Multi-Persona Mobile Computing

Jeremy Andrus

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2015

ABSTRACT

Multi-Persona Mobile Computing

Jeremy Andrus

Smartphones and tablets are increasingly ubiquitous, and many users rely on multiple mobile devices to accommodate work, personal, and geographic mobility needs. Pervasive access to always-on mobile computing has created new security and privacy concerns for mobile devices that often force users to carry multiple devices to meet those needs. The volume and popularity of mobile devices has commingled hardware and software design, and created tightly vertically integrated platforms that lock users into a single, vendor-controlled ecosystem. My thesis is that lightweight mechanisms can be added to commodity operating systems to enable multiple virtual phones or tablets to run at the same time on a physical smartphone or tablet device, and to enable apps from multiple mobile platforms, such as iOS and Android, to run together on the same physical device, all while maintaining the low-latency and responsiveness expected of modern mobile devices. This dissertation presents two lightweight operating systems mechanisms, virtualization and binary compatibility, that enable *multi-persona* mobile computing. First, we present *Cells*, a mobile virtualization architecture enabling multiple virtual phones, or *personas*, to run simultaneously on the same physical cellphone in a secure and isolated manner. *Cells* introduces device namespaces that allow apps to run in a virtualized environment while still leveraging native devices such as GPUs to provide accelerated graphics. Second, we present *Cycada*, an operating system compatibility architecture that runs applications built for different mobile ecosystems, iOS and Android, together on a single Android device. *Cycada* introduces kernel-level code adaptation and diplomats to simplify binary compatibility support by reusing existing operating system code and unmodified frameworks and libraries. Both *Cells* and *Cycada* have been implemented in Android, and can run multiple Android virtual phones, and a mix of iOS and Android apps on the same device with good performance. Because mobile computing has become increasingly important, we also present a new way to teach operating systems in a mobile-centric way that incorporates the concepts of geographic mobility, sensor data acquisition, and resource-constrained design considerations.

Table of Contents

List of Figures

List of Tables

Acknowledgments

My research, and this document, would not have been possible without the help and support of colleagues, friends, and family. Working in collaboration with my advisor, Jason Nieh, and the brilliant students he advises has been a pleasure. Christoffer Dall and Nicolas Viennot have been incredible research collaborators without whom neither the *Cells* project nor the *Cycada* project would have happened. Whenever I would plow myself too quickly or too deep into a piece of code, Christoffer would dogmatically bring me back to principled analysis and a complete understanding. The speed at which Nicolas could read and analyze a complex piece of code continues to awe and inspire me. Key contributions from Alex Van't Hof in both the *Cells*, and *Cycada* work came at opportune times, and his ability to quietly solve problems and deliver results is truly impressive. It was also my pleasure to directly work with Naser AlDuaij who jumped into my binary compatibility project head first and was able to swim his way out with me.

In addition to my colleagues in the Software System Lab, I've had the distinct pleasure of working with talented masters and undergraduate students here at Columbia both as a TA and as co-conspirators in conference submissions. The laborious and careful measurements taken by Qi Ding, and Charles Hastings formed the backbone of my *Cells* evaluation, and Yan Zou flawlessly ran countless micro benchmarks on a complex binary compatibility platform. Without this support my projects would not have been the same.

My wife, Amanda, has endured countless late nights and stressful paper submissions while bearing more than her share of parental duties. Without her support I would not be where I am today, and I most definitely would not have completed this document.

Finally, my research was supported in part by NSF grants CNS-1162447, CNS-1018355, CNS-0914845, CNS-0905246, AFOSR MURI grant FA9550-07-1-0527, a Google Research Award, and a Facebook Graduate Fellowship.

Dad, if I could pencil your name onto my diploma, I would.

Amanda, you pushed me here;
I can't wait to see where we go next.

Chapter 1

Introduction

Smartphones and tablets are increasingly ubiquitous, and many users rely on multiple mobile devices to accommodate work, personal, and geographic mobility needs. Pervasive access to always-on mobile computing has give rise to two inter-related computing phenomena. First, the preferred platform for a user's everyday computing needs is shifting away from traditional desktop and laptop computers towards mobile smartphone and tablet devices [131]. Second, mobile devices are changing the way that computing platforms are designed. The separation of hardware and software concerns in the traditional PC world is shifting to vertically integrated software and hardware platforms that lock users into a particular mobile ecosystem.

Both business and personal users are using their smartphone and tablet devices more than traditional desktop devices. Business professionals rely on mobile devices for pervasive access to email, Web browsing, contact management, and location specific information. These same functions as well as the ability to play games, listen to music, watch movies, and read e-books also make modern mobile devices an ideal personal computing platform. Hundreds of thousands of applications are available for both business and personal users to download and try through various online application stores. The ease of downloading new software imposes a risk on users as malicious software can easily access sensitive data with the risk of corrupting it or even leaking it to third parties [142]. For this reason, companies often lock down the smartphones they allow to connect to the company network, and at least require the ability to wipe clean such smartphones if they are lost, hacked, or their respective owner leaves the company. The result is that many users have to carry separate work and personal phones. Application developers also carry additional phones for development to avoid having a misbehaving application prototype corrupt their primary phone. Parents sometimes wish

they had additional phones when their children use the parent's smartphone for entertainment and end up with unexpected charges due to accidental phone calls or unintended in-app purchases.

In addition to shifting users' preferred computing platform, mobile devices are changing the way that computing platforms are designed. The separation of hardware and software concerns in the traditional PC world is shifting 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 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 [52]. 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.

Both of these computing phenomena have been historically addressed on desktop and server computers using binary compatibility and virtual machines. Previous binary compatibility work has focused on desktop systems. While some user-level solutions have been successful [115], most have incurred high overhead, and none have been complete or efficient enough for resource constrained mobile devices. Virtual machines (VMs) are useful for desktop and server computers to isolate and separate software stacks, and also run applications intended for one platform on a different platform [126; 104]. VM mechanisms have even been proposed that enable two separate and isolated instances of a smartphone software stack to run on the same ARM hardware [97; 28; 43; 69]. These approaches require substantial modifications to both user and kernel levels of the software stack. Paravirtualization is used in most cases since the most prevalent ARM architectures are not virtualizable and new ARM virtualization extensions are not yet widely available

in hardware. While VMs are useful for desktop and server computers, applying these same virtualization techniques to smartphones has two crucial drawbacks. First, smartphones are more resource constrained, and running an entire additional operating system (OS) and user space environment in a VM imposes high overhead and limits the number of instances that can run. High overhead and slow system responsiveness are much less acceptable on mobile devices than on a desktop or laptop computer because smartphones and tablets are often used for minutes or even seconds at a time. Second, mobile devices are tightly integrated hardware platforms that incorporate a plethora of devices using non-standard interfaces. Applications expect to be able to directly use devices such as GPS, cameras, and GPUs. Existing virtualization approaches provide no effective mechanism enabling applications to directly leverage these hardware device features from within VMs This severely limits performance, and makes existing VM-based approaches unusable on mobile devices.

My thesis is that lightweight mechanisms can be added to commodity operating systems to enable multiple virtual phones or tablets to run at the same time on a physical smartphone or tablet device, and to enable apps from multiple mobile platforms, such as iOS and Android, to run together on the same physical device, all while maintaining the low-latency and responsiveness expected of modern mobile devices. This dissertation presents two such lightweight operating system mechanism, virtualization and binary compatibility, that enable *multi-persona* mobile computing. Chapter 2 presents *Cells* [4; 42], a mobile virtualization architecture enabling multiple virtual phones, or *personas*, to run simultaneously on the same physical cellphone. In chapter 3 we refine the concept of a persona from a full virtual phone to the execution mode of a single thread, and present *Cycada* [6], an operating system compatibility architecture that can run applications built for different mobile ecosystems, namely iOS and Android, together on the same device. Chapter 4 presents a detailed application of the *Cycada* architecture in the context of advanced graphics and GPU device support in a binary compatibility system. Chapter 5 presents some experiences and lessons learned while building the *Cycada* OS compatibility solution. Finally, because mobile computing has become increasingly important, we present a new way of teaching operating systems in a mobile-centric way [5]. Chapter 6 presents the details of our approach that incorporates the concepts of geographic mobility, sensor data acquisition, and resource-constrained design considerations.

Chapter 2

Cells

As the preferred platform for a user's everyday computing needs has shifted from desktop and laptop computers to mobile devices, the risk of malicious applications gaining access to sensitive data has increased. Corporations lock down smartphones given to employees, forcing them to carry separate work and personal phones. Similarly, application developers often use multiple devices to mitigate the risk of malicious or buggy applications corrupting their personal phone, and parents wish for a separate device where their child cannot incur unexpected in-app purchases or inadvertently expose personal data to a malicious person or application.

Cells is a lightweight virtualization architecture for enabling multiple *personas*, or virtual phones (VPs), to run simultaneously on the same smartphone hardware with high performance. *Cells* does not require running multiple OS instances. It uses lightweight OS virtualization to provide virtual namespaces that can run multiple VPs on a single OS instance. *Cells* isolates VPs from one another, and ensures that buggy or malicious applications running in one VP cannot adversely impact other VPs. *Cells* provides a novel file system layout based on unioning to maximize sharing of common read-only code and data across VPs, minimize memory consumption and enable additional VPs to be instantiated with very little overhead.

Cells takes advantage of the small display form factors of smartphones, which generally display only a single application at a time, and introduces a usage model having one foreground VP that is displayed and one or more background VPs that are not displayed at any given time. This simple yet powerful model enables *Cells* to provide novel kernel-level and user-level device namespace mechanisms to efficiently multiplex hardware devices across multiple VPs, including proprietary or opaque hardware such as the baseband

processor, while maintaining native hardware performance. The foreground VP is always given direct access to hardware devices. Background VPs are given shared access to hardware devices when the foreground VP does not require exclusive access. Visible applications are always running in the foreground VP and those applications can take full advantage of any available hardware feature, such as hardware-accelerated graphics. Since foreground applications have direct access to hardware, they perform as fast as when they are running natively.

Cells provides individual telephone numbers for each VP without the need for multiple SIM cards through using a VoIP service. Incoming and outgoing calls use the cellular network, not VoIP, and are routed through the VoIP service as needed to provide both incoming and outgoing caller ID functionality for each VP. *Cells* uses this combination of a VoIP service and the cellular network to allow users to make and receive calls using their standard cell phone service while maintaining per-VP phone numbers and caller ID features. *Cells* leverages all of the standard call multiplexing available in standard cellular technologies for handling multiple calls for a single phone to handle multiple calls across virtual phones. For example, if a user switches the foreground VP into the background during a phone call, *Cells* can place the active call on hold and allow the user to make another outgoing call from the new foreground VP. Wi-Fi connections and the cellular network used for data connectivity are fully supported and network connections are completely isolated between VPs.

We have implemented a preliminary *Cells* prototype that supports multiple virtual Android phones on the same mobile device. Each VP can be configured the same or completely different from other VPs. The prototype has been tested to work with multiple versions of Android, including the most recent open-source version, version 4.3. It works seamlessly across multiple hardware devices, including Google Nexus 1 and Nexus S phones, and an NVIDIA developer tablet. Our experimental results, demonstrate that *Cells* imposes almost no runtime overhead and only modest memory overhead. *Cells* scales to support far more phone instances on the same hardware than VM-based approaches. *Cells* is the first virtualization system that fully supports available hardware devices with native performance including GPUs, sensors, cameras, and touchscreens, and transparently runs all applications in VPs without any modifications.

In this chapter, we present the design and implementation of *Cells*. Section 2.1 describes the *Cells* usage model. Section 2.2 provides an overview of the system architecture. Sections 2.3 and 2.4 describe graphics and power management virtualization, respectively, using kernel device namespaces. Sections 2.5 and 2.6 describe telephony and wireless network virtualization, respectively, using user-level device namespaces.

Section 2.7 presents experimental results. Section 2.8 discusses related work. Finally, we present some concluding remarks.

2.1 Usage Model

Cells runs multiple VPs on a single hardware device. Each VP runs a standard Android environment capable of making phone calls, running unmodified Android applications, using data connections, interacting through the touch screen, utilizing the accelerometer, and everything else that a user can normally do on the hardware. Each VP is completely isolated from other VPs and cannot inspect, tamper with, or otherwise access any other VP.

Given the limited size of smartphone screens and the ways in which smartphones are used, *Cells* only allows a single VP, the foreground VP, to be displayed at any time. We refer to all other VPs that are running but not displayed as, background VPs. Background VPs are still running on the system in the background and are capable of receiving system events and performing tasks, but do not render content on the screen. A user can easily switch among VPs by selecting one of the background VPs to become the foreground one. This can be done, for example, using a custom key-combination to cycle through the set of running VPs, or using a swipe gesture on the home screen of a VP. Each VP also has an application that can be launched to see a list of available VPs, and to switch any of these to the foreground. The system can force a new VP to become the foreground VP as a result of an event, such as an incoming call or text message. For security and convenience reasons, a no-auto-switch parameter can be set to prevent background VPs from being switched to the foreground without explicit user action, preventing background VPs from stealing input focus or device data. An auto-lock parameter can also be enabled forcing a user to unlock a VP using a passcode or gesture when it transitions from background to foreground. Section 2.2 discusses how the foreground-background usage model is fundamental to the *Cells* virtualization architecture.

VPs are created and configured on a PC and downloaded to a mobile device via USB. A VP can be deleted by the user, but its configuration is password protected and can only be changed from a PC given the appropriate credentials. For example, a user can create a VP and can decide to later change various options regarding how the VP is run and what devices it can access. On the other hand, IT administrators can also create VPs that users can download or remove from their devices, but cannot be reconfigured by users. This is useful for companies that may want to distribute locked down VPs.

Each VP can be configured to have different access rights for different devices. For each device, a VP can be configured to have no access, shared access, or exclusive access. Some settings may not be available on certain devices; shared access is, for example, not available for the framebuffer since only a single VP is displayed at any time. These per device access settings provide a highly flexible security model that can be used to accommodate a wide range of security policies.

No access means that applications running in the VP cannot access the given device at any time. For example, VPs with no access to the GPS sensor would never be able to track location despite any user acceptances of application requests to allow location tracking. Users often acquiesce to such privacy invasions because an application will not work without such consent even if the application has no need for such information. By using the no access option, *Cells* enables IT administrators to create VPs that allow users to install and run such applications without compromising privacy.

Shared access means that when a given VP is running in the foreground, other background VPs can access the device at the same time. For example, a foreground VP with shared access to the audio device would allow a background VP with shared access to play music.

Exclusive access means that when a given VP is running in the foreground, other background VPs are not allowed to access the device. For example, a foreground VP with exclusive access to the microphone would not allow background VPs to access the microphone, preventing applications running in background VPs from eavesdropping on conversations or leaking information. This kind of functionality is essential for supporting secure VPs. Exclusive access may be used in conjunction with the no-auto-switch to ensure that events cannot cause a background VP to move to the foreground and gain access to devices as a means to circumvent the exclusive access rights of another VP.

In addition to device access rights, *Cells* leverages existing OS virtualization technology to prevent privilege escalation attacks in one VP from compromising the entire device. Both user credentials and process IDs are isolated between VPs; the root user in one VP has no relation to the root user in any other VP.

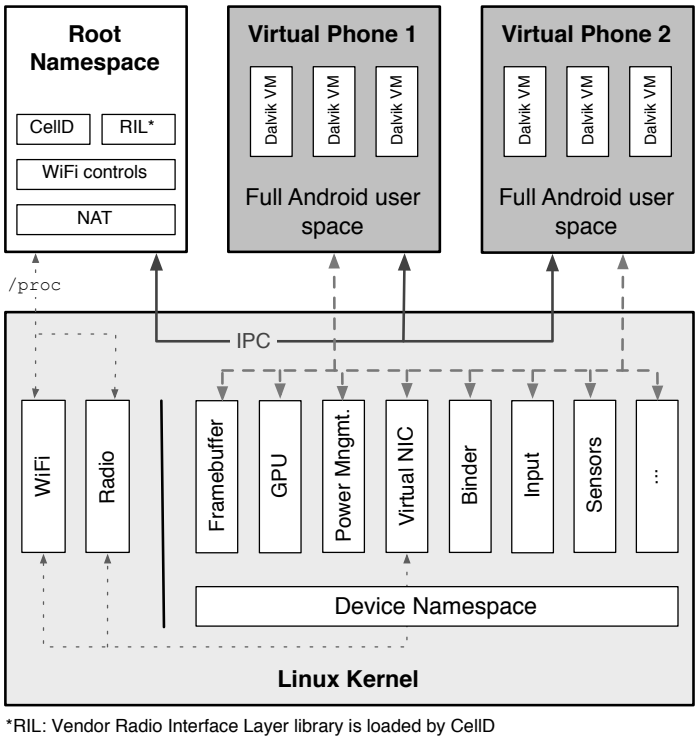


Figure 2.1: Overview of *Cells* architecture

2.2 System Architecture

Figure 2.1 provides an overview of the *Cells* system architecture. We describe *Cells* using Android since our prototype is based on it. Each VP runs a stock Android user space environment. *Cells* leverages lightweight OS virtualization [103; 32] to isolate VPs from one another. Each VP has its own private virtual namespace so that VPs can run concurrently and use the same OS resource names inside their respective namespaces, yet be isolated from and not conflict with each other. This is done by transparently remapping OS resource identifiers to virtual ones that are used by processes within each VP. File system paths, process identifiers (PIDs), IPC identifiers, network interface names, and user names (UIDs) must all be virtualized to prevent conflicts and ensure that processes running in one VP cannot see processes in other VPs. The Linux kernel, including the version used by Android, provides virtualization for these identifiers through namespaces [32]. For example: the file system (FS) is virtualized using mount namespaces that allow different independent views of the FS and provide isolated private FS jails for VPs [80]. *Cells* uses a single OS kernel across all VPs that virtualizes identifiers, kernel interfaces, and hardware resources such that several complete

execution environments, each containing several processes each running in their own Dalvik VM, can exist side-by-side in virtual OS sandboxes.

Cells is complementary to the use of the Dalvik process virtual machine (VM) in Android. The Dalvik VM is similar to a Java VM in that it provides a platform-independent environment for executing Java-like bytecodes. In Android, each application is run using a separate process with its own Dalvik VM instance. Dalvik provides some level of process isolation for each application, but all applications share a single smartphone environment. Dalvik provides no mechanism to support multiple smartphone environments. On the other hand, *Cells* provides an abstraction for the entire Android user environment to support multiple complete and isolated virtual smartphone environments. Each environment has its own system settings, installed applications, and application-specific settings. While Dalvik process virtualization focuses on enabling platform-independence and application isolation, *Cells* OS virtualization focuses on enabling multiple complete and isolated smartphone or tablet environments.

Basic OS virtualization is, however, insufficient to run a complete smartphone or tablet user space environment. OS Virtualization mechanisms have primarily been used in headless server environments with relatively few devices, such as networking and storage, which can already be virtualized in commodity OSes such as Linux. These mechanisms have been extended to desktop computers by virtualizing industry standard interconnection protocols such as PCI or USB. Smartphone applications, however, expect to be able to directly interact with a plethora of hardware devices, many of which are physically not designed to be multiplexed. Furthermore, tightly integrated smartphone hardware is generally accessed through vendor-specific memory-mapped control interfaces, not industry standard interconnection protocols. OS device virtualization support is non-existent for these devices. For Android, at least the devices listed in Table 2.1 must be fully supported, which include both hardware devices and pseudo devices unique to the Android environment. Three requirements for supporting devices must be met: **(1)** Support exclusive or shared access across VPs. **(2)** Never leak sensitive information between VPs. **(3)** Prevent malicious applications in one VP from interfering with device access by another VPs.

Cells meets all three requirements in the tightly integrated, and often proprietary, smartphone ecosystem. It does so by integrating novel kernel-level and user-level device virtualization methods to present a complete virtual smartphone OS environment. Kernel-level mechanisms provide transparency and performance. User-level mechanisms provide portability and transparency when the user space environment provides interfaces

Device	Description
Alarm*	Wake-lock aware RTC alarm timer
Audio	Audio I/O (speakers, microphone)
Binder*	IPC framework
Bluetooth	Short range communication
Camera	Video and still-frame input
Framebuffer	Display output
GPU	Graphics Processing Unit
Input	Touchscreen and input buttons
LEDs	Backlight and indicator LEDs
Logger*	Lightweight RAM log driver
LMK*	Low memory killer
Network	Wi-Fi and Cellular data
Pmem*	Contiguous physical memory allocator
Power*	Power management framework
Radio	Cellular phone (GSM, CDMA)
Sensors	Accelerometer, GPS, proximity

Table 2.1: Android devices

* custom Google drivers

that can be leveraged for virtualization. For proprietary devices with completely closed software stacks, user-level virtualization is necessary.

2.2.1 Kernel-Level Device Virtualization

Cells introduces a new kernel-level mechanism, *device namespaces*, that provides isolation and efficient hardware resource multiplexing in a manner that is completely transparent to applications. Figure 2.1 shows how device namespaces are implemented within the overall *Cells* architecture. Unlike PID or UID namespaces in the Linux kernel, which virtualize process identifiers, a device namespace does not virtualize identifiers. It is designed to be used by individual device drivers or kernel subsystems to tag data structures and to register callback functions. Callback functions are called when a device namespace changes state. Each VP uses a unique device namespace for device interaction. *Cells* leverages its foreground-background VP

usage model to register callback functions that are called when the VP changes between foreground and background state. This enables devices to be aware of the VP state and change how they respond to a VP depending on whether it is visible to the user and therefore the foreground VP, or not visible to the user and therefore one of potentially multiple background VPs. The usage model is crucial for enabling *Cells* to virtualize devices efficiently and cleanly.

Cells virtualizes existing kernel interfaces based on three methods of implementing device namespace functionality. The first method is to create a device driver wrapper using a new device driver for a virtual device. The wrapper device then multiplexes access and communicates on behalf of applications to the real device driver. The wrapper typically passes through all requests from the foreground VP, and updates device state and access to the device when a new VP becomes the foreground VP. For example, *Cells* use a device driver wrapper to virtualize the framebuffer device as described in Section 2.3.1.

The second method is to modify a device subsystem to be aware of device namespaces. For example, the input device subsystem in Linux handles various devices such as the touchscreen, navigation wheel, compass, GPS, proximity sensor, light sensor, headset input controls, and input buttons. The input subsystem consists of the input core, device drivers, and event handlers, the latter being responsible for passing input events to user space. By default in Linux, input events are sent to any process that is listening for them, but this does not provide the isolation needed for supporting VPs. To enable the input subsystem to use device namespaces, *Cells* only has to modify the event handlers so that, for each process listening for input events, event handlers first check if the corresponding device namespace is in the foreground. If it is not, the event is not raised to that specific process. The implementation is simple, and no changes are required to device drivers or the input core. As another example, virtualization of the power management subsystem is described in Section 2.4.

The third method of kernel-level device namespace virtualization is to modify a device driver to be aware of device namespaces. For example, Android includes a number of custom pseudo drivers which are not part of an existing kernel subsystem, such as the Binder IPC mechanism. To provide isolation among VPs, *Cells* needs to ensure that under no circumstances can a process in one VP gain access to Binder instances in another VP. This is done by modifying the Binder driver so that instead of allowing Binder data structures to reference a single global list of all processes, they reference device namespace isolated lists and only allow communication between processes associated with the same device namespace. A Binder device namespace context is only initialized when the Binder device file is first opened, resulting in almost no overhead for

future accesses. While the device driver itself needs to be modified, pseudo device drivers are not hardware-specific and thus changes only need to be made once for all hardware platforms. In some cases, however, it may be necessary to modify a hardware-specific device driver to make it aware of device namespaces. For most devices, this is straightforward and involves duplicating necessary driver state upon device namespace creation and tagging the data describing that state with the device namespace. Even this can be avoided if the device driver provides some basic capabilities as described in Section 2.3.2, which discusses GPU virtualization.

2.2.2 User-Level Device Virtualization

In addition to kernel-level device namespace mechanisms, *Cells* introduces a user-level device namespace proxy mechanism that offers similar functionality for devices, such as the cellular baseband processor, that are proprietary and entirely closed source. *Cells* also uses this mechanism to virtualize device configuration, such as Wi-Fi, which occurs in user space. Sections 2.5 and 2.6 describe how this user-level proxy approach is used to virtualize telephony and wireless network configuration.

Figure 2.1 shows the relationship between VPs, kernel-level device namespaces, and user-level device namespace proxies which are contained in a *root namespace*. *Cells* works by booting a minimal init environment in a root namespace which is not visible to any VP and is used to manage individual VPs. The root namespace is considered part of the trusted computing base and processes in the root namespace have full access to the entire file system. The init environment starts a custom process, CellD, which manages the starting and switching of VPs between operating in the background or foreground. Kernel device namespaces export an interface to the root namespace through the `/proc` filesystem that is used to switch the foreground VP and set access permissions for devices. CellD also coordinates user space virtualization mechanisms such as the configuration of telephony and wireless networking which are discussed in Sections 2.5 and 2.6 respectively.

To start a new VP, CellD mounts the VP filesystem, clones itself into a new process with separate namespaces, and starts the VP's init process to boot up the user space environment. CellD also sets up the limited set of IPC sockets accessible to processes in the VP for communicating with the root namespace. The controlled set of IPC sockets is the only mechanism that can be used for communicating with the root

namespace; all other IPC mechanism are internal to the respective VP. *Cells* also leverages existing Linux kernel frameworks for resource control to prevent resource starvation from a single VP [79].

2.2.3 Scalability and Security

Cells uses three scalability techniques to enable multiple VPs running the same Android environment to share code and reduce memory usage. First, the same base file system is shared read-only among VPs. To provide a read-write file system view for a VP, file system unioning [135] is used to join the read-only base file system with a writable file system layer by stacking the latter on top of the former. This creates a unioned view of the two: file system objects, namely files and directories, from the writable layer are always visible, while objects from the read-only layer are only visible if no corresponding object exists in the other layer. Second, when a new VP is started, *Cells* enables Linux Kernel Samepage Merging (KSM) for a short time to further reduce memory usage by finding anonymous memory pages used by the user space environment that have the same contents, then arranging for one copy to be shared among the various VPs [127]. Third, *Cells* leverages the Android low memory killer to increase the total number of VPs it is possible to run on a device without sacrificing functionality. The Android low memory killer kills background and inactive processes consuming large amounts of RAM. Android starts these processes purely as an optimization to reduce application startup-time, so these processes can be killed and restarted without any loss of functionality. Critical system processes are never chosen to be killed, and if the user requires the services of a background process which was killed, the process is simply restarted.

Cells uses four techniques to isolate all VPs from the root namespace and from one another, thereby securing both system and individual VP data from malicious reads or writes. First, user credentials, virtualized through UID namespaces, isolate the root user in one VP from the root user in the root namespace or the root user in any other VP. Second, kernel-level device namespaces isolate device access and associated data; no data or device state may be accessed outside a VP's device namespace. Third, mount namespaces provide a unique and separate FS view for each VP; no files belonging to one VP may be accessed by another VP. Fourth, CellD removes the capability to create device nodes inside a VP, preventing processes from gaining direct access to Linux devices outside their environment, e.g., by re-mounting block devices. These isolation techniques secure *Cells* system data from each VP, and individual VP data from other VPs. For example, a

privilege escalation or root attack compromising one VP has no access to the root namespace or any other VP, and cannot use device node creation or super-user access to read or write data in any other VP.

2.3 Graphics

The display and its associated graphics hardware are some of the most important devices in smartphones. Applications expect to take full advantage of any hardware display acceleration or graphics processing unit (GPU) available on the smartphone. In fact, modern mobile operating systems make heavy use of hardware graphics acceleration for simple user interactions such as swiping between home screens or displaying menus, and the smooth hardware-assisted graphics is crucial for a rich user experience. Android, for example, uses a process called the *SurfaceFlinger* to compose application windows onto the screen. The SurfaceFlinger process uses the GPU to efficiently blend, animate, or transition application windows for display. Android also makes the GPU available to individual applications through the Open Graphics Library embedded systems API, or *OpenGL ES*. This library specifies a standard interface to accelerated 2D and 3D graphics processing hardware.

Cells virtualizes the display and drawing hardware at two distinct yet interconnected layers: the Linux framebuffer interface used for basic display rendering, and the GPU used by OpenGL for more advanced drawing operations. The standard Linux framebuffer interface used by Android provides an abstraction to a physical display device through a piece of memory called *screen memory*. Screen memory is dedicated to and controlled exclusively by the display device, and its contents correspond exactly to pixels shown on the display. For performance reasons, screen memory is mapped and written to directly by both user space processes and GPU hardware. The GPU, however, is only manipulated by user space processes through an OpenGL API, and while the API itself is open, its implementation is often proprietary. The performance critical nature of graphics processing, direct memory mapping of screen memory to processes and kernel drivers, and the use of proprietary drawing libraries present new challenges for virtualizing mobile devices.

2.3.1 Framebuffer

To virtualize framebuffer access in multiple VPs, *Cells* leverages the kernel-level device namespace and its foreground-background usage model in a new multiplexing framebuffer device driver, `mux_fb`, which serves as a simple, device-independent wrapper to a hardware framebuffer driver. The `mux_fb` driver regis-

ters as a standard framebuffer device and multiplexes access to a single physical device. The foreground VP is given exclusive access to the screen memory and display hardware while each background VP maintains virtual hardware state and renders any output to a virtual screen memory buffer in system RAM, referred to as the *backing buffer*. VP access to the `mux_fb` driver is isolated through the VP's associated device namespace such that a unique virtual device state and backing buffer is associated with each VP. The `mux_fb` driver currently supports multiplexing a single physical framebuffer device, but more complicated multiplexing schemes involving multiple physical devices could be accomplished in a similar manner.

In Linux, the basic framebuffer usage pattern involves three types of accesses: `mmaps`, standard control `ioctls`, and custom `ioctls`. When a process `mmaps` an open framebuffer device file, the driver is expected to map its associated screen memory into the process' address space allowing the process to render directly on the display. A process controls and configures the framebuffer hardware state through a set of standard control `ioctls` defined by the Linux framebuffer interface which can, for example, change the pixel format. Each framebuffer device may also define custom `ioctls` which can be used to perform accelerated drawing or rendering operations.

Cells passes all accesses to the `mux_fb` device from the foreground VP directly to the hardware driver. This includes control `ioctls` as well as custom `ioctls`, allowing applications in the foreground VP to take full advantage of any custom `ioctls` implemented by the physical device driver and used, for example, to accelerate rendering. When an application running in the foreground VP `mmaps` an open `mux_fb` device, the `mux_fb` driver simply maps the physical screen memory controlled by the hardware driver. This creates the same zero-overhead pass-through to the screen memory as on native systems.

Cells does not pass any accesses to the `mux_fb` driver from background VPs to the hardware back end, ensuring that the foreground VP has exclusive hardware access. Standard control `ioctls` are applied to virtual hardware state maintained in RAM. Custom `ioctls`, by definition, perform non-standard functions such as graphics acceleration or memory allocation, and therefore accesses to these functions from background VPs must be at least partially handled by the hardware driver which defined them. Instead of passing the `ioctl` to the hardware driver, *Cells* uses a new notification API that allows the hardware driver to appropriately virtualize the access. If the hardware driver does not register for this new notification, *Cells* can handle custom `ioctls` in one of two ways. One way would be to simply return an error code. Another way would be to block the calling process when the custom `ioctl` is called from a background VP; the process would be unblocked when the VP is switched into the foreground, allowing the `ioctl` to be handled by

the hardware driver. Returning an error code was sufficient for both the Nexus 1 and Nexus S systems. If returning an error code causes the background VP to become unstable, *Cells* can block the calling process allowing the ioctl to be handled by the hardware driver once the VP is switched into the foreground. When an application running in a background VP `mmaps` the framebuffer device, the `mux_fb` driver will map the appropriate backing buffer into the process' virtual address space. This gives applications running in background VPs zero-overhead access to virtualized screen memory.

Not only do background VPs have zero-overhead access to virtual screen memory, but *Cells* also provides each background VP direct access to drawing hardware when it is switched into the foreground. Switching the display from a foreground VP to a background VP is accomplished in four steps, all of which must occur before any additional framebuffer operations are performed:

1. Screen memory remapping.
2. Screen memory deep copy.
3. Hardware state synchronization.
4. GPU coordination.

Screen memory remapping is done by altering the page table entries for each process which has mapped framebuffer screen memory, and redirecting virtual addresses in each process to new physical locations. Processes running in the VP which is to be moved into the background have their virtual addresses remapped to backing memory in system RAM, and processes running in the VP which is to become the foreground have their virtual addresses remapped to physical screen memory. The screen memory deep copy is done by copying the contents of the screen memory into the previous foreground VP's backing buffer and copying the contents of the new foreground VP's backing buffer into screen memory. This copy is not strictly necessary if the new foreground VP completely redraws the screen. Hardware state synchronization is done by saving the current hardware state into the virtual state of the previous foreground VP and then setting the current hardware state to the new foreground VP's virtual hardware state. Because the display device only uses the current hardware state to output the screen memory, there is no need to correlate particular drawing updates with individual standard control `ioctls`; only the accumulated virtual hardware state is needed thus avoiding costly record/replay of framebuffer `ioctls`. GPU coordination, discussed in Section 2.3.2,

involves notifying the GPU of the screen memory address switch so that it can update any internal graphics memory mappings.

To better scale the *Cells* framebuffer virtualization, backing buffers in system RAM could be reduced to a single memory page which is mapped into the entire screen memory virtual address region of background VPs. This optimization not only saves memory, but also eliminates the need for the screen memory deep copy. However, it does require the VP's user space environment to redraw the entire screen when it becomes the foreground VP. Fortunately, redraw overhead is minimal, and Android conveniently provides this functionality through the `fbearlysuspend` driver discussed in Section 2.4.1.

In our testing, we found that the Android *gralloc* library (used to allocate graphics memory for applications) used framebuffer device identification information, such as the name of the driver, to enable or disable graphics functionality. In some cases, the gralloc library would generate errors when unexpected values were returned by the framebuffer driver. To solve this problem, the `mux_fb` driver replicates the identifying information of the hardware driver. Thus, to all user space programs, the device looks exactly like the underlying physical device which is being multiplexed.

2.3.2 GPU

Modern smartphone users expect a smooth, responsive, and highly polished experience from the user interface. Poor quality graphics or clunky animations give users an out-dated or "old" experience, e.g., the lock screen should not simply disappear, it should smoothly transition to a home screen or application perhaps by making icons appear to "fly" onto the screen. Smartphone manufacturers use dedicated graphics processing units, GPUs, to efficently render these complicated animations. Applications running on the smartphone can also expect to use the GPU to render application specific graphics such as scenes in a video game, or custom animations such as a paper-like page curl in a document reader. This means that all VPs running on the smartphone require simultaneous, isolated access to the computational power of the GPU. However, virtualization of graphics resources is an extremely challenging problem for two primary reasons.

First, there is no standard operating system interface to the GPU. A user space application interacts with the GPU solely through a graphics API such as OpenGL. GPU hardware vendors provide an implementation of the OpenGL library which interacts with the physical graphics hardware through often proprietary or opaque interfaces. Details of the GPU hardware and OpenGL library implementation are kept as closely

guarded industry secrets. Graphics driver integration with existing operating system kernel mechanisms such as memory management are generally obfuscated by the closed nature of both the hardware and software using the OS. This leads to unnecessarily duplicated or overlapping functionality in the driver and user space libraries.

Second, graphics APIs such as the OpenGL API guarantee processes graphics memory isolation; one process may not access the graphics memory of another process. Modern GPUs accomplish this using their own hardware memory management unit (MMU) to translate "device virtual addresses" into physically addressable RAM. Memory allocated for graphics processing must therefore be accessible by three different systems, the OS kernel, the user space process, and the GPU, in four different address spaces, user virtual, GPU device virtual, kernel linear, and physical addresses. While it is a primary task of the operating system to manage memory, the closed nature of graphics hardware and software forces GPU device driver and graphics library developers to write custom memory management infrastructures with knowledge of the device specifics. In some cases, small pieces of software run directly on the GPU to handle hardware events such as IRQs and memory page faults [70]. This autonomy from core operating system management infrastructure creates a challenging virtualization problem.

