMU as explained in §3.2. When it comes to Mutex and Binder (which is responsible for interprocess communication between Android services and app instances), DAOW reduces 14% and 49% execution time compared to AOVB-EMU with VT, respectively.

Third, we run user-space benchmarks such as Linpack (Linear system package [16]) and memory copying with 4 cores. For Linpack, DAOW performs only 10% faster than AOVB-EMU with VT. In contrast, AOVB-EMU without VT costs 3× more execution time since it can only provide one core for the guest Android system. For the same reason, AOVB-EMU without VT needs more time for memory copying.

Fourth, we measure the rendering pipeline latency (between the app instance and the graphics driver) which is crucial to graphics processing. Compared to AOVB-EMU with VT, DAOW greatly reduces the latency by 85%, mostly owing to the usage of shared memory for direct bulk data transfer.

Comparison with Bluestacks. Using the same PC mentioned above, we evaluate the performance and overhead of Bluestacks (version 3.56) by manually playing 50 typical heavy 3D games and calculating the smoothness and game startup time. As shown in Figure 19, the smoothness of Bluestacks is slightly higher than that of AOVB-EMU while essentially lower than that of DAOW. Likewise, the average game startup time of Bluestacks (19 seconds) is shorter than that of AOVB-EMU while longer than that of DAOW. In contrast, the average memory usage of Bluestacks (1.86 GB) is larger than that of AOVB-EMU or DAOW. In general, Bluestacks resembles AOVB-EMU in terms of major performance.

6 CONCLUSION

Efficiently emulating heavy Android games on Windows PCs has long been desired, and it is highly challenging. In this paper, we introduce and discuss our design and implementation of DAOW, a widely-adopted direct Android emulation system on Windows x86 PCs. Instead of full virtualization in the cost of the complexity of development, DAOW makes considerate tradeoffs among efficiency, overhead, and compatibility. Real-world user reports solidly confirm the efficacy of DAOW. All in all, our work proves the practical feasibility of efficient cross-OS program execution even for a large number of heavy mobile applications.

[3]It is worth noting that DAOW respects the terms of service of Android apps by *not* attempting to hide its identity as an emulator. In fact, DAOW shares the same Android emulator fingerprints with AOVB-EMU, which are easy to identify for all Android apps.

The idea of direct Android emulation on Windows is also applicable to non-game apps. As a matter of fact, we have observed a few users' running heavy non-game apps with DAOW, indicating that the demand does exist. Although we currently prioritize supporting game apps, our methodology can be extended to other types of mobile apps in principle, of course with more engineering efforts.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful comments, and our shepherd for guiding us through the revision process. Also, we appreciate the valuable suggestions from Pengyu Zhang, Jia Rao, and Rui Zhou. This work is supported in part by the National Key R&D Program of China under grant 2018YFB1004702, the National Natural Science Foundation of China (NSFC) under grants 61822205, 61632013, 61632020, 61432002 and 61471217.

REFERENCES

[1] AMD.Com. 2018. AMD-V Technology for Client Virtualization. https://www.amd.com/en/technologies/virtualization.

[2] Ardalan Amiri Sani, Kevin Boos, Shaopu Qin, and Lin Zhong. 2014. I/O Paravirtualization at the Device File Boundary. In *Proceedings of ACM ASPLOS*. 319–332.

[3] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. 2014. Rio: A System Solution for Sharing I/O Between Mobile Systems. In *Proceedings of ACM MobiSys*. 259–272.

[4] Android-X86.Org. 2016. Android-x86 Vendor Intel Houdini. https://osdn.net/projects/android-x86/scm/git/vendor-intel-houdini/.

[5] Android-X86.Org. 2018. Android-x86 - Porting Android to x86. http://www.android-x86.org/.

[6] Jeremy Andrus, Alexander Van't Hof, Naser AlDuaij, Christoffer Dall, Nicolas Viennot, and Jason Nieh. 2014. Cider: Native Execution of iOS Apps on Android. In *Proceedings of ACM ASPLOS*. 367–382.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of ACM SOSP*. 164–177.

