Chameleon: Multi-Persona Binary Compatibility for Mobile Devices

Jeremy Andrus, Alexander Van't Hof, Naser AlDuaij, Christoffer Dall, Nicolas Viennot, and Jason Nieh
{jeremya, alexvh, alduaij, cdall, nviennot, nieh}@cs.columbia.edu
Department of Computer Science
Columbia University

Technical Report CUCS-011-13
April 2013

Abstract

Mobile devices are vertically integrated systems that are powerful, useful platforms, but unfortunately limit user choice and lock users and developers into a particular mobile ecosystem, such as iOS or Android. We present Chameleon, a multi-persona binary compatibility architecture that allows mobile device users to run applications built for different mobile ecosystems together on the same smartphone or tablet. Chameleon enhances the domestic operating system of a device with *personas* to mimic the application binary interface of a foreign operating system to run unmodified foreign binary applications. To accomplish this without reimplementing the entire foreign operating system from scratch, Chameleon provides four key mechanisms. First, a multi-persona binary interface is used that can load and execute both domestic and foreign applications that use different sets of system calls. Second, compile-time code adaptation makes it simple to reuse existing unmodified foreign kernel code in the domestic kernel. Third, API interposition and passport system calls make it possible to reuse foreign user code together with domestic kernel facilities to support foreign kernel functionality in user space. Fourth, schizophrenic processes allow foreign applications to use domestic libraries to access proprietary software and hardware interfaces on the device. We have built a Chameleon prototype and demonstrate that it imposes only modest performance overhead and can run iOS applications from the Apple App Store together with Android applications from Google Play on a Nexus 7 tablet running the latest version of Android.

1 Introduction

Mobile devices such as tablets and smartphones are changing the way that computing platforms are designed, from the separation of hardware and software concerns in the traditional PC world, to vertically integrated platforms. Hardware components are integrated together in compact devices using non-standard interfaces. Software is customized for the hardware, often using proprietary libraries to interface with specialized hardware. Applications are tightly integrated with particular 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 applications 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 [15]. 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 a massive 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 application base of existing platforms. New platforms without an enormous pool of user applications face the difficult, if not impossible, task of end user adoption, creating huge barriers to entry into the mobile device market.

In the traditional PC world, users can use virtual machines (VMs) such as VMWare Workstation [29] or Parallels Desktop [25] to run applications intended for one platform on a different platform. While VMs are useful for desktop and server computers, 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 an application 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 using non-standard interfaces, such as GPUs. VMs provide no effective mechanism to enable applications to directly leverage these hardware device features, severely limiting performance and making existing VM-based approaches unusable on smartphones and tablets.

To give users greater freedom of choice, make it possible for developers to write once and run on multiple platforms, reduce barriers to entry in the mobile device market, and help spur innovation by making huge application markets available on alternative platforms, we have created Chameleon. Chameleon is a binary compatibility architecture that allows users to run applications written and compiled for different mobile ecosystems simultaneously on the same smartphone or tablet. We introduce the notion of *foreign* binaries, those developed for another OS, the foreign OS, and *domestic* binaries, those developed for the given device's OS, the domestic OS. Chameleon defines a *persona* as an execution mode assigned to each thread on the system, identifying the thread as executing either a foreign or domestic binary. Chameleon supports multiple personas by extending the kernel's application binary interface (ABI) to be aware of both foreign and domestic threads.

Chameleon supports running foreign binaries by enhancing the domestic OS with four key mechanisms to leverage existing unmodified application frameworks and kernel code. First, Chameleon provides a *multi-persona binary interface*. For each persona, Chameleon provides a binary loader to interpret the actual contents of the application binary and associated libraries, a system call interface, and an asynchronous signal delivery mechanism that works with each persona. Chameleon can then load and execute foreign applications written for a different mobile ecosystem that uses a completely different kernel ABI. Second, Chameleon introduces *duct tape*, a novel compile-time code adaptation layer, that allows unmodified foreign kernel code to be directly compiled into the domestic kernel to provide foreign binaries with kernel services not otherwise present in the domestic kernel. Third, Chameleon introduces *API interposition* that allows foreign kernel services to be implemented in user space leveraging existing domestic kernel services via *passport system calls* to export the familiar foreign API to the foreign binary. Using these mechanisms, Chameleon can then execute foreign applications that require kernel-level services not available in the domestic OS. Fourth, Chameleon creates *schizophrenic processes* that can allow foreign applications to use domestic libraries to access proprietary software and hardware interfaces on the device. A schizophrenic process is a process that executes code using multiple binary personas. Because it is common for user space libraries on mobile devices to directly use or manage custom hardware resources, those libraries cannot be used on a different mobile device with different hardware. However, mobile devices typically have similar functionality implemented using different libraries and hardware. Using schizophrenic processes, Chameleon replaces the foreign library used by a foreign application with the existing domestic library used by domestic applications to provide fast and efficient direct access to underlying closed hardware, such as GPUs.

Using these mechanisms, we have built a Chameleon prototype for use on Android devices that can run unmodified iOS applications along with Android applications on the same device. Our approach leverages existing software infrastructure as much as possible, including using existing application frameworks across both iOS and Android ecosystems with no modification. We demonstrate the effectiveness of our prototype by running various iOS applications from the Apple App Store together with Android applications from Google Play on a Nexus 7 tablet running the latest version of Android. Users can interact with iOS applications using multi-touch input, and iOS applications can leverage GPU hardware to display smooth accelerated graphics. Using both system-level and application-level benchmarks, we demonstrate that Chameleon imposes only modest performance overhead and can yield faster performance on iOS applications than their Android counterparts on Android hardware because of the greater efficiencies of running native iOS code instead of Java as used by Android.

2 Overview of Android and iOS

To understand how Chameleon runs iOS applications on Android, we first provide a brief overview of the operation of Android and iOS. We limit our discussion to central components providing application startup, graphics, and input on both systems. A complete explanation of both systems is beyond the scope of this paper.

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

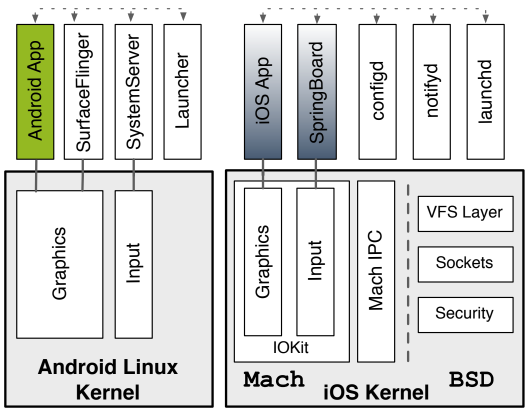


Figure 1: Android and iOS architectures

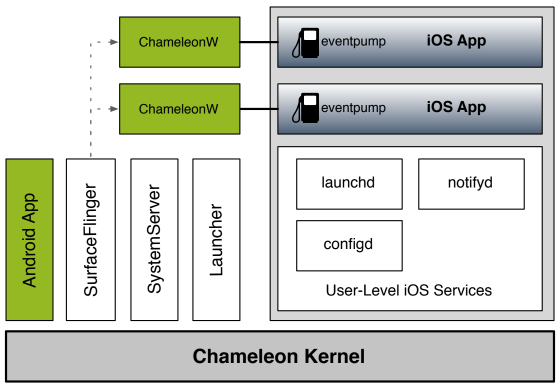


Figure 2: System Integration Overview

for different applications and display the final composed surface to the screen.

