Cider: Native Execution of iOS Apps on Android

Jeremy Andrus, Alexander Van't Hof, Naser AlDuaij,
Christoffer Dall, Nicolas Viennot, and Jason Nieh

Department of Computer Science
Columbia University
{jeremya, alexvh, alduaij, cdall, nviennot, nieh}@cs.columbia.edu

Abstract

We present Cider, an operating system compatibility architecture that can run applications built for different mobile ecosystems, iOS or Android, together on the same smartphone or tablet. Cider enhances the domestic operating system, Android, of a device with kernel-managed, per-thread personas to mimic the application binary interface of a foreign operating system, iOS, enabling it to run unmodified foreign binaries. This is accomplished using a novel combination of binary compatibility techniques including two new mechanisms: compile-time code adaptation, and diplomatic functions. Compile-time code adaptation enables existing unmodified foreign source code to be reused in the domestic kernel, reducing implementation effort required to support multiple binary interfaces for executing domestic and foreign applications. Diplomatic functions leverage per-thread personas, and allow foreign applications to use domestic libraries to access proprietary software and hardware interfaces. We have built a Cider prototype, and demonstrate that it imposes modest performance overhead and runs unmodified iOS and Android applications together on a Google Nexus tablet running the latest version of Android.

Categories and Subject Descriptors C.0 [*Computer Systems Organization*]: General–System architectures; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; D.2.11 [*Software Engineering*]: Software Architectures; D.3.4 [*Programming Languages*]: Processors–Run-time environments; D.4.7 [*Operating Systems*]: Organization and Design; D.4.9 [*Operating Systems*]: Systems Programs and Utilities

ASPLOS '14, March 1–4, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2305-5/14/03...$15.00.
http://dx.doi.org/10.1145/2541940.2541972

Keywords Android, iOS, Mobile Computing, Binary Compatibility, Operating System Compatibility

1. Introduction

Mobile devices such as tablets and smartphones are changing the way that computing platforms are designed, from the separation of hardware and software concerns in the traditional PC world, to vertically integrated platforms. Hardware components are integrated together in compact devices using non-standard interfaces. Software is customized for the hardware, often using proprietary libraries to interface with specialized hardware. Applications (apps) are tightly integrated with libraries and frameworks, and often only available on particular hardware devices.

These design decisions and the maturity of the mobile market can limit user choice and stifle innovation. Users who want to run iOS gaming apps on their smartphones are stuck with the smaller screen sizes of those devices. Users who prefer the larger selection of hardware form factors available for Android are stuck with the poorer quality and selection of Android games available compared to the well populated Apple App Store [21]. Android users cannot access the rich multimedia content available in Apple iTunes, and iOS users cannot easily access Flash-based Web content. Some companies release cross-platform variants of their software, but this requires developers to master many different graphical, system, and library APIs, and creates additional support and maintenance burden on the company. Many developers who lack such resources choose one platform over another, limiting user choice. Companies or researchers that want to build innovative new devices or mobile software platforms are limited in the functionality they can provide because they lack access to the huge app base of existing platforms. New platforms without an enormous pool of user apps face the difficult, if not impossible, task of end user adoption, creating huge barriers to entry into the mobile device market.

While virtual machines (VMs) are useful for desktop and server computers to run apps intended for one platform on a different platform [36, 44], using them for smartphones and tablets is problematic for at least two reasons. First, mobile devices are more resource constrained, and running an entire additional operating system (OS) and user space environment in a VM just to run one app imposes high overhead. High overhead and slow system responsiveness are much less acceptable on a smartphone than on a desktop computer because smartphones are often used for just a few minutes or even seconds at a time. Second, mobile devices are tightly integrated hardware platforms that incorporate a plethora of devices, such as GPUs, that use non-standardized interfaces. VMs provide no effective mechanism to enable apps to directly leverage these hardware device features, severely limiting performance and making existing VM-based approaches unusable on smartphones and tablets.

To address these problems, we created *Cider*, an OS compatibility architecture that can simultaneously run apps written and compiled for different mobile ecosystems, iOS or Android, simultaneously on the same smartphone or tablet. Cider runs *domestic* binaries, those developed for a given device's OS, the domestic OS, and *foreign* binaries, those developed for a different OS, the foreign OS, together on the same device. In our prototype, Android is the domestic OS, running domestic Android apps, and iOS is the foreign OS. We use the terms foreign and iOS, and domestic and Android interchangeably. Cider defines a *persona* as an execution mode assigned to each *thread* in the system, identifying the thread as executing either foreign or domestic code, using a foreign persona or domestic persona, respectively. Cider supports multiple personas within a single process by extending the domestic kernel's application binary interface (ABI) to be aware of both foreign and domestic threads.

Cider provides OS compatibility by augmenting the domestic Android kernel with the ability to simultaneously present both a domestic kernel ABI as well as a foreign kernel ABI. Foreign user space code interacts with a Cider-enabled kernel in exactly the same way as a foreign kernel, i.e., iOS apps trap into the Linux kernel exactly as if they were trapping into a kernel running on an iPhone or iPad. Modifying the domestic kernel in this way allows Cider both to avoid the traditional VM overhead of running a complete instance of a foreign kernel, and reuse and run unmodified foreign user space library code. The latter is essential since mobile ecosystem libraries and frameworks are often complex, closed-source, and proprietary.

To run unmodified foreign libraries and apps on a domestic OS, we had to overcome two key challenges: the difficulty of porting and reimplementing complex functionality of one OS in another, and the use of proprietary and opaque kernel interfaces to access custom hardware. In particular, tightly integrated libraries on mobile devices will often directly access proprietary hardware resources of a particular device through opaque kernel interfaces such as the `ioctl` system call. Proprietary and opaque foreign kernel interfaces cannot easily be implemented in the domestic kernel, and custom foreign hardware is often missing from the domestic device. Our solution takes advantage of two aspects of mobile ecosystems. First, although user space libraries and frameworks are often proprietary and closed, even closed mobile ecosystems increasingly build on open source kernel code through well-defined interfaces; iOS builds on the open source XNU kernel [6]. Second, although libraries on mobile devices will access custom hardware through opaque kernel interfaces, the actual functionality provided is often cross platform as companies mimic the best features of their competitors' devices such as the use of touchscreens for input and OpenGL ES for graphics on mobile ecosystems.

Based on these observations, Cider supports running unmodified foreign apps on a domestic OS through a novel combination of binary compatibility techniques, including two new OS compatibility mechanisms. Cider introduces *duct tape*, a novel compile-time code adaptation layer, that allows unmodified foreign kernel code to be directly compiled into the domestic kernel. Foreign binaries can then use these new kernel services not otherwise present in the domestic kernel. Brute force implementation of these services and functionality can be error-prone and tedious. Duct tape maximizes reuse of available foreign open source OS code to substantially reduce implementation effort and coding errors. Cider introduces *diplomatic functions* to allow foreign apps to use domestic libraries to access proprietary software and hardware interfaces on the device. A diplomatic function is a function which temporarily switches the persona of a calling thread to execute domestic code from within a foreign app, or vice-versa. Using diplomatic functions, Cider replaces calls into foreign hardware-managing libraries, such as OpenGL ES, with calls into domestic libraries that manage domestic hardware, such as a GPU. Diplomatic functions make it possible to deliver the same library functionality required by foreign apps without the need to reverse engineer and reimplement the opaque foreign kernel interfaces used by proprietary foreign libraries.

Using these OS compatibility mechanisms, we built a Cider prototype that can run unmodified iOS and Android apps on Android devices. We leverage existing software infrastructure as much as possible, including unmodified frameworks across both iOS and Android ecosystems. We demonstrate the effectiveness of our prototype by running various iOS apps from the Apple App Store together with Android apps from Google Play on a Nexus 7 tablet running the latest version of Android. Users can interact with iOS apps using multi-touch input, and iOS apps can leverage GPU hardware to display smooth, accelerated graphics. Our microbenchmark and app measurements show that Cider imposes modest performance overhead, and can deliver faster performance for iOS apps than corresponding Android counterparts on Android hardware. The faster performance is due to the greater efficiencies of running native iOS code instead of interpreted bytecode as used by Android.

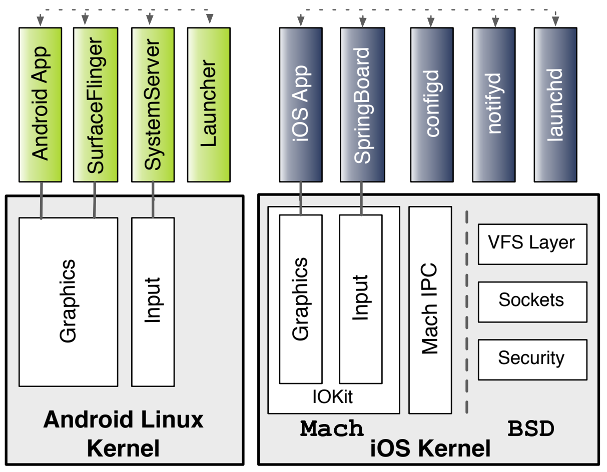


Figure 1: Android and iOS architectures

2. Overview of Android and iOS

To understand how Cider runs iOS apps on Android, we first provide a brief overview of the operation of Android and iOS. We limit our discussion to central components providing app startup, graphics, and input on both systems.