Cells solves these challenges, and virtualizes the GPU by leveraging the existing graphics API and GPU hardware isolation mechanisms in combination with screen memory virtualization similar to the framebuffer virtualization described in Section 2.3.1. Because each VP is essentially an isolated collection of processes running on the same operating system kernel, a VP can be given direct pass-through access to the GPU device. The graphics API, such as OpenGL used in Android, utilizes GPU hardware isolation mechanisms, through the single OS kernel driver, to run each process in its own *graphics context*. Entire VPs are isolated from one another through these graphics contexts, and therefore no further GPU isolation is required. However, each VP requires isolated screen memory on which to compose the final scene displayed to the user, and in general the GPU driver requests and uses this memory directly from within the OS kernel.

Cells solves this problem by leveraging its foreground-background usage model to provide a virtualization solution similar to framebuffer screen memory remapping. The foreground VP uses the GPU to render directly into screen memory, while background VPs will use the same GPU, through the previously discussed isolation mechanisms, to render into their respective backing buffers. When switching the foreground VP, the GPU driver locates all GPU virtual addresses allocated by the current foreground VP and mapped to the screen memory, and remaps them to point to the VP's backing buffer. Correspondingly,

the GPU driver locates GPU addresses allocated by the selected background VP and mapped to its backing buffer, and remaps them to point to screen memory. To accomplish this remapping, *Cells* provides a callback interface from the `mux_fb` driver that provides source and destination physical addresses on each foreground VP switch. This allows the driver to properly locate and switch GPU virtual address mappings.

While this technique necessitates a certain level of access to the GPU driver, it does not preclude the possibility of using a proprietary driver so long as it exposes three basic capabilities. Any driver which implements the *Cells* GPU virtualization mechanism must provide the ability to:

1. Remap GPU device virtual addresses to specified physical addresses.

2. Safely reinitialize the GPU device or ignore re-initialization attempts as each VP running an unmodified user space configuration will attempt to initialize the GPU on startup.

3. Ignore device power management and other non-graphics related hardware state updates, making it possible to ignore such events from a user space instance running in a background VP.

Some of these capabilities were already available in the *Adreno* GPU driver, used in the Nexus 1, but not all. We added a modest number of lines of code to the *Adreno* GPU driver and *PowerVR* GPU driver, used in the Nexus S, to implement these three capabilities. For the Linux kernel version 2.6.35 used with Android version 2.3.3, We added or changed less than 200 lines of code in the *Adreno* GPU driver and less than 500 lines of code in the *PowerVR* GPU driver. These are relatively small changes, given that the *Adreno* GPU driver is almost 6,000 lines of code and the *PowerVR* GPU driver is almost 50,000 lines of code.

While most modern GPUs include an MMU, there are some devices which require memory used by the GPU to be physically contiguous. For example, the *Adreno* GPU can selectively disable the use of the MMU. For *Cells* GPU virtualization to work under these conditions, the backing memory in system RAM must be physically contiguous. This can be done by allocating the backing memory either with `kmalloc`, or using an alternate physical memory allocator such as Google's `pmem` driver or Samsung's `s3c_mem` driver.

2.4 Power Management

To provide *Cells* users the same power management experience as non-virtualized phones, we apply two simple virtualization principles:

1. background VPs should not be able to put the device into a low power mode
2. background VPs should not prevent the foreground VP from putting the device into a low power mode

We apply these principles to Android's custom power management, which is based on the premise that a mobile phone's preferred state should be suspended. Android introduces three interfaces which attempt to extend the battery life of mobile devices through extremely aggressive power management: *early suspend*, *fbearlysuspend*, and *wake locks*, also known as suspend blockers [137].

The *early suspend* subsystem is an ordered callback interface allowing drivers to receive notifications just before a device is suspended and after it resumes. *Cells* virtualizes this subsystem by disallowing background VPs from initiating suspend operations. The remaining two Android-specific power management interfaces present unique challenges and offer insights into aggressive power management virtualization.

2.4.1 Frame Buffer Early Suspend

The *fbearlysuspend* driver exports display device suspend and resume state into user space. This allows user space to block all processes using the display while the display is powered off, and redraw the screen after the display is powered on. Power is saved since the overall device workload is lower and devices such as the GPU may be powered down or made quiescent. Android implements this functionality with two `sysfs` files, `wait_for_fb_sleep` and `wait_for_fb_wake`. When a user process opens and reads from one of these files, the read blocks until the framebuffer device is either asleep or awake, respectively.

Cells virtualizes fbearlysuspend by making it namespace aware, leveraging the kernel-level device namespace and foreground-background usage model. In the foreground VP, reading these two sysfs files functions exactly as a non-virtualized system. Reads from a background VP always report the device as sleeping. When the foreground VP switches, all processes in all VPs blocked on either of the two files are unblocked, and the return values from the read calls are based on the new state of the VP in which the process is running. Processes in the new foreground VP see the display as awake, processes in the formerly foreground VP see the display as asleep, and processes running in background VPs that remain in the background continue to see the display as asleep. This forces background VPs to pause drawing or rendering which reduces overall system load by reducing the number of processes using hardware drawing resources, and increases graphics throughput in the foreground VP by ensuring that its processes have exclusive access to the hardware.

2.4.2 Wake Locks

Power management in Android is predicated on the notion that the base state of the smartphone device should be low-power mode. Processes or kernel drivers must explicitly request that the device remain active. This is accomplished through *wake locks*, or suspend blockers [136; 137], and a corresponding kernel subsystem that uses a timer to opportunistically suspend the device even if processes are still running.

Wake locks are a special kind of OS kernel reference counter with two states: *active* and *inactive*. When a wake lock is "locked", its state is changed to active; when "unlocked," its state is changed to inactive. A wake lock can be locked multiple times, but only requires a single unlock to put it into the inactive state. The Android system will not enter suspend, or low power mode, until all wake locks are inactive. When all locks are inactive, the suspend timer is started. If the timer expires without an intervening lock then the device is powered down.

Wake locks in a background VP interfering with the foreground VP's ability to suspend the device coupled with their distributed use and initialization make wake locks a challenging virtualization problem. Wake locks can be created statically at compile time or dynamically by kernel drivers or user space. They can also be locked and unlocked from user context, kernel context (work queues), and interrupt context (IRQ handlers) independently, making determination of the VP to which a wake lock belongs a non-trivial task.

Cells leverages the kernel-level device namespace and foreground-background usage model to maintain both kernel and user space wake lock interfaces while adhering to the two virtualization principles specified above. The solution is predicated on three assumptions. First, all lock and unlock coordination in the trusted root namespace was correct and appropriate before virtualization. Second, we trust the kernel and its drivers, i.e., when a lock or unlock is called from interrupt context, we perform the operation unconditionally. Third, the foreground VP maintains full control of the hardware.

Under these assumptions, *Cells* virtualizes Android wake locks using device namespaces to tag lock and unlock operations. Multiple device namespaces can independently lock and unlock the same wake lock, and power management operations are only initiated based on the state of the set of locks associated with the foreground VP. The solution comprises the following set of rules:

1. When a wake lock is locked, a namespace "token" is associated with the lock indicating the context in which the lock was taken. A wake lock token may contain references to multiple namespaces if the lock was taken from those namespaces.

2. When a wake lock is unlocked from user context, remove the associated namespace token.

3. When a wake lock is unlocked from interrupt context or the root namespace, remove *all* lock tokens. This follows from our explicit trust of the kernel and its drivers.

4. After a user context lock or unlock, adjust any suspend timeout value based only on locks acquired in the foreground VP's device namespace.

5. After a root namespace lock or unlock, adjust the suspend timeout based on the foreground VP's device namespace.

6. When the foreground VP changes, reset the suspend timeout based on locks acquired in the device namespace of the new foreground VP. This requires per-namespace bookkeeping of suspend timeout values.

One additional mechanism is necessary to implement the *Cells* wake lock virtualization. The set of rules given above implicitly assumes that, aside from interrupt context, the lock and unlock functions are aware of the device namespace in which the operation is being performed. While this is true for operations started from user context, it is not the case for operations performed from kernel work queues. To address this issue, we introduce a mechanism which executes a kernel work queue in a specific device namespace.

2.5 Telephony

Cells provides each VP with separate telephony functionality enabling per-VP call logs, and independent phone numbers. We first describe how *Cells* virtualizes the radio stack to provide telephony isolation among VPs, then we discuss how multiple phone numbers can be provided on a single physical phone using the standard carrier voice network and a single SIM.

2.5.1 RIL Proxy

The Android telephony subsystem is designed to be easily ported by phone vendors to different hardware devices. The Android phone application uses a set of Java libraries and services that handle the telephony state and settings such as displaying current radio strength in the status bar, and selection of different roaming options. The phone application, the libraries, and the services all communicate via Binder IPC with the

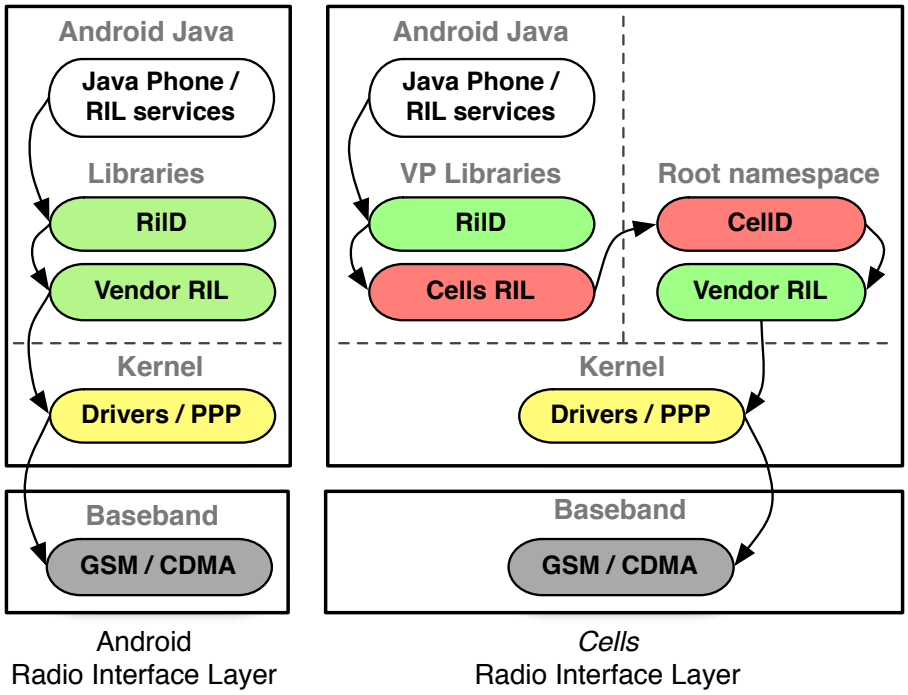


Figure 2.2: *Cells* Radio Interface Layer

Radio Interface Layer (RIL) Daemon (RilD). RilD dynamically links with a library provided by the phone hardware vendor which in turn communicates with kernel drivers and the radio baseband system. The left side of Figure 2.2 shows the standard Android telephony system.

The entire radio baseband system is proprietary and closed source, starting from the user-level RIL vendor library down to the physically separate hardware baseband processor. Details of the vendor library implementation and its communication with the baseband are well-guarded secrets. Each hardware phone vendor provides its own proprietary radio stack. Since the stack is a complete black box, it would be difficult if not impossible to intercept, replicate, or virtualize any aspect of this system in the kernel without direct hardware vendor support. Furthermore, the vendor library is designed to be used by only a single RilD and the radio stack as a whole is not designed to be multiplexed.

As a result of these constraints, *Cells* virtualizes telephony using our user-level device namespace proxy in a solution designed to work transparently with the black box radio stack. Each VP has the standard Android telephony Java libraries and services and its own stock RilD, but rather than having RilD communicate directly with the hardware vendor provided RIL library, *Cells* provides its own proxy RIL library in each

VP. The proxy RIL library is loaded by RilD in each VP and connects to CellD running in the root namespace. CellD then communicates with the hardware vendor library to use the proprietary radio stack. Since there can be only one radio stack, CellD loads the vendor RIL library on system startup and multiplexes access to it. We refer to the proxy RIL library together with CellD as the *RIL proxy*. The right side of Figure 2.2 shows the *Cells* Android telephony system, which has three key features. First, no hardware vendor support is required since it treats the radio stack as a black box. Second, it works with a stock Android environment since Android does not provide its own RIL library but instead relies on a hardware-specific, vendor-provided RIL implementation. Third, it operates at a well-defined interface, making it possible to understand exactly how communication is done between RilD and the RIL library it uses.

Cells leverages its foreground-background model to enable the necessary multiplexing of the radio stack. Users can only make calls from the foreground VP, since only its user interface is displayed. Therefore CellD allows only the foreground VP to make calls. All other forms of multiplexing are done in response to incoming requests from the radio stack through CellD. CellD uses the vendor RIL library in the same manner as Android's RilD, and can therefore take full advantage of the well-defined RIL API. CellD uses existing RIL commands to leverage existing network features to provide call multiplexing among VPs. For example, standard call multiplexing in Android allows for an active call to be placed on hold while answering or making another call. Since CellD intercepts incoming call notifications and outgoing call requests, it can forward an incoming call notification to a background VP even while there is an active call in the foreground VP. CellD can then place the foreground call on hold by issuing the same set of RIL commands that the Android phone application would normally issue to place a call on hold, switch the receiving background VP to the foreground, and let the new foreground VP answer the call. In the same way, if a user switches a foreground VP to the background, and the system is configured to place active foreground calls on hold when this happens, CellD can allow the new foreground VP to place a new call while the existing call is on hold in the background.

The RIL proxy needs to support the two classes of function calls defined by the RIL, *solicited calls* which pass from RilD to the RIL library, and *unsolicited calls* which pass from the RIL library to RilD. The interface is relatively simple, as there are only four defined solicited function calls and two defined unsolicited function calls, though there are a number of possible arguments. Both the solicited requests and the responses carry structured data in their arguments. The structured data can contain pointers to nested data structures and arrays of pointers. The main complexity in implementing the RIL proxy is dealing with the

Call	Class	Category
Dial Request	Solicited	Foreground
Set Screen State	Solicited	
Set Radio State	Solicited	
SIM I/O	Solicited	Initialization
Signal Strength	Unsolicited	Radio Info
Call State Changed	Unsolicited	Phone Calls
Call Ring	Unsolicited	
Get Current Calls	Solicited	

Table 2.2: Filtered RIL commands

implementation assumption in Android that the RIL vendor library is normally loaded in the RilD process so that pointers can be passed between the RIL library and RilD. In *Cells*, the RIL vendor library is loaded in the CellD process instead of the RilD process and the RIL proxy passes the arguments over a standard Unix Domain socket so all data must be thoroughly packed and unpacked on either side.

The basic functionality of the RIL proxy is to pass requests sent from within a VP unmodified to the vendor RIL library and to forward unsolicited calls from the vendor RIL library to RilD inside a VP. CellD filters requests as needed to disable telephony functionality for VPs that are configured not to have telephony access. However, even in the absence of such VP configurations, some solicited requests must be filtered from background VPs and some calls require special handling to properly support our foreground-background model and provide working isolated telephony. The commands that require filtering or special handling are shown in Table 2.2 and can be categorized as those involving the foreground VP, initialization, radio info, and phone calls.

Foreground commands are allowed only from the foreground VP. The *Dial Request* command represents outgoing calls, *Set Screen State* is used to suppress certain notifications like signal strength, and *Set Radio State* is used to turn the radio on or off. *Set Screen State* is filtered from background VPs by only changing a per-VP variable in CellD that suppresses notifications to the issuing background VP accordingly. *Dial Request* and *Set Radio State* are filtered from all background VPs by returning an error code to the calling

background VP. This ensures that background VPs do not interfere with the foreground VP's exclusive ability to place calls.

Initialization commands are run once on behalf of the first foreground VP to call them. The *SIM I/O* command is used to communicate directly with the SIM card, and is called during radio initialization (when turning on the device or turning off airplane mode), and when querying SIM information such as the *IMSI*. The first time a VP performs a *SIM I/O* command, CellD records an ordered log of commands, associated data, and corresponding responses. This log is used to replay responses from the vendor RIL library when other VPs attempt *SIM I/O* commands. When the radio is turned off, the log is cleared, and the first foreground VP to turn on the radio will be allowed to do so, causing CellD to start recording a new log. CellD also records the radio state between each *SIM I/O* command to properly replay any state transitions. The record/replay implementation properly virtualizes *SIM I/O* commands by initializing the proprietary baseband stack only exactly when needed and by simulating expected behavior to background VPs that perform *SIM I/O* initialization commands.

Radio Info commands are innocuous and are broadcast to all VPs. *Signal Strength* is an unsolicited notification about the current signal strength generated by the vendor library. CellD re-broadcasts this information to all VPs with one exception. During initialization, a VP cannot be notified of the signal strength since that would indicate an already initialized radio and generate errors in the VP running the initialization commands.

The *Phone Call* commands, *Call State Changed*, *Call Ring*, and *Get Current Calls*, notify a VP of incoming calls and call state changes. When an incoming call occurs, a *Call State Changed* notification is sent, followed by a number of *Call Ring* notifications for as long as the call is pending. CellD inspects each notification and determines the VP to which it should forward the notification. However, this is somewhat complicated since the notifications are independent and neither notification contains a phone number. When there is an incoming call on an unmodified phone, RilD also receives the *Call State Changed* and *Call Ring* notifications, but issues a *Get Current Calls* command to retrieve the caller ID and display caller information to the user. CellD mirrors this behavior by queueing the incoming call notifications and by issuing the *Get Current Calls* to receive a list of all incoming and active calls. Using tagging information encoded in the caller ID as discussed in Section 2.5.2, CellD determines the target VP and passes the queued notifications into the appropriate VP. When a VP subsequently issues a *Get Current Calls* request, CellD simply forwards

the request (which can be issued multiple times) to the vendor library, but intercepts the data returned from the vendor library and only returns data from calls directed to, or initiated from the requesting VP.

CellD's architecture supports a highly configurable implementation, and there are many valid security configuration scenarios. For example, if the user switches the foreground VP during a call, CellD can either drop the call and switch to the new VP, keep the call alive and switch to a new VP (handling the active call in a background VP), or, deny switching to a new VP until the call is ended by the user. Under all configurations, *Cells* provides strict isolation between every VP by not allowing any information pertaining to a specific VP to be revealed to another VP including incoming and outgoing call information and phone call voice data.

2.5.2 Multiple Phone Numbers

While some smartphones support multiple SIM cards, which makes supporting multiple phone numbers straightforward, most phones do not provide this feature. Since mobile network operators do not generally offer multiple phone numbers per SIM card or CDMA phone, we offer an alternative system to provide a distinct phone number for each VP on existing unmodified single SIM card phones, which dominate the market. Our approach is based on pairing *Cells* with a VoIP service that enables telephony with the standard cellular voice network and standard Android applications, but with separate phone numbers.

The *Cells* VoIP service consists of a VoIP server which registers a pool of subscriber numbers and pairs each of them with the carrier provided number associated with a user's SIM. The VoIP server receives incoming calls, forwards them to a user's actual phone number using the standard cellular voice network, and passes the incoming caller ID to the user's phone appending a digit denoting the VP to which the call should be delivered. When CellD receives the incoming call list, it checks the last digit of the caller ID and chooses a VP based on that digit. *Cells* allows users to configure which VP should handle which digit through the VoIP service interface. CellD strips the appended digit before forwarding call information to the receiving VP resulting in correctly displayed caller IDs within the VP. If the VP is not available, the VoIP service will direct the incoming call to a server-provided voice mail. We currently use a single digit scheme supporting a maximum of ten selectable VPs, which should be more than sufficient for any user. While it is certainly possible to spoof caller ID, in the worst case, this would simply appear to be a case of dialing the wrong phone number. Our VoIP service is currently implemented using an Asterisk [26] server as it

provides unique functionality not available through other commercial voice services. For example, although Google Voice can forward multiple phone numbers to the same land line, it does not provide this capability for mobile phone numbers, and does not provide arbitrary control over outgoing caller ID [60].

The caller ID of outgoing calls should also be replaced with the phone number of the VP that actually makes the outgoing call instead of the mobile phone's actual mobile phone number. Unfortunately, the GSM standard does not have any facility to *change* the caller ID, only to either enable or disable showing the caller ID. Therefore, if the VP is configured to display outgoing caller IDs, *Cells* ensures that they are correctly sent by routing those calls through the VoIP server. CellD intercepts the *Dial Request*, dials the VoIP service subscriber number associated with the dialing VP, and passes the actual number to be dialed via DTMF tones. The VoIP server interprets the tones, dials the requested number, and connects the call.

Note that *Cells* only leverages a VoIP service for multiple phone number support. It does not use VoIP from the smartphone itself. All incoming and outgoing calls from the smartphone are regular calls placed on the cellular network. As a result, *Cells* provides the same kind of call quality and reliability of regular cell phones, especially in geographic locations in which data network coverage can be poor.

2.6 Networking

Mobile devices are commonly equipped with an IEEE 802.11 wireless LAN (WLAN) adapter and cellular data connectivity through either a GSM or CDMA network. Each VP that has network access must be able to use either WLAN or cellular data depending on what is available to the user at any given location. At the same time, each VP must be completely isolated from other VPs. *Cells* integrates both kernel and user-level virtualization to provide necessary isolation and functionality, including core network resource virtualization and a unique wireless configuration management virtualization.

Cells leverages previous kernel-level work [117; 118] that virtualizes core network resources such as IP addresses, network adapters, routing tables, and port numbers. This functionality has been largely built in to recent versions of the Linux kernel in the form of network namespaces [32]. Virtual identifiers are provided in VPs for all network resources, which are then translated into physical identifiers. Real network devices representing the WLAN or cellular data connection are not visible within a VP. Instead, a virtual Ethernet pair is setup from the root namespace where one end is present inside a VP and the other end is in the root namespace. The kernel is then configured to perform Network Address Translation (NAT) between

the active public interface (either WLAN or cellular data) and the VP-end of a virtual Ethernet pair. Each VP is then free to bind to any socket address and port without conflicting with other VPs. *Cells* uses NAT as opposed to bridged networking since bridging is not supported on cellular data connections and is also not guaranteed to work on WLAN connections. Note that since each VP has its own virtualized network resources, network security mechanisms are isolated among VPs. For example, VPN access to a corporate network from one VP cannot be used by another VP.

However, WLAN and cellular data connections use device-specific, user-level configuration which requires support outside the scope of existing core network virtualization. There exists little if any support for virtualizing WLAN or cellular data configuration. Current best practice is embodied in desktop virtualization products such as VMware Workstation [126] which create a virtual wired Ethernet adapter inside a virtual machine but leave the configuration on the host system. This model does not work on a mobile device where no such host system is available and a VP is the primary system used by the user. VPs rely heavily on network status notifications reflecting a network configuration that can frequently change, making it essential for wireless configuration and status notifications to be virtualized and made available to each VP. A user-level library called `wpa_supplicant` with support for a large number of devices is typically used to issue various `ioctls` and netlink socket operations that are unique to each device. Unlike virtualizing core network resources which are general and well-defined, virtualizing wireless configuration in the kernel would involve emulating the device-specific understanding of configuration management which is error-prone, complicated, and difficult to maintain.

To address this problem, *Cells* leverages the user-level device namespace proxy and the foreground-background model to decouple wireless configuration from the actual network interfaces. A configuration proxy is introduced to replace the user-level WLAN configuration library and RIL libraries inside each VP. The proxy communicates with CellD running in the root namespace, which communicates with the original user-level library for configuring WLAN or cellular data connections. In the default case where all VPs are allowed network access, CellD forwards all configuration requests from the foreground VP proxy to the user-level library, and ignores configuration requests from background VP proxies that would adversely affect the foreground VP's network access. This approach is minimally intrusive since user space phone environments, such as Android, are already designed to run on multiple hardware platforms and therefore cleanly interface with user space configuration libraries.

To virtualize Wi-Fi configuration management, *Cells* replaces `wpa_supplicant` inside each VP with

a thin Wi-Fi proxy. The well-defined socket interface used by `wpa_supplicant` is simple to virtualize. The Wi-Fi proxy communicates with CellD running in the root namespace, which in turn starts and communicates with `wpa_supplicant` as needed on behalf of individual VPs. The protocol used by the Wi-Fi proxy and CellD is quite simple, as the standard interface to `wpa_supplicant` consists of only eight function calls each with text-based arguments. The protocol sends the function number, a length of the following message, and the message data itself. Replies are similar, but also contain an integer return value in addition to data. CellD ensures that background VPs cannot interfere with the operation of the foreground VP. For instance, if the foreground VP is connected to a Wi-Fi network and a background VP requests to disable the Wi-Fi access, the request is ignored. At the same time, inquiries sent from background VPs that do not change state or divulge sensitive information, such as requesting the current signal strength, are processed since applications such as email clients inside background VPs may use this information to, for example, decide to check for new email.

For virtualizing cellular data connection management, *Cells* replaces the RIL vendor library as described in Section 2.5, which is also responsible for establishing cellular data connections. As with Wi-Fi, CellD ensures that background VPs cannot interfere with the operation of the foreground VP. For instance, a background VP cannot change the data roaming options causing the foreground VP to either lose data connectivity or inadvertently use the data connection. Cellular data is configured independently from the Wi-Fi connection and VPs can also be configured to completely disallow data connections. Innocuous inquiries from background VPs with network access, such as the status of the data connection (Edge, 3G, HSPDA, etc.) or signal strength, are processed and reported back to the VPs.

2.7 Experimental Results

We have implemented a preliminary *Cells* prototype using Android, and demonstrated its complete functionality across different Android devices, including the Google Nexus 1 [58] and Nexus S [59] phones. The prototype has been tested to work with multiple versions of Android, including the most recent open-source version, version 4.3. However, all of our experimental results are based on Android 2.3.3, which was the latest open-source version available at the time of our experiments. In UI testing while running multiple VPs on a phone, there is no user noticeable performance difference between running in a VP and running natively on the phone. For example, while running 4 VPs on a Nexus 1 device using Android 2.3.3, we

simultaneously played the popular game *Angry Birds* [114] in one VP, raced around a dirt track in the *Reckless Racing* [109] game on a second VP, crunched some numbers in a spreadsheet using the *Office Suite Pro* [90] application in a third VP, and listened to some music using the Android music player in the fourth VP. Using *Cells* we were able to deliver native 3D acceleration to both game instances while seamlessly switching between and interacting with all four running VPs.

2.7.1 Methodology

We quantitatively measured the performance of our unoptimized prototype running a wide range of applications in multiple VPs. Our measurements were obtained using a Nexus 1 (Qualcomm 1 GHz QSD8250, Adreno 200 GPU, 512 MB RAM) and Nexus S (Samsung Hummingbird 1 GHz Cortex A8, PowerVR GPU, 512 MB RAM) phones. The Nexus 1 uses an SD card for storage for some of the applications; we used a Patriot Memory class 10 16 GB SD card. Due to space constraints on the Nexus 1 flash device, all Android system files for all *Cells* configurations were stored on, and run from, the SD card.

The *Cells* implementation used for our measurements was based on the Android Open Source Project (AOSP) version 2.3.3, the most recent version available at the time our measurements were taken. Aufs version 2.1 was used for file system unioning [96]. A single read-only branch of a union file system was used as the `/system` and `/data` partitions of each VP. This saves megabytes of file system cache while maintaining isolation between VPs through separate writable branches. When one VP modified a file in the read-only branch, the modification is stored in its own private write branch of the file system. The implementation enables the Linux KSM driver for a period of time when a VP is booted. To maximize the benefit of KSM, CellD uses a custom system call which adds all memory pages from all processes to the set of pages KSM attempts to merge. While this potentially maximizes shared pages, the processing overhead required to hash and check all memory pages from all processes quickly outweighs the benefit. Therefore, CellD monitors the KSM statistics through the procfs interface and disables shared page merging after the merge rate drops below a pre-determined threshold.

We present measurements along three dimensions of performance: runtime overhead, power consumption, and memory usage. To measure runtime overhead, we compared the performance of various applications running with *Cells* versus running the applications on the latest manufacturer stock image available for the respective mobile devices (Android 2.3.3 build GRI40). We measured the performance of *Cells* when

running 1 VP (1-VP), 2 VPs (2-VP), 3 VPs (3-VP), 4 VPs (4-VP), and 5 VPs (5-VP), each with a fully booted Android environment running all applications and system services available in such an environment. Since AOSP v2.3.3 was used as the system origin in our experiments, we also measured the performance of a baseline system (Baseline) created by compiling the AOSP v2.3.3 source and installing it unmodified.

We measured runtime overhead in two scenarios, one with a benchmark application designed to stress some aspect of the system, and the other with the same application running, but simultaneously with an additional background workload. The benchmark application was always run in the foreground VP and if a background workload was used, it was run in a single background VP when multiple VPs were used. For the benchmark application, we ran one of six Android applications designed to measure different aspects of performance: CPU using Linpack for Android v1.1.7; file I/O using Quadrant Advanced Edition v1.1.1; 3D graphics using Neocore by Qualcomm; Web browsing using the popular SunSpider v0.9.1 JavaScript benchmark; and networking using the `wget` module in a cross-compiled version of BusyBox v1.8.1 to download a single 400 MB file from a dedicated Samsung nb30 laptop (1.66 GHz Intel Atom N450, Intel GMA 3150 GPU, 1 GB RAM). The laptop was running Windows 7, providing a WPA wireless access point via its Atheros AR9285 chipset and built-in Windows 7 SoftAP [89] functionality, and serving up the file through the HFS [65] file server v2.2f. To minimize network variability, a location with minimal external Wi-Fi network interference was chosen. Each experiment was performed from this same location with the phone connected to the same laptop access point. For the background workload, we played a music file from local storage in a loop using the standard Android music player. All results were normalized to 1.0 against the performance of the manufacturer's stock configuration without the background workload.

To measure power consumption, we compared the power consumption of the latest manufacturer stock image available for the respective mobile devices against that of Baseline and *Cells* in 1-VP, 2-VP, 3-VP, 4-VP, and 5-VP configurations. We measured two different power scenarios. In the first scenario, the device configuration under test was fully booted, all VPs started up and KSM had stopped merging pages, then the Android music player was started. In multiple VP configurations, the music player ran in the foreground VP, preventing the device from entering a low power state. The music player repeated the same song continuously for four hours. During this time we sampled the remaining battery capacity every 10 seconds. In the second power scenario, the device configuration under test was fully booted, and then the device was left idle for 12 hours. During the idle period, the device would normally enter a low power state, preventing intermediate measurements. However, occasionally the device would wake up to service

timers and Android system alarms, and during this time we would take a measurement of the remaining battery capacity. At the end of 12 hours we took additional measurements of capacity. To measure power consumption due to *Cells* and avoid having those measurements completely eclipsed by Wi-Fi, cellular, and display power consumption, we disabled Wi-Fi and cellular communication, and turned off the display backlight for these experiments.

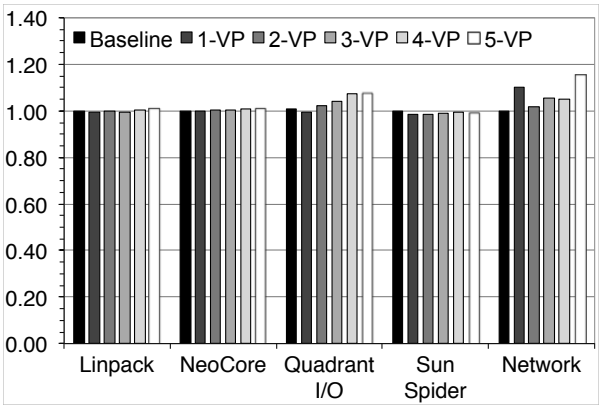
To measure memory usage, we recorded the amount of memory used for the Baseline and *Cells* in 1-VP, 2-VP, 3-VP, 4-VP, and 5-VP configurations. We measured two different memory scenarios. First, we ran a full Android environment without launching any additional applications other than those that are launched by default on system bootup (No Apps). Second, we ran the first scenario plus the Android Web browser, the Android email client, and the Android calendar application (Apps). In both scenarios, an instance of every application was running in all background VPs as well as the foreground VP.

2.7.2 Measurements

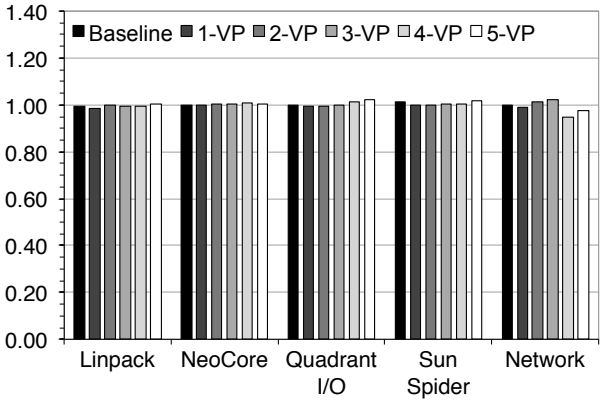
Figures 2.3a to 2.3f show measurement results. These are the first measurements we are aware of for running multiple Android instances on a single phone. In all experiments, Baseline and stock measurements were within 1% of each other, so only Baseline results are shown.

Figures 2.3a and 2.3b show the runtime overhead on the Nexus 1 and Nexus S, respectively, for each of the benchmark applications with no additional background workload. *Cells* runtime overhead was small in all cases, even with up to 5 VPs running at the same time. *Cells* incurs less than than 1% overhead in all cases on the Nexus 1 except for Network and Quadrant I/O, and less than 4% overhead in all cases on the Nexus S. The Neocore measurements show that *Cells* is the first system that can deliver fully-accelerated graphics performance in virtual mobile devices. Quadrant I/O on the Nexus 1 has less than 7% overhead in all cases, though the 4-VP and 5-VP measurements have more overhead than the configurations with fewer VPs. To isolate the impact of VPs on I/O, we configured the Quadrant I/O benchmark to use the internal flash storage for its measurements on the Nexus 1 instead of using the SD card, as the internal flash storage is used for the Baseline system. Experiments using the SD card instead of internal flash for the Quadrant I/O benchmark show that this configuration results in roughly 20% overhead compared to using internal flash.

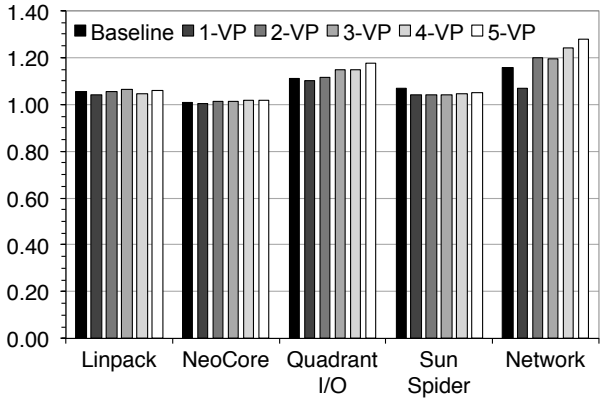
The Network overhead measurements show the highest overhead on the Nexus 1 and the least overhead on the Nexus S. The measurements shown are averaged across ten experiments per configuration. The



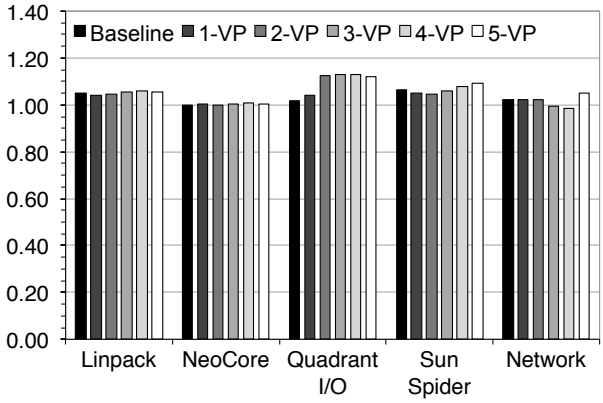
(a) Normalized Nexus 1 results



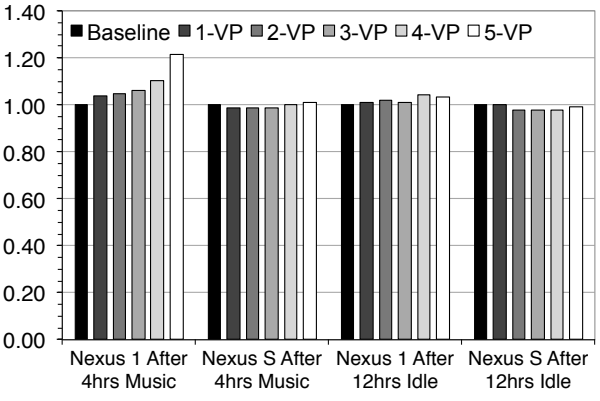
(b) Normalized Nexus S results



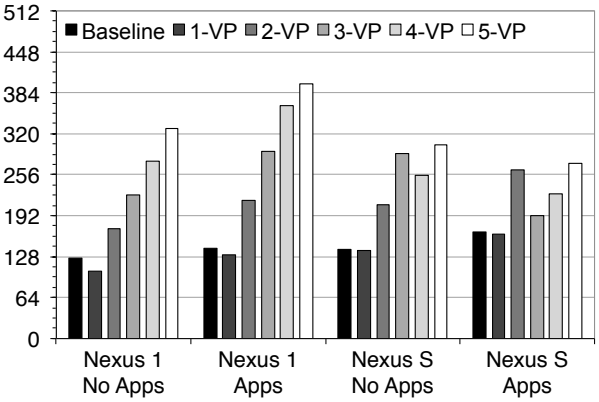
(c) Normalized Nexus 1 + music results



(d) Normalized Nexus S + music results



(e) Normalized battery capacity



(f) Memory usage in MB

Figure 2.3: *Cells* Experimental results

differences here are not reflective of any significant differences in performance as much as the fact that the results of this benchmark were highly variable; the variance in the results for any one configuration was much higher than any differences across configurations. While testing in a more tightly controlled environment would provide more stable numbers, any overhead introduced by *Cells* was consistently below Wi-Fi variability levels observed on the manufacturer's stock system and should not be noticeable by a user.