Each Android application is compiled into Dalvik's Dex bytecode format. Tapping the icon on the homescreen launches a separate Dalvik VM instance to run the application. When a user interacts with Android applications input events are delivered from the Linux device driver through the Android framework to the application. The application displays content by obtaining window memory in the form of a display surface from SurfaceFlinger and draws directly into the window memory. An application can attach an OpenGL context to the window memory and use the OpenGL ES framework to render high-end, animated graphics in the window memory using the GPU.

iOS runs on ARM CPUs like Android, but its software ecosystem is substantially different. iOS is built on the XNU kernel [6], a hybrid combination of a monolithic BSD kernel and a Mach microkernel running in a single kernel address space. XNU leverages the high performance BSD socket and VFS subsystems, but also benefits from the IPC mechanisms provided by Mach IPC. iOS makes extensive use of both the 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 applications. `launchd` starts Mach IPC services such as `configd`, the system configuration daemon, `notifyd`, the asynchronous notification server, and `mediaserverd`, the audio/video server. `launchd` also starts the SpringBoard application, which displays the iOS homescreen from which users can launch applications, handles and routes user input to applications, and uses the GPU to compose display surfaces for applications onto the screen. Springboard is analogous to an amalgamation of SurfaceFlinger, Launcher, and SystemServer in Android.

iOS applications are written in Objective-C and are compiled and run as native binaries in an extended `Mach-O` [3] format. This is in stark contrast to the Java archives used by Android applications, which are not loaded as native binaries, but interpreted by the Dalvik VM. On iOS, binaries are loaded directly by an OS level Mach-O loader, which reads the application binary, loads its text and data segments and jumps to the application entry point. Dynamically linked libraries are loaded by `dyld`, a user space binary, which is called from the Mach-O loader. Examples of frequently used libraries in iOS applications include UIKit, the user interface framework, QuartzCore and OpenGL ES, the core graphics frameworks, and WebKit, the Web browser engine.

3 Operational Model and System Integration

Chameleon provides a familiar user experience when running iOS applications on Android. Applications are launched from the Android home screen, just like any other Android application, and users can switch seamlessly between domestic Android applications and foreign iOS applications. Chameleon accomplishes this without running the iOS kernel or the main SpringBoard application, but instead overlays a file system (FS) hierarchy on the existing Android FS, allowing iOS applications to access familiar iOS paths, and provides background user-level services required by iOS applications. The background user-level services are not visible to users, but establish key OS level functionality in user space necessary to run iOS applications. Since we focus on running iOS applications on Android, we use the terms foreign and iOS, and domestic and Android, interchangeably to help describe Chameleon. Figure 2 shows an overview of the integration of iOS functionality into our Android-based Chameleon prototype.

Chameleon overlays the iOS FS hierarchy onto the existing Android FS, allowing iOS to access files using existing iOS-specific FS paths, while at the same time maintaining the domestic Android FS hierarchy, allowing domestic applications and services to function uninterrupted. Chameleon supports the use of personas to isolate and switch between foreign and domestic views of the FS using familiar techniques such as `chroot` and FS unioning [30], but because there are no naming conflicts between iOS and Android, our prototype implementation avoids the overhead of such approaches and simply creates the iOS FS entries inside the Android FS. Chameleon sets up subdirectories for running iOS applications, including */Applications*, */Documents*, */Library*, and */System/Library*.

iOS applications running on Chameleon need access to a number of application framework components including libraries and user-level services. Instead of re-implementing 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 Chameleon's binary compatibility architecture. However, not all libraries can simply be copied from the iOS device due to optimizations on the iOS device discussed in Section 6. Therefore, we combined binaries from an iOS device with binaries from the Xcode SDK, Apple's development environment. Background user-level services such as `launchd`, `configd`, and `notifyd` were copied from the iOS device, where core framework libraries were copied from the Xcode SDK.

Chameleon fully supports Apple's I/O Kit kernel-level driver support framework as part of the Chameleon kernel, described in more detail in Section 4. I/O Kit works seamlessly with existing Android device drivers to access the domestic device hardware resources. Services such as `launchd` require functional I/O Kit and kernel-level Mach IPC services, also supported by Chameleon, to start and function properly.

However, `launchd` assumes it is the first application on the system and therefore has a PID of 1. It further relies on iOS-specific OS services to be notified of available applications in the file system. Chameleon leverages PID virtualization [24] in the form of namespaces to allow `launchd` to run with a perceived PID of 1, despite it not being the first process on the system. We complement Chameleon's seamless integration of PID namespaces with imported foreign Mach IPC functionality for fully compliant IPC mechanisms.

Since the Android application environment differs substantially from iOS, Chameleon provides some user-level integration mechanisms to ensure a familiar user experience. For example, in iOS, where `launchd` detects new applications and ensures that they appear in SpringBoard, Chameleon introduces *chameleond*, which leverages Linux's existing inotify framework to detect new iOS applications, extract the application icon from the foreign application package, and notify the Android Launcher of the new application. Launcher then places the iOS icon on the Android homescreen, as shown in Figure 6a.

Because the application launch procedure and the binary format of iOS and Android applications are completely different, iOS applications cannot simply be started by the Android Launcher. To provide seamless system integration, and to keep changes to Android core user space components to a minimum, Chameleon introduces a proxy service, *ChameleonW*, which integrates the execution of an application, from tapping an icon in the Android Launcher, to running a foreign binary and integrating with background user-level iOS services. *ChameleonW*, is a standard Android application that the Android Launcher can start directly. It receives events and can be managed by Launcher like any other Android application, providing an integrated user experience when running iOS applications. For example, the iOS application shows up under Android's recent activity list, and can be selected or terminated like a normal Android application. Our approach is also compatible with Android's security model, but a detailed discussion is beyond the scope of this paper. Chameleon launches a separate copy of *ChameleonW* for each iOS application running on the system. *ChameleonW* adapts to the respective iOS application it is running, providing the familiar application icon, and facilitates starting iOS binary applications by issuing Java Native Interface (JNI) calls to call the unmodified iOS `launchctl` command, which in turn requests that the `launchd` service run the iOS application, replicating the application launch mechanism on iOS.

To further provide seamless integration between the two systems, the iOS application must be notified about Android system events, such as the application going into the background, the display rotating, or simply to receive input when the application is active. Because *ChameleonW* is a standard Android application, it will automatically be notified of such events by the Android framework, but it must forward this info to the iOS application. We accomplish this last step by leveraging Chameleon's API interposition to hook into the iOS application startup procedure at a known point and launch a per-application thread, the *eventpump*. The *eventpump* runs in the context of the iOS application and receives events from the Android *ChameleonW* application over a standard BSD socket, a mechanism ubiquitously available on both Android and iOS. *ChameleonW* can then send events via `eventpump` to the iOS application to direct its execution, such as touch input, display rotation, accelerometer, proximity, and volume change events. In fact, *ChameleonW* will even use `eventpump` to notify the iOS application that it is headed into the background, so that the application can save any necessary state and prepare for imminent termination, just as it would when running on an iPhone or iPad.

4 Architecture

To support its operational model, Chameleon needs to be able to run unmodified iOS binaries on Android, including iOS applications, frameworks, and services. This is challenging because iOS binaries are built to run on iOS, not Linux or Android. iOS provides a different system call interface from Linux, and iOS binaries make extensive use of OS services not available on Linux, such as Mach IPC [5].

At a high-level, the solution to providing binary compatibility is straightforward. The interaction between applications and an OS is defined by the kernel application binary interface (ABI). The ABI consists of the binary loader which interprets the physical contents of the application binary and associated libraries, asynchronous signal delivery, and the system call interface. To run iOS binaries, we just need to implement the iOS ABI in Android. However, there are two key challenges. First, iOS is a complex, closed-source system and simply reimplementing the system call interface from scratch would be a tedious, difficult, and error-prone process. Second, the behavior of some system calls is not well-defined. For example, the `ioctl` system call passes in a device-specific request code and a pointer to memory, and its behavior is device-specific. Without any understanding of how ioctls are used, simply implementing the system call itself is of little benefit to applications.