Figure 1 shows an overview of these two systems. Android is built on the Linux kernel and runs on ARM CPUs. The Android framework consists of a number of system services and libraries used to provide app services, graphics, input, and more. For example, SystemServer starts Launcher, the home screen app on Android, and SurfaceFlinger, the rendering engine which uses the GPU to compose all the graphics surfaces for different apps and display the final composed surface to the screen.

Each Android app is compiled into Dalvik bytecode (dex) format, and runs in a separate Dalvik VM instance. When a user interacts with an Android app, input events are delivered from the Linux kernel device driver through the Android framework to the app. The app displays content by obtaining window memory (a graphics surface) from SurfaceFlinger and draws directly into the window memory. An app can attach an OpenGL context to the window memory and use the OpenGL ES framework to render hardware-accelerated graphics into the window memory using the GPU.

iOS runs on ARM CPUs like Android, but has a very different software ecosystem. iOS is built on the XNU kernel [6], a hybrid combination of a monolithic BSD kernel and a Mach microkernel running in a single kernel address space. XNU leverages the BSD socket and VFS subsystems, but also benefits from the virtual memory management [42] and IPC mechanisms [47] provided by Mach. iOS makes extensive use of both BSD and Mach XNU services. The iOS user space framework consists of a number of user space daemons. `launchd` is responsible for booting the system, and starting, stopping, and maintaining services and apps. `launchd` starts Mach IPC services such as `configd`, the system configuration daemon, `notifyd`, the asynchronous notification server, and `mediaserverd`, the audio/video server. `launchd` also starts the SpringBoard app, which displays the iOS home screen, handles and routes user input to apps, and uses the GPU to compose app display surfaces onto the screen. SpringBoard is analogous to an amalgamation of SurfaceFlinger, Launcher, and SystemServer in Android.[1]

iOS apps are written in Objective-C, and compiled and run as native binaries in an extended Mach-O [4] format. In contrast, the Java archives used by Android apps are interpreted by the Dalvik VM, not loaded as native binaries. On iOS, apps are loaded directly by a kernel-level Mach-O loader which interprets the binary, loads its text and data segments, and jumps to the app entry point. Dynamically linked libraries are loaded by `dyld`, a user space binary, which is invoked from the Mach-O loader. Examples of frequently used libraries in iOS apps include UIKit, the user interface framework; QuartzCore and OpenGL ES, the core graphics frameworks; and WebKit, the web browser engine.

3. System Integration

Cider provides a familiar user experience when running iOS apps on Android. Apps are launched from the Android home screen, just like any other Android app, and users can switch seamlessly between domestic Android apps and foreign iOS apps. Cider accomplishes this without running the iOS XNU kernel or the SpringBoard app. Cider overlays a file system (FS) hierarchy on the existing Android FS, and provides several background user-level services required by iOS apps. The overlaid FS hierarchy allows iOS apps to access familiar iOS paths, such as `/Documents`, and the background services establish key microkernel-style functionality in user space necessary to run iOS apps. Figure 2 shows an overview of the integration of iOS functionality into our Android-based Cider prototype.

iOS apps running on Cider need access to a number of framework components including iOS libraries and user-level Mach IPC services. Instead of reimplementing these components, a task which would require substantial engineering and reverse-engineering efforts, we simply copy the existing binaries from iOS and run them on the domestic system, leveraging Cider's OS compatibility architecture. Background user-level services such as `launchd`, `configd`, and `notifyd` were copied from an iOS device, and core framework libraries were copied from the Xcode SDK, Apple's development environment.

To provide seamless system integration, and minimize Android user space changes, Cider introduces a proxy service, *CiderPress*. The launch procedure and binary format of iOS and Android apps are completely different, so iOS app startup and management cannot be directly performed by the Android framework. *CiderPress* is a standard Android app that integrates launch and execution of an iOS app with Android's Launcher and system services. It is directly started by

[1] In newer versions of iOS, graphics rendering and input event pumping have been offloaded to `backboardd`.

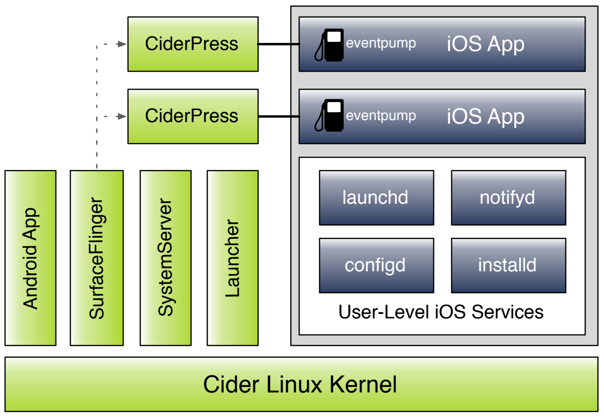


Figure 2: System integration overview

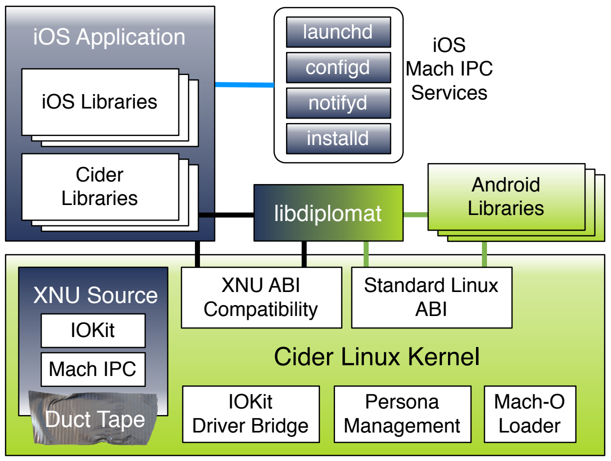


Figure 3: Overview of Cider architecture

Android's Launcher, receives input such as touch events and accelerometer data from the Android input subsystem, and its life cycle is managed like any other Android app. *CiderPress* launches the foreign binary, and proxies its own display memory, incoming input events, and app state changes to the iOS app. An Android Launcher short cut pointing to *CiderPress* allows a user to click an icon on the Android home screen to start an iOS app, and the proxied display surface allows screen shots of the iOS app to appear in Android's recent activity list. Proxied app state changes allow the iOS app to be started, stopped, and paused (put into the background) like a standard Android app.

4. Architecture

The primary goal of Cider is to run unmodified iOS binaries on Android, including iOS apps, frameworks, and services. This is challenging because iOS binaries are built to run on iOS and XNU, not Android and Linux. The XNU kernel provides a different system call (syscall) interface from Linux, and iOS apps make extensive use of OS services not available on Linux, such as Mach IPC [5]. Clearly, more than just binary compatibility is necessary: Cider must make Android compatible with iOS. Cider's multi-persona OS compatibility architecture solves this problem.

Figure 3 provides an overview of the Cider OS compatibility architecture which can be divided into three key components. First, Cider provides XNU kernel compatibility by implementing a Mach-O loader for the Linux kernel, supporting the XNU syscall interface, and facilitating proper signal delivery – together referred to as the kernel application binary interface (ABI). Second, Cider provides a *duct tape* layer to import foreign kernel code that supports syscalls and subsystems not available in Linux. Third, Cider introduces diplomatic functions, implemented in the `libdiplomat` iOS library, to support apps that use closed iOS libraries which issue device-specific calls such as opaque Mach IPC messages or `ioctl`s. We describe these components in further detail in the following sections.

4.1 Kernel ABI

At a high-level, providing an OS compatibility solution is straightforward. The interaction between apps and an OS is defined by the kernel application binary interface (ABI). The ABI defines all possible interactions between apps and the kernel. The ABI consists of a binary loader which interprets the physical contents of the application binary and associated libraries, asynchronous signal delivery, and the syscall interface. To run iOS apps in Android, we need to implement these three components of the XNU ABI in the Linux kernel.

Cider provides a Mach-O binary loader built into the Linux kernel to handle the binary format used by iOS apps. When a Mach-O binary is loaded, the kernel tags the current thread with an iOS *persona*, used in all interactions with user space. Personas are tracked on a per-thread basis, inherited on `fork` or `clone`, and enable processes with multiple threads to simultaneously support multiple personas. Multi-persona processes play a key role in supporting hardware-accelerated graphics as discussed in Section 5.3.

Cider provides a translation layer for asynchronous signal delivery that converts signals from the Linux kernel (generated from events such as an illegal instruction, or a segmentation fault) into signals which would have been generated by the XNU kernel. The XNU signals are then delivered to iOS applications, as appropriate, where they can be properly handled. Cider also converts XNU signals generated programmatically from iOS applications into corresponding Linux signals, so they can be delivered to non-iOS applications or threads. Cider uses the persona of a given thread to deliver the correct signal. Android apps (or threads) can deliver signals to iOS apps (or threads) and vice-versa.