Figures 2.3c and 2.3d show the runtime overhead on the Nexus 1 and Nexus S, respectively, for each of the benchmark applications while running the additional background music player workload. All results are normalized to the performance of the stock system running the first scenario without a background workload to show the overhead introduced by the background workload. As expected, there is some additional overhead relative to a stock system not running a background workload, though the amount of overhead varies across applications. Relative to a stock system, Neocore has the least overhead, and has almost the same overhead as without the background workload because it primarily uses the GPU for 3D rendering which is not used by the music player. Linpack and SunSpider incur some additional overhead compared to running without the background workload, reflecting the additional CPU overhead of running the music player at the same time. Network runtime overhead while running an additional background workload showed the same level of variability in measurement results as the benchmarks run without a background workload. *Cells* network performance overhead is modest, as the variance in the results for any one configuration still exceeded the difference across configurations. Quadrant I/O overhead was the highest among the benchmark applications.

Comparing to the Baseline configuration with an additional background workload, *Cells* overhead remains small in all cases. It incurs less than 1% overhead in all cases on the Nexus 1 except for Network and Quadrant I/O, and less than 4% overhead in all cases on the Nexus S except for Quadrant I/O, although the majority of benchmark results on the Nexus S show nearly zero overhead. Quadrant I/O on the Nexus 1, while running an additional background workload, incurs a maximum overhead of 7% relative to Baseline performance. Quadrant I/O on the Nexus S has less than 2% overhead for the 1-VP configuration when compared to the Baseline configuration. However, configurations with more than 1 VP show an overhead of 10% relative to the Baseline due to higher I/O performance in the Nexus S baseline compared to the Nexus 1. The higher absolute performance of the Nexus S accentuates the virtualization overhead of running multiple VPs.

Figure 2.3e shows power consumption on the Nexus 1 and Nexus S, both while playing music with the

standard Android music player for 4 hours continuously, and while letting the phone sit idle for 12 hours in a low power state. In both scenarios, the background VPs were the same as the foreground VP except that in the second scenario the music player was not running in the background VPs. Note that the graph presents normalized results, not absolute percentage difference in battery capacity usage, so lower numbers are better.

The power consumption attributable to *Cells* during the 4 hours of playing music on the Nexus 1 increased while running more VPs, which involved scheduling and running more processes and threads on the system and resulted in a higher power supply load variation. The nonlinearity in how this variation affects power consumption resulted in the 4-6% overhead in battery usage for 1-VP through 3-VP, and the 10-20% overhead for 4-VP and 5-VP. In contrast, the Nexus S showed no measurable increase in power consumption during the 4 hours of playing music, though the the noisy measurements had some slight variation. Because the Nexus S is a newer device, the better power management may be reflective of what could be expected when running *Cells* on newer hardware.

Nexus 1 power consumption after 12 hours of sitting idle was within 2% of Baseline. Similarly, Nexus S measurements showed no measurable increase in power consumption due to *Cells* after the 12 hour idle period. When the device sat idle, the Android wake lock system would aggressively put the device in a low power mode where the CPU was completely powered down. The idle power consumption results hold even when background VPs are running applications which would normally hold wake locks to prevent the device from sleeping such as a game like *Angry Birds* or the Android music player. This shows that the *Cells*' wake lock virtualization makes efficient use of battery resources.

Figure 2.3f shows memory usage on the Nexus 1 and Nexus S. These results show that by leveraging the KSM driver and file system unioning, *Cells* requires incrementally less memory to start each additional VP compared to running the first VP. Furthermore, the 1-VP configuration uses less memory than the Baseline configuration, also due to the use of the KSM driver. *Cells* device memory use increases linearly with the number of VPs running, but at a rate much less than the amount of memory required for the Baseline.

The Nexus 1 memory usage is reported for both memory scenarios, No Apps and Apps, across all six configurations. The No Apps measurements were taken after booting each VP and waiting until CellD disabled the KSM driver. The Apps measurements were taken after starting an instance of the Android Web browser, email client, and calendar program in each running VP. Leveraging the Linux KSM driver, *Cells* uses approximately 20% less memory for 1-VP than Baseline in the No Apps scenario. The No

Apps measurements show that the memory cost for *Cells* to start each additional VP is approximately 55 MB, which is roughly 40% of the memory used by the Baseline Android system and roughly 50% of the memory used to start the first VP. The reduced memory usage of additional VPs is due to *Cells*' use of file system unioning to share common code and data as well as KSM, providing improved scalability on memory-constrained phones.

As expected, the No Apps scenario uses less memory than the Apps scenario. Starting all three applications in the 1-VP Apps scenario consumes 24 MB. This memory scales linearly with the number of VPs because we disable the KSM driver before starting the applications. It may be possible to reduce the memory used when running the same application in all VPs by periodically enabling the KSM driver, however application heap usage would limit the benefit. For example, while *Cells* uses 20% less memory for 1-VP than Baseline in the No Apps scenario, this savings decreases in the Apps scenario because of application heap memory usage.

The Nexus S memory usage is reported under the same conditions described above for the Nexus 1. The memory cost of starting a VP on the Nexus S is roughly 70 MB. This is higher than the Nexus 1 due to increased heap usage by Android base applications and system support libraries. The memory cost of starting all three apps in the 1-VP Apps scenario is approximately the same as the Nexus 1, and also scales linearly with the number of running VPs.

However, the total memory usage for the Nexus S shown in Figure 2.3f does not continue to increase with the number of running VPs. This is due to the more limited available RAM on the Nexus S and the Android low memory killer. The Nexus S contains several hardware acceleration components which require dedicated regions of memory. These regions can be multiplexed across VPs, but reduce the total available system memory for general use by applications. As a result, although the Nexus 1 and Nexus S have the same amount of RAM, the RAM available for general use on the Nexus S is about 350 MB versus 400 MB for the Nexus 1. Thus, after starting the 4^{th} VP in the No Apps scenario, and after starting the 3^{rd} VP in the Apps scenario, the Android low memory killer begins to kill background processes to free system memory for new applications. While this allowed us to start and interact with 5 VPs on the Nexus S, it also slightly increased application startup time.

2.8 Related Work

Virtualization on embedded and mobile devices is a relatively new area. Bare-metal hypervisors such as OKL4 Microvisor [97] and Red Bend's VLX [113] offer the potential benefit of a smaller trusted computing base, but the disadvantage of having to provide device support and emulation, an onerous requirement for smartphones which provide increasingly diverse hardware devices. For example, we are not aware of any OKL4 implementations that run Android on any phones other than the dated HTC G1. A hosted virtualization solution such as VMware MVP [28] can leverage Android device support to more easily run on recent hardware, but its trusted computing base is larger as it includes both the Android user space environment and host Linux OS. Xen for ARM [69] and KVM/ARM [43] are open-source virtualization solutions for ARM, but are both incomplete with respect to device support. All of these approaches require some level of paravirtualization, and require an entire OS instance in each VM adding to both memory and CPU overhead. This can significantly limit scalability and performance on resource constrained mobile devices. For example, VMware MVP is targeted to run just one VM to encapsulate an Android virtual work phone on an Android host personal phone. Attempts have been made to run a heavily modified Android in a VM without the OS instance [66], but they lack support for most applications and are problematic to maintain.

User-level approaches have also been proposed to support separate work and personal virtual phone environments on the same mobile hardware. This is done by providing either an Android work phone application [51] that also supports other custom work-related functions such as email, or by providing a secure SDK on which applications can be developed [133]. While such solutions are easier to deploy, they suffer from the inability to run standard Android applications, and use an inherently weaker security model.

Efficient device virtualization is a difficult problem on user-centric systems such as desktops and phones that must support a plethora of devices. Most approaches require emulation of hardware devices, imposing high overhead [138]. Dedicating a device to a VM can enable low overhead pass-through operation, but then does not allow the device to be used by other VMs [93]. Bypass mechanisms for network I/O have been proposed to reduce overhead [87], but require specialized hardware support used in high-speed network interfaces not present on most user-centric systems, including phones. GPU devices are perhaps the most difficult to virtualize. For example, VMware MVP simply cannot run graphics applications such as games within a VM with reasonable performance [VMware, personal communication]. There are two

basic GPU virtualization techniques, API forwarding and back-end virtualization [47]. API forwarding adds substantial complexity and overhead to the TCB, and is problematic due to vendor-specific graphics extensions [75]. Back-end virtualization in a type-1 hypervisor offers the potential for transparency and speed, but unfortunately most graphics vendors keep details of their hardware a trade secret precluding any use of this virtualization method.

2.9 Conclusions

Cells is the first OS virtualization solution for mobile devices. Our solution enables a single mobile device to use multiple personas through isolated and secure virtual instances of a mobile operating system. As the preferred platform for everyday computing trends away from desktop computers, multiple personas on a single mobile device can reduce the number of physical devices a user must carry to maintain the isolation, privacy, and security required by different aspects of the user's life.

Mobile devices have a different usage model than traditional computers. Using this observation, *Cells* provides two new virtualization mechanisms, device namespaces and device namespace proxies, that leverage a foreground-background usage model to isolate and multiplex mobile device hardware with near zero overhead.

Device namespaces provide a kernel-level abstraction that is used to virtualize critical hardware devices such as the framebuffer and GPU while providing fully accelerated graphics. Device namespaces are also used to virtualize Android's complicated power management framework, resulting in almost no extra power consumption for *Cells* compared to stock Android. *Cells* proxy libraries provide a user-level mechanism to virtualize closed and proprietary device infrastructure, such as the telephony radio stack, with only minimal configuration changes to the Android user space environment. *Cells* further provides each virtual phone complete use of the standard cellular phone network with its own phone number and incoming and outgoing caller ID support through the use of a VoIP cloud service.

We implemented a *Cells* prototype that runs the latest open-source version of Android on the most recent Google phone hardware, including both the Nexus 1 and Nexus S. We implemented a *Cells* prototype that runs on the Nexus 1, Nexus S, and Nexus 7 devices. The system can use virtual mobile devices to run standard unmodified Android applications downloaded from the Android market. Applications running inside VPs have full access to all hardware devices, providing the same user experience as applications

running on a native phone. Performance results across a wide-range of applications running in up to 5 VPs on the same Nexus 1 and Nexus S hardware show that *Cells* incurs near zero performance overhead, and human UI testing reveals no visible performance degradation in any of the benchmark configurations.

We have provided a public, open source release of the *Cells* prototype for the Nexus 7 tablet. The code can be browsed and downloaded here: `https://cells-source.cs.columbia.edu/`.

Chapter 3

Cycada

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. For example, 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 [52]. 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 [126; 104], 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 present *Cycada* (formerly known as *Cider* [6]), an OS compatibility architecture that can run apps written and compiled for different mobile ecosystems, namely iOS and Android, simultaneously on the same smartphone or tablet. *Cycada* 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. For example, 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. *Cycada* refines the definition of a *persona* from a full virtual phone to an execution mode assigned to each *thread* on the system, identifying the thread as executing either foreign or domestic code: using a foreign persona or domestic persona, respectively. *Cycada* 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.

Cycada 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 *Cycada*-enabled kernel in exactly the same way as it would with a foreign kernel, i.e., iOS apps trap into the *Cycada* kernel exactly as if they were trapping into a kernel running on an iPhone or iPad. Modifying the domestic kernel in this way allows *Cycada* 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. Reuse of foreign code is essential in mobile ecosystems where 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. Additionally, driver source code for proprietary or custom hardware is rarely available.

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 [13]. Second, although libraries on mobile devices will access custom hardware through opaque kernel interfaces, the actual high-level 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 for graphics on mobile devices.

Based on these observations, *Cycada* supports running unmodified foreign apps on a domestic OS through a novel combination of binary compatibility techniques, including two new OS compatibility mechanisms: compile-time code adaptation, and diplomatic functions. Leveraging existing open source implementations and specifications, *Cycada* provides a foreign binary loader for the domestic kernel which interprets the on-disk contents of foreign application binaries and libraries. The domestic kernel also contains a foreign system call table (or tables), and an asynchronous signal delivery mechanism that translates signals from domestic to foreign and vice-versa. These domestic kernel enhancements allow foreign user space code to be loaded into memory, trap into the domestic kernel, and send and receive signals to and from the domestic kernel. Foreign system call semantics and functionality are mapped, as much as possible, onto pre-existing domestic kernel functionality. Foreign kernels, however, often contain subsystems or functionality that do not exist in the domestic kernel, e.g., the Android Linux kernel does not support Mach IPC, the inter-process communication subsystem used heavily by iOS.

Cycada 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 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.

Cycada introduces *diplomatic functions* which allow foreign applications 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, *Cycada* 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 *Cycada* prototype that can run unmodified iOS and Android applications simultaneously on Android devices. We leverage existing software infrastructure as much as possible, including unmodified user and kernel 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 Android 4.3, Jelly Bean. Users can interact with iOS apps using multi-touch input, and iOS apps can leverage GPU hardware to display smooth, accelerated graphics. Microbenchmark and app measurements of our prototype show that *Cycada* 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.

3.1 Overview of Android and iOS

To understand how *Cycada* 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 3.1 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.

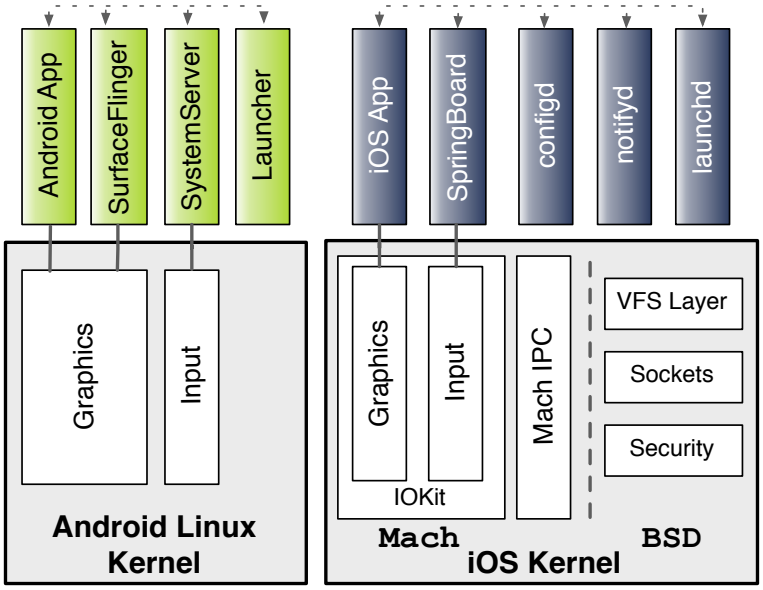


Figure 3.1: Android and iOS Architecture Overview

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 a piece of memory, a graphics "surface", or *window memory*, 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 [13], 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 [112] and IPC mechanisms [141] 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 *SpringBoard* and `backboardd`. SpringBoard displays the iOS home screen. `backboardd` handles and routes user input to apps, and uses the GPU to compose app display surfaces (window memory)

onto the screen. These two services are 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 [10] 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.

3.2 System Integration

Cycada 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. *Cycada* accomplishes this without running the iOS XNU kernel or the SpringBoard app. *Cycada* 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 3.2 shows an overview of the integration of iOS functionality into our Android-based *Cycada* prototype.

iOS apps running on *Cycada* 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 *Cycada*'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, *Cycada* introduces a proxy service, *CycadaSong*. The launch procedure and binary format of iOS and Android apps are com-

[1] In versions of iOS prior to 6.0, graphics rendering, window composition, and input routing were all handled by the SpringBoard app

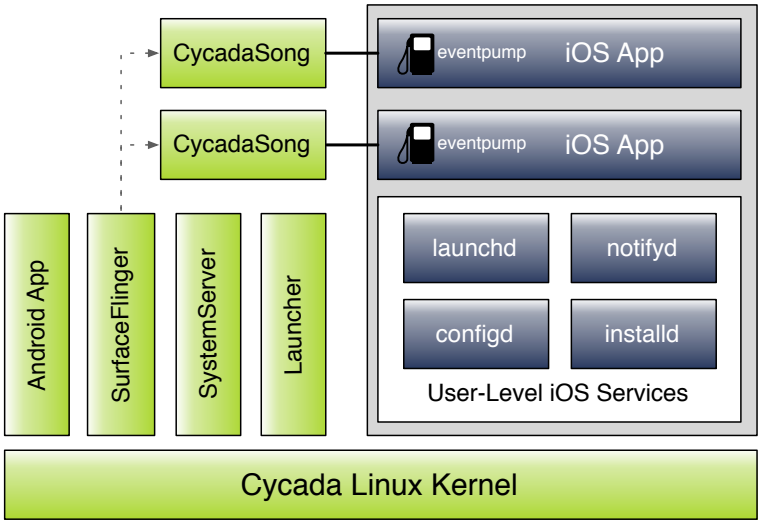


Figure 3.2: System integration overview

pletely different, so iOS app startup and management cannot be directly performed by the Android framework. *CycadaSong* 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 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. *CycadaSong* 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 *CycadaSong* 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.

3.3 Architecture

The primary goal of *Cycada* 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 [12]. Clearly, more than just binary

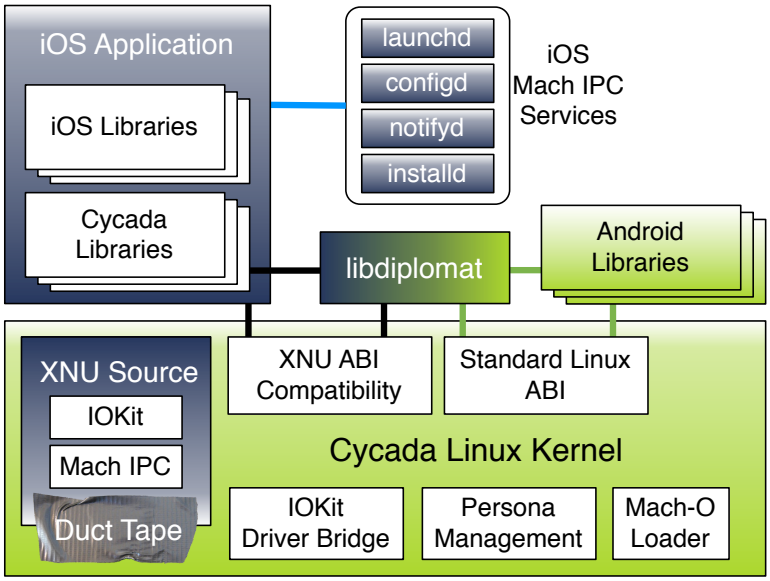


Figure 3.3: Overview of *Cycada* Architecture

compatibility is necessary: *Cycada* must make Android compatible with iOS. *Cycada*'s multi-persona OS compatibility architecture solves this problem.

Figure 3.3 provides an overview of the *Cycada* OS compatibility architecture which can be divided into three key components. First, *Cycada* 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, *Cycada* provides a *duct tape* layer to import foreign kernel code that supports syscalls and subsystems not available in Linux. Third, *Cycada* 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.

3.3.1 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.

Cycada 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* that is used in all subsequent 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, discussed in more detail in Chapter 4.

Cycada 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. *Cycada* also converts XNU signals generated programmatically from iOS applications into corresponding Linux signals, so they can be delivered to non-iOS applications or threads. *Cycada* 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, *Cycada* provides multiple syscall interfaces to support different kernel ABIs. *Cycada* 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. It 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. *Cycada* 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, *Cycada* 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, *Cycada* 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.

3.3.2 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, *Cycada* 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 re-usable 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.

Cycada 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 3.4.1. 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. *Cycada* uses duct tape to directly compile this file without modification.

The Mach IPC subsystem is significantly more complicated than pthread support. It involves many different header files and a large collection of C files. *Cycada* 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 *Cycada* as user space libraries because of the availability of existing open source user-level implementations [62]. Because they did not need to be incorporated into the kernel, they did not use duct tape. Rather, *Cycada* used API interposition [68] to force iOS code to use the *Cycada* entry points instead of XNU system calls.

3.3.3 Diplomatic Functions

XNU kernel ABI support coupled with duct tape allows *Cycada* 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). Apps that link against the OpenGL ES interface can be compiled to run on many devices and platforms, but the library implementations themselves are tied closely to the hardware for which it was written. *Cycada* 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.

Cycada 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.

Cycada'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, *Cycada* 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 *Cycada* 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 3.4.3 (and in more detail in Chapter 4) 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 *Cycada* 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 *Cycada* 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 [55; 49; 50; 39], which only allowed an app to use a single persona, *Cycada* allows an app to switch among personas and use multiple personas simultaneously. *Cycada* 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. Chapter 4 describes how *Cycada* replaces the iOS OpenGL ES library with diplomats that use Android's OpenGL ES libraries.

3.4 iOS Subsystems on Android

We highlight four examples of key iOS subsystems to show how *Cycada*'s OS compatibility architecture supports each subsystem on an Android devices. Section 3.4.1 describes how *Cycada* supports the XNU I/O Kit driver frameworks. Section 3.4.2 details *Cycada*'s multi-touch input support, Section 3.4.4 details *Cycada*'s support for iOS networking facilities such as BSD sockets, multicast DNS, and CoreFoundation networking APIs. Section 3.4.3 outlines our basic support for iOS graphics, however an in-depth discussion of how *Cycada* supports accelerated 2D and 3D graphics through OpenGL can be found in Chapter 4.

3.4.1 Devices

Cycada 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 *Cycada* exactly as they would access Apple devices on iOS.

To directly compile the I/O Kit framework, *Cycada* 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. *Cycada* 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 *Cycada* 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.

Cycada makes devices available via both the Linux device driver framework and I/O Kit. Using a small hook in the Linux `device_add` function, *Cycada* creates a Linux device node I/O Kit registry entry (a device class instance) for every registered Linux device. *Cycada* also provides an I/O Kit driver class for

[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.

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 *Cycada* 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.

3.4.2 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. 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.

Cycada takes a simpler approach to provide complete, interactive, multi-touch input support for iOS. *Cycada* 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 then sends them directly to the Mach IPC port used by apps to receive events. In particular, *Cycada* 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 3.2, listens for events from the Android

CycadaSong 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, *Cycada* 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.

3.4.3 Graphics

Cycada 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.

The following outlines our basic support for iOS graphics, but a more complete and detailed discussion of the graphics subsystem can be found in Chapter 4.

Cycada 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. *Cycada* 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 *Cycada* version of a particular entry point.

To support more complicated 2D and 3D graphics, *Cycada* 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 [77; 74] and Apple-specific *EAGL* [17] extensions. *Cycada* 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, *Cycada* provides a set of diplomats that use the arbitration process, described in Section 3.3.3, 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.

Cycada 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 [76], and this is implemented in an Android EGL library. To support Apple's EAGL extensions, *Cycada* 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 *Cycada* 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 *Cycada* to manage the iOS display in the same manner that all Android app windows are managed.

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

3.4.4 Networking

Networking in iOS involves several high-level APIs which are implemented on top of the Berkeley sockets API (BSD sockets), a proprietary TCP connection library, several I/O Kit drivers, and a handful of key iOS user space daemons. Kernel ABI support, diplomatic functions, and several I/O Kit Linux driver bridges were necessary to fully support iOS networking.

While POSIX specifies API functions and token names, the binary implementation of the API is left unspecified. Both Android and iOS use different binary representations for things such as bits in a flags field, and padding in well-defined structures. *Cycada* uses its XNU kernel ABI support to flip appropriate bits in flag fields, and translate iOS structures into their Linux equivalent. This ABI support allows iOS apps to use BSD sockets. However, iOS layers a significant amount of complexity and functionality on top of BSD sockets which require additional support from *Cycada*.

In iOS, many higher-level APIs, such as `CFNetwork` [16; 11], make use of both BSD sockets as well as a proprietary TCP connection library, and several user space daemons accessed via Mach IPC. The iOS TCP connection library creates and manages TCP connections. This library uses XNU routing sockets, and can dynamically select the best interface over which a connection should route. XNU routing sockets are currently unimplemented in the *Cycada* prototype (see Section 4.9.4), however, using diplomatic functions which interpose on the TCP connection library APIs, *Cycada* can support this critical API. *Cycada* also interposes on key functions in the popular `SCNetworkReachability` API [18] to avoid the need to implement XNU routing sockets.

Functions in iOS such as `gethostbyaddr`, `getaddrinfo`, and many functions in the `SCNetwork-Reachability` framework, communicate with user space daemons such as `mDNS-Responder` and `configd`. These user space daemons are not only responsible for servicing requests for DNS lookups, but they are also responsible for bringing up and configuring network interfaces. iOS keeps the status and properties of its network configuration in both a dynamic and persistent store managed by the `System-Configuration` framework [8]. This framework, along with the user space daemons themselves, use I/O Kit driver interfaces such as `PowerManagement`, `NetworkStack`, and `NetworkInterface` which provide various properties about the available network devices and their capabilities. *Cycada* provides a basic I/O Kit adaptation driver that bridges to appropriate Linux functionality for each of the required I/O Kit interfaces.

User space networking daemons in iOS rely not only on proprietary libraries, and proprietary I/O Kit drivers to provide system-critical information, they also rely on the BSD kqueue asynchronous event deliver kernel subsystem. In iOS, this mechanism has been coupled to Mach IPC, and is a more full-featured replacement for the basic `listen`/`accept`/`read` POSIX-compliant network daemon paradigm. While *Cycada* uses a user space library to replace the `kevent` and `kqueue` APIs (see Section 3.3.2), *Cycada* also needs to implement kernel-level support for event-driven socket notifications. More specifically, *Cycada* extends the Linux `inotify` subsystem to also deliver events such as connection state and socket bytes available to Linux socket inodes. This provides the kernel mechanism that our user space kqueue library uses to map Linux socket events to iOS kevents.

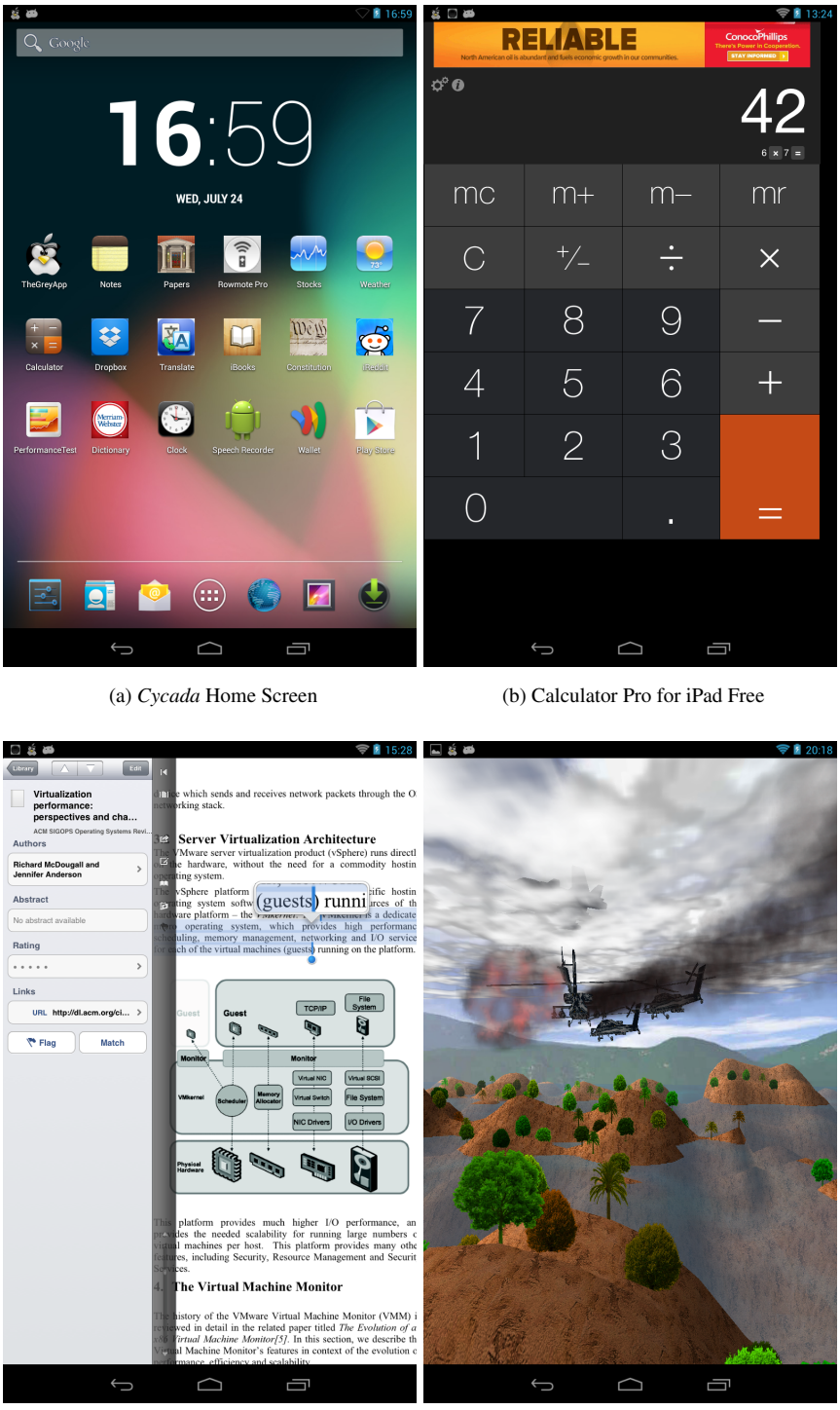
3.5 Experimental Results

We have implemented a *Cycada* 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 *Cycada*:

1. Linux binaries and Android apps running on unmodified (vanilla) Android.
2. Linux binaries and Android apps running on *Cycada*.
3. iOS binaries and apps running on *Cycada*.

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 216ppi 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 163ppi 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 *Cycada*. 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



(a) *Cycada* Home Screen

(b) Calculator Pro for iPad Free

(c) Papers for iOS

(d) PassMark 3D Benchmark

Figure 3.4: *Cycada* Displaying and Running iOS Apps

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 [105] and Google Play [106], 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 *Cycada* performance overhead, and also provides some key observations regarding the characteristics of Android and iOS apps.

3.5.1 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 may be incentivized to provide alternative distribution methods. For example, Google Play might be incentivized to take advantage of *Cycada* 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 [125] to decrypt apps on any jailbroken iOS device using gdb [54]. To illustrate this point, we used both the iPad mini and an old iPhone 3GS running iOS 5.0.1 to decrypt applications. 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 was copied to the *Cycada* prototype, and a small background process automatically unpacked each `.ipa` and created Android shortcuts on the Launcher home screen, pointing each one to the *CycadaSong* Android app. The iOS app icon was used for the Android shortcut. Decrypted iOS apps work on *Cycada* exactly as they would on an iPhone or iPad, including displaying ads using Apple's iAd framework.

Figure 3.4 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 side-by-side, the *Calculator Pro for iPad Free* [7], one of the top three free utilities for iPad at the time of our evaluation,

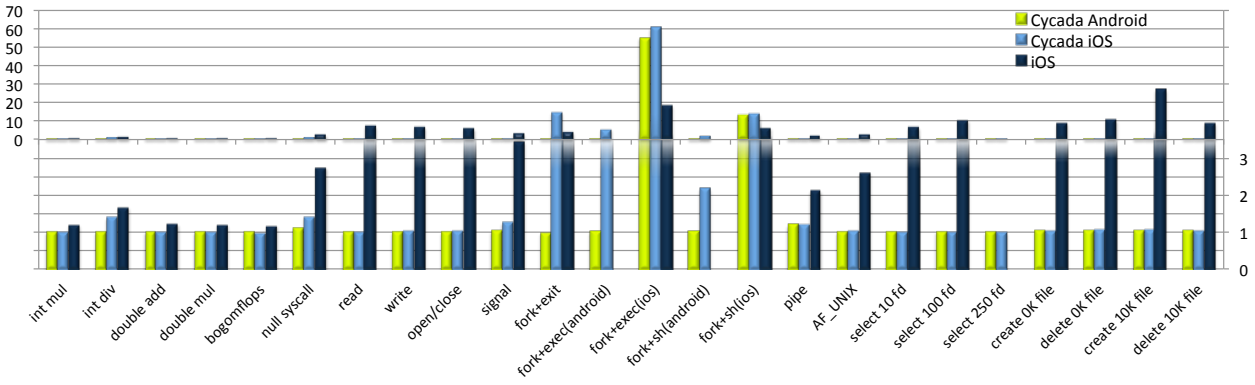


Figure 3.5: Microbenchmark latency measurements normalized to vanilla Android; lower is better performance.

displaying a banner ad via the iAd framework, the highly-rated *Papers* [37] app highlighting text in a PDF, and the *PassMark* [105] app running the 3D performance test.

3.5.2 Microbenchmark Measurements

Figure 3.5 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 3.5 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 3.5 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 *Cy-*

cada on a syscall that does no work, providing a conservative measure of the cost of *Cycada*. 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 *Cycada* is much faster in these syscall measurements than running the same binary on the iPad mini, illustrating a benefit of using *Cycada* to leverage the faster performance characteristics of Android hardware.

The signal handler measurement shows *Cycada*'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 *Cycada* for the signal handler test.

Third, Figure 3.5 shows five sets of process creation measurements, `fork+exit`, `fork+exec`, and `fork+sh` tests. The `fork+exit` measurement shows that *Cycada* 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, *Cycada* 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 *Cycada* on the Android device due to a shared library cache optimization that is not yet supported in the *Cycada* 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 *Cycada*. 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 *Cycada*, 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 3.5 shows all four `fork+exec` measurements using *Cycada*.

The `fork+exec(android)` test forks a child that execs a Linux binary. The *Cycada* Android bar shows results of a Linux test program while the *Cycada* iOS bar shows results of an iOS test program. *Cycada* incurs negligible overhead in the *Cycada* 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. *Cycada* takes 4.8 times longer to run the test in the *Cycada* 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)` *Cycada* 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 *Cycada* Android bar shows results of a Linux test program. The *Cycada* 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 which executes a more heavyweight iOS binary. Nevertheless, using this comparison, Figure 3.5 shows that spawning a child to run an iOS binary is much more expensive. This is because `dyld` loads 90 MB of extra libraries when it starts the iOS binary. Unlike `fork` where copy-on-write can be used to

limit this cost of page table duplication, the cost of `exec` involves not just creating the page tables, but also mapping the libraries themselves into memory. This is very expensive because the *Cycada* 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 *Cycada* on the Android device because of its shared cache optimization which avoids the need to walk the filesystem and load/map each library.