To address these problems, Chameleon provides a multi-persona binary compatibility architecture that provides two key features. First, it provides mechanisms that make it simple to use existing kernel and user code from foreign sources and incorporate them in Android. Although iOS is closed source, it is built on the XNU kernel, which is open source, and there are a variety of open source projects based on XNU. Chameleon makes it possible to import the code from XNU and related projects to into Android to implement substantial portions of the iOS ABI without a substantial reimplementation effort. Second, Chameleon introduces a fundamentally new capability, schizophrenic processes. This mechanism supports iOS applications that use closed iOS libraries which issue device-specific system calls to access iOS proprietary hardware. By replacing those libraries with Android libraries, which provide access

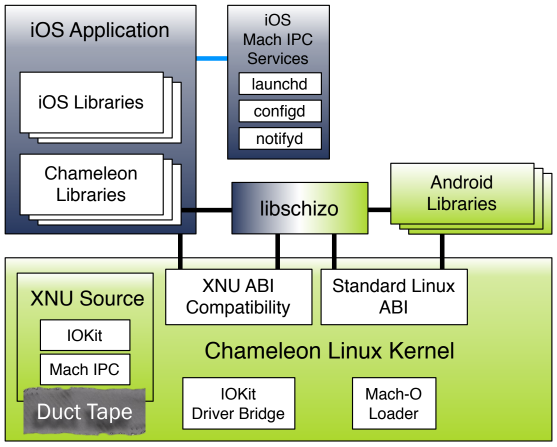


Figure 3: Overview of Chameleon architecture

to Android proprietary hardware, unmodified iOS applications can access Android hardware. This mechanism is based on the fact that closed libraries and closed hardware must rely on open interfaces for applications to use them. Rather than operating on opaque system calls, Chameleon operates at a higher level to leverage the same open interfaces used by applications to provide the required functionality.

Figure 3 provides an overview of the Chameleon architecture, which can be thought of in terms of four components. First, Chameleon provides XNU ABI compatibility by implementing a Mach–O loader for the Linux kernel, the XNU system call interface, and supporting signal delivery. For most system calls, a simple wrapper is implemented which maps arguments from XNU structures to Linux structures, then calls the corresponding Linux system call or reuses existing Linux kernel APIs. Second, Chameleon provides a duct tape layer to import foreign kernel code to support system calls that require a core kernel subsystem not in Linux. Third, Chameleon provides an API interposition layer implemented as iOS libraries to support system calls that can be more easily implemented in user space by using existing foreign user code together with special Linux system calls. Finally, Chameleon provides schizophrenic processes using the `libschizo` iOS library to support applications that use closed iOS libraries which issue device-specific system calls such as `ioctl`. We describe these components in further detail in the following sections.

4.1 Kernel ABI

Before an application is executed, its binary format must be properly loaded and interpreted by the kernel. iOS uses the Mach–O format, not supported in Linux. Chameleon provides a Mach–O loader for the Linux kernel which is registered as a standard binary format handler within the kernel, allowing the kernel to automatically choose to use the Mach–O loader when an iOS binary is executed in user space. When the Chameleon loader launches a new process, it tags the process with a *persona* that is used by the kernel in all subsequent interactions with user space. Personas are tracked on a per thread basis, namely per `task_struct` in Linux, enabling an application process with multiple threads to support multiple personalities.

An application's primary interface to the kernel is through system calls. Each kernel may have a different system call entry and exit path, and a different set of system calls available. Chameleon uses the persona of a thread to select among different kernel ABIs. It maintains one or more system call dispatch tables for each persona, switches among them based on the persona of the calling thread and the system call number, and manages the entry and exit path differences through persona-tagged support functions. For example, the system call ABI on iOS is vastly different from that of Linux in that there are multiple categories of system calls, each with its own calling convention and error reporting mechanism. Chameleon is aware of the foreign kernel's calling conventions, and uses the persona of the calling thread to decide how to handle a system call upon entry to the kernel. Chameleon further translates the foreign calling convention to an internal Linux calling convention, making it possible to, for example, call existing Linux system call implementation functions. Similarly, Chameleon also examines a thread's persona on its return path to user space, and reports the exit status of the system call as per the thread's persona.

Because the iOS kernel is based on a POSIX-compliant BSD, most of the BSD system calls overlap with functionality already provided by the Linux kernel. For most system calls, a simple wrapper is implemented which maps arguments from XNU structures to Linux structures when necessary and then simply calls the Linux implementation. For BSD system calls that have no corresponding Linux system calls, but for which similar Linux functionality exists, the wrapper reuses existing Linux kernel functions to implement the respective system calls. For example, Chameleon implements the `posix_spawn` system call, which is a flexible method of starting a thread or new application, by leveraging the linux `clone` and `exec` system calls.

Chameleon uses the persona of a thread for proper signal delivery. While most OSes can deliver asynchronous signals to an application, the semantics and implementation can vary widely between OSes. When a thread returns to user space from the kernel, Chameleon uses the thread's persona to account for these semantic differences in signal delivery, including the user space signal delivery entry point and function parameters passed to the signal handler. Processes or threads of different personas can send and receive signals amongst one another.

4.2 Duct Tape

Some iOS system calls require a core subsystem that does not exist in the Linux kernel. A prime example of this is the Mach IPC mechanism used extensively by iOS applications. The Mach IPC interface is a rich and complicated API providing interprocess communication and memory sharing. Implementing such a subsystem from scratch in the Linux kernel would be a daunting task.

To address this problem, Chameleon introduces *duct tape*, a novel compile–time code adaptation layer that allows source code from a foreign kernel to be directly compiled into the domestic kernel. The duct tape layer 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 applications. We refer to the process of compiling foreign kernel code in a domestic kernel as *cross–kernel compilation*.

Chameleon successfully uses duct tape for three different subsystems from the XNU kernel [6] into Android's Linux kernel: BSD pthread support, Mach IPC, and Apple's I/O Kit device driver framework, the latter is discussed in Section 5.1. BSD pthread support in iOS differs substantially from Linux in how functionality is separated 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. This support is found in the `bsd/kern/pthread_support.c` file in the open source XNU code provided by Apple. Chameleon used duct tape to directly compile this file without modification.

The Mach IPC subsystem is significantly more complicated than BSD pthread support. It involves many different header files and a large collection of C files found in the `osfmk/ipc/` and `osfmk/kern/` XNU source directories. Chameleon uses duct tape to directly compile the majority of Mach IPC as a part of Linux, but some parts were designed to work in a significantly different kernel environment, and therefore had to be reimplemented. In particular, the XNU kernel stack defaults to 16KB, twice that of the Linux kernel, and the XNU Mach IPC code takes direct advantage of deep call stacks to access queuing structures recursively, which would not work in Linux. Therefore we reimplemented the queuing structures used in the XNU source to better fit with the Linux kernel environment.

4.3 API Interposition

Some iOS system calls require a core subsystem that does not exist in the Linux kernel and cannot be easily implemented using duct tape. For example, the XNU kernel provides the BSD kqueue and kevent notification mechanism. The implementation of this mechanism is distributed throughout the XNU kernel source so that it can notify user space of myriad kernel events. The mechanism is built on a single system call entry point, but data semantics to register for and receive notification of events are highly subsystem-specific. Adding support for this mechanism in Linux, would involve changes to almost all core subsystems of the kernel, and is therefore not an attractive approach. To address this problem, Chameleon uses API interposition to interpose between the foreign binary and the kernel, allowing a user space library to implement the intended notification mechanism on top of existing Linux functionality. Library interposition is as simple as configuring an environment variable to load a library, which overrides symbolic references. As it turns out, existing open source user level implementations of some iOS services are available and can be directly used with API interposition.

Because we are implementing foreign OS services in user space based on domestic kernel functionality, Chameleon needs access to a variety of domestic system calls, not available from a foreign persona. For example, we implement FS notification support using Linux's inotify functionality, but there are not inotify system calls in iOS. To address this issue, we introduce *passport* system calls. A passport system call is a new foreign system call in Chameleon, which allows threads with a foreign personality to indirectly call domestic system calls.