Since an app's primary interface to the kernel is through syscalls, Cider provides multiple syscall interfaces to support different kernel ABIs. Cider maintains one or more syscall dispatch tables for each persona, and switches among them based on the persona of the calling thread and the syscall number. Cider is aware of XNU's low-level syscall interface, and translates things such as function parameters and CPU flags into the Linux calling convention, making it possible to directly invoke existing Linux syscall implementations. Different kernels have different syscall entry and exit code paths. For example, iOS apps can trap into the kernel in four different ways depending on the system call being executed, and many XNU syscalls return an error indication through CPU flags where Linux would return a negative integer. Cider manages these syscall entry and exit path differences through persona-tagged support functions.

Because the iOS kernel, XNU, is based on POSIX-compliant BSD, most syscalls overlap with functionality already provided by the Linux kernel. For these syscalls, Cider provides a simple wrapper that maps arguments from XNU structures to Linux structures and then calls the Linux implementation. To implement XNU syscalls that have no corresponding Linux syscall, but for which similar Linux functionality exists, the wrapper reuses one or more existing Linux functions. For example, Cider implements the `posix_spawn` syscall, which is a flexible method of starting a thread or new application, by leveraging the Linux `clone` and `exec` syscall implementations.

4.2 Duct Tape

Simple wrappers and combinations of existing syscall implementations are not enough to implement the entire XNU ABI. Many XNU syscalls require a core subsystem that does not exist in the Linux kernel. Reimplementing these mechanisms would be a difficult and error-prone process. Mach IPC is a prime example of a subsystem missing from the Linux kernel, but used extensively by iOS apps. It is a rich and complicated API providing inter-process communication and memory sharing. Implementing such a subsystem from scratch in the Linux kernel would be a daunting task. Given the availability of XNU source code, one approach would be to try to port such code to the Linux kernel. This approach is common among driver developers, but is time consuming and error-prone given that different kernels have completely different APIs and data structures. For example, the Linux kernel offers a completely different set of synchronization primitives from the XNU kernel.

To address this problem, Cider introduces *duct tape*, a novel compile-time code adaptation layer that supports *cross-kernel compilation*: direct compilation of unmodified foreign kernel source code into a domestic kernel. Duct tape translates foreign kernel APIs such as synchronization, memory allocation, process control, and list management, into domestic kernel APIs. The resulting module or subsystem is a first-class member of the domestic kernel and can be accessed by both foreign and domestic apps.

To duct tape foreign code into a domestic kernel, there are three steps. First, three distinct coding *zones* are created within the domestic kernel: the domestic, foreign, and duct tape zones. Code in the domestic zone cannot access symbols in foreign zone, and code in the foreign zone cannot access symbols in the domestic zone. Both foreign and domestic zones can access symbols in the duct tape zone, and the duct tape zone can access symbols in both the foreign and domestic zones. Second, all external symbols and symbol conflicts with domestic code are automatically identified in the foreign code. Third, conflicts are remapped to unique symbols, and all external foreign symbols are mapped to appropriate domestic kernel symbols. Simple symbol mapping occurs through preprocessor tokens or small static inline functions in the duct tape zone. More complicated external foreign dependencies require some implementation effort within the duct tape or domestic zone.

Duct tape provides two important advantages. First, foreign code is easier to maintain and upgrade. Because the original foreign code is not modified, bug fixes, security patches, and feature upgrades can be applied directly from the original maintainer's source repository. Second, the code adaptation layer created for one subsystem is directly reusable for other subsystems. For example, most kernel code will use locking and synchronization primitives, and an adaptation layer translating these APIs and structures for one foreign subsystem into the domestic kernel will work for all subsystems from the same foreign kernel. As more foreign code is added, the duct tape process becomes simpler.

Cider successfully uses duct tape to add three different subsystems from the XNU kernel into Android's Linux kernel: pthread support, Mach IPC, and Apple's I/O Kit device driver framework, the latter is discussed in Section 5.1. iOS pthread support differs substantially from Android in functional separation between the pthread library and the kernel. The iOS user space pthread library makes extensive use of kernel-level support for mutexes, semaphores, and condition variables, none of which are present in the Linux kernel. This support is found in the `bsd/kern/pthread_support.c` file in the XNU source provided by Apple. Cider uses duct tape to directly compile this file without modification.

The Mach IPC subsystem is significantly more complicated than pthread support. Cider uses duct tape to directly compile the majority of Mach IPC into the Linux kernel. However, code relying on the assumption of a deeper stack depth in XNU required reimplementation. In particular, XNU's Mach IPC code uses recursive queuing structures, something disallowed in the Linux kernel. This queuing was rewritten to better fit within Linux.

Note that the BSD kqueue and kevent notification mechanisms were easier to support in Cider as user space libraries because of the availability of existing open source user-level implementations [25]. Because they did not need to be incorporated into the kernel, they did not need to be incorporated using duct tape, but simply via API interposition [27].

4.3 Diplomatic Functions

XNU kernel ABI support coupled with duct tape allows Cider to successfully handle most kernel interactions from iOS apps, however, the behavior of several key syscalls is not well-defined. For example, the `ioctl` syscall passes a driver-specific request code and a pointer to memory, and its behavior is driver-specific. Without any understanding of how `ioctls` are used, simply implementing the syscall itself is of little benefit to apps.

Additionally, Mobile apps often make use of closed and proprietary hardware and software stacks. Compressed consumer production time lines and tight vertical integration of hardware and software lead developers to discard cumbersome abstractions present on Desktop PCs in favor of custom, direct hardware communication. For example, the OpenGL ES libraries on both Android and iOS directly communicate with graphics hardware (GPU) through proprietary software and hardware interfaces using device-specific ioctls (Android), or opaque IPC messages (iOS).

Cider cannot simply implement kernel-level support for foreign, closed libraries which directly manipulate hardware through proprietary interfaces. Not only are the semantics of the closed library unknown, but they are likely closely tied to foreign hardware not present on the domestic device.

Cider solves the problem of direct access to proprietary hardware through the novel concept of *diplomatic functions*. A diplomatic function temporarily switches the persona of a calling thread to execute domestic functions from within a foreign app. A thread's persona, or execution mode, selects the kernel ABI personality and thread local storage (TLS) information used during execution. The TLS area contains per-thread state such as `errno` and a thread's ID.

Cider's diplomatic function support comprises three key components: **(1)** The ability to load and interpret domestic binaries and libraries within a foreign app. This involves the use of a domestic loader compiled as a foreign library. For example, Cider incorporates an Android ELF loader cross-compiled as an iOS library. **(2)** Kernel-level persona management at a thread level including both kernel ABI and TLS data management. The Cider kernel maintains kernel ABI and TLS area pointers for every persona in which a given thread executes. A new syscall (available from all personas) named `set_persona`, switches a thread's persona. Different personas use different TLS organizations, e.g., the `errno` pointer is at a different location in the iOS TLS than in the Android TLS. After a persona switch, any kernel traps or accesses to the TLS area will use the new persona's pointers. This allows the thread to invoke functions from libraries compiled for a different persona. For example, in Section 5.3 we describe how, using diplomatic functions, an iOS app can load and execute code in the Android OpenGL ES library and thereby directly interact with the underlying GPU hardware **(3)** The ability to mediate foreign function calls into domestic libraries, appropriately loading, switching, and managing the domestic persona; this is done through *diplomats*. A diplomat is a function stub that uses an arbitration process to switch the current thread's persona, invoke a function in the new persona, switch back to the calling function's persona, and return any results.

The Cider arbitration process for calling a domestic function from foreign code through a diplomat is as follows: **(1)** Upon first invocation, a diplomat loads the appropriate domestic library and locates the required entry point, storing a pointer to the function in a locally-scoped static variable for efficient reuse. **(2)** The arguments to the domestic function call are stored on the stack. **(3)** The `set_persona` syscall is invoked from the foreign persona to switch the calling thread's kernel ABI, and TLS area pointer to their domestic values. **(4)** The arguments to the domestic function call are restored from the stack. **(5)** The domestic function call is directly invoked through the symbol stored in step 1. **(6)** Upon return from the domestic function, the return value is saved on the stack. **(7)** The `set_persona` syscall is invoked from the domestic persona to switch the kernel ABI and TLS area pointer back to the foreign code's values. **(8)** Any domestic TLS values, such as `errno`, are appropriately converted and updated in the foreign TLS area. **(9)** The domestic function's return value is restored from the stack, and control is returned to the calling foreign function.

Because Cider maintains kernel ABI and TLS information on a per-thread basis, a single app can simultaneously execute both foreign and domestic code in multiple threads. For example, while one thread executes complicated OpenGL ES rendering algorithms using the domestic persona, another thread in the same app can simultaneously process input data using the foreign persona. Unlike previous binary compatibility work [13, 19, 20, 23], which only allowed an app to use a single persona, Cider allows an app to switch among personas and use multiple personas simultaneously. Cider can replace an entire foreign library with diplomats, or it can define a single diplomat to use targeted functionality in a domestic library such as popping up a system notification. Section 5.3 describes how Cider replaces the iOS OpenGL ES library with diplomats that use Android's OpenGL ES libraries.

5. iOS Subsystems on Android

We highlight three examples of key iOS subsystems to show how Cider's OS compatibility architecture supports each subsystem on an Android device. Due to space constraints, we only provide an overview of the XNU I/O Kit subsystem, multi-touch input, and graphics support. A full discussion of all supported subsystems is beyond the scope of this paper.