Similar to the `fork+exec` measurement, the `fork+sh` measurement is done in four variations on *Cycada*. The `fork+sh(android)` test launches a shell that runs a Linux binary. *Cycada* 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 3.5 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 3.5 shows local communication and filesystem measurements including `pipe`, `AF_UNIX` sockets, `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 *Cycada* on Android with performance comparable to running a Linux binary on vanilla Android across all measurement variations.

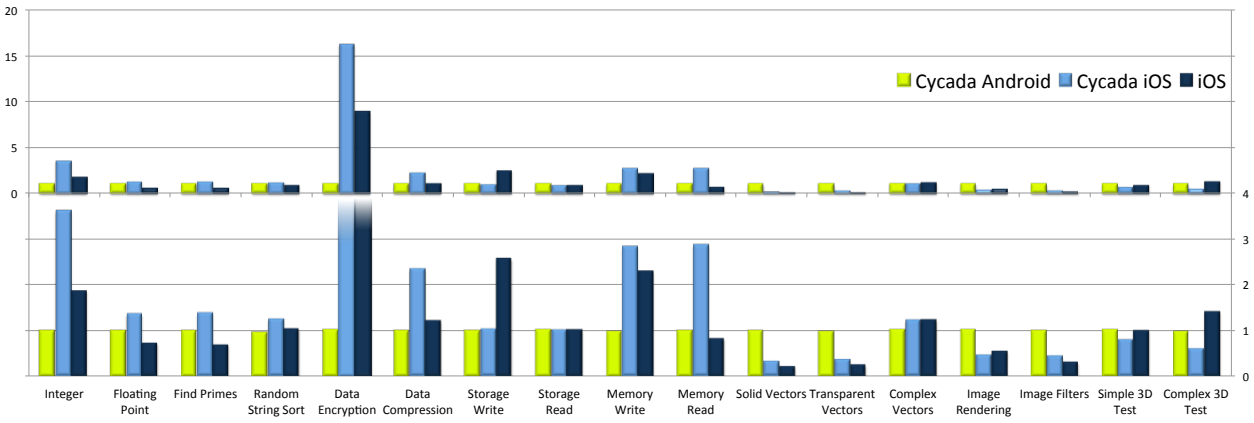


Figure 3.6: App Throughput Measurements Normalized to Vanilla Android; Higher is Better Performance.

3.5.3 Application Measurements

Figure 3.6 shows the results of the iOS and Android PassMark benchmark apps [106; 105] 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, *Cycada* adds negligible overhead to the Android PassMark app. Test results are analyzed in five groups.

First, Figure 3.6 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 *Cycada* 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, *Cycada* outperforms iOS when running the CPU tests from the same iOS PassMark application binary.

Second, Figure 3.6 shows storage operation measurements for write and read tests. *Cycada* 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 *Cycada*. Because storage performance can depend heavily on the OS, these results may reflect differences in both the underlying hardware and the OS.

Third, Figure 3.6 shows memory operation measurements for write and read tests. *Cycada* delivers significantly faster performance when running the iOS PassMark app on Android. This is, again, because *Cycada* runs the iOS app natively while Android interprets the app through the Dalvik VM. *Cycada* 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 3.6 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 *Cycada* 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, *Cycada* generally outperforms iOS due to the Nexus 7's faster CPU. However, bugs in the *Cycada* OpenGL ES library related to "fence" synchronization primitives caused under-performance in the image rendering tests.

Finally, Figure 3.6 shows graphics measurements for simple and complex 3D tests, the latter shown in Figure 3.4d. Because the iPad mini has a faster GPU than the Nexus 7, it has better 3D graphics performance. The iOS binary running on *Cycada* 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.

3.6 Limitations

We have not encountered any fundamental limitations regarding the feasibility of the *Cycada* approach, such as any unbridgeable compatibility issues between iOS and Android. However, while we have implemented an initial *Cycada* 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 cameras, cell phone radio, Bluetooth, and others. Although our prototype supports GPS and location services, full support for the full range of devices available on smartphones and tablets is left to future work. *Cycada* 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 *Cycada*, however, if the iOS app has a fall-back code path it can still partially function.

Implementation of *Cycada* device support varies with device and interface complexity. Devices with a relatively simple interface can be supported with I/O Kit drivers, discussed in Section 3.4.1, 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 the *Cycada* graphics solution, detailed in Section 3.4.3 and Chapter 4, 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.

Our initial prototype contained limited support for advanced graphics. These limitations degraded our graphics performance, however they are addressed in detail in Chapter 4.

Finally, *Cycada* does not map iOS security to Android security, and our system requires a jailbroken iPad or iPhone. 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. Decrypting iOS App Store apps currently requires a jailbroken iPhone or iPad. Extracting decryption keys from an Apple device may mitigate this requirement, but this also is left to future work.

3.7 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 [55; 48; 49] 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 [50]. Similarly, Solaris 10 Zones [92; 100] 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 [102]. *Cycada* 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 [2] 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 [46] 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, *Cycada* provides kernel-level persona management that leverages existing unmodified libraries and frameworks to avoid rewriting huge amounts of user space code. *Cycada*'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 [67] 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 [39]. 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, *Cycada* 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 [35] 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, *Cycada* 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 [40; 108; 140] 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. *Cycada* 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 [61] 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 [78] allows the Linux kernel to load and use Windows NDIS driver DLLs. The project's kernel driver implements the Network Driver Interface Specification (NDIS) [111], 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, *Cycada* 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 [28; 43; 69] 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 [44]. Lightweight OS virtualization [4; 38] claims lower performance overhead, but does not support different OS instances and therefore cannot run foreign apps at all. Unlike *Cycada*, none of these previous approaches can run iOS and Android apps on the same device.

Drawbridge [110] and Bascule [29] 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, *Cycada* 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.

3.8 Conclusions

Cycada 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 *Cycada* to support foreign libraries that are closely tied to foreign hardware by replacing library function calls with diplomats that utilize domestic libraries and hardware. This functionality is essential on mobile devices to support graphics. We built a *Cycada* prototype that reuses existing unmodified user and kernel frameworks across both iOS and Android ecosystems. Our results demonstrate that *Cycada* has modest performance overhead and runs popular iOS and Android apps together seamlessly on the same Android device.

Chapter 4

Cycada Graphics

4.1 Introduction

In the previous chapter, we presented *Cycada*, an OS compatibility architecture that runs applications built for different mobile ecosystems, iOS or Android, together on the same device. *Cycada* was built on the premise that the wide availability of open source software and the use of standardized APIs for mobile app development can be used to build binary compatibility into an existing mobile OS. *Cycada* mimics the ABI of a foreign OS, iOS, enabling the domestic OS, Android, to run unmodified foreign binaries.

Cycada introduces two new binary compatibility mechanisms, compile-time code adaptation, and diplomatic functions. Compile-time code adaptation, or duct tape, allows existing unmodified foreign (iOS) source code to be reused within the domestic (Android) kernel, reducing implementation effort required to support multiple binary interfaces for executing domestic and foreign apps. 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.

A diplomat, or diplomatic function, temporarily switches the persona of a calling thread to execute domestic code 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. Diplomatic functions allow foreign, iOS, apps to leverage domestic, Android, libraries to access proprietary hardware and software interfaces. Diplomats are more than simple API "thunks," "glue code," or "trampolines." Beyond simply

adapting two different API surfaces, a diplomat manages transitions between APIs across different thread-level personas. Each side of the diplomat is assumed to use its own TLS data and calling conventions, and the diplomat translates between the two.

Although Chapter 3 demonstrated the feasibility of the *Cycada* approach, a more detailed examination of iOS and Android compatibility has shown *Cycada* to be incomplete. In particular, the type of graphics acceleration used in libraries such as WebKit [23], the HTML and JavaScript rendering engine, was only partially supported. WebKit consists of over 5 million lines of code [33] and heavily uses the GPU to accelerate web page layout and rendering. GPU device support is of paramount importance given how extensively mobile devices rely on GPUs not only to support 3D graphics, but also to accelerate basic UI functionality, and libraries such as WebKit.

Binary compatible graphics support is a key challenge on mobile platforms because of their vertically integrated stack of proprietary closed-source vendor libraries that communicate directly to the kernel or device drivers through opaque, undocumented calls to control black-box GPU hardware.

In this chapter, we present a graphics-focused study of *Cycada*, and extend its binary compatibility support through three new OS compatibility techniques necessary to build a complete real-world system able to run apps built for different mobile ecosystems, iOS and Android, that require complex GPU-accelerated frameworks such as WebKit, together on the same smartphone or tablet. First, we extend *Cycada*'s basic diplomat construction to perform library-wide prelude and postlude operations in the context of the foreign OS before and after domestic library usage. Using these prelude and postlude functions, we formalize four *diplomat usage patterns*, direct, indirect, data-dependent, and multi, which together provide the complex and rigorous management of resources necessary for bridging between proprietary iOS and Android implementations of complex real-world graphics APIs, such as OpenGL.

Second, we introduce *thread impersonation* which allows thread-specific context to be shared amongst multiple threads running under multiple personas. A thread *impersonating* another thread temporarily takes on the identity of another thread (in all running personas), to perform some action that may be thread-dependent.

Third, we load multiple, independent instances of a single library within the same process through a technique called *dynamic library replication*. Dynamic Library Replication enables a dynamic linker to create separate loaded instances of a dynamic library in a single process including unique virtual addresses for each instance of every symbol in the library including library-global and initialization data.

We extend *Cycada* with these three new OS compatibility techniques to provide binary compatible support for the OpenGL ES (GLES) standard available on both iOS and Android, and heavily used by frameworks such as WebKit. While GLES is standardized, it relies on other graphics infrastructure such as graphics resource management to manage the state information, commands, and resources needed to draw using GLES, and this may not be standardized. Furthermore, the GLES standard is intended to be extensible, and vendor libraries and GPU hardware are free to implement any subset of available extensions or even implement new extensions. We use the diplomat usage patterns, thread impersonation, and dynamic library replication OS compatibility techniques to allow *Cycada*'s graphics compatibility mechanisms to bridge differences in GLES implementations across iOS and Android.

We extend the *Cycada* prototype, and demonstrate its effectiveness in enabling widely-used iOS apps such as Safari that make extensive use of WebKit to run on Android with reasonable graphics performance. We also demonstrate through detailed micro-benchmarks that *Cycada* provides robust binary compatible graphics device support across a broad range of graphics functions. Our detailed study, new OS compatibility techniques, experimental results, and our overall experiences building binary compatible graphics devices support are also useful for other approaches such as virtualization in the context of mobile platforms.

Our experiences with *Cycada* show that our new OS compatibility mechanisms, diplomat usage patterns, thread impersonation, and dynamic library replication, are key to supporting graphics binary compatibility. Our study shows:

1. *Real-world graphics implementations vary greatly between platforms.* Although the GLES API standardized, both iOS and Android take advantage of a number of GLES extensions such that more than half of the extensions used in one platform are not available in the other. Each extensions adds API entry points or modifies the behavior of an existing API. GPU binary compatibility or virtualization approaches that perform simple API forwarding or basic API thunks will not work for these mobile platforms due to the large differences between the resulting extended APIs. Despite their GLES differences, it is possible to map iOS GLES to Android GLES. Most iOS GLES functions, including extension functions, can be supported by leveraging one or more Android GLES functions via diplomat usage patterns.

2. *iOS and Android have substantially different graphics resource management APIs.* Graphics rendering APIs such as GLES require display and memory management APIs to provide window and memory

management. iOS uses an Apple-proprietary API, EAGL, while Android uses the standardized EGL API. These APIs are different enough that running iOS apps on Android requires reimplementing Apple's EAGL APIs, which also requires reverse engineering the EAGL library in the absence of access to non-public Apple specifications and source code. Fortunately, the EAGL API is small, and many functions can leverage aspects of Android's EGL through a combination of diplomat usage patterns and dynamic library replication. However, Android provides an incomplete key EGL extensions which complicates its use.

3. *iOS graphics libraries are designed for multi-threaded use that is not supported in Android's graphics libraries. Cycada* uses thread impersonation and dynamic library replication to allow Android threads to impersonate iOS graphics threads. Each thread uses diplomats to access access multiple, isolated instances of the Android graphics libraries. The isolated graphics libraries and thread impersonation allow each Android thread to perform thread-specific actions and ultimately support multi-threaded iOS graphics functionality.

4. *iOS provides richer support than Android for multiple GLES API versions.* Both iOS and Android support multiple GLES API versions which are useful for different purposes but incompatible. However, iOS allows multiple GLES versions to be used simultaneously by different threads in the same process while Android does not. This is widely used by multi-threaded iOS apps. For example, an iOS game may use GLES v1 APIs to render game graphics, but use a WebKit view to render a basic HTML "about" page which uses GLES v2 APIs. *Cycada* uses dynamic library replication to support iOS apps using multiple GLES API versions on Android.

The rest of this paper is organized as follows. Section 4.2 provides an overview of iOS and Android graphics. Section 4.3 describes the *Cycada* graphics architecture. Sections 4.4 to 4.8 discuss our new OS compatibility techniques in the context of binary compatibility support for various aspects of the iOS graphics system. Section 4.9 presents experimental results. Section 4.10 presents related work. Finally, we present some concluding remarks.

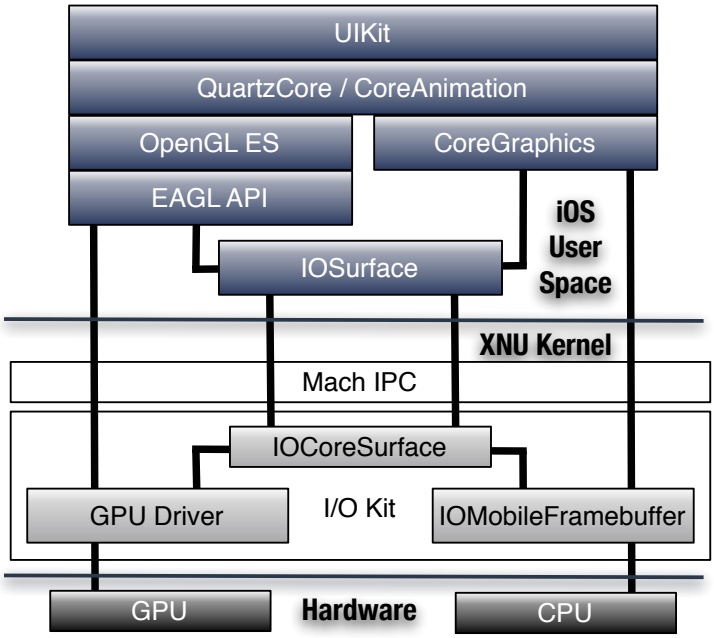


Figure 4.1: Overview of iOS Graphics

4.2 iOS and Android Graphics Overview

Modern graphics subsystems can be broken into three major components: rendering or drawing, display and window management, and memory management. On mobile platforms such as iOS and Android, the most widely-used subsystem for GPU-accelerated graphics is often loosely referred to as OpenGL ES (GLES). However, the GLES API [74] is more properly thought of as a rendering, or drawing, API. GLES takes no responsibility for display and window management. To bridge between the rendered output of GLES and what is actually shown on the screen, GLES relies on the EGL, originally Embedded-System Graphics Library, display API. A native window API such as EGL can be thought of as the canvas on which GLES draws. GLES and EGL objects all require memory to store graphics state. The memory management is done by the OS and generally involves a separate OS-specific API, allowing the resulting memory objects to be efficiently shared between apps or between different drawing APIs, such as between OpenGL and non-OpenGL APIs.

To understand how GLES is supported in iOS and Android, we first review some basics. A GLES *object* is an opaque structure that refers to memory objects which store actual graphics data. Renderbuffers,

framebuffers, and textures are some examples of GLES objects. A GLES *context* is a state container for all GLES objects associated with a given instance of GLES. When a thread calls a GLES function, the function is called in a GLES context to manipulate a GLES object. A thread can create many GLES contexts. Because there are multiple versions of GLES which have different characteristics and are not compatible with each other, an *EGL context*, created with the EGL native window management API, defines the rendering API version used, and therefore the set of GLES functions that can be used within that EGL context.

With this brief background, Figure 4.1 provides an overview of the major graphics components in iOS. iOS apps use user space libraries such as UIKit and CoreAnimation to render user content, such as buttons, text, and images, using CoreGraphics and GLES system libraries. These system libraries communicate directly to the iOS kernel via opaque Mach IPC calls, and 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. For 3D rendering and drawing, iOS uses the standard GLES API [77; 74]. For display and window management, Apple has devised a non-standard native window API, *EAGL* (Embedded Apple GL). The iOS GLES library renders content into a window or display managed by the EAGL library. For memory management, all graphics memory is allocated and manipulated through the IOSurface API which communicates via opaque Mach IPC messages to the IOCoreSurface I/O Kit driver. An IOSurface object is a memory abstraction that facilitates zero-copy transfers of large graphics buffers between apps and rendering APIs. Most GLES objects, such as textures, reference IOSurface memory objects for storing graphics data. For 2D graphics, the CoreGraphics or QuartzCore APIs are used to draw directly into IOSurfaces. IOSurfaces from both CoreGraphics and GLES are composited together using the IOMobileFramebuffer kernel driver, again accessed as an I/O Kit driver via opaque Mach IPC calls. Opaque Mach IPC calls are inter-process communication messages where both client and server ends of the communication either hide or obfuscate the details of messages being passed.

Figure 4.2 provides an overview of the major graphics components in Android. Android Java apps use the android.graphics.canvas and android.opengl APIs to render both 2D and 3D graphics. These Java APIs make extensive use of Java native calls to system libraries which communicate to the Android Linux kernel via opaque `ioctl`s and Binder IPC. Opaque `ioctl`s are `ioctl` system calls on a proprietary driver where both the command and the arguments are intentionally obfuscated or hidden creating an opaque interface into the kernel. For rendering and drawing, Android uses the standard GLES API. All 3D drawing is

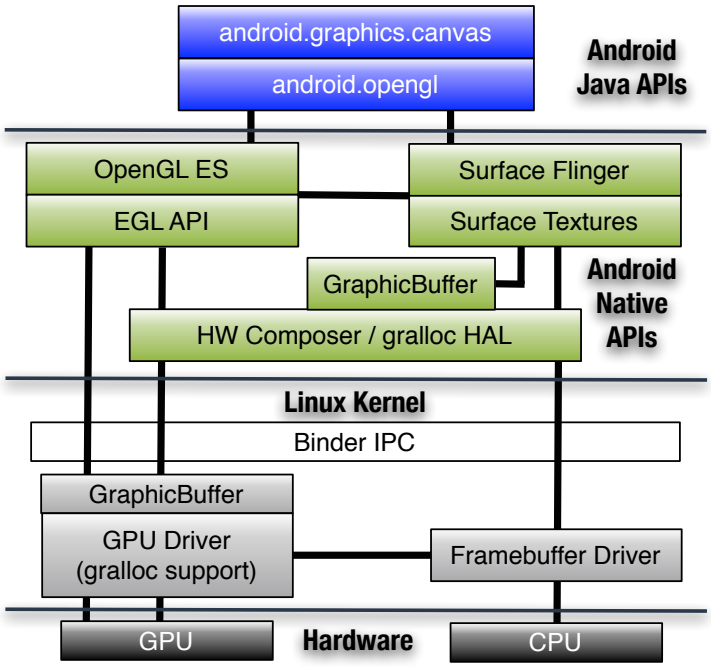


Figure 4.2: Overview of Android Graphics

done via GLES, and as of Android 4.0, all 2D drawing is also accelerated by GLES [120]. For display and window management, Android uses the Khronos standardized EGL [76] API, unlike the proprietary EAGL used by iOS. The Android GLES library renders content into a window or display managed by the EGL library. For memory management, all graphics memory is managed by the GraphicBuffer API, and allocated through the HW Composer or gralloc APIs which use non-standard, often opaque, Linux kernel driver interfaces. Similar to iOS, Android GraphicBuffer objects facilitate cross-process and cross-API zero-copy memory transfers. Unlike their IOSurface counterparts in iOS, GraphicBuffer objects are a much lower-level abstraction managed by the Surface Texture API. Surface Textures are use by both 2D and 3D drawing APIs, and are composited together by the Surface Flinger which uses the HW composer API and Linux kernel framebuffer driver.

4.3 *Cycada* Graphics Architecture

Graphics binary compatibility support in Android for iOS is a key challenge because the graphics sub-systems in both OSes are driven by closed-source libraries that discard all abstractions and communicate

directly with kernel drivers through opaque, undocumented Mach IPC calls and `ioctl`s, which in turn control complex, black-box pieces of hardware. Given the tight coupling of user space libraries to opaque, undocumented kernel APIs, rewriting any complex libraries or drivers, or emulating hardware would be at best an enormous and difficult reverse engineering effort attempting to keep up with product development cycles of large companies. Alternatively, any attempt to interpose on the kernel ABI would be useless without an understanding of the driver-specific `ioctl` commands or opaque Mach IPC messages.

At a high level, *Cycada* addresses this problem by leveraging the fact that the GLES standard is used across mobile platforms. Loosely speaking, instead of having iOS apps use their own iOS GLES libraries, *Cycada* has them use Android GLES libraries. This is done by introducing diplomats [6] which allow foreign apps to use domestic libraries to access proprietary software and hardware interfaces on the device. In *Cycada*, a thread has two personas, or execution modes, a foreign one for executing foreign code with a foreign kernel ABI (iOS) and a domestic one for executing domestic code with a domestic kernel ABI (Android). A diplomat 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 diplomats, *Cycada* replaces calls into foreign hardware-managing libraries, such as GLES, with calls into domestic libraries that manage domestic GPU hardware. Each diplomat maps iOS functionality onto equivalent Android functionality.

We extend the basic *Cycada* diplomat construction to include a *prelude* and *postlude* operation in the context of the foreign persona. Before our extended diplomats switch the persona of the calling thread, they invoke a prelude function that executes in the foreign persona. After invoking the domestic function and switching the persona of the calling thread back to the foreign persona, our new diplomats invoke a postlude function in the context of the foreign persona. These prelude and postlude functions allow *Cycada* to support the necessary multiplexing of multiple loaded instances of a single Android graphics library. This is discussed in more detail in Section 4.8.

The complete process of calling a domestic function from foreign code through a diplomat is:

1. Upon first invocation, a diplomat loads the appropriate domestic library and locates the required entry point (function), storing a pointer to the function in a locally-scoped static variable for efficient reuse.

2. A prelude function executes foreign code using the foreign persona. This function is common to all diplomats and is specified at compile time.

3. Arguments to the domestic function call are stored on the stack.

4. A new `set_persona` system call is invoked from the foreign persona to switch the calling thread's kernel ABI and TLS area pointer to their domestic persona values.

5. Arguments to the domestic function call are restored from the stack.

6. The domestic function call is directly invoked through the symbol stored in step 1.

7. Upon return from the domestic function, the return value is saved on the stack.

8. The `set_persona` syscall is invoked from the domestic persona to switch the kernel ABI and TLS area pointer back to their foreign persona values.

9. Any domestic TLS values, such as `errno`, are appropriately converted and updated in the foreign TLS area.

10. A postlude function executes foreign code using the foreign persona. based on the foreign library being replaced. Similar to the prelude, the postlude function is common to all diplomats and is specified at compile time.

11. The domestic function's return value is restored from the stack, and control is returned to the calling foreign function.

Using diplomat usage patterns, thread impersonation, and dynamic library replication, we complete the *Cycada* graphics compatibility architecture to run unmodified iOS binaries on Android, including iOS apps and graphics frameworks. Figure 4.3 depicts the components of this architecture. Components shown in grey represent new *Cycada* code, components in blue represent iOS code, and components in green represent Android code. Components containing both blue and green represent libraries containing diplomats. Components can be loosely grouped based on graphics compatibility functionality they provide, and we discuss our new OS compatibility techniques in this logical order. Section 4.4 discusses iOS GLES support implemented by the diplomatic OpenGL ES library which includes OpenGL ES Bridge Logic to support *indirect* and *data-dependent* diplomats. Section 4.5 discusses iOS display and window management API support through a re-implemented Apple EAGL API that leverages *multi* diplomats composed in the diplomatic libEGLbridge library. Section 4.6 discusses iOS graphics memory management support which is implemented using a diplomatic IOSurface library and LinuxCoreSurface, a reimplementation of the iOS

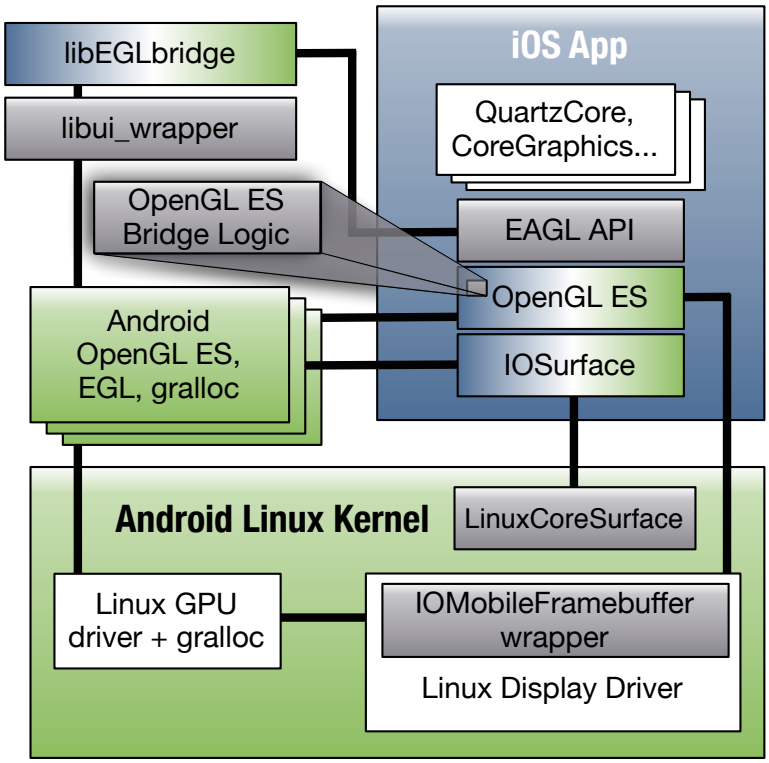


Figure 4.3: *Cycada* iOS Graphics Compatibility

kernel framework, IOCoreSurface. Section 4.7 discusses *Cycada*'s multi-threaded iOS OpenGL ES support using thread impersonation. Section 4.8 discusses iOS EAGL Multi-Context Support using dynamic library replication. This support incorporates the diplomatic libEGLbridge library and a new Android implementation library, *libui_wrapper*.

4.4 GLES

GLES is the rendering, or drawing, API used by both iOS and Android, and its specification has been standardized by the Khronos Group, so at first glance, it seems straightforward to simply replace iOS GLES standard C-function symbols with diplomats that call into the Android GLES library to run iOS apps. However, the GLES standard is intended to be extensible [75], and vendor library implementations are free to implement any subset of available extensions or even implement new extensions. The set of available extensions depends on both the GPU hardware and the vendor library used. Because Apple provides both

OpenGL ES	**iOS**	**Android**	**Khronos**
1.0 Standard Functions	145	145	145
2.0 Standard Functions	142	142	142
Extension Functions	94	42	285
Common Extension Functions	27	27	-
Extensions	50	60	174
Extensions not in Android	33	0	-
Extensions not in iOS	0	43	-

Table 4.1: OpenGL ES Implementation Breakdown

the vendor library and GPU hardware for iOS platforms, iOS GLES implements a similar set of extensions across all iOS platforms of a given generation. However, Android runs on a multitude of platforms provided by many different manufacturers, so the GLES extensions implemented can vary and depend on the particular vendor library and GPU hardware. Apps on both platforms are expected to query GLES to determine what extensions are available and adjust their behavior accordingly.

Table 4.1 gives a summary of standard and extension GLES functions implemented in iOS and Android, as well as the total number of GLES functions reported by Khronos. The Android function list comes from a Nexus 7 tablet with an NVIDIA Tegra 3 GPU. We focus on GLES v1 and v2 standard functions, as GLES v3 is only supported by a minority of both iOS [20] and Android [121] devices. Additionally, the table only considers GLES functions added by extensions, not functionality added to existing functions. Table 4.1 shows that iOS and Android implement the complete set of GLES standard functions, but vastly differ in extensions and extension functions they implement. As a result, there is no possible one-to-one mapping from iOS to Android GLES functions.

4.4.1 Diplomat Usage Patterns

To support the complete set of iOS GLES functions, including extensions, on Android, we taxonomize diplomat usage based on common patterns uncovered through our study of iOS and Android graphics. Similar to the Gang of Four's design patterns [56], our diplomat usage patterns allow *Cycada* to quickly

identify recurring solutions to binary compatibility problems. We formalize four diplomat usage patterns, *direct*, *indirect*, *data-dependent*, and *multi*, that together provide the rigorous and complex management of resources necessary to bridge the intricacies of two mis-matched APIs.

First, standard GLES functions which are not augmented in any way by extensions can be implemented using our extended *Cycada* diplomats. We refer to these as *direct* diplomats. A direct diplomat uses the procedure listed in Section 4.3 to directly invoke a corresponding Android function. For iOS functions where direct invocation of an Android function is not possible, we introduce indirect, data-dependent, and multi diplomats.

An indirect diplomat uses a small amount of custom logic or wrapper code around a standard diplomat. The custom logic runs in the foreign, iOS, context and can re-direct APIs to similar Android APIs with different names, or can manipulate input data to match an existing Android implementation. For example, APPLE_fence [57] is an extension implemented in iOS but not in Android. *Cycada* supports this extension using an indirect diplomat that maps APPLE_fence APIs to a similar extension, NV_fence [73], present on the NVIDIA Nexus 7 tablet. A small amount of iOS code performs minor input re-arranging within each APPLE_fence API before calling into a corresponding Android GLES NV_fence API.

Data-dependent diplomats augment standard diplomats by performing input-dependent logic or implementation before optionally calling the Android function. For instance, if an iOS extension adds the ability to render a new pixel format which is unsupported in Android, GLES functions that allocate or manipulate textures would need data-dependent diplomats that can understand the iOS texture format and manipulate it into a form understood by Android functions. For example, the standard GLES `glGetString` function in iOS has been modified by Apple to accept a non-standard parameter name which is unknown in Android. That parameter name is intended to return Apple–proprietary extensions available on the platform. *Cycada* uses a data-dependent `glGetString` diplomat that interprets the input parameter and either calls the Android function, or returns a custom string indicating that no Apple-proprietary extensions are available. Some data-dependent diplomats may not invoke an Android function at all due to a lack of corresponding Android functionality. For example, the APPLE_row_bytes [72] extension handes two extra parameters to the `glPixelStorei` function, `PACK_ROW_BYTES_APPLE` and `UNPACK_ROW_BYTES_APPLE`, and maintains state associated with the current GLES context which controls how three GLES functions, `glTexImage2D`, `glTexSubImage2D`, and `glReadPixels`, read in or write out pixel data.

Type of Support	**Functions**
Direct Diplomats	312
Indirect Diplomats	15
Data-dependent Diplomats	5
Multi-Diplomats	2
Unimplemented (never called)	10
Total	344

Table 4.2: *Cycada* iOS OpenGL ES Support Breakdown

These three GLES functions are implemented using data-dependent diplomats such that when the APPLE_row_bytes [72] extension is being used, *Cycada* reads in and writes out the packed data manually.

Finally, multi diplomats are necessary when iOS functions or extension logic do not map cleanly to a single Android function, and the behavior is too complex for wrapper or glue logic. These diplomats leverage several different Android library functions through two or more coalesced diplomats. *Cycada* uses multi diplomats to implement a variety of window management and memory management functionality discussed in Sections 4.5 and 4.6.

Table 4.2 indicates how effective our diplomat usage patterns are in supporting iOS GLES on Android. The majority of GLES functions are supported via direct diplomats. Twenty GLES functions are supported via indirect or data-dependent diplomats, and 2 GLES functions require multi diplomats. While indirect diplomats are simple, data-dependent diplomats can, in some cases, require a hundred lines of additional code, and multi diplomats involve the most implementation complexity. No GLES functions need to be completely reimplemented. For completeness, we also list the number of iOS GLES functions that are not implemented in our prototype because they are never called. Note that the total number of functions in Table 4.2 does not match Table 4.1 because the numbers in Table 4.1 are not mutually exclusive; for example, some GLES v1 and v2 standard functions are the same. This table shows that our diplomat usage patterns successfully bridge the gap between iOS and Android GLES APIs.

4.5 EAGL

Graphics resource management, including display and window management, is done in iOS using Apple's own EAGL Objective-C API, but in Android using the Khronos standardized EGL API. There is not a direct mapping from EAGL to EGL which requires *Cycada* to implement substantial logic to support EAGL. However, the Android EGL library performs conceptually similar functions the Apple's EAGL, and the EAGL API itself is small. Using this insight, it is possible to construct an EAGL implementation from a combination of Android EGL and GLES libraries using multi diplomats with a modest amount of support and glue logic. For efficiency, we coalesced our multi diplomats into a single Android library called *libEGLbridge* (see Figure 4.3). This allows us to pay the overhead of a single diplomat which calls into a custom Android API that uses several standard Android functions and libraries to perform the required function.

Table 4.3 shows the EAGL API, which consists of only 17 Objective-C methods. Six methods were supported using multi diplomats, 10 required implementation from scratch, and 1 was not implemented in the current *Cycada* prototype as it was never called. The 10 EAGL functions implemented from scratch required less than 30 lines of code in total as most of them called other EAGL functions, or returned class properties. In contrast, the methods supported by multi diplomats were much more complicated and required approximately 5000 lines of code to implement these 6 methods. To provide clearer insight into how multi diplomats were used to implement EAGL methods, we now describe two examples: render-target object attachment, and double buffering.

4.5.1 Render-Target Object Attachment

Recall from Section 4.2 that the window management API provides a canvas on which the rendering APIs draw. Apple's EAGL provides this functionality, in part, through APIs that attach GLES objects, such as textures and renderbuffers, to memory buffers wrapped in objects called *IOSurfaces*. In iOS, an IOSurface object provides a fast, zero-copy API to manipulate and share large memory allocations (see Section 4.6 for more detail). When connected to an IOSurface, a GLES object use the associated memory buffer to store the results of rendering operations. EAGL can then take the rendered results from the memory buffer and display them on the screen.

GLES functions are standardized and only manipulate GLES objects, not platform-specific objects like

Multi-diplomats		
Type	**EAGLContext method**	**Public**
+	setCurrentContext:	YES
-	initWithAPI:properties:	YES
-	presentRenderbuffer:	YES
-	renderbufferStorage:fromDrawable:	YES
-	texImageIOSurface:target: internaFormat:width:height: format:type: plane:invert:	NO
-	swapNotification:forTransaction: onLayer:	NO

Reimplemented		
Type	**EAGLContext method**	**Public**
+	currentContext	YES
-	API	YES
-	sharegroup	YES
-	initWithAPI:	YES
-	initWithAPI:sharedWithCompute:	YES
-	initWithAPI:sharegroup:	YES
-	getMacroContextPrivate	NO
-	getParameter:to:	NO
-	setParameter:to:	NO
-	attachImage:toCoreSurface: invertedRender:	NO

Unimplemented (never called)		
Type	**EAGLContext method**	**Public**
-	sendNotification:forTransaction: onLayer:	NO

Table 4.3: EAGLContext Objective-C API

IOSurfaces. Apple's EAGL library provides the bridge between platform and window management specific data, and standard GLES objects. Implementing this bridge without low-level OS assistance would result in painfully slow data transfer between processes and APIs making advanced graphics nearly unusable. Fortunately, Android's EGL implementation provides the *EGLImage* extension [71] that introduces EGLImage objects that are used in approximately the same way IOSurfaces are used in iOS. For example, Android's *Surface Texture* API uses EGLImage objects to associate GLES textures with memory objects.

Cycada implements EAGL functions for associating GLES objects with IOSurfaces using multi diplomats. The functions create EGLImage objects using the Android EGL APIs, and connect them to GLES objects using GLES extension APIs. However, Android's EGLImage implementation is incomplete and only allows textures to be tied to memory objects, not renderbuffers. Because iOS allows either textures or renderbuffers to be associated with IOSurfaces for GLES drawing, *Cycada* forces the use of textures to leverage Android's EGLImage extension. This is done by interposing on iOS's EAGL and GLES functions that manipulate renderbuffers and transparently translating their functionality, using multi diplomats, to manipulate the underlying Android GLES textures used in place of renderbuffers.