Chameleon successfully combines API interposition and passport system calls to provide two important iOS kernel subsystems: (1) The kqueue and kevent kernel notification mechanism, and (2) pthread workqueues, a set of APIs introduced by Apple that extend the pthread API. Chameleon implements kqueue and kevent system calls in the `libkqueue` library, based on the open source XDispatch project [22]. A number of bug fixes and semantic changes to the original XDispatch were incorporated into libkqueue to fully support iOS binaries.

The pthread workqueue API is a feature in the iOS kernel that extends basic pthread support with a number of features to support prioritized workqueue threads. Pthread workqueues are used heavily by Apple's Grand Central Dispatch [4] libraries, which are linked into almost all applications and system services running in user space. Implementing pthread workqueue support in Linux would involve radical changes to existing process management components due to the semantics of the API, and other attempts to implement such functionality in existing kernels are not yet complete [16]. Chameleon implements pthread workqueue support in user space using existing Linux pthread support. Chameleon pthread workqueue support library is called `pthread_workqueue` and is also based on XDispatch.

4.4 Schizophrenic Processes

Applications on mobile devices often make use of closed and proprietary hardware and software stacks. For example, the OpenGL ES libraries on both Android and iOS directly communicate with the graphics hardware through proprietary software and hardware interfaces using device-specific ioctls, or opaque IPC messages.

Chameleon cannot simply implement kernel-level support for foreign closed libraries because the semantics of ioctls or opaque IPC messages used by the closed libraries are unknown. Further, the semantics are likely to be closely tied to the underlying foreign hardware which is not present on the domestic device. Because of the tight vertical integration across hardware and software on mobile devices, library developers often discard cumbersome abstractions present on desktop PCs in favor of direct communication with hardware.

Chameleon solves the problem of direct access to proprietary hardware through the novel concept of *schizophrenic processes*. A schizophrenic process is a process which executes code using multiple personas. Endowing processes with the ability to execute code using multiple personas allows foreign applications to use domestic libraries to interact with the domestic hardware. For example, iOS applications in Chameleon can load and execute code from the Android OpenGL ES library and thereby directly interact with the hardware on the underlying Android device. Chameleon leverages the fact that while the implementation of proprietary libraries and their interface to kernel drivers is closed, the API on the application side is well-known, and is typically similar across platforms such as iOS and Android.

Note that schizophrenic processes differ from both API interposition and passport system calls. API interposition allows the execution of new foreign user space code in a foreign binary. Passport system calls allow the issuing of specifically-identified domestic system calls from a foreign binary. In contrast, schizophrenic processes allow the execution of domestic user space binaries within a foreign binary, including the issuing of all domestic system calls.

Chameleon encapsulates schizophrenic process support in a library called `libschizo`. This library is compiled as a foreign binary library, so it can link directly with the foreign binary that wishes to use domestic services. It provides a *switching* API comprising three key components. First, it provides the ability to interpret and load domestic binaries, which involves the use of a complete domestic dynamic loader within libschizo. For example, Chameleon incorporates a cross-compiled, library version of the Android ELF loader.

Second, libschizo provides the ability to switch the thread local storage (TLS) pointer from one persona to another so that each persona used by a thread can maintain its own state. The contents of the TLS vary from system to system, and initializing domestic libraries would wipe out any state already initialized by the foreign application. For example, the `errno` variable is stored in a particular location in the TLS which may differ between the foreign and domestic systems.

Third, libschizo provides the ability to switch from the foreign persona to the domestic persona and back again using a passport system call. This passport system call allows the foreign calling thread to tell the domestic kernel to switch personas, and using the same system call within the domestic persona to switch back to the foreign persona.

Chameleon replaces a foreign library with domestic functionality by generating a *surrogate library*, which is used to direct calls to foreign libraries from the application into domestic libraries. Surrogate libraries are created by scanning the foreign library being replaced for entry points and generating a wrapper function, called a *surrogate function*, for each possible entry point. The surrogate library is dynamically loaded instead of the original foreign library thereby intercepting calls made to the foreign library.

When a foreign application calls a surrogate function, the surrogate uses the libschizo switching API to enact a schizophrenic function call as follows:

1. Upon first invocation, the surrogate function uses libschizo's ability to load and interpret domestic binaries to load the appropriate domestic library and locate the required entry point. The libschizo switching API stores a pointer to the entry point in a locally scoped static variable for efficient reuse.

2. The arguments to the function call are stored on the stack.

3. The TLS pointer is changed to point to the domestic TLS instead of the foreign TLS.

4. A passport system call is executed that switches the calling thread's persona from foreign to domestic.

5. The arguments to the function call are restored from the stack.

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

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

8. A domestic system call is executed that switches the calling thread's persona from domestic back to foreign.

9. The TLS pointer is changed to point to the foreign TLS instead of the domestic TLS, and any TLS values such as `errno` are copied into the proper foreign TLS location.

10. The function return value is restored from the stack, and control is returned to the calling function.

Because Chameleon maintains kernel binary personas on a per thread basis, a foreign application can execute using multiple personas simultaneously from multiple threads, and each thread executing schizophrenically within a foreign application is free to use all of the facilities of the domestic OS including spawning additional threads which will inherit the current persona. Chameleon leverages a schizophrenic OpenGL ES library, discussed in detail in Section 5.3, to provide fully accelerated graphics to foreign iOS applications. Chameleon also uses schizophrenic function calls to a custom integration library to enable iOS applications to perform actions normally performed by SpringBoard such as displaying alerts and opening URLs.

5 iOS Subsystems on Android

To describe more clearly how the binary compatibility mechanisms work, we give a few examples of how Chameleon uses them together to provide binary compatibility in the context of key iOS subsystems. We discuss devices, input, and graphics. A discussion of all subsystems is beyond the scope of this paper.

5.1 Devices

Chameleon uses duct tape to make underlying hardware devices available via I/O Kit to iOS applications and libraries in the same manner as I/O Kit is used on iOS. 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. To directly compile the I/O Kit framework, Chameleon added a basic C++ runtime to the Linux kernel. The runtime support was based on Android's Bionic libc, and kernel Makefile support was added such that the compilation of C++ files from within the kernel required nothing more than assigning the object name to the `obj-y` Makefile variable. Chameleon used 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. [1] In fact, we initially compiled Chameleon with the I/O Kit framework found in XNU v1699.24.8, but later directly applied source code patches to upgrade to the I/O Kit framework found in XNU v2050.18.24.

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

For example, iOS applications 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 Chameleon prototype added a single C++ file in the Linux 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 with I/O Kit through a small interface function called on Linux kernel boot. The duct taped I/O Kit code will then match the C++ driver with the Linux device node (previously added from the Linux `device_add` function). After the driver class instance is matched to the device class instance, iOS user space can query and use the device as a standard iOS device. We believe that a similar process can be done for most devices found on a tablet or smartphone.

5.2 Input

No user-facing application would be complete without input from both the user and devices such as the accelerometer. In iOS, every application 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.

Chameleon creates a new thread in each iOS application to act as a bridge between the Android input system and the Mach IPC port expecting input events. This thread, the *eventpump*, simply listens for events from the Android `ChameleonW` wrapper application via a BSD socket and pumps those events into the iOS application via Mach IPC. In the future, this intermediary thread could be avoided with a minimal Linux Mach IPC wrapper ABI.

[1] 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.