[8] F. Bellard. 2016. QEMU, A Fast and Portable Dynamic Translator. In *Proceedings of USENIX ATC*. 41–46.

[9] BigNox.Com. 2018. Nox Android Emulator. https://www.bignox.com/.

[10] BlueStacks.Com. 2018. BlueStacks 3 Android Emulator. https://www.bluestacks.com/bluestacksgaming-platform-bgp-android-emulator.html.

[11] Amy Chen. 2017. Amy Chen, General Manager of BlueStacks China: Ingenuity of the $100 Billion Market. https://www.facebook.com/bluestacksTW/posts/367309563685501.

[12] Riad Chikhani. 2015. The History of Gaming. https://techcrunch.com/2015/10/31/the-history-of-gaming-an-evolving-community/.

[13] Chromium.Org. 2018. Chrome OS Supporting Android Apps. https://www.chromium.org/chromium-os/chrome-os-systems-supporting-android-apps.

[14] Xingmin Cui, Da Yu, Patrick Chan, Lucas CK Hui, Siu-Ming Yiu, and Sihan Qing. 2014. Cochecker: Detecting Capability and Sensitive Data Leaks from Component Chains in Android. In *Proceedings of Springer Australasian Conference on Information Security and Privacy*. 446–453.

[15] Christoffer Dall, Jeremy Andrus, Alexander Van't Hof, Oren Laadan, and Jason Nieh. 2012. The Design, Implementation, and Evaluation of Cells: A Virtual Smartphone Architecture. *ACM Transactions on Computer Systems* 30, 3 (2012), 9:1–9:31.

[16] Jack Dongarra. 2007. Frequently Asked Questions on the Linpack Benchmark. http://www.netlib.org/utk/people/JackDongarra/faq-linpack.html.

[17] FutureMark.Com. 2010. 3DMark 11 Whitepaper. http://s3.amazonaws.com/download-aws.futuremark.com/3DMark_11_Whitepaper.pdf.

[18] Genymotion.Com. 2018. Genymotion Android Emulator. https://www.genymotion.com/.

[19] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. 2016. Mining Test Repositories for Automatic Detection of UI Performance Regressions in Android Apps. In *Proceedings of ACM Mining Software Repositories Conference*. 13–24.

[20] Songtao He, Yunxin Liu, and Hucheng Zhou. 2015. Optimizing Smartphone Power Consumption Through Dynamic Resolution Scaling. In *Proceedings of ACM MobiCom*. 27–39.

[21] Ding-Yong Hong, Jan-Jan Wu, Pen-Chung Yew, Wei-Chung Hsu, Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2014. Efficient and Retargetable Dynamic Binary Translation on Multicores. *IEEE Transactions on Parallel and Distributed Systems* 25, 3 (2014), 622–632.

[22] Jon Howell, Bryan Parno, and John R. Douceur. 2013. How to Run POSIX Apps in a Minimal Picoprocess. In *Proceedings of USENIX ATC*. 321–332.

[23] Chanyou Hwang, Saumay Pushp, Changyoung Koh, Jungpil Yoon, Yunxin Liu, Seungpyo Choi, and Junehwa Song. 2017. RAVEN: Perception-aware Optimization of Power Consumption for Mobile Games. In *Proceedings of ACM MobiCom*. 422–434.

[24] Intel. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel.

[25] Jide.Com. 2018. Remix OS. http://www.jide.com/remixos.

[26] Khronos.Org. 2018. Multisample anti-aliasing. https://www.khronos.org/opengl/wiki/Multisampling.

[27] Koplayer.Com. 2018. Koplayer Android Emulator. http://www.koplayer.com/.

[28] Robert LiKamWa and Lin Zhong. 2015. Starfish: Efficient Concurrency Support for Computer Vision Applications. In *Proceedings of ACM MobiSys*. 213–226.

[29] LinxTestProject. 2018. LTP - Linux Test Project. http://linux-test-project.github.io/.

[30] Timothy Lottes. 2011. FXAA: Fast Approximate Anti-Aliasing. *NVIDIA white paper* (2011).

[31] Memuplay.Com. 2018. MEmu Android Emulator. http://www.memuplay.com/.