4.5.2 Double Buffering

Care must be taken when rendering using GLES APIs. The memory backing the render target is necessarily accessible to the window management API and could potentially be displayed on the screen at any time. To prevent corrupt, or half-rendered output, window management APIs usually provide some method of double (or triple, or quadruple) buffering the output. Double buffering allows the rendering APIs to draw into a memory buffer while the window management APIs send a different memory buffer out to the screen.

GLES uses an object called a *framebuffer* to represent the abstracted memory and render target provided by the window management API. Programmers can create many framebuffers, but the first, or default, framebuffer always represents the display screen or window area. The standardized EGL window management API uses a function named, `eglSwapBuffers` to swap the rendering target of the default framebuffer between a "front" buffer (the GLES render target) and a "back" buffer (the memory sent out to the screen or window). In contrast, Apple's EAGL API only allows rendering to an off-screen (non-default) framebuffer. When frame rendering is complete, the programmer must call the `presentRenderbuffer` function that copies the off-screen framebuffer into the display screen or window area.

Because EAGL does not use the default framebuffer, the standard Android `eglSwapBuffers` will not work to transfer rendered frame data to the screen or window memory - the rendered data was never put into memory associated with the default framebuffer! To display the contents of an off-screen framebuffer into which an iOS app has rendered content, *Cycada* implements the EAGL `presentRenderbuffer` function using a multi diplomat. This diplomat uses simple GLES vertex and fragment shader programs, via several Android GLES APIs, to render the off-screen framebuffer contents into the default framebuffer. From the default framebuffer, *Cycada* can use `eglSwapBuffers` to display the content. This implementation is obviously inefficient, and could be improved through a more complicated management of underlying graphics memory or deeper ties into the Android EGL/GLES libraries. However, due to time constraints, *Cycada* prototype performance, detailed in Section 4.9.2, suffers from this inefficiency.

4.6 Memory Management

The massive size, low latency requirements, and cross–process composition of graphics objects requires an efficient, zero–copy mechanism that allows graphics memory to be shared between libraries and applications. iOS uses *IOSurface* objects for efficient graphics memory management. Kernel-level IOSurface support (see IOCoreSurface in Figure 4.1), provides the zero-copy support which allows IOSurface objects to be efficiently passed between libraries and applications. In the absence of internal Apple information, we needed to reverse engineer IOCoreSurface kernel APIs and functionality. The resulting module is shown as *LinuxCoreSurface* within the Android Linux kernel in Figure 4.3.

Android manages efficient graphics memory transfers using *GraphicBuffer* objects. While these objects perform the same high-level functions as iOS IOSurface objects, the IOSurface API is significantly more complicated and offers a richer interface for manipulating, sharing, and remapping graphics memory. *Cycada* must therefore provide a mapping between IOSurfaces and GraphicBuffers for GLES to function correctly. We discuss two key aspects of IOSurfaces and how they are supported in *Cycada*: object life cycle management, and cross-API object sharing.

4.6.1 IOSurface Life Cycle Management

IOSurfaces are created using `IOSurfaceCreate`. This function must allocate the necessary memory buffer, and connect the allocated region to the supporting kernel infrastructure. To provide the necessary

kernel support for advanced IOSurface memory operations, *Cycada* interposes on `IOSurfaceCreate` using an indirect diplomat to create an Android GraphicBuffer object as the underlying backing graphics memory for an IOSurface. Similarly, as the created IOSurface is associated with GLES textures, or other library objects, *Cycada* uses indirect diplomats to interpose Android GraphicBuffer management. For example, *Cycada* interposes on the `glDeleteTextures` API and removes any corresponding connection to the underlying Android GraphicBuffer.

4.6.2 Cross-API Object Sharing

An IOSurface, much like its GraphicBuffer counterpart, can be used by 3D as well as other 2D rendering APIs. These 2D APIs, such as CoreGraphics, use the CPU to draw directly into IOSurfaces as opposed to sending commands to the GPU to render content into the memory. To allow 2D and 3D APIs to share IOSurfaces, iOS provides the `IOSurfaceLock` and `IOSurfaceUnlock` functions to lock and unlock an IOSurface for CPU-only access, during which time the GPU may not access it. Although the Android GraphicBuffer object can be locked for CPU-only access, it *cannot* be CPU locked after it has been associated with a GLES texture (via an EGLImage). As discussed in Section 4.5, iOS associates IOSurfaces, and thus GraphicBuffers, with GLES textures for 3D drawing. Thus the exact scenario in which we need to lock a GraphicBuffer to support IOSurfaces is unsupported by the Android API!

To circumvent this limitation, *Cycada* interposes on the `IOSurfaceLock` and `IOSurfaceUnlock` functions with multi diplomats. When an IOSurface is locked, *Cycada* disassociates the Android GraphicBuffer from the connected GLES texture allowing it to be locked for CPU-only access. However, this process is non-trivial, and unsupported by current Android GLES and EGL APIs. A GLES texture is required to be associated with some memory object, so while the IOSurface is locked for CPU access the *Cycada* multi diplomat rebinds the GLES texture to a single-pixel buffer allocated by `glTexImage2D`. The multi diplomat can then destroy the EGLImage object associated with the texture which implicitly disassociates the Android GraphicBuffer. [1] At this point, the GraphicBuffer can be locked for CPU access. By assuming correct IOSurface locking behavior of the iOS application, we know that no Open GL function calls will occur that will try to render using the texture.

[1]The GraphicBuffer object is tied to an EGLImage, and the EGLImage is used as the backing object for the GLES texture. Destroying the EGLImage object explicitly disassociates the GraphicBuffer fromt the EGLImage, and ultimately from the GLES texture.

Cycada also interposes on the `IOSurfaceUnlock` function with another multi diplomat. Here, we create a new EGLImage object and rebind it, and the GraphicBuffer, back to the GLES texture. Since GLES did not have access to the IOSurface (or GraphicBuffer) while it was locked, the disassociation and re-association process is transparent to iOS's GLES.

4.7 Multi-Threaded GLES

GLES and EGL are commonly used in heavily multi-threaded environments, however there are some restrictions based on the specifications. First, GLES functions are not thread safe, so apps are expected to synchronize access to GLES state outside of the GLES API. Second, an EGLContext object (see Section 4.2) defines the set of GLES functions available, and it is possible for a standards-compliant EGL implementation to allow only a single context to be created per thread group [76]. Thus, the accepted standard for multi-threaded apps using GLES/EGL is to use a single thread dedicated for rendering.

iOS and Android are both heavily multi-threaded environments, but differ in the level of GLES threading support. iOS allows any thread to use a GLES context; one thread can create a GLES context and another can use it. Apple's Grand Central Dispatch (GCD) is used heavily and relies on this feature to asynchronously dispatch GLES jobs such as texture loading or off-screen rendering. Each thread in the system has its own context, and implicitly takes on the GLES and EAGL context of the thread that submitted the asynchronous job. Similarly, the iOS WebKit library spawns a rendering thread that allocates and initializes its own GLES context which is used by other threads related to WebKit. This level of multi-threaded support does not exist in Android GLES or EGL libraries, which only allow a GLES context to be used by a thread if it or its thread group leader created the context. In other words, a GLES context created by Android thread 1 could not be used by Android thread 2 unless thread 1 also happened to be the "main" thread.

4.7.1 Thread Impersonation

To support multi-threaded iOS GLES apps on Android, *Cycada* introduces thread impersonation. Thread impersonation allows thread-specific context to be shared amongst multiple threads enabling one thread to temporarily assume the persona of another thread. While more limited forms of this impersonation have been used in security contexts, *Cycada* thread impersonation presents a generalized mechanism to impersonate a

thread across *all* personas in which the thread may execute. In each of these personas, a subset of thread-specific data may be used or shared by the impersonating thread.

In *Cycada*, iOS threads attempting to perform GLES operations will impersonate the Android thread that created an Android GLES context. We refer to the Android thread which created the GLES context as the target thread, and the iOS thread that invoked a GLES function (through a diplomat) the running thread.

At a high level, diplomats invoked by the running thread will leverage the prelude and postlude functions, discussed in Section 4.3, to migrate thread local data between the target and running threads. Both Android and iOS TLS state must be migrated. Although somewhat unintuitive, this is necessary because the target thread created the GLES context through a diplomat. Thus the target thread has both iOS and Android graphics state in their respective thread local storage.

However, not all data in thread local storage (TLS) needs to be, or should be, migrated. TLS is an array of void pointers unique to each persona of thread. Each array entry is a slot. Some TLS slots are reserved for system use for things such as a thread-local `errno` value, but apps can reserve other slots using the `pthread_key_create` function, which returns a globally-unique TLS slot ID. A given thread passes the returned slot ID into the `pthread_getspecific` or `pthread_setspecific` functions to get or set a thread-local, or thread-private, value. *Cycada* thread impersonation allows selective migration of TLS data by modifying Android's libc to send out a notification whenever a new TLS key is reserved through `pthread_key_create` and destroyed through `pthread_key_delete`; this is a trivial 12 line patch. By registering for hook that is invoked on every `pthread_key_create` and `pthread_key_delete` call, we can selectively monitor TLS slot allocation.

Because *Cycada* migrates graphics contexts, we monitor graphics-specific TLS slot allocations by gating the Android `pthread_key_create` and `pthread_key_delete` hooks in the prelude and postlude of each graphics diplomat. In this way we selectively migrate only the graphics-relevent TLS data between the target and running thread's Android personas. We also migrate well-known iOS TLS slots used by Apple graphics libraries. [2] Since vendor graphics libraries are opaque (even on Android), we can assume that the TLS slots they reserve are not used by any other subsystems because their use of TLS slots are also opaque to other subsystems.

[2]Well-known iOS TLS slots are used by other Apple graphics libraries, such as CoreGraphics, when invoking GLES or EAGL APIs.

More formally, *Cycada* thread impersonation for graphics is done as follows:

1. *Cycada* identifies graphics-related TLS state using `pthread_key_create` and `pthread_key_delete` Android libc hooks as described above.
2. Whenever an GLES context is created or modified, *Cycada* ties the graphics-related TLS of the thread that created the GLES context to the GLES context itself.
3. Whenever a thread calls a GLES function using a GLES context it did not create, *Cycada* saves the running thread's graphics-related TLS state, in both its iOS and Android personas, and replaces it with TLS data associated with the GLES context.
4. Updates are made to the TLS values as needed as the thread executes graphics functions, and these updates are reflected back into the TLS associated with the GLES context.
5. Upon GLES function return, *Cycada* restores the running thread's original graphics-related TLS state.

In *Cycada*, a thread has both an iOS and an Android persona, and each persona has its own TLS. *Cycada* must ensure that graphics-related use of TLS in the iOS persona matches what is expected by iOS apps. In general, this is done by relying on iOS libraries to manipulate the TLS in the iOS persona as needed without any additional work by *Cycada*. However, when a thread submits an asynchronous job to GCD, *Cycada* must associate the iOS TLS data of the submitting thread with the EAGL context such that when the GCD job is run on a different thread, that thread's iOS TLS can be properly updated.

Cycada must also ensure that when diplomats are used, thread migration in Android is done to match the necessary iOS GLES and EAGL contexts, i.e., thread migration must occur for each persona. For example, if thread A passes its context to thread B before calling a diplomat, thread B must impersonate thread A in both iOS *and* Android. However, iOS and Android use separate TLS areas for execution, and the only place with knowledge of both TLS areas is the kernel. Thus to effect thread migration, *Cycada* introduces two new system calls: `locate_tls` and `propagate_tls`. The `locate_tls` syscall can extract TLS values from any given persona in which a thread has executed. Similarly, the `propagate_tls` syscall pushes TLS values into any given persona. Using these two syscalls, *Cycada* ensures proper GLES functionality across multi-persona thread migration.

Although Android's EGL implementation is not designed for the type of multi-threaded support available in iOS, *Cycada* can still guarantee correct behavior of the calling GLES functions. The GLES spec-

ification explicitly requires external thread synchronization. By assuming that an iOS app or framework has correctly synchronized calls to GLES functions, *Cycada* can guarantee that calls to the Android GLES library, through diplomats will be properly synchronized.

4.8 EAGL Multi-Context Support

The iOS EAGL library can instantiate multiple *EAGLContext* objects, each with their own GLES connection. Each GLES connection can use a different API version. For example, consider a simple iOS game with an initial menu interface, and a UIWebView to render an HTML "about" page. The UIWebView class uses the WebKit library to render HTML which implicitly creates its own EAGLContext object connected to GLES API v2. If the game uses GLES, it will create its own EAGLContext object with its own connection to the GLES API. The iOS app is free to use either GLES v2 *or* GLES v1. If the game uses GLES v1, the process has now instantiated two unique EAGLContext objects each using a different version of the GLES API. There is no EGL or Android mechanism to support this paradigm. Only a single EGL connection to a single GLES API version can be made per-process.

4.8.1 Dynamic Library Replication

To support multiple EAGLContexts in a single process, we introduce *dynamic library replication*. At a high level, our solution reloads and re-initializes, or re-instances, all the Android graphics libraries. Each new library instance, or *replica*, is loaded and linked as if no other libraries have been loaded. [3] This causes each replica to occupy its own virtual memory space, and invoke its own pseudo-private copies of all library functions and their constituent dependencies. A replica, then, is essentially a library namespace which include all dependent libraries. For example, the NVIDIA graphics support library, *libGLESv2_tegra.so* requires the *libnvrm.so* library which, in turn, requires the *libnvos.so* library. Each replica of the *libGLESv2_tegra.so* library would not only occupy its own virtual address space, it would also link against privately loaded copies of all required libraries such as *libnvrm.so* and *libnvos.so*.

To better understand the need for dynamic library replication, we first discuss Android's EGL implementation. Android's EGL implementation can be broken into two pieces: an open source library that

[3] We do *not* reload libc; all library instances use a single, shared instance of libc.

exports all the standardized EGL functions, and a vendor-provided, device-specific EGL implementation. Android applications link against the open source library, and when an app initializes the EGL interface using the `eglInitialize` function, the open source library loads the vendor-specific EGL and OpenGL ES libraries. The restriction of a single EGL-to-GLES connection per process is seemingly arbitrary, but enforced by both the open source library and the vendor-specific libraries.

Modifying the open source EGL implementation to maintain per-thread EGL connection information only partially solves the problem. While individual threads in a process can separately initialize and maintain EGL-to-GLES connections through the open source library, the vendor provided EGL and GLES libraries are proprietary and closed source and operate under the assumption of a single, process-wide EGL connection.

To bypass arbitrary vendor restrictions on singleton EGL connections, *Cycada* introduces the concept of dynamic library replication (DLR) - a dynamic linker mechanism that loads multiple, independent instances of a dynamic library. We refer to a duplicate loaded instance as a *replica*. Normally, on a call to `dlopen` the linker will not re-initialize or reload a library if it's already been loaded. The linker will simply return a handle to the previously loaded instance. The *Cycada* DLR-enabled linker introduces a new function, `dlforce`, which opens a library (the replica), and all its dependencies, just as if they were never previously loaded. The replica and all of its dependencies will have unique virtual addresses, and all of their library constructors will be called. The linker keeps track of each replica, and the same `dlforce` function can be used to modify the behavior of other linker functions such as `dlsym` and `dlopen` to search only those libraries loaded from the given `dlforce` handle. This allows library code within a replica, or any of its dependencies, to use the dynamic loader normally, and essentially creates isolated trees of libraries.

4.8.1.1 EGL Extension: *multi_context*

Cycada uses dynamic library replication in the Android EGL open source library through a custom EGL extension named *EGL_CU_multi_context* and a supporting library, *libui_wrapper* (see Figure 4.3). This extension API, shown in Figure 4.4, adds four new EGL functions for creating and manipulating EGLContext objects that maintain isolated, unique GLES connections within the same process: `eglReInitializeMC`, `eglSwitchMC`, `eglGetTLSMC`, and `eglSetTLSMC`. The libui_wrapper library links against the vendor GLES and EGL libraries and encapsulates other Android system libraries which implicitly link against

```
EGLBoolean eglReInitializeMC(EGLNativeDisplayType display,
                             EGLDisplay *dpy,
                             EGLint *major, EGLint *minor);
EGLBoolean eglSwitchMC(EGLContext new_ctx, EGLContext old_ctx);
EGLBoolean eglGetTLSMC(void **tls_vals, int nvals);
EGLBoolean eglSetTLSMC(void **tls_vals, int nvals);
```

Figure 4.4: Columbia EGL Extension: EGL_CU_multi_context

GLES or EGL. The `eglReInitializeMC` function creates a replica of the vendor EGL and GLES libraries. The `eglSwitchMC` functions allows a thread to select which replica, and thus which GLES connection, it will use by setting the thread's EGLContext object to the one contained within the replica.

Creating EGL and GLES replicas, through a modified Android open source EGL library, results in unique GLES connection management challenges related to TLS. The unmodified Android EGL library allows one EGL-to-GLES connection (*EGLConnection* object) per-process, and it stores this information in a library-static global variable. Creating replicas of the vendor EGL and GLES libraries allows multiple threads to use different EGLConnections concurrently. A single, global EGLConnection variable no longer suffices, so the *EGL_CU_multi_context* extension stores this per-thread EGLConnection object in the TLS. A common paradigm in GLES programming is to create a context on one thread (generally the main thread), and pass the context information to another thread which will perform rendering or texture loading functions. Because the *EGL_CU_multi_context* extension has moved the previously global GLES connection information into a thread-local variable, we require the ability to copy, or *migrate*, TLS values between threads. This is accomplished using the thread impersonation mechanism discussed in Section 4.7 through the `eglGetTLSMC` and `eglSetTLSMC` extension functions.

4.8.2 Unintended Consequences

Moving data from a presumed global location into thread-specific variables, and creating multiple copies of the same library can have unintended consequences that are not easily predicted. *Cycada*'s EAGL implementation relies on a custom library, *libEGLbridge*, described in Section 4.5, that provides targeted

Android functionality through a set of diplomats. In particular, this library uses Android GraphicBuffer objects which use the "gralloc" or "HW Composer" APIs to allocate memory. These APIs are provided in vendor-proprietary libraries that link against the same libraries used by the vendor-proprietary EGL and GLES libraries. In other words, libEGLbridge implicitly links against the vendor EGL and GLES libraries. The mechanism by which GraphicBuffer memory is shared between APIs such as GLES and EGL requires that the gralloc and HW composer APIs use the *same* GLES connection, established by EGL, that the GLES rendering functions use. In other words, a GraphicBuffer allocated by libEGLbridge using the first instantiation of the EGL and GLES libraries cannot be used by GLES functions in replicas created by multiple iOS EAGLContexts.

To avoid this morass of library dependencies, *Cycada* separates the libEGLbridge functionality into two pieces. The first piece, labeled in Figure 4.3 as "libEGLbridge," contains all the diplomats used by the iOS code, and carefully avoids linking against any libraries. The second piece, labeled "libui_wrapper," contains all of the actual logic that links against Android graphics libraries. Whenever a new EAGLContext object is created, a diplomat in libEGLbridge simultaneously creates a replica of the libui_wrapper library and the EGL and GLES libraries by using the prelude functionality of diplomats discussed in Section 4.3. In this way, the libui_wrapper functionality uses the same replica of GLES as the gralloc functions which allocate a GraphicBuffer.

4.9 Evaluation

We have extended the *Cycada* prototype, described in Section 2.7, to more completely bridge the differences between iOS and Android graphics subsystems. Our extended prototype runs widely used iOS apps on Android, including those that make extensive use of WebKit, such as the Apple-only apps Safari and iBooks, and third-party apps Yelp, Holy Bible, and Wikipedia. Support for other devices such as GPS and networking is also completely functional, so apps such as Yelp can be used to find local restaurants and other businesses. We present some experimental results that both demonstrate the feasibility of our approach and measure its performance using both application-level benchmarks and targeted micro-benchmarks. For our experiments, we used *Cycada* on a Nexus 7 tablet with Android 4.2.2, and compared its performance with an iPad mini running iOS 6.1.2.

To round-out our evaluation, in Section 4.9.3 we present a complete list of all applications that we

downloaded, decrypted, and tested on an Android device along with a short description of the current state of the app on *Cycada*. Finally, Section 4.9.4 describes the current state of the *Cycada* prototype including known bugs and unimplemented features.

4.9.1 iOS WebKit Functionality

We first ran the iOS Safari Web browser on *Cycada* to demonstrate its functionality. Safari is an excellent test app because it uses a wide range of complex graphics functionality via WebKit. While advanced graphics APIs, such as GLES, are often thought of exclusively in the context of games or third-party applications, GPUs and their associated APIs are also used to accelerate many different aspects of computation. For example, WebKit uses CoreImage, QuartzCore, CoreGraphics, and IOSurface libraries in iOS which together use GLES to accelerate image and graphics processing. Additionally, the deep vertical integration of iOS allows library designers, as opposed to third-party developers, to bypass standard GLES extension query mechanisms, and make simplifying assumptions about available GLES extensions. Our prototype supports these implicit assumptions by mapping missing extension functionality onto existing Android GLES functions. Thus, running an app such as Safari in *Cycada* requires a near-complete GLES bridge implementation.

We performed two sets of experiments. First, we used Safari to browse the main page of the top 30 websites in the US [1] and compared the visual results between Safari running on *Cycada* on a Nexus 7 tablet, and Safari running on an iPad mini. All of the top 30 websites rendered their content correctly and appeared visually similar to the respective content on the iPad mini. Second, we used Safari to run the most recent Acid3 test [129], a web test page from the Web Standards Project that checks browser compliance with web standards, including the Document Object Model (DOM) and JavaScript. Safari on *Cycada* passes the test, showing smooth animation, a score of 100/100, and having the final page look exactly, pixel for pixel, like the reference rendering.

4.9.2 Performance

We compared four different Android and iOS system configurations to measure the performance of *Cycada*: an iOS app running on *Cycada* (*Cycada* iOS), an Android app running on *Cycada* (*Cycada* Android), an iOS app running on iOS (iOS), and an Android app running on Android. We normalize our results to the Android app running on Android. A Nexus 7 tablet with Android 4.2.2 was used in all cases except for iOS

Null Syscall	
System	**Time (ns)**
Stock Android	225.2
Cycada Android	244.1
Cycada iOS	304.6
iPad mini iOS	575.1

Diplomatic Calls	
Function Call	**Time (ns)**
Standard Function	9
Diplomat	815.8
Diplomat +Pre/Post	827.7
Diplomat + GL Pre/Post	932.8

Table 4.4: Kernel-level / ABI Micro-Benchmarks

app on iOS which was run on an iPad mini. Both tablets were released around the same time frame and have a similar form factor, providing a useful point of comparison.

We first ran a simple set of micro benchmarks using the `lmbench` test suite to measure the raw overhead of using diplomats. A diplomat involves two system calls: one to switch the thread from the iOS persona to Android persona, and one to switch back. We first ran the null system call `lmbench` micro-benchmark which invokes system calls that perform no work within the kernel. Using *Cycada*, we then ran a custom micro-benchmark using the `lmbench` infrastructure that measures the time to invoke a standard iOS function, a diplomat with no prelude or postlude, a diplomat with an empty prelude and postlude, and a diplomat using the *Cycada* GLES prelude and postlude functions.

Table 4.4 shows the results of our kernel and ABI micro-benchmarks. For all tests run on the Nexus 7, the CPU was pinned to 1.3HGz. Unfortunately, there is no interface to pin the CPU frequency in iOS, so we used the lmbench framework in a best effort to "warm up" the caches, and ramp the CPU frequency to its maximum of 1GHz. The null syscall results on the four different Android and iOS system configurations show that *Cycada* adds approximately 8% overhead to an Android kernel trap and 35% to an iOS trap due to an unoptimized kernel entry path. It is interesting to note that the iPad mini had a significantly higher cost to trap into the kernel due primarily to the addition of protection logic guarding against return-to-user attacks [25]. The diplomatic call results on *Cycada* show that standard function calls are obviously much faster than system calls and diplomats. The additional prelude and postlude mechanisms introduced by *Cycada* add little overhead over to diplomats without them, and the fully functional GLES prelude and postlude cost roughly 100ns. A GLES diplomatic call costs approximately the same as three system calls.

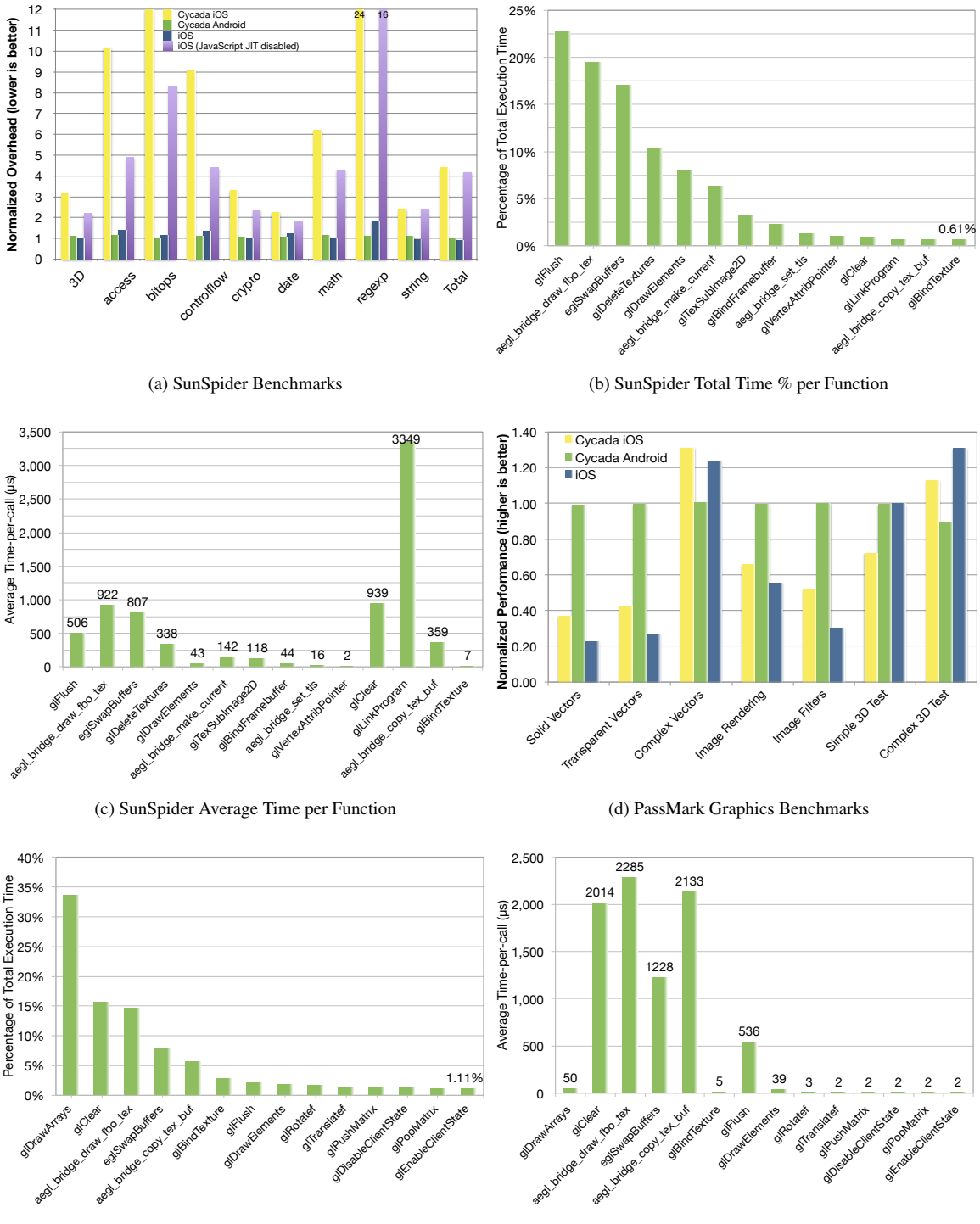
We then used two app benchmarks that could be run on both Android and iOS: a web browser running

SunSpider [19], and the PassMark [105; 106] app. SunSpider is a widely-used JavaScript benchmark that stresses many aspects of the browser's JavaScript engine including bit operations, cryptography, raytracing, JSON input, and pure math. We ran SunSpider using Safari on *Cycada* and iOS, and Chrome, the default Android browser, on *Cycada* and Android. PassMark is a freely available, cross-platform benchmark suite, and we used its 2D and 3D tests to measure graphics performance. We ran the iOS PassMark app on *Cycada* and iOS, and the Android PassMark app on *Cycada* and Android.

Figure 4.5a shows the SunSpider latency measurements normalized to the performance of the stock Android browser on Android. Lower numbers are better. The Android browser on *Cycada* and Safari on iOS perform similar to the stock Android browser on Android. However, Safari on *Cycada* is more than four times slower overall, and over ten times slower for "access" and "bitops" tests. The majority of this slow down results from a lack of Just-In-Time (JIT) compilation of JavaScript on *Cycada* due to a Mach VM memory bug in our prototype that prevents JIT from working properly. For comparison, we also disabled JIT for JavaScript on iOS, and we present the results, normalized to iOS performance, as the purple bar in Figure 4.5a. Relative to standard iOS performance, disabling JIT results in a 4.2x slowdown on iOS. This is approximately equal to the 4.4x slowdown on *Cycada* with the additional overhead resulting from our unoptimized system call path (see Table 4.4). The high overhead of running SunSpider without JIT, especially in the "regex" test, generally falls in line with measurements done by the WebKit team on the first introduction of ARM JIT [130], and subsequent speedups introduced by the data flow graph (DFG) JIT [53]. [4]

Figure 4.5b shows a breakdown of the Android GLES functions called while running SunSpider in Safari on *Cycada*. Note that the SunSpider test itself does not invoke graphics functions, rather the WebKit framework uses OpenGL ES to render the resulting dynamic HTML output. We show the percentage of time consumed by each GLES function relative to the total time consumed by all GLES functions, with the functions ordered in descending order based on how much total time they consume in running the benchmark. We show the top 14 functions, which consume over 90% of the total time. Function names starting with *gl* correspond to direct, indirect, or data-dependent diplomats to Android GLES functions, names start-

[4]The Fourth Tier LLVM JIT optimizations referenced in the WebKit FTL JIT article [53] were not implemented in the version of WebKit used in our measurements, however the article presents DFG JIT performance versus baseline JIT which indicates a greater than 3x speedup versus baseline and 30x speedup versus non-JIT on at least one representative benchmark.



(a) SunSpider Benchmarks

(b) SunSpider Total Time % per Function

(c) SunSpider Average Time per Function

(d) PassMark Graphics Benchmarks

(e) PassMark Total Time % per Function

(f) PassMark Average Time per Function

Figure 4.5: *Cycada* Application Benchmarks

ing with *egl* correspond to multi diplomats to Android EGL functions, and names starting with *aegl* are custom multi diplomats, located in libEGLbridge, supporting the *Cycada* EAGL implementation (see Section 4.5 and Figure 4.3). Approximately 40% of the graphics-related time is spent in EAGL implementation related functions such as `aegl_bridge_draw_fbo_tex`. Clearly there is room for improvement in our unoptimized prototype.

Figure 4.5c shows average execution time per function call for the same GLES functions in Figure 4.5b. Of the top 14 functions, only 1 costs less than 10μs, and the most time consuming functions take, on average, over 300μs. Since Table 4.4 shows the cost of a diplomatic call is less than 1μs, the overhead due to diplomats is small for most GLES functions, and certainly small for GLES functions that account for most of the GLES execution time in running SunSpider in Safari. The dominant cost of *Cycada* is in the actual graphics logic required to bridge between iOS and Android, not in the diplomat mechanism.

Figure 4.5d shows the PassMark 2D and 3D graphics measurements normalized to the performance of the stock Android app on Android. Higher numbers are better. This measurement generally matches the PassMark graphics measurements first reported in Section 3.5.3. However, through the enhanced graphics support described in this chapter, and some preliminary optimizations of our prototype, *Cycada* now outperforms Android in the GPU-intensive complex 3D test by more than 20% running on the same Nexus 7 tablet. Some of this performance increase could also be attributed to slight variations in how the 3D scenes were rendered on each platform, or differences in the exact OpenGL ES calls made on either platform due to differences in supported GLES extensions.

Similar to results found in Section 3.5.3, all the PassMark graphics measurements show that *Cycada* iOS performance relative to Android is highly correlated to iOS performance relative to Android, even though *Cycada* is running on the Nexus 7 while iOS is running on the iPad mini. For the 2D tests in which stock iOS does significantly worse than stock Android, *Cycada* iOS also does significantly worse than *Cycada* Android. In the complex vectors and 3D tests in which stock iOS does noticeably better than stock Android, *Cycada* iOS also does noticeably better. The reason for this is that both *Cycada* and iOS use the same frameworks and libraries, which in some cases have better performance than Android and in some cases are worse. Comparing *Cycada* and iOS, we see that *Cycada* performs better than iOS on the 2D tests and worse on the 3D tests. Because the 2D tests make heavier use of the CPU, *Cycada* outperforms iOS because it uses the faster Nexus 7 CPU versus the slower iPad mini CPU. Because the 3D tests are highly GPU intensive, they provide a better indicator of the graphics overhead of *Cycada* compared to iOS. The overhead is higher

on the simple 3D test because that test is designed to maximize frame-rate and thus stresses our unoptimized EAGL implementation which is responsible for moving rendered scenes onto the display. The complex 3D tests involve more processing intensive GPU functions to render complex scenery, so since the GLES calls used are more expensive, the overhead of *Cycada* is less.

Figures 4.5e and 4.5f show the percentage of total GLES execution time and average execution time per GLES function, ordered from the function with the largest to the smallest total execution time. We show the top 14 functions, which consume over 90% of the total time. The two most heavily used GLES functions are `glDrawArrays`, which draws an array of vertices, and `glClear`, which clears the framebuffer. These are both standard GLES functions, heavily used by the simple and complex 3D tests, called via direct diplomats. Based on their average execution time shown in Figure 4.5f relative to the cost of diplomats shown in Table 4.4, diplomat overhead is small. The primary overhead of *Cycada* is due to functions such as `aegl_bridge_draw_fbo_tex` and `aegl_bridge_copy_tex_buf`, which consume roughly 20% of the GLES execution time. These functions correspond to a highly optimized hardware supported path in iOS on the iPad mini.

4.9.3 iOS Application List

We used many different real-world iOS applications to further test, debug, and benchmark *Cycada*. A complete list of these applications can be found in Table 4.5. Note that each of these apps was downloaded from the App Store, decrypted on an iOS device (described in Chapter 3 Section 3.5.1), and run on the *Cycada* prototype. Table 4.5 lists the application name, the App Store category where it was found, and some brief notes on how well the app performed while running on *Cycada*. Note that apps in the *iPhone System App* and *iPad System App* category are applications that ship with iOS on an iPhone and iPad respectively. These applications did not need to be decrypted, they were simply copied directly from a jailbroken iOS device. In addition to full GUI applications, we also tested numerous command line utilities such as `ioreg`, `bash`, and `openssh`. Nearly all command line utilities we ran worked without issue on *Cycada*.