5.1 Devices

Cider uses duct tape to make Android hardware devices available to iOS apps via Apple's I/O Kit. I/O Kit is Apple's open source driver framework based on NeXTSTEP's DriverKit. It is written primarily in a restricted subset of C++, and is accessed via Mach IPC. iOS Apps and libraries access Android devices via I/O Kit drivers in Cider exactly as they would access Apple devices on iOS.

To directly compile the I/O Kit framework, Cider added a basic C++ runtime to the Linux kernel based on Android's Bionic. Linux kernel Makefile support was added such that compilation of C++ files from within the kernel required nothing more than assigning an object name to the `obj-y` Makefile variable. Cider uses duct tape and its Linux kernel C++ runtime to directly compile the majority of the I/O Kit code, found in the XNU `iokit` source directory, without modification.[2] In fact, we initially compiled Cider with the I/O Kit framework found in XNU v1699.24.8, but later directly applied source code patches upgrading to the I/O Kit framework found in XNU v2050.18.24.

Cider makes devices available via both the Linux device driver framework and I/O Kit. Using a small hook in the Linux `device_add` function, Cider creates a Linux device node I/O Kit registry entry (a device class instance) for every registered Linux device. Cider also provides an I/O Kit driver class for each device that interfaces with the corresponding Linux device driver. This allows iOS apps to access devices as well as query the I/O Kit registry to locate devices or properties.

For example, iOS apps expect to interact with the device framebuffer through a C++ class named `AppleM2CLCD` which derives from the `IOMobileFramebuffer` C++ class interface. Using the C++ runtime support added to the Linux kernel, the Cider prototype added a single C++ file in the Nexus 7 display driver's source tree that defines a class named `AppleM2CLCD`. This C++ class acts as a thin wrapper around the Linux device driver's functionality. The class is instantiated and registered as a driver class instance with I/O Kit through a small interface function called on Linux kernel boot. The duct taped I/O Kit code matches the C++ driver class instance with the Linux device node (previously added from the Linux `device_add` function). After the driver class instance is matched to the device class instance, iOS user space can query and use the device as a standard iOS device. We believe that a similar process can be done for most devices found on a tablet or smartphone.

5.2 Input

No user-facing app would be complete without input from both the user and devices such as the accelerometer. In iOS, every app monitors a Mach IPC port for incoming low-level event notifications and passes these events up the user space stack through gesture recognizers and event handlers. The events sent to this port include mouse, button, accelerometer, proximity and touch screen events. A system service, SpringBoard, is responsible for communicating with the lower-level iOS input system, determining which app should receive the input, and then sending corresponding input events to the app via its respective Mach IPC port.

[2] Portions of the I/O Kit codebase such as `IODMAController.cpp` and `IOInterruptController.cpp` were not necessary as they are primarily used by I/O Kit drivers communicating directly with hardware.

Replicating the interactions between SpringBoard and the lower-level input system on Android would potentially involve reverse engineering complex device driver interactions with input hardware that is not present in Android devices.

Cider takes a simpler approach to provide complete, interactive, multi-touch input support for iOS. Cider does not attempt to replicate the lower-level iOS input system. Instead, it simply reads events from the Android input system, translates them as necessary into a format understood by iOS apps, and sends them to the Mach IPC port used by apps to receive events. In particular, Cider creates a new thread in each iOS app to act as a bridge between the Android input system and the Mach IPC port expecting input events. This thread, the *eventpump* seen in Figure 2, listens for events from the Android *CiderPress* app on a BSD socket. It then pumps those events into the iOS app via Mach IPC. In the future, this intermediary thread could be avoided with a minimal Linux Mach IPC wrapper ABI. Using this approach, Cider is able to provide complete, interactive, multi-touch input support in a straightforward manner. Panning, pinch-to-zoom, iOS on-screen keyboards and keypads, and other input gestures are also all completely supported.

5.3 Graphics

Cider leverages kernel-level personas through diplomatic functions to provide 2D and 3D graphics support in iOS apps. User space libraries such as WebKit, UIKit, and CoreAnimation render content, such as buttons, text, web pages, and images, using the OpenGL ES and IOSurface iOS libraries. These libraries communicate directly to the iOS kernel via Mach IPC. They use I/O Kit drivers to allocate and share graphics memory, control hardware facilities such as frame rates and subsystem power, and perform more complex rendering tasks such as those required for 3D graphics.

Supporting the iOS graphics subsystem is a huge OS compatibility challenge; highly optimized user space libraries are tightly integrated with mobile hardware. Libraries such as OpenGL ES, call into a set of proprietary, closed source, helper libraries which, in turn, use opaque Mach IPC messages to closed source kernel drivers that control black-box pieces of hardware. Opaque Mach IPC calls to kernel drivers are essentially used as device-specific syscalls. Unlike the modern desktop OS, there are no well-defined mobile interfaces to graphics acceleration hardware, such as the Direct Rendering Infrastructure used by the X Window system. Neither implementing kernel-level emulation code nor duct taping a piece GPU driver code, if it were even available, will solve this problem.

Cider enables 2D and 3D graphics in iOS apps through a novel combination of I/O Kit Linux driver wrappers, and diplomatic IOSurface and OpenGL ES libraries. The IOSurface iOS library provides a zero-copy abstraction for all graphics memory in iOS. An IOSurface object can be used to render 2D graphics via CPU-bound drawing routines, efficiently passed to other processes or apps via Mach IPC, and even used as the backing memory for OpenGL ES textures in 3D rendering. Cider interposes diplomatic functions on key IOSurface API entry points such as `IOSurfaceCreate`. These diplomats call into Android-specific graphics memory allocation libraries such as `libgralloc`. Well-known API interposition techniques are used to force iOS apps to link against the Cider version of a particular entry point.

To support more complicated 2D and 3D graphics, Cider replaces the entire iOS OpenGL ES library with diplomats. We leverage the fact that while the implementation of proprietary libraries, such as OpenGL ES, and their interface to kernel drivers is closed, the app-facing API is well-known, and is typically similar across platforms such as iOS and Android.[3] The iOS OpenGL ES library consists of two parts: the standardized OpenGL ES API [29, 31] and the Apple-specific *EAGL* [7] extensions. Cider provides a replacement iOS OpenGL ES library with a diplomat for every exported symbol in both of these categories.

For standard OpenGL ES API entry points, Cider provides a set of diplomats that use the arbitration process, described in Section 4.3, to load, initialize, and call into the Android OpenGL ES libraries. Because each of these entry points has a well-defined, standardized function prototype, the process of creating diplomats was automated by a script. This script analyzed exported symbols in the iOS OpenGL ES Mach-O library, searched through a directory of Android ELF shared objects for a matching export, and automatically generated diplomats for each matching function.

Cider provides diplomats for Apple's EAGL extensions that call into a custom Android library to implement the required functionality. Apple-specific EAGL extensions, used to control window memory and graphics contexts, do not exist on Android. Fortunately, the EAGL extensions are Apple's replacement for the Native Platform Graphics Interface Layer (EGL) standard [30], and this is implemented in an Android EGL library. To support Apple's EAGL extensions, Cider uses a custom domestic Android library, called `libEGLbridge`, that utilizes Android's `libEGL` library and SurfaceFlinger service to provide functionality corresponding to the missing EAGL functions. Diplomatic EAGL functions in the Cider OpenGL ES library call into the custom Android `libEGLbridge` library to fully support Apple's EAGL APIs in iOS apps. Allocating window memory via the standard Android SurfaceFlinger service also allows Cider to manage the iOS display in the same manner that all Android app windows are managed.

6. Experimental Results

We have implemented a Cider prototype for running both iOS and Android apps on an Android device, and present some experimental results to measure its performance. We compared three different Android system configurations to measure the performance of Cider: (1) Linux binaries and Android apps running on unmodified (vanilla) Android, (2) Linux binaries and Android apps running on Cider, and (3) iOS binaries and apps running on Cider. For our experiments, we used a Nexus 7 tablet with a 1.3 GHz quad-core NVIDIA Tegra 3 CPU, 1 GB RAM, 16 GB of flash storage, and a 7" 1280x800 display running Android 4.2 (Jelly Bean). We also ran iOS binaries and apps on a jailbroken iPad mini with a 1 GHz dual-core A5 CPU, 512 MB RAM, 16 GB of flash storage, and a 7.9" 1024x768 display running iOS 6.1.2. Since the iPad mini was released around the same time as the Nexus 7 and has a similar form factor, it provides a useful point of comparison even though it costs 50% more.