[32] Microsoft.Com. 2016. Windows Subsystem for Linux. https://blogs.msdn.microsoft.com/wsl/2016/04/22/windows-subsystem-for-linux-overview/.

[33] Microsoft.Com. 2018. WSL LTP result. https://github.com/MicrosoftDocs/WSL/tree/live/LTP_Results/16273.

[34] George Osborn. 2016. The Big Screen Opportunity in Southeast Asia. https://newzoo.com/insights/articles/the-big-screen-opportunity-in-southeast-asia/.

[35] Prweb.Com. 2018. BlueStacks Releases the First Android Gaming Platform Ever to Run Android N. https://www.prweb.com/releases/2018/01/prweb15098178.htm.

[36] Rusty Russell. 2008. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM Operating Systems Review* 42, 5 (2008), 95–103.

[37] Bor-Yeh Shen, Wei-Chung Hsu, and Wuu Yang. 2014. A Retargetable Static Binary Translator for the ARM Architecture. *ACM Transactions on Architecture and Code Optimization* 11, 2 (2014), 18:1–18:25.

[38] SuperEvilMegaCorp.Com. 2018. Vainglory. https://play.google.com/store/apps/details?id=com.superevilmegacorp.game.

[39] Tencent.Com. 2018. DAOW Android Game Emulator. https://syzs.qq.com/en/.

[40] Tencent.Com. 2018. MyApp Android Market. https://sj.qq.com/myapp/.

[41] Tencent.Com. 2018. PUBG Mobile. https://play.google.com/store/apps/details?id=com.tencent.ig.

[42] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E Porter. 2016. A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting. In *Proceedings of ACM EuroSys*. 16.

[43] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. 2005. Intel Virtualization Technology. *IEEE Computer* 38, 5 (2005), 48–56.

[44] Unity3d.Com. 2018. Unity - Multiplatform - Publish your game to over 25 platforms. https://unity3d.com/unity/features/multiplatform.

[45] UnrealEngine.Com. 2018. Unreal Engine - Platform Development. https://docs.unrealengine.com/en-us/Platforms.

[46] Timothy Vidas and Nicolas Christin. 2014. Evading Android Runtime Analysis via Sandbox Detection. In *Proceedings of ACM AsiaCCS*. 447–458.

[47] VirtualBox.Org. 2018. Open-source Oracle VM VirtualBox. https://www.virtualbox.org/.

[48] VirtualBox.Org. 2018. VirtualBox - Guest Additions. https://www.virtualbox.org/manual/ch04.html#guestadd-3d.

[49] VirtualBox.Org. 2018. VirtualBox - Technical background. https://www.virtualbox.org/manual/ch10.html#technical-components.

[50] Yu Wang, Rui Tan, Guoliang Xing, Jianxun Wang, Xiaobo Tan, and Xiaoming Liu. 2016. Energy-Efficient Aquatic Environment Monitoring Using Smartphone-Based Robots. *ACM Transactions on Sensor Networks* 12, 3 (2016), 25:1–25:28.

[51] Benjamin Watson, Victoria Spaulding, Neff Walker, and William Ribarsky. 1997. Evaluation of the Effects of Frame Time Variation on VR Task Performance. In *Proceedings of IEEE Virtual Reality Annual International Symposium*. 38–44.

[52] Tom Wijman. 2018. Mobile Revenues Account for More Than 50% of the Global Games Market in 2018. https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/.

[53] Tom Wijman. 2018. TinyDancer: An android library for displaying smoothness from the choreographer. https://github.com/friendlyrobotnyc/TinyDancer.

[54] Da Yu and Wushao Wen. 2012. Non-Access-Stratum Request Attack in E-UTRAN. In *Proceedings of IEEE Computing, Communications and Applications Conference*. 48–53.

[55] Peng Zhao, Kaigui Bian, Tong Zhao, Xintong Song, Jung-Min Jerry Park, Xiaoming Li, Fan Ye, and Wei Yan. 2017. Understanding Smartphone Sensor and App Data for Enhancing the Security of Secret Questions. *IEEE Transactions on Mobile Computing* 16, 2 (2017), 552–565.