Table 4.5: iOS applications tested under *Cycada*

	App Name	Category	Status
1	3D Benchmark	Utilities	Fully functional
2	Apple Remote	Entertainment	Fully functional
3	Calculator	iPhone System App	Fully functional
4	Calculator Pro	Utilities	Fully functional
5	Calendar	iPad System App	Fully functional
6	Clock	iPhone System App	Fully functional
7	Constitution	Reference	Fully functional
8	Holy Bible	Reference	Fully functional
9	iBenchmark	Utilities	Fully functional
10	Keynote Remote	Utilities	Fully functional
11	Mobile Mouse	Utilities	Fully functional
12	My Apps Lite	Business	Fully functional
13	Papers	Productivity	Fully functional
14	PerformanceTest	Utilities	Fully functional
15	RowmotePro	Utilities	Fully functional
16	Stocks	iPhone System App	Fully functional
17	SystemStatusLite	Utilities	Fully functional
18	UrbanDictionary	Reference	Fully functional
19	Yelp	Travel	Fully functional
20	AnTuTu	Utilities	Works, but GPU test crashes
21	Dark Sky	Weather	Works, but WebView overlay occludes UI
22	Google Translate	Reference	Works, but no audio
23	iBooks	Reference	Works, but can't read encrypted (purchased) content
24	iReddit	News	Works, but can only load a single web page view
25	Padgram	Utilities	Works, but images fail to render in multiple WebView contexts
26	QuranMajeedLite	Reference	Works, but no audio
27	Safari	iPad System App	Works, but has some graphics glitches
28	TextPics	Social Networking	Works, but no copy/paste
29	Weather	iPhone System App	Works when emulating an iPhone
30	App Store	iPad System App	Loads with errors (missing daemons [5])
31	CNN app for iPhone	News	ACAccountStore/XPC error
32	Dropbox	Productivity	Partially works: current version uses hardcoded screen size, (no keyboard access)
33	Google Chrome	Utilities	Interface works, but won't fetch url (after pressing enter)
34	iTunes	iPad System App	Loads with errors (missing daemons [6])

Continued on next page

[5] App 30: missing support for daemons: `itunesstored`

[6] App 34: missing support for daemons: `itunesstored`

Table 4.5 – continued from previous page

	App Name	Category	Status
35	Keynote	Productivity	Initial UI works, requires iCloud interaction
36	Mail	iPad System App	Interface works: says iCloud password is not provided
37	Maps (Apple)	iPad System App	Loads, but no map images on screen (MapKit)
38	Messages	iPad System App	Interface works, but cannot send an iMessage (missing daemons [7])
39	Music	iPad System App	Interface works (missing daemons [8])
40	Nook	Books	Loads
41	WhatsApp	Social Networking	"Your device is not supported:" (relies on cellular plan)
42	Wikipedia	Reference	Loads, but unable to fetch new pages
43	YouTube	iPad System App	Interface works, cannot connect to youtube (SSL errors)
44	Amazon	Lifestyle	Displays interface then crashes with SSL errors
45	Angry Birds: Star wars	Games	Loads, but crashes
46	Bible	Reference	Loads, but eventually hangs
47	candycrushsaga	Games	Loads, but hangs
48	Clash of Clans	Games	Loads, then hangs
49	Craigslist	News	Does not load
50	Dictionary.com	Reference	Loads, but hangs
51	Dolphin Browser	Productivity	Does not load
52	EAT24	Food & Drink	Running process, but no window
53	eBay	Lifestyle	Does not load
54	Facebook	Social Networking	Running process, but no window
55	Facebook Messenger	Social Networking	Running process, but no window
56	Flashlight	Utilities	Running process, but no window
57	Geekbench	Utilities	Running process, but no window
58	Gmail	Productivity	Shows an empty dialog then crashes
59	Google Search	Reference	Loads, then crashes
60	Groupon	Lifestyle	Loads, then crashes with unhandled exception
61	iTunes U	Education	Running process, but no window
62	Kindle	Books	Loads main screen
63	LinkedIn	Social Networking	Loads, but freezes after a couple of seconds
64	Merriam Webster Dict	Reference	Loads, but crashes
65	Notes	iPad System App	Loads, but hangs after a few seconds
66	PayPal	Finance	Running process, but no window
67	Petting Zoo	Games	Loads, but crashes
68	Pokedex	Reference	Does not load
69	Starbucks	Food & Drink	Running process, but no window

[7] App 38: missing support for daemons: `apsd`

[8] App 39: missing support for daemons: `itunesdaemon`

In total, 29 apps are either fully functional or work well-enough to be usable with minor bugs; 14 apps load and are interactive, but require services from currently unsupported libraries or daemons such as MapKit or `itunesstored`; 26 apps either fail to load, crash or are otherwise unresponsive / unusable. The iOS libraries, such as MapKit, which are currently unsupported in *Cycada* should only require engineering effort to find and implement missing pieces of compatibility.

Many of the 28 non-functional applications tested on *Cycada* either hang or fail to load / work properly due to missing or non-functional daemons or system services. Most of these daemons or system services are not supported simply due to lack of time. Most of the work to support these daemons would fall into two categories. First, daemons such as `CommCenter` communicate to proprietary hardware such as the telephony, or cellular baseband, device. These devices (and daemons which use them) could be supported using methods similar to those described in this chapter. Although not all of these devices have a standardized interface, they at least have a documented interface used by the daemons. This should be sufficient to construct diplomatic support libraries on *Cycada* which map functionality onto Android or Linux devices.

Second, many daemons and libraries utilize device-specific encryption or certificate based authentication that communicates to Apple servers using previously *activated* SSL certificates. Decrypting existing content, such as iTunes purchases, requires using keys that are generated locally on an Apple devices and stored in a secure location in Apple hardware. The initial entropy for these keys comes from device-specific parameters such as the BlueTooth MAC, the WiFi MAC, the IMEI, and other device-specific values. Much of this data can be copied from a single Apple device, but supporting a generic mechanism to decrypt this content on any device is cryptographically impossible. Similarly, certificates used in certain SSL transactions with Apple computers requires device-specific entropy that's been previously provided to Apple. Given enough implementation time, these services could be made to work, but they could, at best, only emulate a single, existing, previously activated, Apple device. Reverse engineering the activation protocols to generate new device-specific keys and certificates is left to future work.

Finally, there are also many daemons that communicate with other daemons that require support in one or both of the above categories. While these daemons may not have any additional compatibility requirements, they still block application progress because they require other unsupported daemons.

4.9.4 Prototype: Current State

The *Cycada* prototype has been demonstrated numerous times, including in a YouTube video [9]. Although the demonstration shows several useful and interesting iOS apps running on a Nexus 7 tablet, there are pieces of the prototype which are incomplete, buggy, or missing.

There is a known bug in the duct tape implementation of XNU thread call APIs which uses Linux delayed workqueues. In our current prototype, there is a race condition when adding an item to a workqueue which may already be on an existing queue. Careful examination of the locking used in the Duct Tape function `__thread_call_enter` (the core of the thread call API implementation) should resolve this issue.

The networking support in the *Cycada* prototype properly supports most of the BSD sockets API, however there are some types of networking operations performed by iOS frameworks and applications which are not currently supported. The two primary pieces of iOS networking which are unsupported are: traffic classes, set through the `SO_TRAFFIC_CLASS` socket option, and XNU routing sockets. Traffic classes are useful for applying different QoS policies to different types of network traffic. While this should not affect correctness, it could affect network performance. Routing sockets are not used extensively in iOS 5.x, but they are used in iOS 6. The *Cycada* prototype was able to demonstrate near complete networking support on iOS 5.1, but in order to properly support iOS 6 and beyond, full routing socket support will be necessary.

Cycada supports most iOS OpenGL ES APIs through techniques described in this chapter, however there are pieces of this API which are not supported due to lack of time, and, as Section 4.9.2 shows, there is a sub-optimal mapping of some basic buffer and window management iOS APIs. A prime example of missing OpenGL ES support in *Cycada* is Vertex Array Objects [31; 21] (VAOs). Unfortunately, the OpenGL ES version and graphics hardware in the *Cycada* prototype do not support VAOs. To support this feature, *Cycada* would need to either manually implement this functionality on top of Vertex Buffer Objects [74], or port the prototype to a different platform which supports this feature. Vertex Array Objects are used extensively in the `OGRE` [124] framework which is a cross-platform 3D graphics library used by many games and benchmarks.

The iOS EAGL library uses proprietary interfaces to a closed-source device driver to optimize transfer of final graphics memory buffers to the GPU and display. Unfortunately, due to an imperfect reverse-engineering of the EAGL API and a sub-optimal buffer-swapping implementation, performance of the *Cy-*

[9] `https://www.youtube.com/watch?v=Uaple0Ec1Dg`

cada prototype demonstrates high overhead as seen in Section 4.9.2. More specifically, without the help of a proprietary interface for zero-copy transfer of off-screen render buffers onto the screen, *Cycada* must use a combination of Android OpenGL ES and EGL functions to accomplish the same task. This adds significant overhead in both performance and memory use - at least one extra memory copy of each frame is performed.

Finally, Just-In-Time (JIT) compilation of JavaScript in iOS applications does not work in the *Cycada* prototype. Memory allocation and permissions setup by the `JavaScriptCore` iOS library have been verified, and the library successfully generates code into the allocated memory. However, the iOS application crashes soon after jumping into the JIT compiled JavaScript code. The root cause of this failure is currently unknown, although two vectors of investigation have been identified. First, cache flushing and interaction between icache and dcache could be the cause of the failure, and the efficacy of the cache flushing mechanism mapped to the iOS cache flush operation should be investigated. Second, the output of the JIT compiler could be encrypted, hashed, or otherwise code-signed by the iOS library under the assumption that the XNU kernel can decrypt, unhash, or verify the executable memory segment. The *Cycada* prototype assumes unencrypted binaries (see Section 3.5.1), and would be unable to decrypt dynamically generated encrypted code.

4.10 Related Work

Section 3.7 discusses many different approaches to run applications from multiple OSes on the same hardware [2; 55; 48; 49; 50; 39; 67; 92; 101; 99; 88; 68; 110; 29; 46]. However, as previously discussed in Chapter 3, graphics support remains a key challenge. Various approaches rely on the assumed ubiquity of X Windows, and assume that apps simply conform to the X Windows standard. This is insufficient to support GPU-intensive apps that rely on a wide range of graphics support, and is not applicable to the vertically integrated, proprietary graphics stacks common on mobile platforms in lieu of X. Wine, a Windows API reimplementation project for Linux, provides incomplete support for Microsoft's advanced graphics API, DirectX, on top of Linux using desktop-class OpenGL functions. This is in contrast to *Cycada* which leverages existing iOS frameworks and libraries instead of requiring a massive reimplementation effort.

Desktop virtualization solutions have developed several solutions that allow guest VMs to use GPU acceleration through mediated access to host hardware resources and libraries. These solutions can be grouped into four basic categories: API remoting or forwarding, device emulation, split-mode or mediated

pass-through drivers, and direct pass-through. Desktop or server API remoting [116] ("indirect rendering") solutions, such as VirtualGL [122], rely on desktop windowing protocols such as GLX [139] to stream OpenGL commands to a remote server for execution. Mobile OSes such as iOS discard abstractions such as GLX in favor of custom APIs targeted for mobile usage patterns and low-latency direct hardware communication. Similarly, API forwarding solutions rely on being able to forward API calls from one platform to another because they share the same API. However, mobile OSes such as iOS use some standards, but not others, and the ones they use, they may extend, such that there is no longer a complete, shared API to forward from iOS to Android, making API forwarding or remoting problematic.

Device emulation is only used for simple, 2D hardware [30] due to the massive complexity of GPU hardware. The approach is even more problematic on resource constrained mobile platforms such as iOS with proprietary interfaces that would be difficult to reverse engineer and emulate.

Split-mode, or mediated pass-through, solutions such as VMGL [83], VMware's vGPU [118; 47], and XenGT [123], take a hybrid approach of forwarding some aspects of the guest API, direct mapping some host GPU resources, and emulating other aspects of a graphics driver. The difficulty of emulation depends in part on the API supported. For example, VMware's approach focuses on Direct3D, which requires apps to perform their own graphics resource management and lacks the additional challenges of support GLES which requires EGL, or in the case of iOS, a proprietary EAGL.

Direct pass-through solutions, such as NVIDIA GRID [95], leverage traditional hardware virtualization and require specialized hardware only recently available in desktop-class GPUs [94]. They rely on hardware support not available in mobile platforms, and do not address the problems with providing graphics device support in the context of mapping vertically integrated graphics stacks such as iOS to other platforms.

Although none of these previous approaches by themselves can support graphics device support across mobile platforms such as iOS and Android, many facets of *Cycada* may be adapted and applied in the context of these other approaches, for example to enable mobile graphics virtualization. For example, API forwarding could leverage some of the mechanisms introduced by *Cycada* to provide similar graphics support for VMs. The primary difference in applying these techniques would be some of the performance costs. For example, a hosted mobile graphics virtualization solution based on KVM for ARM [44] would require a world-switch that at bare minimum costs 5000 cycles, in contrast to using diplomats to switch between thread personas from iOS to Android at the much lower cost of a couple of system calls.

4.11 Conclusions

We have presented a graphics-focused study of *Cycada*, and extended its binary compatible graphics support for running iOS apps on Android through three new OS compatibility mechanisms: **(1)** extended diplomat construction and new diplomat usage patterns, direct, indirect, data-dependent, and multi, **(2)** thread impersonation which allows one thread to temporarily assume the persona of another thread, and **(3)** dynamic library replication which allows the linker to create separate loaded instances of a dynamic library. These mechanisms proved essential to support our prototype in running widely used iOS apps, many of which utilize WebKit and other frameworks that rely heavily on the GPU. We discussed our experiences with these mechanisms in the context of GPU device support, and demonstrated their feasibility by using iOS's Safari on *Cycada* to visit popular websites and run browser benchmarks.

Chapter 5

Cycada: Experiences and Lessons Learned

Building a binary compatibility solution able to run apps extracted from one of the world's most closely guarded mobile operating systems is a massive and ambitious undertaking. The paradigms, techniques, and methodologies described in Chapters 3 and 4 construct a viable framework within which a project such as *Cycada* can be developed, however, real world implementation of this project embodied an intricate web of binary-level reverse engineering, humorously simple solutions, tedious hours of bit manipulation, and obstinate debugging sessions worthy of a Greek hero. This chapter recounts our experiences building *Cycada* through observation and anecdote, and identifies some key lessons learned along the way.

The process of running iOS apps on Android devices began by investigating a simple static binary compiled on an iMac using the iOS toolchain and debugging facilities provided by Apple through their *Xcode* IDE [24]. These tools are intended for use by Application developers, and include a full-featured debugger with GUI integration and a complete set of binary utilities for manipulating and investigating object files. So, with very little effort we built a working iOS app, ran it on an iPhone, and introspected it. With only marginally more effort, we extended this support to command-line, statically linked iOS programs: something not supported by default in Xcode, but only involved discovering the proper arguments to pass to `gcc`/`clang` in a simple `Makefile`.

Xcode supports iOS development through an SDK which ships with the IDE. The iOS SDK that ships with Xcode also contains all the source headers and object files necessary to build iOS apps. Interestingly, the SDK contains full copies of nearly all the system libraries used on an iPhone or iPad. Although the actual device uses the libraries in a pre-linked shared cache, the object files provided in the SDK contain

the exact same compiled code which can be investigated by the iOS toolchain and binary utilities. In other words, Apple actually ships fully-functional ARM binaries for almost all iOS libraries.

In addition to the libraries and framworks within the iOS SDK, Apple also releases the source code for many libraries and daemons, as well as the XNU kernel, on the `http://opensource.apple.com/` website. However, the released source code has been at least partially sanitized of all iOS, ARM, or embedded platform specific bits. For example, the XNU source released by Apple contains no platform support for any iOS device, but it does compile for the x86_64 platform. We were able to leverage some of the platform agnostic pieces of kernel source, as well as the sources of several other libraries. Section 3.3.2 describes how we took large portions of XNU source and compiled them into the Linux kernel. Outside the kernel, we were also able to use projects such as *libdispatch* where we could compile a working iOS version of the library and enable or disable features. Unfortunately, key pieces of code such as the ARM XNU system call binary interface were missing.

Although the combination of open source code and working ARM libraries gave *Cycada* a jump start, none of the nice iOS debugging tools and command line utilities shipped with Xcode work on Android devices. Debugging a process, using a tool such as `gdb`, requires intimate coordination with the kernel and low-level system libraries. You can't use the iOS debugger to diagnose what went wrong when you tried to run that same debugger on top of a Linux kernel, and the Android debugger only knows about Android/Linux programs.

So, it was without a debugger, but with bits of source code and some working iOS libraries, that we attempted to run our first statically linked iOS binary on an Android device. Of course, it failed miserably. After many long hours of pouring over binary formats, ARM assembly, sparse documentation, and broken tools, we were able to see the words "Hello World" printed by an iOS binary running on an Android device. It was only the beginning.

With humble deference to Lampson's hints for system design [84], we offer up several hints, or guideposts, for the reverse engineer, the binary compatibility hacker who may consider an endeavor similar to our own. These guideposts will not exhaustively light the way to a complete binary compatibility solution, nor will they necessarily apply more generally outside the scope of *Cycada*. As their name implies, they construe a boundary to a particular problem space. They do not mark a path through that space.

We have organized these guideposts along three axes, or three sets of counter-balanced ideals: the axis of work, the axis of tools, and the axis of human nature. On one side of each axis lie the three great virtues of

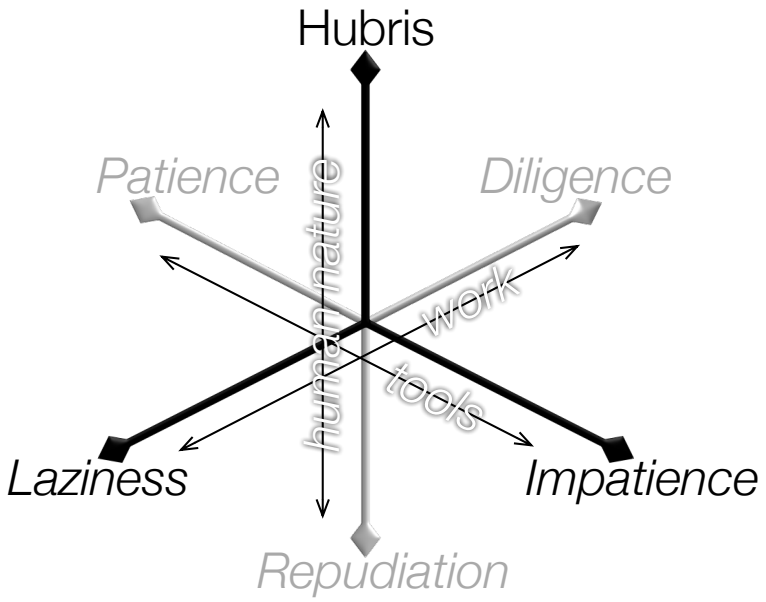


Figure 5.1: Axes of Reverse Engineering

a programmer, *laziness*, *impatience*, and *hubris*, first described by Larry Wall [128]. These virtues provide clarity and insight to the process of mapping one complex system onto another. Regrettably, virtue alone could not run even a single, statically linked iOS binary on an Android device. No matter how much code we reused, or how much work our tools performed for us, or how much leverage we gained from the collective hubris of iOS developers, we found ourselves falling into what can only be described as vice. On the other side of each axis lie the counter-balancing ideals of *diligence*, *patience*, and *repudiation*. We would often require *diligence* to complete a brute-force implementation, or work through bugs arising from implicit assumptions made about iOS hardware. A complete lack of tools required *patience* to carefully construct an environment robust enough to debug an iOS binary on the Linux kernel. Finally, under extreme duress, the vice-filled reverse engineer must *repudiate* first principles; they must know when a bit of duct tape, or a little white lie will do the job better than a well-engineered, maintainable interface. Figure 5.1 depicts each of

Axis of Work

Laziness	*Diligence*
Force/reuse existing code paths:	Standardized API tokens, different binary representation
avoid the "commpage" and dyld shared cache	Subtle API variations
Don't implement every system call	User / kernel boundary differences: the system call return path
	Implicit assumptions

Axis of Tools

Impatience	*Patience*
IDA scripts for automated binary analysis	Debugging:
SDK-provided analysis tools and custom shell scripts	in-kernel signal handlers for user space debugging
Enable all the logs	custom log formatting / parsing

Axis of Human Nature

Hubris	*Repudiation*
IOSurface reverse engineering	Ignore the unnecessary: code signing / sandboxing
GLES Extension mapping	Fake it: faux hardware names
	Close enough: just overlay system call variants
	Duct tape the code you don't want to write

Table 5.1: *Cycada* Development Examples

the three axes that map out the three dimensional space illuminated by our reverse engineering guideposts. Table 5.1 provides an overview of the various stories and experiences that fall under each guidepost.

We describe, in Section 5.1, the three axes of counter-balanced virtue and vice. Each virtue is described through example, occasionally in gory detail. The chapter concludes, in Section 5.2, with a discussion of porting *Cycada* to iOS 6 and beyond.

5.1 Virtue and Vice

The Axis of Work

The first great virtue of a programmer according to Wall, laziness, characterizes the amount and type of work performed by a programmer. Wall describes laziness as,

> *[t]he quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful, and document what you wrote so you don't have to answer so many questions about it. Hence, the first great virtue of a programmer.*

As in most other programming endeavors, when writing a binary compatibility system the reverse engineer should never write unnecessary code, and should never solve a problem that doesn't really need solving. Unlike other endeavors, a binary compatibility system maps one large body of code onto another large body of code; the less new code introduced in this mapping the better. Thus, when presented with two possible solutions, the reverse engineer should always choose the one that involves less code (or no code at all).

The virtue of laziness, however, does not completely cover the work a reverse engineer must do to build a binary compatibility solution. Many problems or situations require a *diligent*, brute force approach. Often this is either the only way to accomplish your goal, or it's actually faster than planning a more elegant, but eminently more complicated solution. Being in opposition to a virtue, one must call this vice.

5.1.1 Laziness

When building a binary compatibility solution, Wall's notion of laziness can be extended to encompass all of the foreign binaries being made to run on the host system. This means that not only should you strive to reuse and document code that you write, it also means that you should look at the library or application you're trying to run to see if it already does what you need it to do. For example, the iOS dynamic linker, `dyld` [9], and `Libc` library use variants of certain functions such as `gettimeofday` which use data exported to user space in a shared memory region called the *commpage*. Exporting this dynamic data to iOS user space from the Linux kernel would involve a time-consuming reverse engineering effort as the format and exact bits of the iOS ARM commpage are not released. By inspection of the disassembly, we found that each of the functions in question had a "slow" variant that called down into the kernel directly to obtain

the necessary information. By modifying the TEXT segment of the `dyld` and `Libc` binaries, we forced the code to invoke the non-commpage function variants thus reusing code that already existed in the iOS binaries.

Another example of lazily reusing existing iOS code to solve a harder problem involves the iOS `dyld` shared cache. In iOS and OS X, Apple uses a single-file representation for most system libraries. This file, the *shared cache*, is an optimized concatenation of all constituent libraries. When the user space linker, `dyld`, loads a library, either through explicit or implicit dependencies marked in the Mach-O binary format, it first looks to see if that library exists in the shared cache. If so, the linker simply uses a pre-bound version of the library from within the cache instead of opening, mapping, and binding the on-disk file. The shared cache is mapped once for each process in the system resulting in a significant reduction in startup time, but a significant increase in the complexity of the kernel loader necessary to support *Cycada*.

Through careful reading of the `dyld` documentation, and cross-referencing with the disassembled binary, we noticed that the binary also supported manually walking the file system to search for a given library. *Cycada* uses environment variables and a small binary hack to `dyld` to force the linker to look for libraries in the file system. While this avoids having to implement complicated support for shared caches, application load time on *Cycada* suffers from having to locate, open, map, and bind every shared library. This is shown in more detail in Figure 3.5 and Chapter 3, Section 3.5.2.

In addition to reusing iOS code, *Cycada* also took a lazy approach to system call implementation. During the initial development of *Cycada*, only system calls that were actually called during program execution were implemented. This means that instead of laboriously pouring over the specification of each system call and iteratively implementing each one, we simply put stubs, or place-holder functions, in every system call slot which logged an error message whenever they were invoked. In this way, we focused our limited programming resources on the exact bits of compatibility work that needed to be done. When a system call was invoked, its functionality was determined, and a concerted effort was made to reuse as much existing Linux functionality as possible. This allowed us to further reduce implementation time and effort.

5.1.2 Diligence

Not all binary compatibility problems can be solved with lazily written, well documented, reusable code. In fact, many problems either require a brute force approach, or custom, narrowly–scoped, non-reusable

code written for the specifics of the given problem. In *Cycada*, several problems of this nature arose from differences in interpretation of a standard, or minor variations on well defined interfaces. For example, the POSIX standardized flag to the `open` call which should create a file if it doesn't exist is `O_CREAT`. While POSIX specifies this token name, it does not specify its underlying binary representation. In other words, POSIX defines source-level compatibility, not a binary compatibility. In XNU, Apple's iOS kernel, the `O_CREAT` flag is designated by the bit, `0x200`, while in Linux this same token is designated by the bit, `0x100`. Although both OSes are mostly POSIX compliant, they are not binary compatible. To support these variations in implementation, *Cycada* uses a set of translation tables and inline functions which map iOS binary tokens and structures to Linux tokens and structures. This work was, by far, the most tedious and painstaking part of *Cycada*.

In addition to differences in the binary representation of standardized tokens, both Linux and iOS often contain minor variations on a well-defined interface. These differences must be bridged either through a brute force implementation of the iOS variation, or through emulation using existing mechanisms. For example, iOS uses a networking stack derived from the FreeBSD kernel. Thus, it supports a socket option named `SO_REUSEPORT` which allows two sockets to bind to exactly the same IP:PORT combination. While Android's Linux kernel uses the BSD socket interface, support for `SO_REUSEPORT` was not present until Linux kernel v3.9, and very few Android devices support a kernel version that new. To support this iOS networking feature, used by several daemons, we manually back-ported the `SO_REUSEPORT` support found in Linux 3.9 to the kernel versions that support the Android devices used for running *Cycada*.

A brute force implementation was also necessary to support the iOS syscall return path. In iOS, the return from a syscall is always a 64-bit value, even on 32-bit architectures. In Android, the syscall return path is a single register which is 32-bits or 64-bits depending on the platform. On 32-bit platforms this means that we only return half the number of bits an iOS syscall expects. The Android syscall entry and exit paths comprise a collection of hand-coded, heavily optimized assembly functions. The return value from a syscall handler is, by default, stored in a single register which is carefully preserved by the exit path assembly as user space state is restored. To support 64-bit iOS syscall return values on 32-bit Android devices in *Cycada*, we refactored all of the hand-coded assembly functions invoked on the syscall exit path to preserve two register return values instead of just one.

Another fascinating problem space that required non-reusable, often tedious and brute force, solutions involves assumptions made by user space code. A user space environment assumes many different things

about its underlying kernel, and some of these assumptions can foil even the most carefully crafted compatibility interfaces. Occasionally these assumptions are documented, but many times they are not. Undocumented implicit assumptions about libraries, frameworks, or the underlying kernel can cause very subtle failures in seemingly disparate parts of the system. Here are several of the most interesting implicit assumption bugs that we encountered while developing *Cycada*.

First, the Mach timer interface was straightforward to implement using Linux kernel timers (and the duct taped Mach IPC code). For the majority of the time, this simplistic solution worked. However, there is an implicit assumption in the CoreFoundation library, which uses Mach timers as part of the `CFRunLoop` implementation, that a timer will *never* fire early. Within the CoreFoundation code that handles timer events, a timer that appears to fire early is treated an invalid, the error is silently squashed, and the timer is never re-armed. Mach timers in XNU are designed never to fire before their specified deadline or timeout. Linux timers, on the other hand, are by default sloppy, and are free to fire before the specified deadline or timeout period has completely expired. In iOS user space, a timer that fires before its anticipated deadline is treated as invalid, disregarded, and never rearmed *or serviced*. In *Cycada*, this resulted in applications which appeared to hang at seemingly random points in their execution because one or more timers had stopped firing. *Cycada* handles this implicit assumption by wrapping the Linux kernel timer API. If a Linux kernel timer fires early, it is re-armed for the amount of remaining time (or shortest possible timer duration). Thus *Cycada* ensures that a Mach timer event will never be generated until after the specified deadline.

Another interesting implicit assumption was made by code that called the `getattrlist` syscall to obtain extended attributes of a file or directory. Initially, *Cycada* simply returned an error indicating that certain attributes were unsupported. However, the user space code assumed that this call would always succeed, and proceeded to use the invalid contents of an uninitialized file system attribute structure. This lead to `SEGFAULT`s or stack corruptions at various points in the code depending on exactly where the invalid attribute had been propagated. The only solution to this problem was to do a brute force implementation of the missing attributes, or return dummy values for attributes that simply don't exist on the Android file system.

The BSD socket API, implemented by both iOS and Android, allows a server process to create a non-blocking socket file descriptor, and then invoke the `accept` socket API call on it. If there are no connections ready to be accepted, both systems return `EWOULDBLOCK` (or `EAGAIN`). However, on iOS (and on some BSD variants) if a client connection is successfully created and returned from an `accept` call

made on a non-blocking socket, the resulting file descriptor is *also* marked as non-blocking. In other words, the `O_NONBLOCK` socket option on the server `accept` socket is *inherited* by the client socket. This is not the case in Linux. User space iOS code assumed the resulting file descriptor was non-blocking and proceeded to perform otherwise blocking calls such as `read` on the client descriptor expecting them to return `EWOULDBLOCK`. *Cycada* need to manually propagate the non-blocking state from the `accept` file descriptor to client file descriptor.

Finally, assumptions in iOS user space about how the XNU scheduler would select threads and processes to run would occasionally lead to busy loops and deadlocks in user space. In iOS, many locking primitives rely on tight integration with the XNU scheduler which can re-prioritize threads holding a resource when a high-priority thread requests it. Under this assumption, it is safe for a high priority thread to repeatedly call syscalls such as `sched_yield` or `switch_pri` in the slow (or contended) path of iOS user space locking primitives. The *Cycada* mapping of these CPU yielding functions called Linux near-equivalent functions, but the Android Linux scheduler has no integration with locking primitives. Occasionally, a high-priority iOS thread would begin a tight loop in user space making repeated calls to `sched_yield` or `switch_pri` expecting the scheduler to push directly on the thread holding the contended resource. Unfortunately, the Linux scheduler would continue to pick the high priority thread to run impeding further user space progress. We attempted to mitigate these scheduling differences by either forcing the current thread to sleep, or attempting to direclty yield to another runnable thread.

The Axis of Tools

The second great virtue of a programmer, impatience, describes a programmer's desire and need for great tools. Impatience arises from,

> *[t]he anger you feel when the computer is being lazy. This makes you write programs that don't just react to your needs, but actually anticipate them. Or at least pretend to. Hence, the second great virtue of a programmer.*

Perhaps even more than the average programmer, a reverse engineer must leverage great tools to accomplish great tasks. There are amazing binary introspection and debugging tools available, and a successful binary compatibility system requires extensive use of these tools. There will be times when writing an IDA [64] python module to tease out an interface from the disassembly of a proprietary library will be the only way to continue.

Other times, however, will require custom tools or ad-hoc techniques. In all likelihood, the reverse engineer must chart unknown waters, going where debuggers and integrated development environments have never gone before. For this work, the vice of patience sustains the programmer through countless hours of tedious debugging and custom tool building.

5.1.3 Impatience

The reverse engineer must insist on the computer doing as much work as possible. When creating a binary compatibility system, that work generally takes the form of disassembly, static analysis, and debugging. The *Interactive DisAssembler*, IDA, from Hex-Rays [64] integrates all three of these capabilities into a single, powerful program that includes an extensible scripting interface. *Cycada* leveraged the power and capability of IDA's static analysis, disassembly, and scripting interface. Unfortunately, IDA's debugger could not be used due to incompatibilities discussed in Section 5.1.4.

Many hours were spent pouring over disassembly and graphs output by IDA, but the virtue of impatience is best exemplified through plugin scripts written against IDA's python APIs. For example, Apple does not release any ARM or iOS–specific code for their kernel, XNU. This means that there was no complete map of system call numbers used on iOS devices. Fortunately, most system calls use wrapper functions contained in the `libsystem_kernel.dylib` library. Using IDA's python APIs, we wrote a small script that would iterate through all exported functions in a library, and, for each function, parse the disassembly looking for the `SVC` instruction. When found, the script would inspect the register state, and output the name of the current function and the value of a particular register. [1]The resulting script output served as an easily parsable system call table.

In a similar example, we used an IDA script to analyze the nearly indecipherable OpenGL ES binary, and output the GL dispatch table – an internal table used to dispatch GL ES functions. This allowed us to wrap up the iOS GL ES library interface in user space using diplomats (see Chapter 3, Section 3.3.3). We also used IDA scripts to discover MIG (Mach Interface Generator) [15] and IOKit interfaces in proprietary libraries and applications such as `IOSurface`, `SpringBoard`, and `mediaserverd`.

[1]In iOS, the system all number is passed in one of three different registers depending on type of system call:
If ip (`r12`) $==$ `0x80000000` then the syscall number is in `r3`. If $ip < 0$ then the system call is a Mach trap and the trap number is $-ip$. If $ip == 0$ then the system call is `syscall` and the actual system call number is in `r0`. Otherwise, the system call number is ip.

Although powerful and extensible, IDA is not always practical for tasks ideally embedded in a shell script, or run on a mobile device. The iOS SDK ships with a complete set of binary utilities for linking, inspecting, and debugging iOS binaries. Even though the GUI IDE, *Xcode*, doesn't easily allow creation or compilation of command line utilities, other tools in the SDK can easily be used to create and inspect command line tools, libraries and applications. Using SDK and system tools, such as `otool`, `dwarfdump`, `lipo`, and `gdb`, we wrote a shell script to wrap an entire Mach-O library with diplomats. We also modified a widely used script [125] to decrypt iOS applications. Creating a library with a set of diplomats that reflect functionality from Android into iOS is tedious work that the computer is perfectly capable of doing. Our shell script takes as input the iOS dylib and a set of directories containing Android ELF libraries. The script finds all exported entry points in the iOS library, and either matches them to a corresponding ELF entry point in an Android library or writes a small stub function in C. We used this script to create the initial OpenGL ES wrapper library that was later extended with diplomat usage patterns (see Chapter 4, Section 4.4).

As described in Chapter 3 Section 3.5.1, a script, named `DCrypt.sh`, runs on any iPhone or iPad, discovers the ASLR slide of an iOS application, uses `gdb` to decrypt the TEXT segment, and then re-packages the application for easy installation on and Android device. This script provided the primary mechanism for obtaining iOS applications from the App Store.

One final area where the virtue of impatience guided *Cycada* development was logging. Log processing in *Cycada* was two-fold. First, we developed a specially formatted log output that was sent, by the Linux kernel, directly to an in-memory file. Our logging facility uses an in-kernel ring buffer that is read out from user space using a special file on a ramfs. Log lines are formatted, through a preprocessor macro, to allow easy post-processing through scripts or editor macros. Log entries are easy to add throughout the *Cycada* kernel using this macro. We developed a custom `vim` highlighting scheme and several log parsing scripts to perform tasks such as highlight error conditions, and aid in reconstruction of multi-process communication.

Second, we not only parsed our own logs, we collated and parsed system and app-specific logs. Using environment variables and program arguments, we enabled as much logging in each application and framework as was reasonable. We collected all of these logs in a single directory, and rotated the directory on reboot or relaunch of an application. We wrote several scripts to either comb these logs for useful information, or simply aggregate them and display them concurrently with application execution.

5.1.4 Patience

Shiny disassemblers and shell scripts are excellent tools for static analysis, but most bugs a reverse engineer encounters while building a binary compatibility system occur at runtime. For these bugs, a debugger is needed. Unfortunately, debugging a running process is highly system specific. The iOS debugger does not use the standard Linux debugging interface, `ptrace`. Instead, it uses iOS–specific Mach IPC APIs such as task exception ports, thread register state, and Mach VM memory flags, for thread and process introspection and control. Enabling the iOS gdb binary on an Android devices would have involved implementation of many different complex Mach IPC interfaces that are sparsely documented, and not used by any other app or library. These facilities, in aggregate, would require nearly as much effort to develop as the rest of the *Cycada* system. Subtle bugs introduced in these complex interfaces would also lead to compounding difficulty, and time lost debugging iOS apps on Android devices.

In a valiant attempt to maintain virtue, *Cycada* attempted to reuse the Android debugger, `gdb`, to debug iOS apps and libraries. Unfortunately, the Android `gdb` variant has no support for Mach-O binaries or `dyld` library loading. Without knowledge of the binary format, or proper OS support, debugging iOS apps with Android's `gdb` on *Cycada* lacks symbols, and requires a-priori knowledge of where libraries are loaded and which ARM variant (Thumb or ARM) was to encode the instructions being debugged. This makes even the most basic debugging operations such as setting a breakpoint and single stepping over instructions extremely difficult. Even if proper `gdb` support was readily available, attempting to debug a system like iOS, where user space daemons rely on each other in an indecipherable spiderweb of Mach IPC, is rarely as easy as attaching to the current process and taking a backtrace.