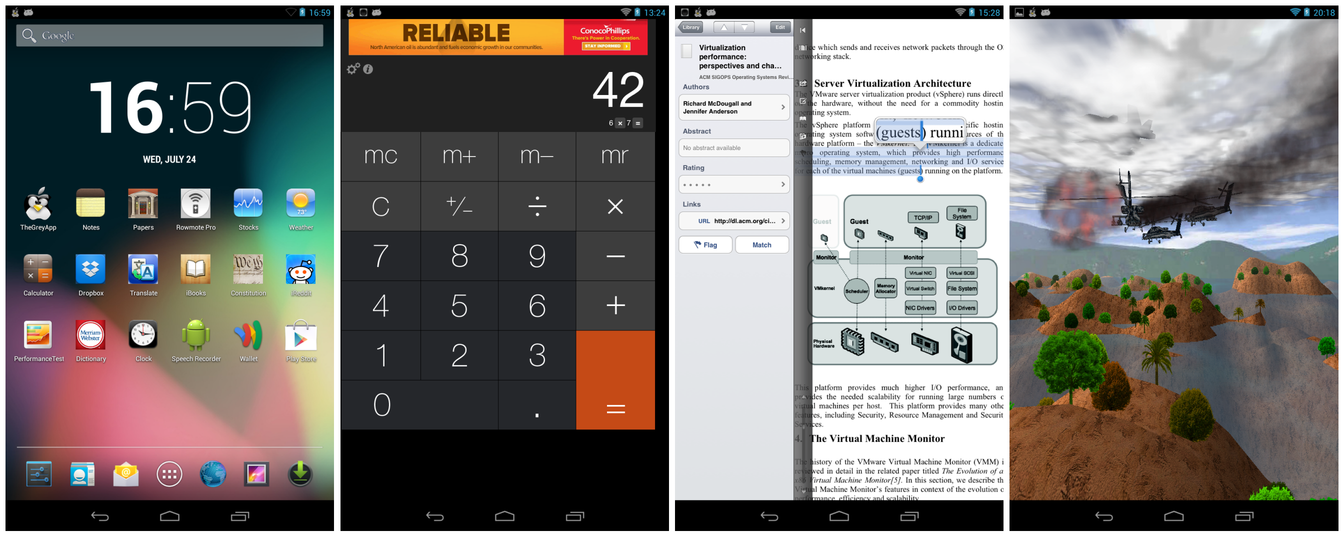
We used both microbenchmarks and real apps to evaluate the performance of Cider. To measure the latency of common low-level OS operations, we used microbenchmarks from `lmbench` 3.0 and compiled two versions: an ELF Linux binary version, and a Mach-O iOS binary version, using the standard Linux GCC 4.4.1 and Xcode 4.2.1 compilers, respectively. We used four categories of `lmbench` tests: basic operations, syscalls and signals, process creation, and local communication and file operations. To measure real app performance, we used comparable iOS and Android PassMark apps available from the Apple App Store [37] and Google Play [38], respectively. PassMark conducts a wide range of resource intensive tests to evaluate CPU, memory, I/O, and graphics performance. We used PassMark because it is a widely used, commercially-supported app available on both iOS and Android, and provides a conservative measure of various aspects of app performance. We normalize all results using vanilla Android performance as the baseline to compare across systems. This is useful to measure Cider performance overhead, and also provides some key observations regarding the characteristics of Android and iOS apps.

6.1 Obtaining iOS Apps

The iOS apps used in our evaluation were downloaded from the Apple App Store. In the future, we envision that developers and app distributors would be incentivized to provide alternative distribution methods. For example, Google Play might be incentivized to take advantage of Cider to make a greater number and higher quality of apps available for Android devices. However, using the App Store required a few more steps to install the applications on an Android device because of various security measures used by Apple.

App Store apps, unlike iOS system apps such as Stocks, are encrypted and must be decrypted using keys stored in encrypted, non-volatile memory found in an Apple device. We modified a widely used script [43] to decrypt apps on any jailbroken iOS device using gdb [22]. To illustrate this point, we used an old iPhone 3GS with iOS 5.0.1 for this purpose. The script decrypts the app, and then re-packages the decrypted binary, along with any associated data files, into a single `.ipa` file (iOS App Store Package). Each `.ipa` file

[3] Cider could also leverage a Direct3D to OpenGL translation layer from the Wine project to support Windows Mobile devices.


16:59
WED, JULY 24
RELIABLE
42
mc
m+
m–
mr
C
+/–
÷
×
7
8
9
–
4
5
6
+
1
2
3
0
.
=
Server Virtualization Architecture
The Virtual Machine Monitor

(a) Cider home screen (b) Calculator Pro for iPad Free (c) Papers for iOS by Mekentosj B.V. (d) PassMark 3D Benchmark

Figure 4: Cider displaying and running iOS apps

was copied to the Cider prototype, and a small background process automatically unpacked each `.ipa` and created Android shortcuts on the Launcher home screen, pointing each one to the *CiderPress* Android app. The iOS app icon was used for the Android shortcut. Decrypted iOS apps work on Cider exactly as they would on an iPhone or iPad, including displaying ads using Apple's iAd framework.

Figure 4 shows screenshots of the Nexus 7 tablet with various iOS apps that we installed and ran on the device (from left to right): the Nexus 7 home screen with iOS and Android apps, the *Calculator Pro for iPad Free* [3], one of the top three free utilities for iPad, displaying a banner ad via the iAd framework, the highly-rated *Papers* [11] app highlighting text in a PDF, and the *PassMark* [37] app running the 3D performance test.

6.2 Microbenchmark Measurements

Figure 5 shows the results of running `lmbench` microbenchmarks on the four system configurations. Vanilla Android performance is normalized to one in all cases, and the results are not explicitly shown. Measurements are latencies, so smaller numbers are better. Measurements are shown at two scales to provide a clear comparison despite the wide range of results. Results are analyzed in four groups.

First, Figure 5 shows basic CPU operation measurements for integer multiply, integer divide, double precision floating point add, double precision floating point multiply, and double precision bogomflop tests. They provide a comparison that reflects differences in the Android and iOS hardware and compilers used. The basic CPU operation measurements were essentially the same for all three system configurations using the Android device, except for the integer divide test, which showed that the Linux compiler generated more optimized code than the iOS compiler. In all cases, the measurements for the iOS device were worse than the Android device, confirming that the iPad mini's CPU is not as fast as the Nexus 7's CPU for basic math operations.

Second, Figure 5 shows syscall and signal handler measurements including null syscall, `read`, `write`, `open`/`close`, and signal handler tests. The null syscall measurement shows the overhead incurred by Cider on a syscall that does no work, providing a conservative measure of the cost of Cider. The overhead is 8.5% over vanilla Android running the same Linux binary. This is due to extra persona checking and handling code run on every syscall entry. The overhead is 40% when running the iOS binary over vanilla Android running the Linux binary. This demonstrates the additional cost of using the iOS persona, and translating the syscall into the corresponding Linux syscall. These overheads fall into the noise for syscalls that perform some useful function, as shown by the other syscall measurements. Running the iOS binary on the Nexus 7 using Cider is much faster in these syscall measurements than running the same binary on the iPad mini, illustrating a benefit of using Cider to leverage the faster performance characteristics of Android hardware.

The signal handler measurement shows Cider's signal delivery overhead when the signal is generated and delivered within the same process. This is a conservative measurement because no work is done by the process as a result of signal delivery. The overhead is small: 3% over vanilla Android running the same Linux binary. This is due to the added cost of determining the persona of the target thread. The overhead is 25% when running the iOS binary over vanilla Android running the Linux binary. This shows the overhead of the iOS persona which includes translation of the signal information and delivery of a larger signal delivery structure expected by iOS binaries. Running the iOS binary on the iPad mini takes 175% longer than running the same binary on the Nexus 7 using Cider for the signal handler test.

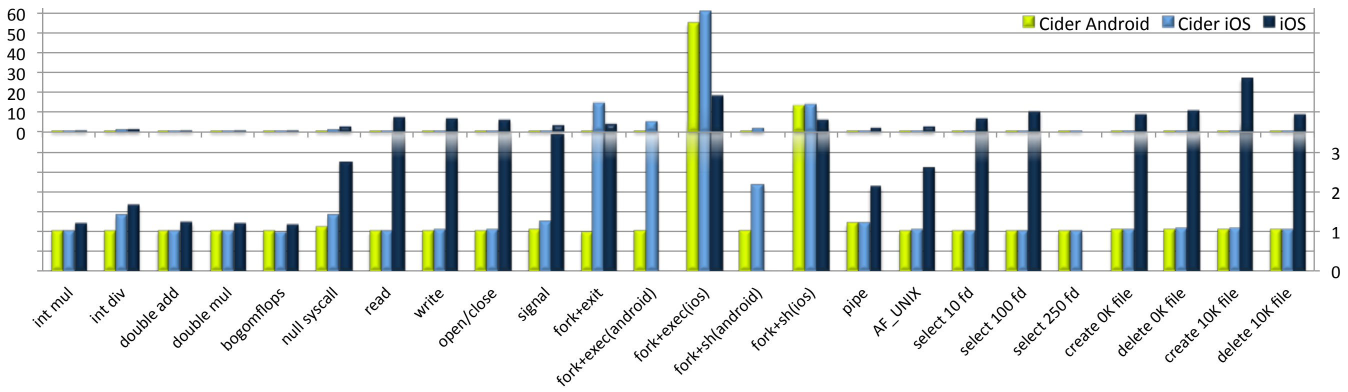


Figure 5: Microbenchmark latency measurements normalized to vanilla Android; lower is better performance.