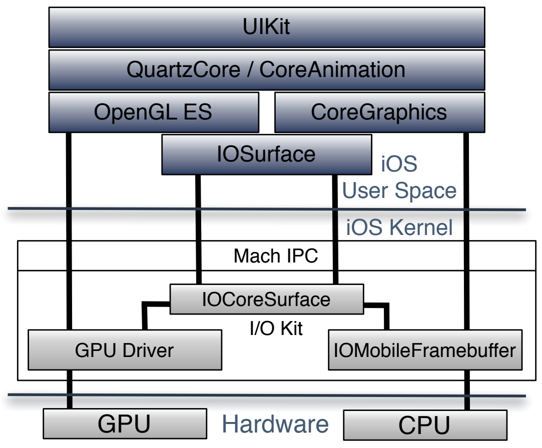


Figure 4: Overview of iOS Graphics Components

5.3 Graphics

Chameleon uses a combination of kernel ABI emulation, duct taped driver interfaces, API interposition, and schizophrenic processes to provide binary compatibility for 2D and 3D graphics. To understand how this works, we first provide an overview of the iOS graphics subsystem, as shown in Figure 4. User space libraries such as UIKit and CoreAnimation render user content, such as buttons, text, and images, using CoreGraphics and OpenGL ES system libraries. These system libraries communicate directly to the iOS kernel via Mach IPC, 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. 2D graphics in iOS uses the CoreGraphics or QuartzCore API to draw into graphics memory allocated by IOSurface and the IOCoreSurface I/O Kit driver, and then display the results using the IOMobileFramebuffer driver. 3D graphics uses the standard OpenGL ES API [21, 23] as well as some Apple-specific extensions and a custom Objective-C interface to the iOS native windowing system.

The iOS graphics subsystem presents a particularly interesting challenge to any mobile binary compatibility system: highly optimized user space libraries are

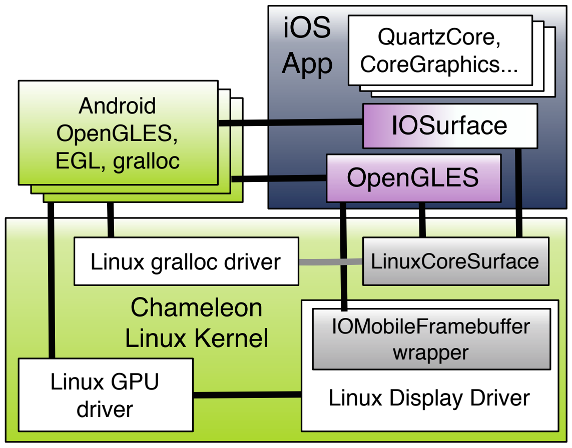


Figure 5: Chameleon Graphics Compatibility Architecture

tightly integrated with mobile hardware. Libraries such as QuartzCore and OpenGL ES, call into a set of proprietary, closed source, helper libraries which, in turn, use closed source opaque calls via Mach IPC to closed source kernel drivers which control black-box pieces of hardware. The Mach IPC calls to the kernel drivers are essentially used as device-specific system calls. Unlike the modern desktop OS, there are no well-defined 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 of non–existent GPU driver code will solve this problem.

Chameleon enables 2D and 3D graphics in iOS applications through a novel combination of I/O Kit Linux driver wrappers, a schizophrenic OpenGL ES library, and API interposition. Figure 5 shows an overview of how the iOS graphics subsystem is mapped to Android. Chameleon uses all of the original unmodified iOS libraries from Figure 4, except for OpenGL ES, and replaces iOS kernel components from Figure 4 with Linux kernel components.

For 2D graphics, Chameleon supports the iOS CoreGraphics or QuartzCore API using the original unmodified CoreGraphics and QuartzCore/CoreAnimation libraries, but it interposes on the IOSurface library to allocate the graphics memory needed using the unmodified Android gralloc library and the underlying Linux gralloc driver. Chameleon uses API interposition and schizophrenic calls to the Android gralloc library to tie the allocated memory to Android GraphicBuffer objects which are designed to be shared between graphics hardware via OpenGL ES and the CPU via direct pixel manipulation. Instead of using the iOS IOMobileFramebuffer driver to display the results, Chameleon uses an IOMobileFramebuffer I/O Kit interface wrapped around the existing Linux display driver to display the results. To provide the zero-copy memory semantics expected by the IOSurface library, Chameleon provides LinuxCoreSurface, a reverse-engineered IOCoreSurface I/O Kit driver. However, 2D graphics support is insufficient for iOS applications to work because 3D graphics is used extensively by iOS drawing and composing functions.

For 3D graphics, Chameleon supports the iOS OpenGL ES API by replacing the entire iOS OpenGL ES library using schizophrenic processes to call the Android graphics libraries. The iOS OpenGL ES API consists of two parts, the standard OpenGL ES API [21, 23] and some Apple-specific EAGL extensions. Chameleon provides an iOS replacement OpenGL ES library with a schizophrenic surrogate entry point for every exported symbol in both of these categories.

For the standard OpenGL ES API entry points, Chameleon provides a set of surrogate functions conforming to the schizophrenic switching API in Chameleon's iOS replacement OpenGL ES library which then call the Android OpenGL ES library. Because each of these entry points has a well-defined function prototype, the process of creating surrogate functions is automated by a script that analyzes exported symbols in the iOS OpenGL ES Mach-O library, searches through a directory of Android ELF shared objects for a matching export, and automatically generates a surrogate function.

For the Apple-specific EAGL extensions, these do not exist on Android. These extensions are used to control window memory and graphics context. The script used to analyze exported symbols in the iOS OpenGL ES Mach-O library will generate a C function stub for the EAGL exported symbols which do not match any export in any Android ELF library. These then need to be implemented in some fashion. However, the EAGL extensions are Apple's replacement for the Native Platform Graphics Interface Layer (EGL) standard, and this is implemented in an Android EGL library. Chameleon uses schizophrenic calls to the Android EGL functions to implement the EAGL extensions. Android's EGL library, used in combination with Android's SurfaceFlinger service, manages windows memory into which iOS applications render. Allocating this memory via the standard Android SurfaceFlinger service allows Chameleon to manage the iOS display in the same manner all Android application displays are managed.

The Chameleon graphics subsystem allows many iOS applications to render without issue and supports the necessary 3D graphics drawing and composing functions used by iOS. However, because our initial prototype was designed to run system services, such as the window composer, in the same process as the application itself, and because the OpenGL ES standard only allows a single rendering thread per process, our prototype is limited to a single graphics context. This means advanced 3D games, or any application that explicitly creates an OpenGL ES context will not work properly. Many applications, however, exclusively use the QuartzCore 2D drawing APIs that utilize the system OpenGL ES context. These applications render without issue.

6 Experimental Results

We have implemented a Chameleon prototype for running iOS and Android applications 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 Chameleon: (1) Linux binary and Android applications running on vanilla Android, (2) Linux binary and Android applications running on Chameleon, and (3) iOS binary applications running on Chameleon. 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 7" LED-backlit IPS LCD display with 1280x800 screen resolution at 216ppi, running Android 4.2, also known as Jelly Bean, the latest version of Android. We also ran iOS binary applications on a jailbroken iPad mini with a 1 GHz dual-core A5 CPU, 512 MB RAM, 16 GB of flash storage, and 7.9" LED-backlit IPS LCD display with 1024x768 screen resolution at 163ppi, running iOS 6.1.2, the latest version of iOS. Since the iPad Mini was released at roughly the same time frame as the Nexus 7 and has a similar form factor, it provides a useful point of comparison even though it costs over 50% more than the Nexus 7.

We used both micro-benchmarks and real applications to evaluate the performance of Chameleon. To measure the latency of common low-level OS operations, we used micro-benchmarks from `lmbench` 3.0 and compiled two versions, a ELF Linux binary and a Mach–O iOS binary, using the standard Linux GCC 4.4.1 and Xcode 4.2.1 compilers, respectively. We used four categories of `lmbench` tests: basic operations, system calls and signals, process creation, and local communication and file operations. To measure real application performance, we used comparable iOS and Android PassMark applications available from the Apple App Store [26] and Google Play [27], respectively. PassMark conducts a wide range of resource intensive throughout tests to evaluate CPU, memory, disk, and graphics performance. For example, it measures how many graphics operations it can do per second. We used PassMark because it is a widely used, commercially-supported application available on both iOS and Android, and provides a conservative measure of application performance. We normalize all results using the vanilla Android performance as the baseline to compare across systems. This is useful to measure Chameleon performance overhead, but also provides some key observations regarding the characteristics of Android and iOS applications.