If impatience is the anger you feel when the computer is being lazy, then patience is the sadness you feel when the computer fails you. Debugging on *Cycada* embraces the vice of patience. Unable to force either iOS or Android debugging tools to work properly, we assembled an ad-hoc collection of techniques for rudimentary debugging of iOS apps and basic analysis of complex IPC dependencies.

To debug running applications, *Cycada* uses a tool, such as IDA or a simple hex editor, to insert an illegal instruction sequence into the TEXT segment of an application or library by replacing an existing instruction. When program execution would naturally execute the replaced instruction, a `SIGILL` signal is generated instead. Simply receiving this signal indicates that the particular code path was taken, however using an in-kernel signal handler we can glean even more information and potentially even restore the

original instruction(s) to continue execution. From an in-kernel `SIGILL` handler, we can further inspect user space register state, and even walk user space frames to generate a back-trace. When cross-referenced with loaded library offsets from `/proc/PID/maps` and basic disassembler output, this `SIGILL` can be translated into a symbolic backtrace of the user space path taken to reach the given point in a library or application. While significantly slower than a traditional debugger, this process allows basic debugging of iOS apps, libraries, and frameworks running on *Cycada*.

In addition to the in-kernel `SIGILL` handler, *Cycada* also handles other signals sent to iOS apps such as `SEGFAULT`. For memory related signals, such as `SEGFAULT`, the *Cycada* handler can not only generate a user space backtrace, but can also dump out the user space page of memory on which the process faulted. The stack traces and memory dumps can be symbolicated post-mortem using a combination of iOS binary tools such as `otool`, and a disassembler such as IDA.

To debug complex user space interdependencies without support from a formal debugger, *Cycada* uses the specially formatted kernel log output, described in Section 5.1.3, through a custom logging macro. Log entries are easy to add throughout the *Cycada* kernel using this macro. The resulting logs present a more complete picture of system activity by tagging each line with the PID, kernel function name, and timestamp. We used these logs extensively to reconstruct complicated Mach IPC communication paths, and pinpoint issues such as missing files or unresponsive daemons. While some of the log parsing was done through scripts in an impatient manner, reconstructing IPC communication and following the interaction of multiple processes was a manual process that required extreme amounts of patience.

The Axis of Human Nature

The third great virtue of a programmer, hubris, speaks to a desire deep within the programmer to write not only functional, but also beautiful, maintainable, and laudable code. Hubris is,

> *[e]xcessive pride, the sort of thing Zeus zaps you for. Also the quality that makes you write (and maintain) programs that other people won't want to say bad things about. Hence, the third great virtue of a programmer.*

Countless hours of patiently staring at disassembled binaries led us to a key insight: seemingly indecipherable assembly code was actually generated by real programmers who, presumably, must write and maintain

the binary using some higher-level code [2]. This meant that we were not staring at a random collection of instructions. Even though partially obscured by optimizing compilers and linkers, there was structure and order that could be extracted by following clues, such as function names and calling conventions, left by hubris–filled iOS developers.

Lamentably, the reverse engineer cannot be so filled with hubris (or the hubris of others) that they miss short cuts, hacks, and otherwise deplorable code that make the job of binary compatibility feasible. A reverse engineer must be willing to repudiate first principles, know when to fake it, and must not be afraid of a little duct tape. Although related to laziness, the vice of repudiation never involves well–documented, reusable code. It involves sneaky pre-processor macros that switch in-and-out ugly static inline functions for mapping one API onto another. It involves blindly returning success from an unimplemented function.

5.1.5 Hubris

The `IOSurface` iOS framework is a private interface that communicates directly to a proprietary driver in Apple's XNU kernel. THe XNU `IOSurface` driver implements zero-copy inter-process graphics memory transfer and management. Both the user space APIs and kernel interfaces used by `IOSurface` are private and undocumented. However, `IOSurface` objects are used as the backing-store for almost all graphics related memory management in iOS. Both library and kernel module code that implement a feature as complicated as zero-copy inter-process memory transfer can be nearly impossible to re-construct from constituent assembler instructions.

A reasonable first-order solution to enable a library, such as `IOSurface`, to run on *Cycada* might be to instrument the Android kernel and observe access patterns and input data. However, this approach is insufficient when the kernel API itself is nothing more than a conduit for opaque data blobs being passed between a proprietary user space library and a proprietary kernel module. The `IOSurface` user space library uses a MIG interface to communicate to an XNU kernel module. The IPC messages passed to the kernel use undocumented structures and opaque data types (such as `void *`) which are interpreted by the proprietary kernel module.

A better solution to enabling the `IOSurface` APIs on *Cycada* leverages the hubris embedded into the proprietary library by iOS engineers. By exploring the disassembler output, applying higher-level semantic

[2] Although apparent in hindsight, this insight cost us many sleepless nights and more than a few gray hairs!

reasoning, and using readily-available developer tools for debugging user space applications, we were able to reconstruct both the user space MIG interface and the kernel functionality of the `IOSurface` API.

A disassembled view of the user space `IOSurface` library reveals that there is a single point of entry into the kernel: `IOConnectCallMethod` (or its asynchronous variant). This function can be found in the open source IOKitUser library [14], and its implementation crosses the user-kernel boundary using Mach IPC to a kernel I/O Kit driver. Instrumenting the Mach IPC path is not particularly enlightening because the interface from user space to the kernel is defined by data packed into the messages by the proprietary code on either end. However, looking again at the disassembly output of the `IOSurface` library, there are several intermediate function calls between exported library functions and `IOConnectCallMethod`. These intermediate functions represent the MIG interface to the kernel module. Unfortunately, manual inspection the assembly for each of these functions would be time consuming and ultimately infeasible for a single grad student.

The iOS SDK ships with ARM variants of most libraries and frameworks, and most of these contain some level of debugging symbols. Using the insight that real, hubris–filled iOS programmers wrote the code that was compiled into the `IOSurface` binary, the available symbols from the SDK and the symbol names of the intermediate functions used to invoke `IOConnectCallMethod` disclose quite a bit about the nature of their higher-level function. Further, repeated patterns in the disassembly inform a basic re-construction of function arguments. With a carefully crafted regular expression, we were able to re-construct the MIG interface to `IOSurface`, and enable further investigation into kernel functionality and API data structures from even higher level tools.

In addition to writing the code that compiles into the `IOSurface` library, iOS programmers wrote and used debugging tools that are also available in the iOS SDK. Although the ultimate goal of *Cycada* is to run iOS apps on Android devices, those same iOS apps can be debugged on an iOS device using all of the fancy features provided by the SDK and IDE. In other words, we can write a custom application to exercise private APIs, and debug the app on an iOS device where we can single-step through all of user space, and even inspect memory and register contents. To further investigate the `IOSurface` API, we wrote a custom iOS app that linked against (or directly called into) private `IOSurface` APIs. In the absence of header files, local C-style *extern* prototypes gleaned from the SDK symbols and libraries worked quite well. We then used the XCode IDE to debug our own app on an iPhone and iPad. This custom application used standard security and reverse engineering techniques, such as input fuzzing and memory inspection, to reconstruct

the data structures used by the `IOSurface` kernel module. Correlating this debugging output with the higher-level intuition gained from symbols in the SDK, we were able to reverse engineer `IOSurface` data structures, and implement necessary pieces of the kernel logic to enable the user space iOS library to run without modification on *Cycada*.

Another example of leveraging the hubris of other developers is *Cycada*'s mapping of iOS OpenGL ES extensions to Android OpenGL ES extensions. Table 4.1 shows a rather bleak picture of the discrepancy between iOS extensions and Android extensions. Less than 50% of implemented GL ES extensions on either platform overlapped with the other. While some iOS extensions did require a diligent, brute-force implementation effort, some of them could be mapped onto extensions provided by Android's Open GL ES implementation. By realizing that Apple engineers were working under similar constraints and solving similar problems to Android engineers, we were able to find either a directly corresponding Android GL ES extension, or a set of Android GL functions that could implement the iOS GL ES extension. For example, as described in Section 4.4, iOS uses the APPLE_fence extension which is not present in Android. However, both iOS and Android developers needed a way to synchronize their rendering pipelines based on the completion of previous GL ES commands. Using this knowledge, we found that the NV_fence extension implemented on Android provides identical functionality to the APPLE_fence extension. Using a small amount of glue logic, in the form of indirect diplomats, we were able to support the iOS APPLE_fence API using the Android NV_fence API.

5.1.6 Repudiation

Binary compatibility work can be messy. While a good reverse engineer should strive to be lazy through nicely packaged reusable code, many problems will have one-off, bedraggled solutions that you won't ever want to share with your colleagues. Other problems could potentially be solved using a proper framework or diligent brute force enablement, but at a cost of man-months or years. The vice of repudiation frees the reverse engineer to think creatively, hack freely, and dismiss formalism with prejudice. This methodology almost always comes with an associated cost in either efficiency, security, or isolation. This is not code that wins you the Turing award. It's code that gets the job done, and occasionally makes for a great story. What follows are a few of those stories from the *Cycada* development.

Binaries run on iOS can be both encrypted and cryptographically signed. Encryption ensures that the

contents of the binary are only used by authorized devices. All apps downloaded from Apple's App Store are encrypted, and *Cycada* used a jailbroken iOS device to decrypt the binary contents (see Section 3.5.1). From Apple's documentation [22], cryptographic signing of applications, or *code signing*,

> *. . . is a security technique that can be used to ensure code integrity, to determine who developed a piece of code, and to determine the purposes for which a developer intended a piece of code to be used.*

Code signing uses a collection of checksums or hashes of the various pieces of the binary or application, a digital signature, and a unique identifier. Nearly all binaries in iOS are code signed, and the code signature is verified by the kernel binary loader before execution. While it may be technically possible to write an entire code signature verification engine out of reusable hashing and decryption modules, the effort would be monumental and the gain minimal.

Because *Cycada* provides its own Linux kernel Mach-O loader, we simply skip code signature verification altogether by instructing the *Cycada* loader to return success for all code signing queries. While skipping code signature verification of binaries allows *Cycada* to run application, library, and framework code, it also means that it does not perform any actual verification of the code signatures.

Application sandboxing [34] is another example of a complex iOS subsystem that would take many man–months or years to fully implement. Sandboxing is Apple's method of restricting the data and system features an app may use. This includes restrictions on both file system paths, and use of certain syscalls. The iOS sandbox is a kernel feature, and thus can be easily bypassed in *Cycada*. When user space attempts to enter a given application into a sandbox, the *Cycada* kernel simply returns success indicating that the app is now in a sandbox. Handling the iOS sandbox in this way allows *Cycada* to run iOS binaries, but obviously does not afford them the same security. We rely on Android's model of user and process isolation to implement security, and thus cannot give the same guarantees. Further compatibility between the Android and iOS security models is left for future research.

Sometimes, supporting an iOS feature or kernel subsystem can be as easy as re-directing or renaming an existing Android or Linux feature. In other words, occasionally binary compatibility is really as easy as just pretending to be iOS. For example, in iOS the default network adaptor name is expected to be `en0` while in Android it is `eth0`. By putting a small translation table in network adaptor functions, we easily translate between the two naming schemes and convince iOS user space it has a network adaptor with the

correct name. In similar fashion, when iOS user space queries the current device or hardware on which it is running, via a `sysctl`, *Cycada* just returns a predetermined string such as "iPhone2,1" or "iPad2,4".

Some iOS user space code iterates over all available I/O Kit devices looking for devices with particular names or classes. In some cases this code isn't used to interact with the device, but rather to gate the execution of another piece of code. For example, the `configd` daemon is responsible for attaching a name to a particular network interface. To do this, it first waits for an I/O Kit service named `IONetworkStack`. When it sees this service, it then watches I/O Kit for any device that conforms to the `IONetworkInterface` class.

In order for iOS user space to properly discover resources and devices mapped from Android to iOS, the names of devices and their supporting driver instances in the *Cycada* I/O Kit implementation must match what is expected by user space. Often these devices use the same, or similar, hardware on both iOS and Android, but have significantly different names. The iOS driver framework, I/O Kit uses a special flattened device tree to initialize and load drivers. *Cycada* hooks into the Linux `device_add` function and calls out into a small support function that translates Linux device names into valid iOS names, and inserts a node into the *Cycada* I/O Kit device tree. Using a small I/O Kit driver class that matches the translated name and leverages existing Linux driver code, *Cycada* can support iOS user space libraries looking for devices with specific, non-Linux names. More details can be found in Chapter 3, Section 3.4.1.

Several iOS system calls that have a corresponding Linux implementation, such as `open` and `read`, also have iOS–specific variants that are not present in Linux. For example, the *_nocancel* postfix was added to many different system calls resulting in two variants, e.g., `read` and `read_nocancel`. In all cases we encountered, the postfixed variant of a previously defined system call either bracketed the original system call with an invariant, or used an ancillary security mechanism such as an opaque token that was associated with each subsequent interaction with the returned resource. Calls postfixed with *_nocancel* were used to enable in-kernel pthread cancellation support. Because *Cycada* uses Linux-style threads (light weight processes created via `clone`), the *_nocancel* system call variants were simply mapped to their standard counterpart.

The other postfix we encountered, *_guarded*, added a parameter to the original system call which functioned as a security token. The *guard* parameter was generated by the kernel, and verified on subsequent use of the returned resource. For example, `open_guarded` returns an opaque kernel–generated token to user space. The resulting file descriptor can only be used with the corresponding `read_guarded`, `write_-`

`guarded`, and `close_guarded` system calls. The kernel verifies the security token passed from user space, and guarantees that an invalid security token will not be able to operate on the given file descriptor. This guards against malicious (or buggy) user space code from closing or hijacking a file descriptor. *Cycada* ignores these *guard* parameters and simply calls the appropriate existing system call. By doing this *Cycada* can support the *_guarded* interface, but it cannot guard against file descriptor hijacking.

Finally, implementation of the *Cycada* duct tape code-adaptation layer often required messy, one-off hacks that would vex a virtuous programmer's desire to reuse and brag about their code. While we were able to develop a set of design principles, discussed in Chapter 3 Section 3.3.2, the nitty gritty implementation would do things such as define a static inline function in multiple ways depending on how certain pre-processor macros were set. We would also use similar tricks to hide, redefine, or redirect symbols that appeared in both XNU and Linux kernel code.

5.2 The Future: Forward-Porting *Cycada*

Every new release of iOS brings with it new functionality not only at the user-visible level, but also at the kernel level. Our *Cycada* prototype is based on iOS 5.1, but we began an initial port of *Cycada* to iOS 6. The implementation is incomplete, but our experience indicates that the hints, virtues, and vices described in Section 5.1 can guide a successful forward-port of *Cycada* to iOS 6 and beyond. The remainder of this section describes our initial porting effort, and exemplifies our use of the hints, virtues, and vices.

The porting effort began by simply fixing a large number of bugs in the existing *Cycada* prototype implementation. Many of these bugs were exposed due to OS behavior or features exercised in iOS 6, but unused in iOS 5.1. Beyond initial bug fixing, porting *Cycada* to iOS 6 was guided by the hints, virtues, and vices described in Section 5.1. Identifying key virtues or vices that may aid in the binary compatible support of a new iOS feature greatly reduces the effort required to support it. While the port is still incomplete, there have not been any fundamentally limiting problems, and we expect it is feasible, given enough time, to forward port *Cycada* to the latest version of iOS. In fact, many issues were solved by simply applying patches from Apple's open source software to their duct taped counterparts in *Cycada* (Section 3.4.1 describes patching iokit to a newer version).

New additions to `launchd` and `launchctl` in iOS 6 disallow unsigned startup scripts (plist files) to be run on boot. An immutable set of plist files was put into the shared cache and code-signed. The signature

is verified by launchd before running any startup jobs. At first glance this problem appears complicated and untenable, however, both `launchd` and `launchctl` are open source, and a quick inspection of the code revealed a simple solution. Before verifying the code signature and location of startup plist files, launchd checks the XNU boot arguments for a particular string. By providing this particular string in *Cycada*'s faux boot arguments, we were able to bypass this new feature completely.

Other features added to iOS 6 were variations on existing compatibility solutions already in place in *Cycada*. For example, the `kevent` syscall added a 64-bit variant with a new structure, and the `pthread_workqueue` interface was refactored. Both of these interfaces are supported by the XNU kernel, but *Cycada* implements both as user space libraries. Adding support for both of these new features involved refactoring the existing *Cycada* `kevent` and `pthread_workqueue` libraries to support the new interfaces.

Another addition to the `kevent` interface was the ability to receive a Mach message simltaneously with `kevent` delivery. A new flag, `MACH_RCV_MSG`, was added to the `kevent` interface, and *Cycada* was able to support this feature in our user space `libkqueue` implementation through a single additional function that called `mach_msg` when a `kevent` was delivered with this flag.

There were also new features in iOS 6 for which either more re-direction and mis-information, or an adjustment of previous mis-information resolved the missing support. For example, Apple's sandbox interface was slightly adjusted. Previously, the function which requested that an app be moved into a sandbox required a couple bytes to be set in a user space structure by the kernel sandbox syscall. The new version of iOS slightly adjusted the position of those bytes in the user space structure.

Finally, there were some new assumptions made by `launchd` and `notifyd` which *Cycada* had to deal with. The iOS "init" program, `launchd`, can output a lot of extremely helpful debugging logs, however, the thread that outputs these logs waits for a `kevent` of type `EVFILT_FS` to indicate that `/var/log` has been mounted. Previous versions of iOS had made the same `kevent` call, but never depended on the output to initiate logging. We had to implement basic `EVFILT_FS` support in the *Cycada* `libkqueue` implementation in order to enable iOS `launchd` logging in iOS 6.

The `notifyd` daemon is a critical part of the iOS ecosystem - it's the process that delivers asynchronous notifications of system or custom events. Upon receiving or sending its first message, `notifyd` implicitly assumes that timezone, TZ, information has been properly bootstrapped. This occurs as a result of a cycle in the message handling code where the message reception function invokes either `localtime` or

`strftime`. Both of these functions attempt to set `TZ` information. If the `TZ` information has not been set, the function sets it and then attempts to send out a notification that the time zone changed. In order to send out the timezone change notification, the code calls into `libnotify`, the client library for communicating to `notifyd`. Beacuse the code is already synchronously handling a client request (the one that generated a call to `localtime` or `strftime`) this call from client-side `libnotify` to the `notifyd` server deadlocks! By properly bootstrapping the timezone data, we were able to get the iOS 6 version of `notifyd` fully functional.

Chapter 6

Teaching OS Using Android

6.1 Introduction

Hands-on learning through programming projects plays a key role in computer science education, and hands-on kernel programming projects are especially important in the area of operating systems (OS). Many approaches to designing these kernel programming projects have been proposed and implemented. Several pedagogical OSes exist [107; 119; 41] where students fill in or build missing subsystems. In recent years, many institutions have also begun using Linux to teach OS [3; 91; 63]. All these solutions focus primarily on desktop or server environments, whether physical or virtual.

The computing landscape, however, is shifting. The dominant computing platform is becoming the mobile device [45; 132]. The real-world constraints and operating environment of mobile devices are quite different from traditional desktop or server computers. It is important for students to learn in this new environment, and its prevalence and popularity can be used to create engaging programming projects.

We present our work using Android to teach OS through hands-on kernel programming projects. We chose Android for several reasons. First, as a production system it enables students to learn about real-world OS issues which are hard to glean from simplified pedagogical projects. Second, since Android is based on the open-source Linux kernel, students can leverage a wealth of Linux tools and documentation. Third, Android's use of the Linux kernel provides a familiar transition path from courses already using Linux to teach OS. Fourth, Android is the fastest growing mobile platform to date, and its popularity makes it of tremendous interest to students. Fifth, Android is open-source which allows exploration of a complete

production system including the OS kernel, user space libraries, and a graphical user environment written in Java. Sixth, as a commercial platform, Android continues to be developed and improved which naturally evolves the platform as a pedagogical tool, enabling students to learn in a modern context. Finally, as a commercial platform, there is no need for us to maintain or update Android or any of its development tools. This allows us to focus limited resources on teaching rather than time consuming in-house OS development.

To facilitate our use of Android to teach OS, we created an Android virtual lab where students learn about operating systems using both emulated and physical mobile devices. We manage the complexity of device cross-compilation and production kernel development tools by providing a virtual appliance pre-configured with all the software tools necessary to develop an Android Linux kernel. A virtual appliance can be readily deployed, downloaded and used by students without the installation or configuration necessary to deploy Android development tools natively on their personal computers. Additionally, our Android virtual lab uses a distributed version control system and live demonstration infrastructure to develop, distribute, submit and grade homework projects. Our use of the Android emulator in conjunction with this infrastructure allows remote and distance learning students to take full advantage of our Android virtual lab, while also facilitating close collaboration with on-campus students using physical mobile devices.

Students work in groups to complete five Android Linux kernel programming projects. These projects require students to read and understand core Android Linux components, and then either modify or add components as required. The projects build in complexity, and cover various important OS topics: (1) system calls and processes based on the unique process hierarchy of an Android device, (2) synchronization where a global resource such as the orientation sensor is shared amongst many processes, (3) scheduling by exploiting Android's single-application usage model to increase device responsiveness, (4) virtual memory and the exact size and nature of Android's inter-process shared memory, and (5) file systems with automatic geo-tagging using Android sensors.

We successfully used Android in a 100 student introductory OS course at Columbia University. Over 80% of students surveyed enjoyed applying OS concepts to the Android platform, and the majority of students preferred Android over traditional desktop development.

6.2 Android Virtual Lab

Central to any OS development lab is universal access to proper development tools for all students. Our Android virtual lab provides each student with a pre-configured VMware virtual appliance containing all the Android and Linux development tools necessary to complete each programming project. The set of tools includes all tools necessary to boot and test a real device as well as the Android SDK comprising the Android emulator, a tool to create virtual devices, and a device debug GUI tool. We also include a cross-compilation toolchain, Android's Bionic C library, and several shell scripts to mitigate the complexity of embedded development.

Although Android development tools are available for a wide variety of platforms, we provide a pre-configured virtual appliance for four important reasons. First, we avoid mistakes or incompatibilities in development tool installation. For example, when the course was offered, the Android development tools did not support Mac OS 10.6, Mac OS versions prior to 10.4, or Linux distributions other than Ubuntu. Students may also use non-standard PC configurations which would put unnecessary management burden on the instructional staff. Second, the virtual appliance can be used as a safety net for students who corrupt their development tool installation or experience complete system failure. The VMware Workstation snapshot feature can be used to make incremental backups of student work and any changes they make to the development environment. A snapshot is also an easy way for students to begin a homework assignment from a known good configuration. Third, by pre-configuring all of the development tools, we avoid complicated cross-compilation setup by providing simple, standard Makefiles and shell scripts for both kernel and user-level development. The Android SDK is designed primarily for GUI application development not kernel development, so some configuration of the compiler and Android runtime libraries is necessary. A standardized environment allows us to provide simple Makefile examples for user-level test programs and simple Linux kernel cross-compilation instructions. Finally, the virtual appliance gives us freedom to customize both the tools and the Android user space. Customizing the Android user space allows us to create more engaging projects as well as to overcome deficiencies in development tools. We use a customized version of the Android emulator which enables students to use the *Sensor Simulator* program from OpenIntents [98] to inject orientation and acceleration data into the emulator by interacting with a 3D model of a phone instead of manually entering numbers into a shell prompt. We also use a custom device drawing library to support the kernel scheduling project described in Section 6.3.3.

To manage homework project preparation, distribution, submission and grading, we use the Git distributed version control system, already used by Android and Linux for source code version control. Each homework assignment consists of a single Git repository that typically contains a complete Linux kernel tree, template user-space projects as necessary, any additional tools needed to complete the assignment, and a Makefile used to prepare the student's emulator or device for project development. Instructional staff prepare each project repository and push it to a central Git server. The repository is then replicated such that all student groups are given access to their own private repository on the Git server. The central Git server also facilitates distributed group collaboration even when one or more students are remote or distance learning students.

To maximize grading efficiency, we extend the previously developed concept of live demonstrations [81]. Three or four student groups are assigned an hour long demo time slot. During the first 20-30 minutes, all student groups perform a complete clone of their homework submission Git repository, cross-compile the Linux kernel for their mobile device, and install and boot the new kernel. In the last 30-40 minutes, the staff meets individually with each group. During this time, groups demonstrate functionality required by the homework, further explain their solution methodology, and participate in a basic code review. This time helps instructional staff to better understand the group's submission, correct common mistakes, and see how group members contributed to an assignment. Live demos also provide opportunities for instructional staff to explain solutions which can facilitate a more complete understanding of difficult and challenging assignments.

Our Android virtual lab is designed to support remote or distance learning students, though such students may only have access to the emulator and not a real Android device. These students can still participate in demonstrations using freely available screen-sharing applications such as *Skype*, *join.me* or *VNC*.

6.3 Kernel Projects

Using our Android virtual lab, students work in groups to complete five kernel programming projects. These projects require students to read, understand, and modify core Linux components. While some projects require writing a simple user space test program, students are never required to compile the entire Android code base or write any GUI applications. The five projects focus on five important OS concepts, and infuse Android or mobile device specific investigation into the assignment. The five areas covered by the

assignments are: system calls and processes, synchronization, scheduling, virtual memory, and file systems. Corresponding Android-related topics incorporated in these areas are: the *zygote* process and Java worker threads, device sensors, display-prioritized scheduling, multi-process working set via copy-on-write shared memory, and location aware file systems.

The assignments progress in complexity, building not only on OS principles learned in earlier assignments, but also on Android specific knowledge gained. For example, the first assignment requires students to investigate the Android process tree and note how all GUI programs are children of a process named zygote. In a subsequent homework, they investigate the cross-process memory sharing method used by the zygote to save system RAM. We formalize each project's Android and mobile device investigation by asking students to answer a small number of questions designed to make them reflect on how the particular OS concept was applied in the context of an Android or mobile device.

All assignments are intended to keep students focused on the OS principles being taught, and designed such that a group of two or three students can complete them with no prior kernel or Android experience. We provide detailed step-by-step instructions on cross-compiling and device or emulator use. Android specific aspects of the assignments are presented as practical application of the core principle, and most of the Linux kernel modifications necessary to complete the assignments are contained in architecture independent code. Thus, the solution complexity remains manageable and homework setup and prerequisite topic knowledge is kept to a minimum, yet students engage with a complex, real-world system.

6.3.1 System Calls and Processes

The first project lays the groundwork for future projects as students investigate the primary application abstraction, the process, and its primary interface into the kernel, the system call. Our focus in this assignment is process creation, termination, properties, and relationships in the context of a mobile device. In completing this assignment, students also gain an understanding of kernel data structures, such as linked lists, and their APIs. All subsequent assignments require students to be intimately familiar with these concepts.

Students write a new system call which returns the system process tree in DFS order. This involves modifying architecture specific system call entry points, manipulating and traversing the kernel data structures representing processes and threads, and managing data transfer to and from the kernel. To test the system

call, they write a simple user space application to invoke this new system call and print out the process tree similar to the UNIX *ps* utility.

The system call allows students to examine Android's process tree and application startup method, which provide insight into a key system design that drives the entire Java-based user environment. Java applications are interpreted in the *Dalvik* virtual machine, and are represented in the OS as processes which are children of a special process called the zygote. The Dalvik virtual machine starts several worker threads for each application to handle things like input events and garbage collection. Thus, the process tree of Android devices shows not only the relationship between the *init* process and its children, but also the relationship that all Java applications have to the zygote and the symmetry of their component threads.

Students investigate the zygote process using their test program, and are asked to reason why an embedded or mobile system might use such a process. This reflection is designed to connect the pedagogical concept of process creation using copy-on-write memory to a real-world mobile device with memory and disk constraints. Benefits of the zygote include faster application startup time, and cross-process memory sharing of core library code and static data.

The Android emulator enables remote or distance learning students taking the class to complete this assignment without a physical device. The Android emulator provides a complete machine emulation in which a standard version of the Android runtime is installed and run. Thus, remote or distance learning students can investigate the Android process tree using the emulator in the same way on-campus students use real devices.

6.3.2 Synchronization

The second project focuses on synchronization, a critical aspect of a modern multi-tasking OS. The wealth of sensors available on modern mobile devices provides an excellent pedagogical vehicle to demonstrate real-world applications of synchronizing concurrent or interleaved access to a single resource. In completing this assignment, students also gain an appreciation for manipulating and interacting with embedded system sensors.

Students implement a novel synchronization primitive, the *orientation event*, which allows multiple processes to block until the mobile device has been put into a particular orientation. For example, a process can block until the phone is placed face down on a table. To accomplish this, they first write a user space

daemon which reads device orientation through a standard Android hardware abstraction library, and then passes the data into the kernel through a new system call. The orientation event interface is implemented as a set of three new system calls: `orientevt_open`, `orientevt_close`, and `orientevt_wait`. The daemon process passing device orientation into the kernel functions as the signal which wakes up any blocked process. Students test this new interface by writing several small test programs. Each test program forks multiple children, and each child process blocks on an orientation event opened by its parent. When the device is moved within the range of the desired orientation, all child processes should be unblocked.

Orientation and acceleration sensors are an integral part of the mobile device experience, and incorporating them into a synchronization project gives students an experience that desktop or server machines cannot provide. The ability for multiple processes to wait for a device to enter a particular orientation or acceleration profile has many possibilities in real-world user applications and system services such as interactive game controls and pre-fall system shutdown.

As students investigate sensor-based synchronization on Android, they are also exposed to real device interaction using a hardware abstraction layer. This interaction necessarily includes basic understanding and manipulation of sensor data. We provide several helper functions to keep the students focused on the primary topic of synchronization, however the exposure to real-world device data is a valuable experience which can be directly applied in the workforce.

The Android emulator provides the ability for remote or distance learning students to complete this assignment without a physical device. We provide a modified version of the Android emulator, a daemon process to run on the emulator, and a Java-based host application which simulates [98] a mobile device using a 3D wire-frame model. As students manipulate the model in the Java application, orientation information is sent to the daemon process which updates emulator state. The Android hardware abstraction layer reads this emulator state. In this way, remote students can have a similar experience to on-campus students

6.3.3 Scheduling

The third project focuses on scheduling. Mobile devices generally operate as a single user environment, and thus have significantly different scheduling requirements from desktop or server machines. For example, Android users typically view a single application at a time, not multiple applications in multiple windows.

In what is one of the most challenging projects, students write a new scheduling policy for the Linux

kernel. This is the first assignment which requires students to manipulate a substantial portion of core Linux kernel code. To mitigate the daunting task of implementing a new scheduler, we leverage the modular scheduling framework in Linux which provides several examples of self-contained schedulers. We encourage students to use these existing schedulers as templates. The new scheduling policy trades fairness and throughput for responsiveness, a key metric in mobile devices. We call our new scheduler, "Display Boosted Multi-level Container" (*DBMC*) scheduling, and the primary objective of this scheduler is to "boost" the priority of foreground applications. Since mobile devices typically display a single application at at time, boosting the priority of this single, foreground, application decreases its execution time and increases the apparent device responsiveness.

To support DBMC scheduling, we made a small (15 line) change to the Android user space environment that leverages the Android application usage and security model. In contrast with desktop Linux systems, Android only allows a single application to use the display at a time. While multiple applications can run simultaneously, only one draws on the screen. When an application is installed onto an Android device, a unique user ID is generated and assigned to the application. The application is always run using these unique credentials. We wrote a simple, 15-line patch to a core Android drawing library, `libsurfaceflinger.so`, which informs the kernel of the process ID of the application currently drawing on the screen. Students use the unique user ID associated with the process to easily assign the associated threads and processes to a single scheduling entity, the container. The container is used for scheduling so that the priority boost is applied to all of the threads and processes of the foreground application.

Students are asked to run Android using their new scheduler and consider what qualitative impact there is on system performance compared to the existing Linux scheduler. A correct solution implemented on the Google ADP1 booted the GUI 5–10 seconds faster than the standard Linux kernel, but initialized the network and cellular connections significantly slower. Students are also asked to reflect on this scheduler's impact on graphics-intensive games where process starvation can occur and overall game play can suffer.

The Android emulator provides a similarly satisfying experience for remote and distance learning students. The same modified drawing library is used in the emulator. A correct solution implemented on the emulator booted the GUI faster, and made the entire UI qualitatively more usable and responsive.

6.3.4 Virtual Memory

The fourth project explores memory management, a critical aspect of mobile device OSes, with a focus on virtual memory and paging Using a mobile device to investigate virtual memory and paging highlights real-world system constraints and provides a unique platform to investigate creative solutions in memory sharing and allocation.

Students write a new monitoring mechanism to track the working set of specified processes, and a new system call to extract the recorded data. The working set is defined as the set of pages accessed (read or write) by a process during some time period. To test this mechanism, students add a set of processes to the monitoring mechanism, and then write a user space program that invokes the new system call and displays usage information for each process.

Here we follow up the investigation of the zygote process seen in the first project as discussed in Section 6.3.1. This process loads several libraries and Java classes, pre-initializes Dalvik virtual machine state, and listens on a socket for connections from a client. To start a Java application, Android connects to the zygote socket and requests that the process fork. The child begins executing Java code at a particular method of a specified class, and the parent resumes listening on the socket. When the child forks, all memory mapped by the parent is shared copy-on-write with the child. This includes all loaded libraries and initialized Dalvik virtual machine state. This is drastically different from a traditional desktop or server where processes are spawned from a shell or init process that has little or nothing in common with the application being started.

Students use their working set monitor to investigate the cross-process shared memory of the zygote and its children. We define the set of pages originally mapped by the zygote as the, "Android working set." Students use their user space utility to calculate the intersection of the Android working set with the working set of each zygote child process. For simplicity, we assume that no process un-maps or re-maps a region of memory originally mapped by the zygote; this allows us to use the virtual addresses returned by our new system call without needing a more complicated virtual to physical address mapping. By investigating the unique implementation of a zygote process, students gain an appreciation for the memory constraints of a real device and can measure the effectiveness of Android's zygote solution for reducing memory usage.

The Android emulator provides the ability for remote or distance learning students to complete this assignment without a physical device. The assignment does not require the use of any physical device features.

6.3.5 File Systems

The final project focuses on file systems. Similar to virtual memory, file systems tend to be large and complex pieces of code, so we have students implement extensions to an existing file system code base. This assignment requires students to gain practical understanding of how the virtual file system (VFS) infrastructure is designed, which is the key file system abstraction layer that every file system designer needs to understand. In addition to the file abstraction, students are exposed to issues that arise in real-world systems, such as the endian-ness of permanent data storage, data consistency and reliability, and embedded data retrieval.

Students modify an existing disk-based file system to automatically include GPS information so that this information can be used by any application. We refer to this as the geo-tagged file system. All files and directories in a geo-tagged file system include embedded location information in the form of latitude and longitude values. Students write a new system call and user-level daemon to inform the kernel of the current device location, which is retrieved from the GPS sensor via the Android hardware abstraction library similar to the one used in the second project discussed in Section 6.3.2. Students then modify the `ext2` file system to retrieve the last known location data and update a file or directory every time it is created or modified. Students test the geo-tagged file system by writing a second system call to retrieve the embedded location data for a given pathname. This assignment brings together several topics from earlier homework assignments including system calls, synchronization, and sensor data management. As a more advanced challenge, students can implement VFS layer interfaces which would allow any disk-based file system to implement the geo-tagging.

While it is possible to use a GPS sensor with a desktop or laptop system, the integration of location services in a mobile device is much more engaging, and exposes the student to real-world issues of sensor data reliability and permanent storage. In addition to the actual device location returned by the hardware abstraction layer, we also ask students to store the relative "age" of the location data (the number of seconds since the location was last updated in the kernel). This provides a basic confidence metric that can be used when later retrieving a file's location.

Students discover the practicality of their solution as they create a file system image which contains at least three files with unique location data. Students can have fun visiting different places while keeping

track of exactly where they were with a simple shell command on their phone such as: `echo "HERE" > `date +"%s"`.txt`.