Third, Figure 5 shows five sets of process creation measurements, `fork+exit`, `fork+exec`, and `fork+sh` tests. The `fork+exit` measurement shows that Cider incurs negligible overhead versus vanilla Android running a Linux binary despite the fact that it must do some extra work in Mach IPC initialization. However, Cider takes almost 14 times longer to run the iOS binary version of the test compared to the Linux binary. The absolute difference in time is roughly 3.5 ms, the Linux binary takes 245 μs while the iOS binary takes 3.75 ms. There are two reasons for this difference. First, the process running the iOS binary consumes significantly more memory than the Linux binary because the iOS dynamic linker, `dyld`, maps 90 MB of extra memory from 115 different libraries, irrespective of whether or not those libraries are used by the binary. The `fork` syscall must then duplicate the page table entries corresponding to all the extra memory, incurring almost 1 ms of extra overhead. Second, an iOS process does a lot more work in user space when it forks because iOS libraries use a large number of `pthread_atfork` callbacks that are called before and after `fork`. Similarly, for each library, `dyld` registers a callback that is called on `exit`, resulting in the execution of 115 handlers on exit. These user space callbacks account for 2.5 ms of extra overhead. Note that the `fork+exit` measurement on the iPad mini is significantly faster than using Cider on the Android device due to a shared library cache optimization that is not yet supported in the Cider prototype. To save time on library loading, iOS's `dyld` stores common libraries prelinked on disk in a shared cache in lieu of storing the libraries separately. iOS treats the shared cache in a special way and optimizes how it is handled.

The `fork+exec` measurement is done in several unique variations on Cider. This test spawns a child process which executes a simple hello world program. We compile two versions of the program: a Linux binary and an iOS binary. The test itself is also compiled as both a Linux binary and an iOS binary. A vanilla Android system can only run a Linux binary that spawns a child to run a Linux binary. Similarly, the iPad mini can only run an iOS binary that spawns a child to run an iOS binary. Using Cider, the test can be run four different ways: a Linux binary can spawn a child to run either a Linux or an iOS binary, and an iOS binary can spawn a child to run either a Linux or an iOS binary. To compare the different `fork+exec` measurements, we normalize performance against the vanilla Android system running a Linux binary that spawns a child running a Linux binary. Figure 5 shows all four `fork+exec` measurements using Cider.

The `fork+exec(android)` test forks a child that execs a Linux binary. The Cider Android bar shows results of a Linux test program while the Cider iOS bar shows results of an iOS test program. Cider incurs negligible overhead in the Cider Android case. The actual test run time is roughly 590 μs, a little more than twice the time it takes to run the `fork+exit` measurement, reflecting the fact that executing the hello world program is more expensive than simply exiting. Cider takes 4.8 times longer to run the test in the Cider iOS case. The extra overhead is due to the cost of an iOS binary calling `fork`, as discussed previously in the `fork+exit` measurement. Interestingly, the `fork+exec(android)` Cider iOS measurement is 3.42 ms, less than the `fork+exit` measurement because the child process replaces its iOS binary with the hello world Linux binary. This is less expensive than having the original iOS binary exit because of all the exit handlers that must execute.

The `fork+exec(ios)` test forks a child that execs an iOS binary. The Cider Android bar shows results of a Linux test program. The Cider iOS and iOS bars show results of an iOS test program. This test is not possible on vanilla Android, thus the comparison is intentionally unfair and skews the results against this test. Nevertheless, using this comparison, Figure 5 shows that spawning a child to run an iOS binary is much more expensive. This is because the Cider prototype uses non-prelinked libraries, and `dyld` must walk the filesystem to load each library on every `exec`. The extra overhead of starting with an iOS binary versus a Linux binary is due to the cost of the iOS binary calling `fork`, as discussed previously in the `fork+exit` measurement. Running the `fork+exec` test on the iPad mini is faster than using Cider on the Android device because of its shared cache optimization which avoids the need to walk the filesystem to load each library.

Similar to the `fork+exec` measurement, the `fork+sh` measurement is done in four variations on Cider. The `fork+sh(android)` test launches a shell that runs a Linux binary. Cider incurs negligible overhead versus vanilla Android when the test program is a Linux binary, but takes 110% longer when the test program is an iOS binary. The extra overhead is due to the cost of an iOS binary calling `fork`, as discussed previously in the `fork+exec` measurement. Because the `fork+sh(android)` measurement takes longer, 6.8 ms using the iOS binary, the relative overhead is less than the `fork+exec(android)` measurement.

The `fork+sh(ios)` test launches a shell that runs an iOS binary. This is not possible on vanilla Android, so we normalize to the `fork+sh(android)` test, skewing the results against this test. Using this comparison, Figure 5 shows that spawning a child to run an iOS binary is much more expensive for the same reasons as the `fork+exec` measurements. Because the `fork+sh(ios)` test takes longer, the relative overhead is less than the `fork+exec(ios)` measurement.

Finally, Figure 5 shows local communication and filesystem measurements including `pipe`, `AF_UNIX`, `select` on 10 to 250 file descriptors, and creating and deleting 0 KB and 10 KB files. Measurements were quite similar for all three system configurations using the Android device. However, measurements on the iPad mini were significantly worse than the Android device in a number of cases. Perhaps the worst offender was the `select` test whose overhead increased linearly with the number of file descriptors to more than 10 times the cost of running the test on vanilla Android. The test simply failed to complete for 250 file descriptors. In contrast, the same iOS binary runs using Cider on Android with performance comparable to running a Linux binary on vanilla Android across all measurement variations.

6.3 Application Measurements

Figure 6 shows the results of the iOS and Android PassMark benchmark apps [37, 38] on the four different system configurations. Vanilla Android performance is normalized to one in all cases, and not explicitly shown. Measurements are in operations per second, so larger numbers are better. In all tests, Cider adds negligible overhead to the Android PassMark app. Test results are analyzed in five groups.

First, Figure 6 shows CPU operation measurements for integer, floating point, find primes, random string sort, data encryption, and data compression tests. Unlike the basic `lmbench` CPU measurements, the PassMark measurements show that Cider delivers significantly faster performance when running the iOS PassMark app on Android. This is because the Android version is written in Java and interpreted through the Dalvik VM while the iOS version is written in Objective-C and compiled and run as a native binary. Because the Android device contains a faster CPU than the iPad mini, Cider outperforms iOS when running the CPU tests from the same iOS PassMark application binary.

Second, Figure 6 shows storage operation measurements for write and read tests. Cider has similar storage read performance to the iPad mini when running the iOS app. However, the iPad mini has much better storage write performance than either the iOS or Android app running on Cider. Because storage performance can depend heavily on the OS, these results may reflect differences in both the underlying hardware and the OS.

Third, Figure 6 shows memory operation measurements for write and read tests. Cider delivers significantly faster performance when running the iOS PassMark app on Android. This is, again, because Cider runs the iOS app natively while Android interprets the app through the Dalvik VM. Cider outperforms the iPad mini running the memory tests from the same iOS PassMark app binary, again reflecting the benefit of using faster Android hardware.

Fourth, Figure 6 shows graphics measurements for a variety of 2D graphics operations including solid vectors, transparent vectors, complex vectors, image rendering, and image filters. With the exception of complex vectors, the Android app performs much better than the iOS binary on both Cider and the iPad mini. This is most likely due to more efficient/optimized 2D drawing libraries in Android. Additionally, since these tests are CPU bound, Cider generally outperforms iOS due to the Nexus 7's faster CPU. However, bugs in the Cider OpenGL ES library related to "fence" synchronization primitives caused under-performance in the image rendering tests.

Finally, Figure 6 shows graphics measurements for simple and complex 3D tests, the latter shown in Figure 4d. Because the iPad mini has a faster GPU than the Nexus 7, it has better 3D graphics performance. The iOS binary running on Cider performs 20-37% worse than the Android PassMark app due to the extra cost of diplomatic function calls. Each function call into the OpenGL ES library is mediated into the Android OpenGL ES library through diplomats. As the complexity of a given frame increases, the number of OpenGL ES calls increases, which correspondingly increases the overhead. This can potentially be optimized by aggregating OpenGL ES calls into a single diplomat, or by reducing the overhead of a diplomatic function call. Both optimizations are left to future work.