6.1 Obtaining iOS Applications

For our experiments, we downloaded iOS applications from the Apple App Store. In the future, we envision that developers and application distributors would be incentivized to provide alternative distribution methods. For example, Google Play might be incentivized to take advantage of Chameleon to make a greater number and higher quality of applications 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 the App Store.

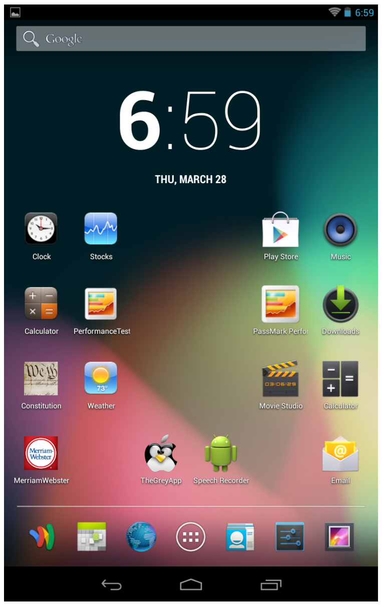
App Store applications, unlike iOS system applications such as Stocks, are encrypted and can only be decrypted using the key stored on an iOS device. We use a script that can download and decrypt applications using any jailbroken iOS device. To illustrate this point, we used an old iPhone 3GS updated with iOS 5.0.1 for this purpose. The decryption process we based our method on is widely used and documented [28].

Figure 6 shows screenshots of the Nexus 7 tablet with various iOS applications that we installed and ran on the device. The homescreen shows iOS and Android applications side by side that users can execute in a seamless manner. The iOS Stocks application receives live market data over WiFi; quotes are from March 27, 2013. The iOS Constitution application by Clint Bagwell Consulting is only available on the Apple App Store and provides a superior user experience to those available on Google Play [9].

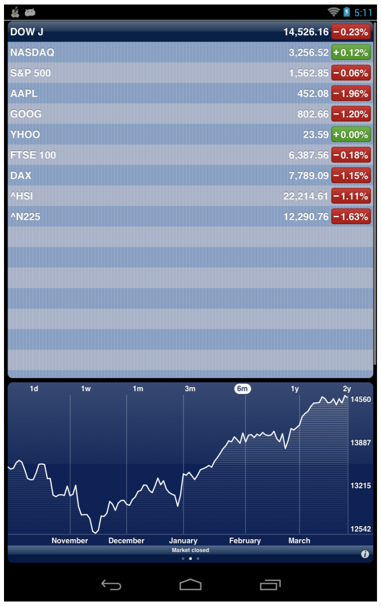
6.2 Microbenchmark Measurements

Figure 7 shows the results of running `lmbench` microbenchmarks on the four different system configurations. Vanilla Android performance is normalized to one in all cases, so those results are not explicitly shown. These 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.

First, Figure 7 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. None of these tests exercise Chameleon functionality, but they

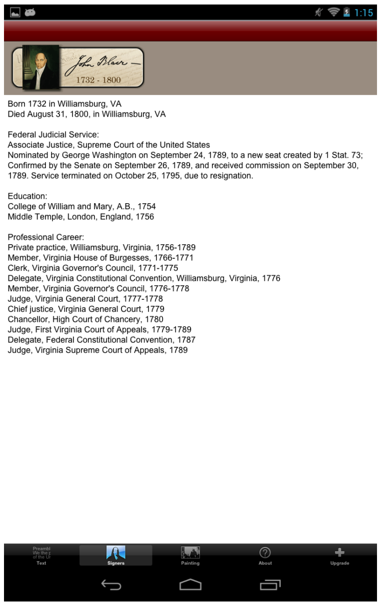


(a) Android homescreen with iOS applications



DOW J	14,526.16	−0.23%
NASDAQ	3,256.52	+0.12%
S&P 500	1,562.85	−0.06%
AAPL	452.08	−1.96%
GOOG	802.66	−1.20%
YHOO	23.59	+0.00%
FTSE 100	6,387.56	−0.18%
DAX	7,789.09	−1.15%
^HSI	22,214.61	−1.11%
^N225	12,290.76	−1.63%

(b) iOS Stocks application



Born 1732 in Williamsburg, VA
Died August 31, 1800, in Williamsburg, VA

Federal Judicial Service:
Associate Justice, Supreme Court of the United States
Nominated by George Washington on September 24, 1789, to a new seat created by 1 Stat. 73; Confirmed by the Senate on September 26, 1789, and received commission on September 30, 1789. Service terminated on October 25, 1795, due to resignation.

Education:
College of William and Mary, A.B., 1754
Middle Temple, London, England, 1756

Professional Career:
Private practice, Williamsburg, Virginia, 1756-1789
Member, Virginia House of Burgesses, 1766-1771
Clerk, Virginia Governor's Council, 1771-1775
Delegate, Virginia Constitutional Convention, Williamsburg, Virginia, 1776
Member, Virginia Governor's Council, 1776-1778
Judge, Virginia General Court, 1777-1778
Chief justice, Virginia General Court, 1779
Chancellor, High Court of Chancery, 1780
Judge, First Virginia Court of Appeals, 1779-1789
Delegate, Federal Constitutional Convention, 1787
Judge, Virginia Supreme Court of Appeals, 1789

(c) Constitution for iPhone and iPod Touch

Figure 6: Chameleon running iOS applications

provide a comparison that reflect 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 7 shows system call and signal handler measurements including null system call, `read`, `write`, `open/close`, and signal handler tests. The null system call measurement shows the overhead incurred by Chameleon on a system call that does no work, providing a conservative measure of the cost of Chameleon. The overhead is 8.5% over vanilla Android in running the same Linux binary, which is due to the need to determine the persona of the calling thread and route the call to the corresponding system call dispatch table. The overhead is 40% for running the iOS binary over vanilla Android running the Linux binary, which shows the additional cost of using the iOS persona and translating the system call into the corresponding Linux system call. These overheads end up in the noise for system calls that actually perform some function, as shown by the measurements for the other system calls. Running the iOS binary on the Nexus 7 using Chameleon is much faster on these system call measurements than running the same binary on the iPad mini, illustrating a benefit of using Chameleon to leverage the different performance characteristics of Android hardware instead of being limited to only iOS hardware.

The signal handler measurement shows the overhead incurred by Chameleon in delivering a signal to the same process that generated the signal. 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 in running the same Linux binary, which is due to the added cost of determining the persona of the target thread. The overhead is 25% for running the iOS binary over vanilla Android running the Linux binary, which shows the additional cost of using the iOS persona. This involves some translation of the signal information and using a larger signal delivery structure with more information, as 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 Chameleon for the signal handler test.

Third, Figure 7 shows five sets of process creation measurements, `fork+exit`, `fork+exec`, and `fork+sh` tests. The `fork+exit` measurement shows that Chameleon incurs negligible overhead versus vanilla Android in running a Linux binary despite the fact that it must do some extra work in the form of Mach IPC initialization. However, Chameleon 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, with the Linux binary taking 245 μs and the iOSbinary taking 3.75 ms.