The Android emulator provides an emulated GPS sensor, allowing remote and distance learning students to fully participate in the assignment. The emulated location is accessed using the same hardware abstraction layer library used on the real device, and can be updated using the emulator's debug interface. The Android development tools also provide a graphical utility which can update the emulator's location by reading `.kml` files generated in Google Earth. Thus remote and distance learning students can take a virtual trip using Google Earth and save files in their geo-tagged file system from each location they visit.

6.4 Experiences

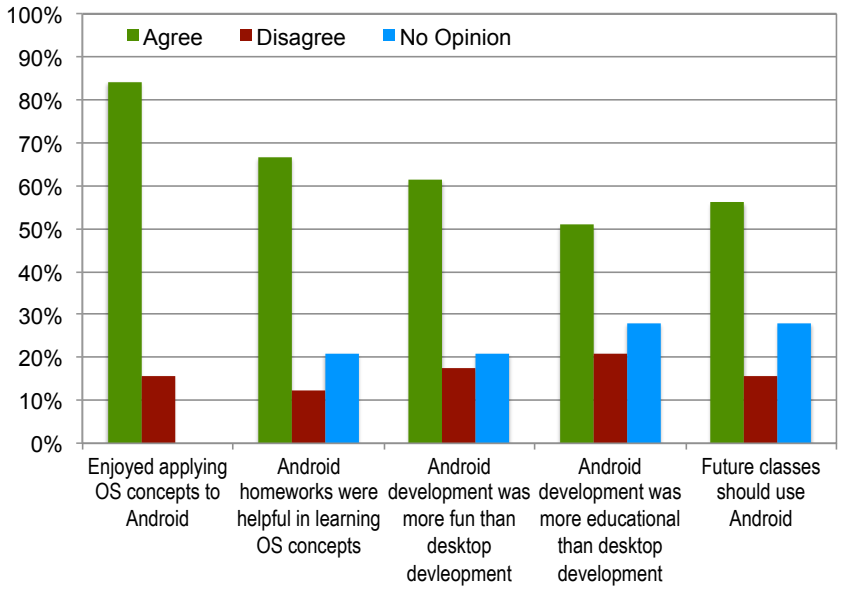


Figure 6.1: OS Course Survey Results

We used Android as the homework project platform in an introductory course on operating systems at Columbia University in Fall 2010. This was the first time we taught the entire course using Android devices and Android-kernel projects. Despite our mistakes and mis-steps, the overall response from students was overwhelmingly positive. Approximately one hundred students enrolled in the course, and we asked all of them to complete evaluation surveys at the end of the course. Sixty percent of the students completed the

survey. Figure 6.1 shows the results. Of those students who completed the survey, 80% said they enjoyed using Android in the course. Most students who completed the survey said Android was both helpful in learning OS concepts and more fun than traditional desktop development; students preferred Android to traditional desktop development by a ratio of 3 to 1. In addition, not more than 21% of the students actively disagreed with any given survey question.

Students were also asked to comment on what they enjoyed the most and what they enjoyed the least about the course. Students found the Android kernel projects fun, exciting, engaging and educational. They also appreciated the practical skills gained from their experience. When asked the question, "What did you like best about using Android for this course?" students responded with:

- "*T*he practicality of it,"
- "*M*ore fun and exciting,"
- "*I*t's more like real work, not just homework,"
- "*I*ncreased curiosity about mobile platforms,"
- "*U*sing a modern system,"
- "*A*ll sensor related assignments were fun."

Negative feedback was concentrated around the speed of the emulator and debugging of the embedded systems (emulator and mobile device). Students felt that the emulator was too slow, and that debugging an embedded kernel was overly complicated. Unfortunately, the speed of the emulator is directly correlated to the speed of the laptop on which it is run, and because the mobile device uses an ARM processor the instruction set must be emulated which is a slow process. In the future, as SMP support for ARM is integrated into QEMU, and as laptop processor speeds increase, it should be possible to speed up the emulator. Debugging an embedded system is an inherently complicated task, and simplifying the process is not easy. One approach to this problem is to create more explicit and detailed instructions on the use of Android and kernel level debugging techniques. such as the use of `/proc/last_kmsg` which stores the kernel log message buffer of the last booted kernel and can be used to diagnose the previous kernel crash. It also may be possible to provide hardware-modified devices that expose either a JTAG connection for low-level

debugging, or a serial port for simplified kernel debugging. The Google ADP1, for example, exposes serial RX/TX pins on its *ExtUSB* connector.

Finally, although students had some difficulty with device debugging, the overall experience using Android as the homework project platform in our introductory OS course was positive. Students not only learned the OS principles being taught, but also gained valuable real-world skills such as practical device debugging and embedded code development. We have already heard from students who were able to directly apply the skills they learned in this course to a professional job or internship opportunity.

6.5 Related Work

In recent years, courses using the Linux kernel for OS programming projects have become increasingly popular [3; 63; 91; 82]. However, these approaches focus primarily on Linux desktop and server environments and provide no hands-on experience with mobile platforms. Lawson *et al* [85] describe a single programming project where students modify a Linux kernel designed to run on an iPod. However, this single project does not incorporate any mobile device specific pedagogy and does not provide a rigorous, project-based curriculum for understanding OS principles in the context of mobile platforms. The BabyOS [86] and embedded XINU [36] projects focus on embedded systems, but lack the real-world applicability of a production Linux kernel. Furthermore, because Android is based on the Linux kernel, our approach provides a straightforward transition path for courses that already use the Linux kernel to incorporate the mobile and embedded concepts embodied in our Android projects. Our approach provides a mapping of structured Linux kernel programming projects onto the Android platform where students gain insight into real-world systems and learn practical skills immediately applicable in today's job market.

Various pedagogical OSes have also been developed [107; 119; 41], but none of them offer students practical experience or insight into modern mobile platforms. Atkin *et al* [27] developed a pedagogical OS focused on portability and mobility. However, all programming projects are done at user level in a simulator, and do not offer the engaging, real-world experience of using a popular, mobile computing platform such as Android on real mobile devices.

Several institutions offer courses in mobile application development [134] . These courses focus on mobile application APIs, and do not teach OS concepts. They offer no insight into the lower-level software infrastructure of mobile platforms on which applications run. In contrast, our method of using Android to

teach OS provides students with a real understanding of how things work under the covers as embodied by the unique OS environment created by the Linux kernel running on a mobile device.

6.6 Conclusions

We have developed a series of hands-on Android Linux kernel programming projects designed to immerse students in a mobile computing environment while simultaneously teaching core OS principles. The projects progressively introduce OS principles and mobile computing, and implicitly teach students about real-world embedded device development. We use key aspects of modern mobile devices, such as orientation sensors, to enrich the hands-on experience and increase student engagement.

We created an Android virtual lab where both on campus and remote students complete Android Linux kernel programming projects using an emulator or mobile device. We also leverage a distributed version control system and live demonstration infrastructure for homework design, distribution, submission, and grading. Our experience with 100 students using Android to teach OS demonstrates that students enjoy using mobile devices while learning OS principles, and appreciate the practical skills they gain.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

My research has demonstrated, through two key systems, that lightweight mechanisms can be added to commodity operating systems to enable multiple virtual phones or tablets to run at the same time on a physical smartphone or tablet device, and to enable apps from multiple mobile platforms, such as iOS and Android, to run together on the same physical device, all while maintaining the low-latency and responsiveness expected of modern mobile devices.

First, we presented *Cells*, the first OS virtualization solution for mobile devices that enables a single mobile device to use multiple personas through isolated and secure virtual instances of a mobile operating system. *Cells* does this through two new virtualization mechanisms, device namespaces and device namespace proxies, that leverage a foreground-background usage model to isolate and multiplex mobile device hardware with near zero overhead. Device namespaces provide a kernel-level abstraction that is used to virtualize critical hardware devices such as the framebuffer and GPU, as well as Android's complicated power management framework resulting in fully accelerated graphics in all personas and almost no extra power consumption for *Cells* compared to stock Android. *Cells* device namespace proxies provide a mechanism to virtualize closed and proprietary device infrastructure such as the telephony radio stack.

Second, we presented *Cycada*, the first system that can run unmodified iOS applications on non-Apple devices allowing users to maximize the benefit of iOS and Android on the same mobile device. *Cycada* accomplishes this through a novel combination of binary compatibility techniques including four new oper-

ating system compatibility mechanisms: duct tape, diplomatic functions, thread impersonation, and dynamic library replication.

Duct tape allows source code from a foreign kernel to be compiled, without modification, into a domestic kernel. This avoids the difficult, tedious, and error prone process of porting or implementing new foreign kernel subsystems.

Diplomatic functions leverage per-thread personas, and mediate foreign function calls into domestic libraries. This enables *Cycada* to support foreign libraries that are closely tied to foreign hardware by replacing library function calls with diplomats that utilize domestic libraries and hardware. From our in-depth study of iOS and Android graphics, we also presented four diplomat usage patterns: direct, indirect, data-dependent, and multi. Similar to the Gang of Four's design patterns, our diplomat usage patterns allow *Cycada* to quickly identify recurring solutions to binary compatibility problems. Diplomatic functions, and their systematic usage through our defined patterns, is essential on mobile devices to support graphics.

Thread impersonation allows thread-specific context to be shared amongst multiple threads enabling one thread to temporarily assume the persona of another thread. *Cycada* thread impersonation presents a generalized mechanism to impersonate a thread across *all* personas in which the thread may execute. In each of these personas, a subset of thread-specific data may be used or shared by the impersonating thread. This is key to supporting multi-threaded iOS graphics rendering on Android devices.

Dynamic library replication allows a dynamic linker to load multiple, independent instances of a single dynamic library. Each instance, or replica, is linked to its own independent set of dependent library replicas. Dynamic library replication allows *Cycada* to bypass arbitrary restrictions in proprietary libraries that require singleton instances of key system components. This is key to supporting iOS graphics window management on Android devices.

Finally, because mobile computing has become increasingly important, we presented a series of hands-on Android Linux kernel programming projects designed to immerse students in a mobile computing environment while simultaneously teaching core OS principles. The projects progressively introduce OS principles and mobile computing, and implicitly teach students about real-world embedded device development. We use key aspects of modern mobile devices, such as orientation sensors, to enrich the hands-on experience and increase student engagement.

7.2 Future Work

My research has demonstrated the feasibility of lightweight multi-persona mobile computing, however, there are still important areas of research which should be explored in future work. First, *Cycada* OS compatibility does not present a complete security solution. Because mobile devices are an extremely personal method of computing, users have begun to keep immense amounts of private data on their smartphones and tablets. Further research is necessary to map the security semantics provided by iOS onto the security semantics provided by Android.

Second, while *Cells* allows users to maintain multiple virtual phones on the same device, it requires each of these phones to use the same OS kernel. By combining *Cells* lightweight OS virtualization with *Cycada* OS compatibility, a user could securely run an entire isolated, virtual iOS device on their Android smartphone or tablet. This avenue of research would further integrate these two mobile platforms, and potentially give future mobile device users a more intuitive way of interacting with multiple personas on their single mobile device by maintaining UI-consistency within a virtual phone or virtual tablet.

Finally, my work on the *Cycada* project revealed that despite the fact that OSes maintain similar conceptual functionality, their implementations can be radically different, often with surprising performance implications. Modern mobile operating systems are not only vertically integrated in user space software, but this vertical integration also extends into the kernel. Opaque, proprietary kernel devices, drivers, and whole subsystems are common on both iOS and Android. This trend is in direct opposition to the vision statement of the OpenGroup, "Boundaryless Information Flow ^{TM} achieved through global interoperability in a secure, reliable and timely manner." To achieve their vision, the OpenGroup has many different standards including the POSIX standard operating system interface. It should be clear that proprietary OS interfaces, especially those communicating to proprietary hardware, fall outside the scope of the POSIX standard. In order to better characterize how modern mobile platforms actually define and use the operating system interface, I propose a detailed study of the use of the POSIX API on both Android and iOS. This would be the first step in a broader investigation which would attempt to more closely align the design patterns and usage model of real-world mobile platforms with standard operating system interfaces.

7.3 Final Thoughts

My work has studied two mobile platforms: iOS and Android. Both platforms provide users with ubiquitous, convenient computing platforms on which they increasingly depend. Both platforms provide arguably similar user interfaces and manage similar hardware resources. However, the design and implementation of both platforms is wildly different. The vertically integrated nature of iOS has allowed its OS designers to freely move functionality in and out of the kernel, and optimize user experience metrics through targetted low-level features and subsystems such as pthread workqueues (Grand Central Dispatch), kqueue Mach IPC message reception, and IOSurface memory management (see Section 5.2. Although iOS maintains POSIX compliance, it is clear through the proliferation of user space daemons and extensive use of non-standard OS mechanisms such as Mach IPC, that a standardized OS interface has taken a lower priority to user experience, and raw performance.

Following the development of Android in particular, it has been fascinating to watch the drift away from interpreted code. For various reasons, Google chose to use Java as the primary platform on which to build Android, but as the project and platform has matured it has drifted decidedly away from Java principles. More and more native code is put into JNI accessible functions for performance reasons, and the advent of Google's successor to Dalvik, ART (the Android RunTime), completely bypasses the virtual machine in favor of direct compilation to native code.

Android chose to leverage the Linux kernel while still attempting to provide third party vendors with enough security to promulgate the platform into near ubiquity. While this allows vendors to develop an entire platform relatively quickly, it has led to different challenges at the OS level. Specifically, the interface to the Linux kernel is much more rigid than Apple's XNU interface. While Google is free to add or remove functionality as they see fit, if they want to maintain good standing in the Linux community and continue to leverage the massive open source development effort surrounding the kernel, they need to respect the processes set in place by the community. Coding standards and review practices for submitting patches to the upstream Linux Kernel are rigorous, time consuming, and potentially political. High profile examples such as suspend blockers (wake locks), and Binder IPC, show that being nimble in the mobile Linux kernel world is difficult. The results of this are two fold: Android kernel development is completely fragmented - each device manufacturer maintains their own development tree, Google maintains some kernel repositories, and bugifxes and patches from upstream may only partially make their way into any given branch / repo /

tree. Second, maintaining a hybrid open source / proprietary development model has lead to avoidable headaches and inefficiencies at the OS level. Specifically subsystems such as graphics, wifi, and GPS where manufacturers provide binary blobs that are read by carefully crafted bits of GPL code inserted into the Linux kernel. Simple things such as reading acclerometer data or passing touch input coordinates to an application result in many many kernel-user space boundary crossings, and a code path that's almost impossible to follow.

Bibliography

[1] ALEXA INTERNET, INC. Alexa - Top Sites in United States. `http://www.alexa.com/topsites/countries/US`, Apr. 2014. Accessed: 05/01/2014.

[2] AMSTADT, B., AND JOHNSON, M. K. Wine. *Linux Journal 1994*, 4es (Aug. 1994).

[3] ANDERSON, C. L., AND NGUYEN, M. A Survey of Contemporary Instructional Operating Systems for use in Undergraduate Courses. *Journal of Computing Sciences in Colleges 21* (October 2005), 183–190.

[4] ANDRUS, J., DALL, C., VAN'T HOF, A., LAADAN, O., AND NIEH, J. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the* 23^{rd} *ACM Symposium on Operating Systems Principles* (Cascais, Portugal, Oct. 2011), pp. 173–187.

[5] ANDRUS, J., AND NIEH, J. Teaching Operating Systems Using Android. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Feb. 2012), SIGCSE '12, pp. 613–618.

[6] ANDRUS, J., VAN'T HOF, A., ALDUAIJ, N., DALL, C., VIENNOT, N., AND NIEH, J. Cider: Native Execution of iOS Apps on Android. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2014), ASPLOS '14, pp. 367–382.

[7] 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.

[8] APPLE, INC. Components of the System Configuration Framework. `https://developer.apple.com/library/mac/#documentation/Networking/Conceptual/`

SystemConfigFrameworks/SC_Components/SC_Components.html, Feb. 2006. Accessed: 3/22/2013.

[9] APPLE, INC. dyld(1) OSX Manual Page. http://developer.apple.com/library/mac/#documentation/Darwin/Reference/Manpages/man1/dyld.1.html, Dec. 2009. Accessed: 3/31/2013.

[10] 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.

[11] APPLE, INC. Networking & Internet Starting Point. http://developer.apple.com/library/ios/#referencelibrary/GettingStarted/GS_Networking_iPhone/index.html, Apr. 2011. Accessed: 3/22/2013.

[12] 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.

[13] APPLE, INC. Source Browser. http://www.opensource.apple.com/source/xnu/xnu-2050.18.24/, Aug. 2012. Accessed: 3/21/2013.

[14] APPLE, INC. Source Browser. http://opensource.apple.com/source/IOKitUser/IOKitUser-755.18.10/, Aug. 2012. Accessed: 10/10/2014.

[15] APPLE, INC. Kernel Programming Guide: Mach Overview. https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html, Aug. 2013. Accessed: 10/12/2014.

[16] APPLE, INC. Networking Overview. http://developer.apple.com/library/ios/#documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/Introduction/Introduction.html, Jan. 2013. Accessed: 3/22/2013.

[17] 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: 12/04/2013.

[18] APPLE, INC. SCNetworkReachability Reference. https://developer.apple.com/library/mac/#documentation/SystemConfiguration/Reference/SCNetworkReachabilityRef/Reference/reference.html, Jan. 2013. Accessed: 3/22/2013.

[19] APPLE, INC. SunSpider 1.0.2 JavaScript Benchmark. https://www.webkit.org/perf/sunspider/sunspider.html, 2013. Accessed: 03/29/2013.

[20] APPLE, INC. iOS Device Compatibility Reference: OpenGL ES Graphics. https://developer.apple.com/library/ios/documentation/DeviceInformation/Reference/iOSDeviceCompatibility/OpenGLESPlatforms/OpenGLESPlatforms.html#//apple_ref/doc/uid/TP40013599-CH106-SW1, Feb. 2014. Accessed: 04/27/2014.

[21] APPLE, INC. OpenGL ES Programming Guide for iOS: Best Practices for Working with Vertex Data. https://developer.apple.com/library/ios/documentation/3ddrawing/conceptual/opengles_programmingguide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html, July 2014. Accessed: 10/12/2014.

[22] APPLE, INC. OpenGL ES Programming Guide for iOS: Drawing to Other Rendering Destinations. https://developer.apple.com/library/ios/documentation/3ddrawing/conceptual/opengles_programmingguide/WorkingwithEAGLContexts/WorkingwithEAGLContexts.html, Mar. 2014. Accessed: 04/24/2014.

[23] APPLE, INC. The WebKit Open Source Project. http://www.webkit.org/, Apr. 2014. Accessed: 04/30/2014.

[24] APPLE, INC. Xcode Overview: About Xcode. https://developer.apple.com/library/mac/documentation/ToolsLanguages/Conceptual/Xcode_Overview/About_Xcode/about.html, Mar. 2014. Accessed: 10/12/2014.

[25] APPLE KERNEL ENGINEER. Personal communication, Mar. 2014.

[26] ASTERISK, 2011. `http://www.asterisk.org`.

[27] ATKIN, B., AND SIRER, E. G. PortOS: An Educational Operating System for the Post-PC Environment. In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2002), SIGCSE '02, ACM, pp. 116–120.

[28] BARR, K., BUNGALE, P., DEASY, S., GYURIS, V., HUNG, P., NEWELL, C., TUCH, H., AND ZOPPIS, B. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *ACM SIGOPS Operating Systems Review 44*, 4 (Dec. 2010), 124–135.

[29] BAUMANN, A., LEE, D., FONESCA, P., GLENDENNING, L., LORCH, J. R., BOND, B., OLINSKY, R., AND HUNT, G. C. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, Apr. 2013), pp. 239–252.

[30] BELLARD, F. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), USENIX Association, pp. 41–46.

[31] BEN BOWMAN AND YUAN WANG AND BENJ LIPCHAK. GL_OES_vertex_array_object. `https://www.khronos.org/registry/gles/extensions/OES/OES_vertex_array_object.txt`, Apr. 2009. Accessed: 10/12/2014.

[32] BHATTIPROLU, S., BIEDERMAN, E. W., HALLYN, S., AND LEZCANO, D. Virtual Servers and Checkpoint/Restart in Mainstream Linux. *ACM SIGOPS Operating Systems Review 42*, 5 (July 2008), 104–113.

[33] BLACK DUCK SOFTWARE, INC. WebKit Open Source Project on Ohloh. `http://www.ohloh.net/p/WebKit`, Apr. 2014. Accessed: 04/30/2014.

[34] BLAZAKIS, D. The Apple Sandbox. In *Blackhat DC* (Jan. 2011).

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

[36] BRYLOW, D. An Experimental Laboratory Environment for Teaching Embedded Operating Systems. In *Proceedings of the 39th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2008), SIGCSE '08, ACM, pp. 192–196.

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

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

[39] CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. FX!32: A Profile-Directed Binary Translator. *IEEE Micro 18*, 2 (Mar. 1998), 56–64.

[40] CONNELLY, D., BALL, T., AND STANGER, K. j2objc - A Java to iOS Objective-C translation tool and runtime. - Google Project Hosting. `https://code.google.com/p/j2objc/`. Accessed: 7/23/2013.

[41] COX, R., FREY, C., YU, X., ZELDOVICH, N., AND CLEMENTS, A. Xv6 – A Simple Unix-like Teaching Operating System. `http://pdos.csail.mit.edu/6.828/xv6/`.

[42] DALL, C., ANDRUS, J., VAN'T HOF, A., LAADAN, O., AND NIEH, J. The Design, Implementation, and Evaluation of Cells: A Virtual Smartphone Architecture. *ACM Transactions on Computer Systems (TOCS) 30*, 3 (Aug. 2012), 9:1–9:31.

[43] DALL, C., AND NIEH, J. KVM for ARM. In *Proceedings of the Ottawa Linux Symposium* (Ottawa, Canada, June 2010).

[44] DALL, C., AND NIEH, J. 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), ASPLOS '14.

[45] DELOITTE DEVELOPMENT, LLC. Deloitte Predictions for the Technology, Media and Telecommunications Sector, 2011. `http://www.deloitte.com/us/telecompredictions2011`.

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

[47] DOWTY, M., AND SUGERMAN, J. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review 43* (July 2009), 73–82.

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

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

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

[51] Enterproid, Inc., 2011. http://www.enterproid.com.

[52] 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.

[53] FILIP PIZLO. Surfin' Safari - Blog Archive - Introducing the WebKit FTL JIT. https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit/, May 2014. Accessed: 02/03/2015.

[54] FREE SOFTWARE FOUNDATION. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/, Dec. 2013. Accessed: 12/10/2013.

[55] FREEBSD DOCUMENTATION PROJECT. Linux Binary Compatibility. In *The FreeBSD Handbook 3rd Edition, Vol.1: User Guid*, B. N. Handy, R. Murphey, and J. Mock, Eds. FreeBSD Mall, Mar. 2004, ch. 11.

[56] GAMME, E., JOHNSON, R., HELM, R., AND JOHN, V. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Boston, MA, USA, Oct. 1994.

[57] GEOFF STAHL. GL_APPLE_fence. https://www.opengl.org/registry/specs/APPLE/fence.txt, Aug. 2002. Accessed: 04/24/2014.

[58] GOOGLE. Nexus One - Google Phone Gallery, May 2011. http://www.google.com/phone/detail/nexus-one.

[59] GOOGLE. Nexus S - Google Phone Gallery, May 2011. http://www.google.com/phone/detail/nexus-s.

[60] GOOGLE INC. Google Voice, Feb. 2011. http://www.google.com/googlevoice/about.html.

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

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

[63] HESS, R., AND PAULSON, P. Linux Kernel Projects for an Undergraduate Operating Systems Course. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2010), SIGCSE '10, ACM, pp. 485–489.

[64] HEX-RAYS SA. IDA: About. https://www.hex-rays.com/products/ida/, Jan. 2014. Accessed: 04/24/2014.

[65] HFS ~ HTTP File Server, 2011. http://www.rejetto.com/hfs/.

[66] HILLS, M. Android on OKL4. http://www.ertos.nicta.com.au/software/androidokl4/.

[67] HOHENSEE, P., MYSZEWSKI, M., AND REESE, D. Wabi CPU Emulation. In *Hot Chips 8* (Palo Alto, CA, Aug. 1996).

[68] HUNT, G. C., AND BRUBACHER, D. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium* (Seattle, WA, July 1999).

[69] HWANG, J., SUH, S., HEO, S., PARK, C., RYU, J., PARK, S., AND KIM, C. 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* (Las Vegas, NV, Jan. 2008), pp. 257–261.

[70] IMAGINATION TECHNOLOGIES LTD. PowerVR Series 5 SGX Architecture Guide for Developers, Nov. 2011.

[71] JEFF JULIANO AND GARY KING AND JON LEECH AND JONATHAN GRANT AND BARTHOLD LICHTENBELT AND AAFTAB MUNSHI AND ACORN POOLEY AND CHRIS WYNN. EGL_KHR_image_base. https://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_image_base.txt, June 2013. Accessed: 12/04/2013.

[72] JOHN ROSASCO AND ANDREW BARNES. GL_APPLE_row_bytes. http://www.opengl.org/registry/specs/APPLE/row_bytes.txt, Oct. 2006. Accessed: 04/24/2014.

[73] JOHN SPITZER AND MARK KILGARD AND ACORN POOLEY. GL_NV_fence. https://www.khronos.org/registry/gles/extensions/NV/fence.txt, Dec. 2008. Accessed: 04/24/2014.

[74] 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.

[75] KHRONOS GROUP. OpenGL Extensions – OpenGL.org, 2011. http://www.opengl.org/wiki/OpenGL_Extensions.

[76] 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: 12/04/2013.

[77] KHRONOS GROUP. OpenGL ES – The Standard for Embedded Accelerated 3D Graphics. http://www.khronos.org/opengles/, Jan. 2013. Accessed: 3/22/2013.

[78] KISZKA, J., PEMMASANI, G., AND FUCHS, P. SourceForge.net: ndiswrapper. http://ndiswrapper.sourceforge.net/. Accessed: 7/23/2013.

[79] KOLYSHKIN, K. Recent Advances in the Linux Kernel Resource Management, 2011. http://www.cse.wustl.edu/~lu/control-tutorials/im09/slides/virtualization.pdf.

[80] LAADAN, O., BARATTO, R., PHUNG, D., POTTER, S., AND NIEH, J. DejaView: A Personal Virtual Computer Recorder. In *Proceedings of the* 21^{st} *Symposium on Operating Systems Principles* (Stevenson, WA, Oct. 2007).

[81] LAADAN, O., NIEH, J., AND VIENNOT, N. Teaching Operating Systems Using Virtual Appliances and Distributed Version Control. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Mar. 2010), SIGCSE '10, pp. 480–484.

[82] LAADAN, O., NIEH, J., AND VIENNOT, N. Structured Linux Kernel Projects for Teaching Operating Systems Concepts. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Mar. 2011), SIGCSE '11, pp. 287–292.

[83] LAGAR-CAVILLA, H. A., TOLIA, N., SATYANARAYANAN, M., AND DE LARA, E. VMM-independent Graphics Acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (New York, NY, USA, 2007), VEE '07, ACM, pp. 33–43.

[84] LAMPSON, B. W. Hints for Computer System Design. *SIGOPS Operating Systems Review 17*, 5 (Oct. 1983), 33–48.

[85] LAWSON, B., AND BARNETT, L. Using iPodLinux in an Introductory OS Course. In *Proceedings of the 39th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2008), SIGCSE '08, ACM, pp. 182–186.

[86] LIU, H., CHEN, X., AND GONG, Y. BabyOS: A Fresh Start. In *Proceedings of the 38th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2007), SIGCSE '07, ACM, pp. 566–570.

[87] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. K. High Performance VMM-bypass I/O in Virtual Machines. In *Proceedings of the 2006 USENIX Annual Technical Conference* (Boston, MA, June 2006).

[88] LXC LINUX CONTAINERS, 2011. `http://linuxcontainers.org/`.

[89] MICROSOFT. About the Wireless Hosted Network, 2011. `http://msdn.microsoft.com/en-us/library/dd815243(v=vs.85).aspx`.

[90] MOBILE SYSTEMS. Office Suite Pro (Trial) – Android Market, 2011. https://market.android.com/details?id=com.mobisystems.editor.office_with_reg.

[91] NIEH, J., AND VAILL, C. Experiences Teaching Operating Systems Using Virtual Platforms and Linux. In *Proceedings of the 36th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2005), SIGCSE '05, ACM, pp. 520–524.

[92] NIEUWEJAAR, N., SCHROCK, E., KUCHARSKI, W., BLAINE, R., PILATOWICZ, E., AND LEVENTHAL, A. 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/.

[93] NVIDIA CORPORATION. NVIDIA SLI MultiOS, Feb. 2011. http://www.nvidia.com/object/sli_multi_os.html.

[94] NVIDIA CORPORATION. High Performance Computing (HPC) and Supercomputing — NVIDIA Tesla — NVIDIA. http://www.nvidia.com/object/tesla-supercomputing-solutions.html, 2014. Accessed: 05/01/2014.

[95] NVIDIA CORPORATION. Shared Virtual GPU (vGPU) Technology — NVIDIA. http://www.nvidia.com/object/virtual-gpus.html, 2014. Accessed: 05/01/2014.

[96] OKAJIMA, J. R. AUFS, 2011. http://aufs.sourceforge.net/aufs2/man.html.

[97] OPEN KERNEL LABS. OKL4 Microvisor. http://www.ok-labs.com/products/okl4-microvisor, Mar. 2011. Accessed: 9/10/2011.

[98] OPENINTENTS. SensorSimulator – openintents – Sensor Simulator for simulating sensor data in real time. – Make Android applications work together. – Google Project Hosting. http://code.google.com/p/openintents/wiki/SensorSimulator.

[99] OPENVZ, 2011. http://openvz.org/Main_Page.

[100] 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.

[101] ORACLE CORPORATION. Consolidating Applications with Oracle Solaris Containers. http://www.oracle.com/technetwork/server-storage/solaris/documentation/consolidating-apps-163572.pdf, July 2011. Accessed: 12/05/2013.

[102] 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.

[103] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The Design and Implementation of Zap: a System for Migrating Computing Environments. In *Proceedings of the* 5^{th} *Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002).

[104] PARALLELS IP HOLDINGS GMBH. Parallels Desktop. http://www.parallels.com/products/desktop/. Accessed: 3/22/2013.

[105] 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.

[106] 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.

[107] PFAFF, B., ROMANO, A., AND BACK, G. The Pintos Instructional Operating System Kernel. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2009), SIGCSE '09, ACM, pp. 453–457.

[108] PHONEGAP. PhoneGap — Home. http://phonegap.com/. Accessed: 7/23/2013.

[109] POLARBIT. Reckless Racing – Android Market, 2011. https://market.android.com/details?id=com.polarbit.RecklessRacing.

[110] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, CA, Mar. 2011), pp. 291–304.

[111] PRINTING COMMUNICATIONS ASSOC., I. NDIS Developer's Reference. `http://www.ndis.com/`. Accessed: 7/24/2013.

[112] RASHID, R., TEVANIAN, A., YOUNG, M., GOLUB, D., BARON, R., BLACK, D., BOLOSKY, W., AND CHEW, J. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *ACM SIGOPS Operating Systems Review 21*, 4 (Oct. 1987), 31–39.

[113] RED BEND SOFTWARE. VLX Mobile Virtualization, 2011. `http://www.redbend.com`.

[114] ROVIO MOBILE LTD. Angry Birds – Android Market, 2011. `https://market.android.com/details?id=com.rovio.angrybirds`.

[115] SHANKLAND, S., AND FRIED, I. The Brains behind Apple's Rosetta: Transitive – CNET News. `http://news.cnet.com/The-brains-behind-Apples-Rosetta-Transitive/2100-1016_3-5736190.html`, June. Accessed: 12/27/2013.

[116] STEGMAIER, S., MAGALLÓN, M., AND ERTL, T. A Generic Solution for Hardware-Accelerated Remote Visualization. In *Proceedings of the Symposium on Data Visualisation 2002* (Aire-la-Ville, Switzerland, Switzerland, 2002), VISSYM '02, Eurographics Association, pp. 87–ff.

[117] SU, G. *MOVE: Mobility with Persistent Network Connections*. PhD thesis, Columbia University, Oct. 2004.

[118] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference* (Boston, MA, June 2001).

[119] TANENBAUM, A. S. A UNIX Clone with Source Code for Operating Systems Courses. *SIGOPS Operating Systems Review 21* (January 1987), 20–29.

[120] THE ANDROID OPENSOURCE PROJECT. Graphics — Android Developers. `https://source.android.com/devices/graphics.html`, 2013. Accessed: 04/23/2014.

[121] THE ANDROID OPENSOURCE PROJECT. Dashboards — Android Developers. `http://developer.android.com/about/dashboards/index.html`, Apr. 2014. Accessed: 04/27/2014.

[122] THE VIRTUALGL PROJECT. VirtualGL — Main / The VirtualGL Project. http://www.virtualgl.org/, May 2014. Accessed: 05/01/2014.

[123] TIAN, K. Graphics Virtualization (XenGT) — 01.org. https://01.org/xen/blogs/srclarkx/2013/graphics-virtualization-xengt, 2014. Accessed: 05/01/2014.

[124] TORUS KNOT SOFTWARE LTD. OGRE - Open Source 3D Graphics Engine. http://www.ogre3d.org/, May 2014. Accessed: 10/12/2014.

[125] TUNG, C. K. 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.

[126] VMWARE, INC. VMware Workstation. http://www.vmware.com/products/workstation/, 2011. Accessed: 3/22/2013.

[127] WALDSPURGER, C. A. Memory Resource Management in VMware ESX Server. In *Proceedings of the* 5^{th} *Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002).

[128] WALL, L., CHRISTIANSEN, T., AND SCHWARTZ, R. L. *Programming Perl*. O'Reilly Media, Sebastopol, CA, USA, Oct. 1996.

[129] WEB STANDARDS PROJECT. The Acid3 Test. http://www.acidtests.org/, Mar. 2008. Accessed: 05/01/2014.

[130] WEBKIT COMMUNITY. Bug 24986 – [multi-patch] ARM JIT port. https://bugs.webkit.org/show_bug.cgi?id=24986, June 2009. Accessed: 02/03/2015.

[131] WEINTRAUB, S. Industry First: Smartphones Pass PCs in Sales – Goole 24/7 – Fortune Tech. http://tech.fortune.cnn.com/2011/02/07/idc-smartphone-shipment-numbers-passed-pc-in-q4-2010, Feb. 2011. Accessed: 03/01/2011.

[132] WIKIPEDIA. Mobile Device – Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Handheld_device.

[133] WORKLIGHT, INC. WorkLight Mobile Platform, 2011. http://www.worklight.com.

[134] WOYKE, E. iPhone and Android Apps 101. http://www.forbes.com/2008/11/11/mobile-apps-colleges-tech-wire-cx_ew_1111mobileapps.html.

[135] WRIGHT, C. P., DAVE, J., GUPTA, P., KRISHNAN, H., QUIGLEY, D. P., ZADOK, E., AND ZUBAIR, M. N. Versatility and Unix Semantics in Namespace Unification. *ACM Transactions on Storage 2*, 1 (Feb. 2006), 74–105.

[136] WYSOCKI, R. J. An Alternative to Suspend Blockers, 2011. http://lwn.net/Articles/416690/.

[137] WYSOCKI, R. J. Technical Background of the Android Suspend Blockers Controversy. http://lwn.net/images/pdf/suspend_blockers.pdf, 2011.

[138] XEN PROJECT. Architecture for Split Drivers Within Xen, 2011. http://wiki.xensource.com/xenwiki/XenSplitDrivers.

[139] X.ORG FOUNDATION. GLX. http://dri.freedesktop.org/wiki/GLX/, Apr. 2013. Accessed: 05/01/2014.

[140] YEECCO, LTD. www.yeeccoo.com. http://http://www.yeecco.com/stella. Accessed: 6/27/2013.

[141] YOUNG, M., TEVANIAN, A., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J. J., BOLOSKY, W. J., BLACK, D., AND BARON, R. 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* (Austin, TX, Nov. 1987), ACM, pp. 63–76.

[142] ZDNET. Stolen Apps that Root Android, Steal Data and Open Backdoors Available for Download from Google Market, 2011. http://zd.net/gGUhOo.