6.4 Limitations

We have not encountered any fundamental limitations regarding the feasibility of the Cider approach, such as any unbridgeable compatibility issues between iOS and Android. However, while we have implemented an initial Cider prototype that successfully runs many iOS apps on Android devices, the implementation is incomplete. In particular, smartphones and tablets incorporate a plethora of devices that apps expect to be able to use, such as GPS, cameras, cell phone radio, Bluetooth, and others. Support for the full range of devices available on smartphones and tablets is beyond the scope of this paper. Cider will not currently run iOS apps that depend on such devices. For example, an app such as Facetime that requires use of the camera does not currently work with Cider. If the iOS app has a fall-back code path, it can still partially function. For example, the iOS Yelp [46]

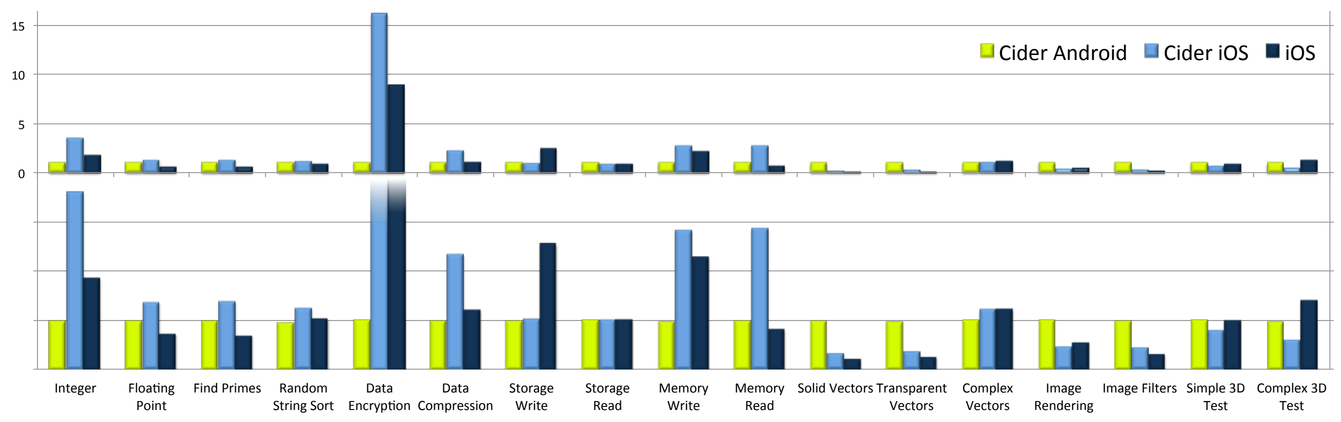


Figure 6: App throughput measurements normalized to vanilla Android; higher is better performance.

app runs on Cider even though GPS and location services are currently unsupported. Yelp simply assumes the user's current location is unavailable, and continues to function as it would on an Apple device with location services disabled.

Implementation of Cider device support varies with device and interface complexity. Devices with a simple interface, such as GPS, can be supported with I/O Kit drivers, discussed in Section 5.1, and diplomatic functions. Devices that use more standardized interfaces can be supported either through duct tape given an open source implementation, or diplomatic functions given a well-defined API. More complicated devices such as the camera and cell radio were not investigated as part of this research, but we believe that techniques similar to those used in Section 5.3 could be used to support these devices. For example, iOS exposes a camera API to apps. By replacing these API entry points with diplomatic functions that interact with native Android hardware, it may be possible to provide camera support for iOS apps.

The current prototype also has some other implementation limitations. Incorrect "fence" synchronization primitive support in the Cider OpenGL ES library, discussed in Section 6.3, degraded our graphics performance, and the iOS WebKit framework is only partially supported due to its multi-threaded use of the OpenGL ES API. We expect these limitations to be removed with additional engineering effort.

Finally, Cider does not map iOS security to Android security. Android's permission-based security model differs significantly from the more dynamic iOS security, which is enforced at runtime. A complete mapping of the two models is left to future work.

7. Related Work

Many approaches have tried to run apps from multiple OSes on the same hardware, though primarily in the context of desktop computers, not mobile devices. Several BSD variants maintain binary compatibility layers for other OSes [18, 19, 23] at the kernel level. BSD reimplements foreign syscalls in the OS, using a different syscall dispatch table for each OS to glue the calls into the BSD kernel. It works for foreign OSes that are close enough to BSD such as Linux, but attempts to extend this approach to Mac OS X apps only provide limited support for command-line tools, not GUI apps [20]. Similarly, Solaris 10 Zones [33, 34] provided an lx brand mechanism that emulates outdated Linux 2.4 kernel system call interfaces to run some Linux binaries on Solaris, though this feature is no longer available as of Solaris 11 [35]. Cider goes beyond these approaches by introducing duct tape to make it easier to add foreign kernel code to an OS and diplomatic functions to support the use of opaque foreign kernel interfaces, providing a richer OS compatibility layer that supports GUI apps with accelerated graphical interfaces.

Operating at user instead of kernel level, Wine [1] runs Windows apps on x86 computers running Linux. It achieves this by attempting to reimplement the entire foreign user space library API, such as Win32, using native APIs. This is tedious and overwhelmingly complex. Wine has been under development for over 20 years, but continues to chase Windows as every new release contains new APIs that need to be implemented. Darling [17] takes a similar approach to try to run Mac OS X apps on Linux, though it remains a work in progress unable to run anything other than simple command-line apps. In contrast, Cider provides kernel-level persona management that leverages existing unmodified libraries and frameworks to avoid rewriting huge amounts of user space code. Cider's duct tape layer and diplomatic function calls facilitate this by incorporating existing foreign kernel code without tedious reimplementation, and allowing foreign apps to directly leverage existing domestic libraries.

Wabi [26] from Sun Microsystems ran Windows apps on Solaris. It supported apps developed for Windows 3.1, but did not support later versions of Windows and was discontinued. Unlike Wine, it required Windows 3.1 and leveraged existing Windows libraries except for the lowest layers of the Windows environment, for which it replaced low-level Windows API libraries with versions that translated from Windows to Solaris calls. It also provided CPU emulation to allow x86 Windows apps to run on Sparc, similar to binary translation systems such as DEC's FX!32 [13]. Wabi ran on top of Solaris and provided all of its functionality outside of the OS, limiting its ability to support apps that require kernel-level services not available in Solaris. In contrast, Cider provides binary personality support in the OS to support foreign kernel services, uses diplomatic functions to make custom domestic hardware accessible to foreign apps, and does not need to perform any binary translation since both iOS and Android run on ARM CPUs.

While most previous approaches have not considered mobile software ecosystems, AppPlayer from BlueStacks [10] allows users to run Android apps on a Windows PC or Apple OS X computer by utilizing a cross-compiled Dalvik VM and ported Android services such as `SurfaceFlinger`. This is possible because Android is open source, and the whole system can be easily cross-compiled. Since many Android apps are entirely Java-based and consist of bytecodes run in a VM, it is relatively easy to run them anywhere. However, many popular Android apps increasingly incorporate native libraries for performance reasons – these apps will not work. Similarly, this approach will not work for iOS apps which are native binaries running on a proprietary system for which source code is not available. In contrast, Cider utilizes kernel-level persona management and diplomatic function calls to support unmodified iOS binaries which link against unmodified iOS libraries and communicate with unmodified iOS support services such as `notifyd` and `syslogd`.

Some developer frameworks [14, 39, 45] allow mobile app developers to target multiple OSes from a single code base. This requires apps to be written using these frameworks. Because such frameworks are often more limited than those provided by the respective mobile software ecosystems, the vast majority of apps are not written in this manner, and are thus tied to a particular platform. Cider does not require developers to rewrite or recompile their apps to use specific non-standard frameworks, but instead runs unmodified iOS binaries on an Android device.

Other partial solutions to OS compatibility have been explored for desktop systems. Shinichiro Hamaji's Mach-O loader for Linux [24] can load and run some desktop OS X command-line binaries in Linux. This project supports binaries using the "misc" binary format, and dynamically overwrites C entry points to syscalls. NDISWrapper [32] allows the Linux kernel to load and use Windows NDIS driver DLLs. The project's kernel driver implements the Network Driver Interface Specification (NDIS) [41], and dynamically links the Windows DLL to this implementation. It is narrowly focused on a single driver specification, does not support incorporation of general foreign kernel subsystems, and does not support app code. In contrast, Cider makes it possible to incorporate general kernel subsystems, such as Mach IPC, without substantial implementation effort and provides a complete environment for foreign binaries including graphics libraries and device access.

VMs are commonly used to run apps requiring different OS instances on desktop computers. Various approaches [8, 15, 28] have attempted to bring VMs to mobile devices, but they cannot run unmodified OSes, incur higher overhead than their desktop counterparts, and provide at best poor, if any, graphics performance within VMs. Some of these limitations are being addressed by ongoing work on KVM/ARM using ARM hardware virtualization support [16]. Lightweight OS virtualization [2, 12] claims lower performance overhead, but does not support different OS instances and therefore cannot run foreign apps at all. Unlike Cider, none of these previous approaches can run iOS and Android apps on the same device.

Drawbridge [40] and Bascule [9] provide user mode OS personalities for desktop apps by refactoring traditional OSes into library OSes that call down into a host OS. Refactoring is a complex and tedious process, and there is no evidence that these systems would work for mobile devices. For example, Bascule has no support for general inter-process sharing, and relies on an external X11 server using network-based graphics, running on the host OS, to support GUI apps. It is unclear how these systems might support vertically integrated libraries that require direct communication to hardware, or multi-process services based on inter-process communication. In contrast, Cider provides duct tape to easily incorporate kernel subsystems that facilitate the multi-process communication required by iOS apps, and leverages diplomatic function calls to support direct communication with closed or proprietary hardware – a feature crucial for vertically integrated mobile devices.