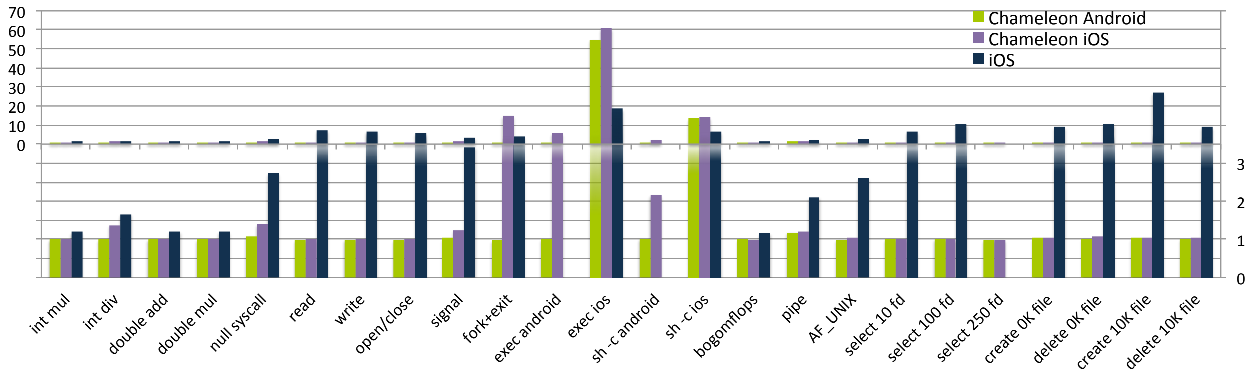


Figure 7: Microbenchmark latency measurements normalized to vanilla Android; lower is better performance.

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 those libraries are used by the binary. `fork` must then duplicate the page table entries corresponding to all of that 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 `pthread_atfork` to register 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 Chameleon on the Android device due to a shared library cache optimization that is not yet supported in the Chameleon prototype implementation. To save time on library loading, iOS's `dyld` stores common libraries pre-linked 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 Chameleon made possible by using schizophrenic processes. The test spawns a child process which executes a simple hello world program, but 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. On a vanilla Android system, the only way to run the test is to run a Linux binary that spawns a child to run a Linux binary. Similarly, on an iOS system, the only way to run the test is to run an iOS binary that spawns a child to run an iOS binary. Using Chameleon, 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.

Figure 7 shows all `fork+exec` measurements using Chameleon. `exec android` tests spawn a child to run a Linux binary. Chameleon incurs negligible overhead versus vanilla Android when the test program is a Linux binary that spawns a child to run a Linux binary. The actual time it takes to run this test 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. Chameleon takes 4.8 times longer to run the test when the test program is an iOS binary that spawns a child to run a Linux binary. The extra overhead is due to the cost of an iOS binary calling fork, as discussed previously in the case of the `fork+exit` measurement. Interestingly, the `fork+exec` measurement is 3.42 ms, which is less than the `fork+exit` measurement. This is because the child process replaces its iOS binary with the hello world Linux binary, and this is less expensive than having the original iOS binary exit because of all the exit handlers that it has to execute.

`exec ios` tests spawn a child to run an iOS binary. This is not possible on vanilla Android, so there is no vanilla Android performance that we can normalize all results against as a baseline. To compare the different `fork+exec` measurements, we instead normalize the performance against the vanilla Android system running a Linux binary that spawns a child running a Linux binary. This comparison is intentionally unfair and skews the results against the tests that spawn a child running the more heavyweight iOS binary. Nevertheless, using this comparison, Figure 7 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` in which copy-on-write can be used to limit this cost to the duplicating page tables, the cost for `exec` involves not just creating the page tables, but also mapping in the actual libraries themselves. This is very expensive because the Chameleon prototype implementation does not use the shared cache optimization, but uses non-prelinked libraries and `dyld` walks the filesystem to load all the libraries on every `exec`. The extra overhead for starting with an iOS binary instead of a Linux binary is due to the cost of the iOS binary calling fork, as discussed previously in the case of the `fork+exit` measurement. Running the `fork+exec` test on the iPad mini is faster than using Chameleon on the Android device because of its use of the shared cache optimization, which avoids the need to walk the filesystem to load all the libraries.

Similar to the `fork+exec` measurement, the `fork+sh -c` measurement is also done in several variations on Chameleon made possible by using schizophrenic processes. `sh -c android` tests launch a shell to run a Linux binary. Chameleon again 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 case of the `fork+exec` measurement, but because the `sh -c android` measurement takes longer, 6.8 ms using the iOS binary, the relative overhead is less than in the case of `exec android`.

`sh -c ios` tests launch a shell to run an iOS binary. This is not possible on vanilla Android, so we again use the same `sh -c android` baseline for comparison, skewing the results against the `sh -c ios` tests. Nevertheless, using this comparison, Figure 7 shows that spawning a child to run an iOS binary is much more expensive for the same reasons as for the `fork+exec` measurement. Because the `sh -c ios` test takes longer, the relative overhead is less than in the case of `exec ios`.

Fourth, Figure 7 shows local communication and filesystem measurements including `pipe`, `AF_UNIX`, `select` on 10 to 250 file descriptors, and creating and deleting 0 KB and 10 KB files. The measurements were quite similar for all three system configurations using the Android device. However, the measurements for the iOS device 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, and simply failed to complete for 250 file descriptors. In contrast, the same iOS binary runs using Chameleon on Android with performance that was the same as running a Linux binary on vanilla Android across the wide range in the number of file descriptors used for the measurements.

6.3 Application Measurements

Figure 8 shows the results of running the iOS and Android PassMark benchmark applications [26, 27] on the four different system configurations. Vanilla Android performance is normalized to one in all cases, so those results are not explicitly shown. These measurements are in operations per second, so larger numbers are better.

Figure 8 shows CPU operation measurements for integer, floating point, find primes, random string sort, data encryption, and data compression tests. Because PassMark is a full graphical application, running the application on Chameleon necessarily exercises its functionality, but because these tests are primarily CPU intensive, the resulting overhead for using Chameleon is negligible. This is evident by comparing the results for running the Android PassMark application on vanilla Android versus Chameleon. However, unlike the basic CPU operation measurements obtained using `lmbench`, the PassMark measurements show that Chameleon delivers significantly faster performance when running the iOS PassMark application on Android. This is because the Android version is written in Java and is interpreted through the Dalvik VM while the iOS version is written in Objective-C and is compiled and run as a native binary. Because the Android device is faster than the iOS device, Chameleon outperforms iOS in running the CPU tests for the same iOS PassMark application.

Figure 8 shows storage operation measurements for write and read tests. There is negligible overhead caused by Chameleon for these tests as evident by comparing the results for running the Android PassMark application on vanilla Android versus Chameleon. However, Chameleon does have somewhat slower performance on the storage write test when running the iOS version of the application. This is not due to any performance overhead caused by Chameleon, but the modest difference is more likely caused by small differences in the native libraries used by the respective applications. Chameleon has much better storage read performance when running the iOS application than vanilla Android. In comparison with the iOS device, Chameleon has better storage read performance but worse storage write performance. Because storage performance can depend heavily on the OS, these results may reflect differences in both the hardware and the OS.

Figure 8 shows memory operation measurements for write and read tests. There is negligible overhead caused by Chameleon for these tests as evident by comparing the results for running the Android PassMark application on vanilla Android versus Chameleon. Chameleon delivers significantly faster performance when running the iOS PassMark application on Android. This is again because Chameleon can run the iOS application natively

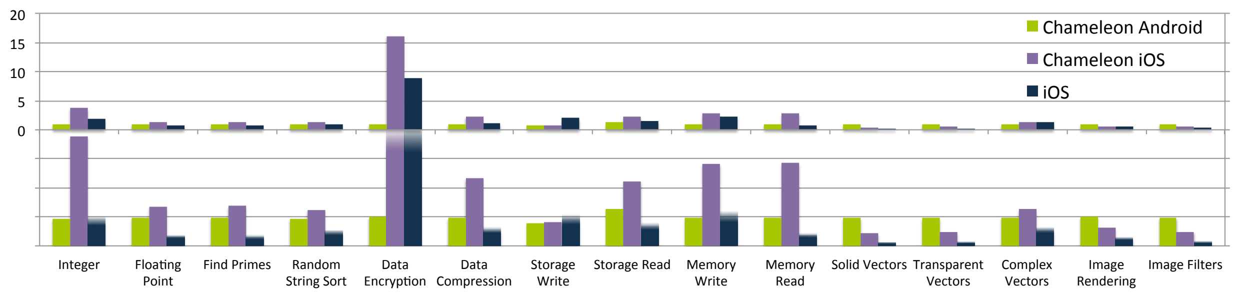


Figure 8: Application throughput measurements normalized to vanilla Android; higher is better performance.

while Android runs the application through the Dalvik VM. Chameleon outperforms iOS in running the memory tests for the same iOS PassMark application, again reflecting the benefit of using faster Android hardware.

Figure 8 shows graphics measurements for a variety of graphics operations, including solid vectors, transparent vectors, complex vectors, image rendering, and image filters. There is negligible overhead caused by Chameleon for these tests as evident by comparing the results for running the Android PassMark application on vanilla Android versus Chameleon. However, Chameleon has slower performance when running the iOS PassMark application on Android. Both versions use the same native Android OpenGL ES library, which is why the Android application performs well since the GPU-intensive operations that dominate this test are performed natively. However, Chameleon incurs additional overhead due to a suboptimal mapping in our unoptimized prototype between the `IOMobileFramebuffer` interface and the Linux device driver. Despite this cost, Chameleon still significantly outperforms iOS in running the same iOS PassMark application across all graphics operations because of the faster GPU hardware in the Android device. For example, the solid vector test runs almost twice as fast using Chameleon versus iOS. Note that PassMark's 3D graphics measurements are not included in the results because the Chameleon prototype implementation does not yet have complete support for advanced 3D rendering.

7 Related Work

Various previous approaches have explored binary compatibility in the context of desktop computers, but little work has been done for mobile devices. Wine [1] runs Windows applications on x86 computers running Linux, including widely-used Microsoft Office productivity applications. It achieves this by attempting to reimplement the entire foreign user space library API, such as Win32, using native APIs. This approach is tedious and overwhelmingly complex, and Wine has been under development for a long time, but continues to chase Windows as every new release contains new APIs that need to be implemented. Darling [11] takes a similar approach to try to run Mac OS X applications on Linux, though it remains a work in progress unable to run anything other than very simple applications. In contrast, Chameleon provides binary personality support in the OS to leverage existing unmodified application libraries and frameworks and avoid rewriting huge amounts of user space code.

Wabi [19] was a product from Sun Microsystems for running Windows applications on Solaris. It supported applications developed for Windows 3.1, but did not support later versions of Windows and was discontinued. Unlike Wine, it required Windows 3.1. It replaced low-level Windows API libraries with versions that translated from Windows to Solaris calls. Wabi ran on top of Solaris and provided all of its functionality outside of the OS, limiting its ability to support applications that require kernel-level services not available in Solaris. In contrast, Chameleon is not limited to API interposition and provides binary personality support in the OS to support foreign kernel services.

Several BSD variants maintain a binary compatibility layer for other OSes [12, 13, 17]. The BSD approach reimplements foreign system calls in the OS, using a different system call dispatch table for each OS to glue the calls to the BSD kernel. It works for foreign OSes that are close enough to BSD such as Linux, but attempts to extend this approach to supporting Mac OS X applications only provide limited support for command line tools, not Mac OS X GUI applications [14]. In contrast, Chameleon provides duct tape and interposition layers to simplify adding unmodified foreign kernel code to an OS, and introduces schizophrenic processes that can run mixtures of foreign and domestic binaries. Unlike previous approaches, this unique mechanism provides a solution for supporting graphical applications in mobile devices where hardware acceleration is often exposed directly to user space and libraries such as OpenGL call device and platform specific functions.

Other partial solutions to binary compatibility have also been explored. For example, Shinichiro Hamaji's Mach-O loader for Linux [18] can load and run some desktop Apple OS X command line binaries in Linux. This project supports only command line binaries using the Linux "misc" binary format and dynamically overwriting C entry points to system calls. In contrast, Chameleon provides a complete environment for foreign binaries including graphics libraries and device access.

Virtual machines have perhaps become the most popular way to run foreign applications on desktop computers. However, they require purchasing and running a full guest OS instance, using much more memory and disk space, and incur additional performance overhead. Various approaches [7, 10, 20] have attempted to bring VMs to mobile devices, but these approaches cannot run unmodified OSes and incur even higher overhead than their desktop counterparts; memory overhead is a key problem on memory constrained smartphones. Lightweight mobile virtualization [2, 8] claims lower virtualization overhead than other mobile counterparts, but does not support running different OS instances and therefore cannot run foreign applications at all. Unlike Chameleon, none of these previous virtualization approaches can run foreign iOS applications on Android.

8 Conclusions

Chameleon is the first system to allow users to run unmodified mobile applications written for one OS on another OS. It provides novel binary compatibility mechanisms including (1) a multi-persona binary interface to allow foreign binaries to run directly on top of a domestic OS kernel, (2) a static code compilation translation layer, duct tape, to allow foreign open-source kernel code to be directly compiled as a part of the domestic kernel without modification, (3) API interposition with passport system calls to implement OS level services in user space to support foreign binaries, and (4) schizophrenic processes, which take advantage of both foreign and domestic OS services at the same time. We have built a Chameleon prototype and demonstrate that it imposes only modest performance overhead and successfully runs widely used iOS applications seamlessly with Android applications.

9 Acknowledgments

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

References

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

[2] 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.

[3] 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.

[4] APPLE, INC. Grand Central Dispatch (GCD) Reference. `http://developer.apple.com/library/ios/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html`, Nov. 2011. Accessed: 3/28/2013.

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

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

[7] 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* (Dec. 2010), 124–135.

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

[9] CLINT BAGWELL CONSULTING. Constitution for iPhone. `https://itunes.apple.com/ca/app/constitution-for-iphone-ipod/id288657710`, Mar. 2013. Accessed: 3/20/2013.

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

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

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

[13] DREYFUS, E. IRIX Binary Compatibility, Part 1. `http://onlamp.com/lpt/a/2623`, Aug. 2002. Accessed: 5/11/2012.

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

[15] FARADAY, OWEN. Android is a desolate wasteland when it comes to games. `http://www.wired.co.uk/news/archive/2012-10/31/android-games`, Oct. 2012. Accessed: 3/21/2013.

[16] FREEBSD. Grand Central Dispatch (GCD) on FreeBSD. `https://wiki.freebsd.org/GCD`, May 2011. Accessed: 3/21/2013.

[17] FREEBSD DOCUMENTATION PROJECT. Linux binary compatibility. In *FreeBSD Handbook*, B. N. Handy, R. Murphey, and J. Mock, Eds. 2000, ch. 11.

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

[19] HOHENSEE, P., MYSZEWSKI, M., AND REESE, D. Wabi CPU emulation. In *Hot Chips 8* (1996).

[20] 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 5th Consumer Communications and Network Conference* (Las Vegas, NV, Jan. 2008).

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

[22] MLBA TEAM. XDispatch - Overview. `http://opensource.mlba-team.de/xdispatch/docs/current/index.html`, Jan. 2013. Accessed: 3/27/2013.

[23] MUNSHI, A., AND LEECH, J. 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.

[24] 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 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002).

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

[26] PASSMARK SOFTWARE, INC. Performancetestmobile for iphone, ipod touch, and ipad on the itunes app store. `https://itunes.apple.com/us/app/performancetest-mobile/id494438360?ls=1&mt=8`, June 2012. Accessed: 3/14/2013.

[27] 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.

[28] 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.

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

[30] 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* (Feb. 2006), 74–105.