8. Conclusions

Cider is the first system that can run unmodified iOS apps on non-Apple devices. It accomplishes this through a novel combination of binary compatibility techniques including two new operating system compatibility mechanisms: duct tape and diplomatic functions. Duct tape allows source code from a foreign kernel to be compiled, without modification, into the domestic kernel. This avoids the difficult, tedious, and error prone process of porting or implementing new foreign subsystems. Diplomatic functions leverage per-thread personas and mediate foreign function calls into domestic libraries. This enables Cider to support foreign libraries that are closely tied to foreign hardware by replacing library function calls with diplomats that utilize domestic libraries and hardware. We built a Cider prototype that reuses existing unmodified frameworks across both iOS and Android ecosystems. Our results demonstrate that Cider has modest performance overhead and runs popular iOS and Android apps together seamlessly on the same Android device.

9. Acknowledgments

Yan Zou helped with running benchmarks to obtain many of the measurements in this paper. Lin Zhong provided helpful comments on earlier drafts of this paper. This work was supported in part by a Facebook Graduate Fellowship and NSF grants CNS-1162447, CNS-1018355, and CNS-0905246.

References

[1] B. Amstadt and M. K. Johnson. Wine. *Linux Journal*, 1994 (4es), Aug. 1994. ISSN 1075-3583.

[2] J. Andrus, C. Dall, A. Van't Hof, O. Laadan, and J. Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the* 23^{rd} *ACM Symposium on Operating Systems Principles*, pages 173–187, Cascais, Portugal, Oct. 2011.

[3] Apalon Apps. Calculator Pro for iPad Free on the App Store on iTunes. `https://itunes.apple.com/us/app/calculator-pro-for-ipad-free/id749118884`, Dec. 2013. Accessed: 12/20/2013.

[4] Apple, Inc. OS X ABI Mach-O File Format Reference. `https://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html`, Feb. 2009. Accessed: 3/20/2013.

[5] Apple, Inc. Porting UNIX/Linux Applications to OS X. `https://developer.apple.com/library/mac/#documentation/Porting/Conceptual/PortingUnix/background/background.html`, June 2012. Accessed: 3/27/2013.

[6] Apple, Inc. Source Browser. `http://www.opensource.apple.com/source/xnu/xnu-2050.18.24/`, Aug. 2012. Accessed: 3/21/2013.

[7] Apple, Inc. OpenGL ES Programming Guide for iOS: Configuring OpenGL ES Contexts. `https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/WorkingwithOpenGLESContexts/WorkingwithOpenGLESContexts.html`, Sept. 2013. Accessed: 12/4/2013.

[8] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *ACM SIGOPS Operating Systems Review*, 44(4):124–135, Dec. 2010. ISSN 0163-5980.

[9] A. Baumann, D. Lee, P. Fonesca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 239–252, Prague, Czech Republic, Apr. 2013.

[10] BlueStacks. Run Mobile Apps on Window PC or Mac With BlueStacks — Android App Player. `http://www.bluestacks.com/`. Accessed: 7/23/2013.

[11] M. B.V. Papers on the App Store on iTunes. `https://itunes.apple.com/us/app/papers/id304655618`, Oct. 2013. Accessed: 12/10/2013.

[12] Cellrox. Cellrox ThinVisor Technology. `http://www.cellrox.com/how-it-works/`, Feb. 2013. Accessed: 4/5/2013.

[13] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX!32: A Profile-Directed Binary Translator. *IEEE Micro*, 18(2):56–64, Mar. 1998. ISSN 0272-1732. .

[14] D. Connelly, T. Ball, and K. Stanger. j2objc - A Java to iOS Objective-C translation tool and runtime. - Google Project Hosting. `https://code.google.com/p/j2objc/`. Accessed: 7/23/2013.

[15] C. Dall and J. Nieh. KVM for ARM. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, June 2010.

[16] C. Dall and J. Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, Mar. 2014.

[17] L. Doleel. The Darling Project. `http://darling.dolezel.info/en/Darling`, Aug. 2012. Accessed: 4/5/2013.

[18] E. Dreyfus. Linux Compatibility on BSD for the PPC Platform. `http://onlamp.com/lpt/a/833`, May 2001. Accessed: 5/11/2012.

[19] E. Dreyfus. IRIX Binary Compatibility, Parts 1–6. `http://onlamp.com/lpt/a/2623`, Aug. 2002. Accessed: 5/11/2012.

[20] E. Dreyfus. Mac OS X binary compatibility on NetBSD: challenges and implementation. In *Proceedings of the 2004 EuroBSDCon*, Karlsruhe, Germany, Oct. 2004.

[21] Faraday, Owen. Android is a desolate wasteland when it comes to games (Wired UK). `http://www.wired.co.uk/news/archive/2012-10/31/android-games`, Oct. 2012. Accessed: 3/21/2013.

[22] Free Software Foundation. GDB: The GNU Project Debugger. `https://www.gnu.org/software/gdb/`, Dec. 2013. Accessed: 12/10/2013.

[23] FreeBSD Documentation Project. Linux Binary Compatibility. In B. N. Handy, R. Murphey, and J. Mock, editors, *The FreeBSD Handbook 3rd Edition, Vol.1: User Guide*, chapter 11. Mar. 2004.

[24] S. Hamaji. Mach-O Loader for Linux. `https://github.com/shinh/maloader`, Mar. 2011. Accessed: 3/15/2013.

[25] M. Heily. libkqueue. `http://www.heily.com/~mheily/proj/libkqueue/`, Mar. 2011. Accessed: 1/3/2014.

[26] P. Hohensee, M. Myszewski, and D. Reese. Wabi CPU Emulation. In *Hot Chips 8*, Palo Alto, CA, Aug. 1996.

[27] G. C. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.

[28] J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park, and C. Kim. Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones. In *Proceedings of the* 5^{th} *Consumer Communications and Network Conference*, pages 257–261, Las Vegas, NV, Jan. 2008.

[29] Khronos Group. OpenGL ES Common Profile Specification Version 2.0.25 (Full Specification). `http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf`, Nov. 2010. Accessed: 4/8/2013.

[30] Khronos Group. Khronos Native Platform Graphics Interface (EGL Version 1.4). `http://www.khronos.org/registry/egl/specs/eglspec.1.4.20130211.pdf`, Feb. 2013. Accessed: 12/4/2013.

[31] Khronos Group. OpenGL ES – The Standard for Embedded Accelerated 3D Graphics. `http://www.khronos.org/opengles/`, Jan. 2013. Accessed: 3/22/2013.

[32] J. Kiszka, G. Pemmasani, and P. Fuchs. SourceForge.net: ndiswrapper. `http://ndiswrapper.sourceforge.net/`. Accessed: 7/23/2013.

[33] N. Nieuwejaar, E. Schrock, W. Kucharski, R. Blaine, E. Pilatowicz, and A. Leventhal. Method for Defining Non-Native Operating Environments. US 7689566, Filed Dec. 12, 2006, Issued Mar. 30, 2010. `http://www.patentlens.net/patentlens/patent/US_7689566/`.

[34] Oracle Corporation. System Administration Guide: Oracle Solaris Containers-Resource Management and Oracle Solaris Zones. `http://docs.oracle.com/cd/E19253-01/817-1592/817-1592.pdf`, Sept. 2010. Accessed: 1/3/2014.

[35] Oracle Corporation. Transitioning From Oracle Solaris 10 to Oracle Solaris 11. `http://docs.oracle.com/cd/E23824_01/pdf/E24456.pdf`, Mar. 2012. Accessed: 1/3/2014.

[36] Parallels IP Holdings GmbH. Parallels Desktop. `http://www.parallels.com/products/desktop/`. Accessed: 3/22/2013.

[37] PassMark Software, Inc. PerformanceTest Mobile on the App Store on iTunes. `https://itunes.apple.com/us/app/performancetest-mobile/id494438360`, June 2012. Accessed: 12/10/2013.

[38] PassMark Software, Inc. PassMark PerformanceTest – Android Apps on Google Play. `https://play.google.com/store/apps/details?id=com.passmark.pt_mobile`, Jan. 2013. Accessed: 3/14/2013.

[39] PhoneGap. PhoneGap — Home. `http://phonegap.com/`. Accessed: 7/23/2013.

[40] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–304, Newport Beach, CA, Mar. 2011.

[41] I. Printing Communications Assoc. NDIS Developer's Reference. `http://www.ndis.com/`. Accessed: 7/24/2013.

[42] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *ACM SIGOPS Operating Systems Review*, 21(4):31–39, Oct. 1987. ISSN 0163-5980.

[43] C. K. Tung. CK's IT blog: How To Decrypt iPhone IPA file. `http://tungchingkai.blogspot.com/2009/02/how-to-decrypt-iphone-ipa-file.html`, Feb. 2009. Accessed: 3/14/2013.

[44] VMware, Inc. VMware Workstation. `http://www.vmware.com/products/workstation/`. Accessed: 3/22/2013.

[45] Yeecco, Ltd. www.yeeccoo.com. `http://http://www.yeecco.com/stella`. Accessed: 6/27/2013.

[46] Yelp. Yelp on the App Store on iTunes. `https://itunes.apple.com/us/app/yelp/id284910350`, Dec. 2013. Accessed: 1/8/2014.

[47] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. J. Chew, W. J. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, Austin, TX, Nov. 1987. ACM.