POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing

Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh
{vatlidak, jeremya, roxana, dimitro, nieh}@cs.columbia.edu
Columbia University

Abstract

The POSIX standard, developed 25 years ago, comprises a set of operating system (OS) abstractions that aid application portability across UNIX-based OSes. While OSes and applications have evolved tremendously over the last 25 years, POSIX, and the basic set of abstractions it provides, has remained largely unchanged. Little has been done to measure how and to what extent traditional POSIX abstractions are being used in modern OSes, and whether new abstractions are taking form, dethroning traditional ones. We explore these questions through a study of POSIX usage in modern desktop and mobile OSes: Android, OS X, and Ubuntu. Our results show that new abstractions are taking form, replacing several prominent traditional abstractions in POSIX. While the changes are driven by common needs and are conceptually similar across the three OSes, they are not converging on any new standard, increasing fragmentation.

1. Introduction

The Portable Operating System Interface (POSIX) is the IEEE standard operating system (OS) service interface for UNIX-based systems. It describes a set of fundamental abstractions needed for efficient construction of applications. Born out of work in the early 1980s, when the fragmentation of UNIX was of concern, it was created to enable application developers to easily write application source code that would be portable across multiple diverse OSes. While perfect portability was never a reality, the level of uniformity added by POSIX has been valuable both for application developers and for educators alike. Application developers can code atop the same rough abstractions, and educators can teach widely applicable abstractions in their OS courses.

Since its creation over 25 years ago, POSIX has evolved to some extent (e.g., the most recent update was published in 2013 [55]), but the changes have been small overall. Meanwhile, applications and the computing platforms they run on have changed dramatically: Modern applications for today's smartphones, desktop PCs, and tablets, interact with multiple layers of software frameworks and libraries implemented atop the OS. Although POSIX continues to serve as the single standardized interface between these software frameworks and the OS, little has been done to measure whether POSIX abstractions are effective in supporting modern application workloads, or whether new, non-standard abstractions are taking form, dethroning traditional ones. Such measurements can be valuable to developers, educators, researchers, and standards bodies alike, who can adapt their applications, teachings, and optimization and standardization efforts toward the new or changed abstractions.

We present the first study of POSIX usage in modern OSes focusing on three of today's most widely used mobile and desktop OSes – Android, OS X, and Ubuntu – and popular consumer applications characteristic to these OSes. We built a utility, called *libtrack*, that supports both dynamic and static analyses of POSIX use in applications. Dynamic analyses give us detailed and precise POSIX usage patterns, but can only be run at limited scale (e.g., tens of popular applications for each OS). Static analyses let us generalize trends at large scale (e.g., 1.1M applications in our Android study), but conclusions are less precise. Our study sheds light into a number of important questions regarding the use of POSIX abstractions in modern OSes, including: which abstractions work well; which appear to be used in ways for which they were never intended; which are being replaced by new and non-standard abstractions; and whether the standard is missing any fundamental abstractions needed by modern workloads. Our findings can be summarized as follows:

First, *usage is driven by high-level frameworks, which impacts POSIX's portability goals.* The original goal of the

EuroSys '16, April 18–21, 2016, London, United Kingdom

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901350

POSIX standard was application source code portability. However, modern applications are no longer being written to standardized POSIX interfaces. Instead, they rely on platform-specific frameworks and libraries that leverage high-level abstractions for inter-process communication (IPC), thread pool management, relational databases, and graphics support. These frameworks and libraries are often implemented using underlying standard POSIX APIs, but are also free to depart from POSIX and use OS-specific interfaces. Thus, instead of the POSIX API serving as the interface between applications and the OS environment, modern OSes – such as Ubuntu, Android, and OS X– provide a more layered programming model with "taller" interfaces. Applications directly link against high-level frameworks, which invoke other frameworks and libraries that may eventually utilize POSIX. This new, layered programming model imposes challenges with respect to application portability, and has given rise to many different cross-platform SDKs [2, 9, 22, 24, 39] that attempt to fill the gap left by a standard which has not evolved with the rest of the ecosystem. However, these cross-platform SDKs are often challenging to develop and maintain up-to-date with OS changes.

Second, *extension APIs, namely* `ioctl`*, dominate modern POSIX usage patterns as OS developers resort to them to build support for abstractions missing from the POSIX standard.* Extension APIs have become the standard way for developers to circumvent POSIX limitations and facilitate hardware-supported, high-level functionality for graphics, sound, and IPC. For example, the `ioctl` interface is now regularly used to mediate complex graphics commands between the high-level OpenGL library and the graphics driver.

Third, *new abstractions are arising driven by the same POSIX limitations across the three OSes, but the new abstractions are not converging.* To deal with abstractions missing from the aging POSIX standard, modern OSes are implementing new abstractions to support higher-level application functionality. Although these interfaces and abstractions are driven by similar POSIX limitations and are conceptually similar across OSes, they are not converging on any new standard. Traditional POSIX threading models, IPC interfaces, and file system access are being replaced by platform and vendor-specific APIs and frameworks such as Grand Central Dispatch [18], Binder [29], DBus [25], and SQLite [1].

We believe that our findings have broad implications related to the future of POSIX-compliant OS portability, which the systems research community and standards bodies will likely need to address in the future. To support further studies across a richer set of UNIX-based OSes and workloads, which we anticipate will be needed to establish a rigorous course of action, we make the *libtrack* source code, along with the application workloads and traces, available at:

`https://columbia.github.io/libtrack/`

This paper is organized as follows. Section 2 presents two motivating examples for our study and formulates our goals. Sections 3 and 4 give background and detail our study's methodology. Section 5 presents our measurement results. Sections 6 and 7 discuss the broader implications of our findings in the context of related work and in general.

2. Motivation

Motivating Examples. Our measurement study is motivated by our experience building two very different systems – an Android data protection system called *Pebbles* [52] and an Android/iOS binary compatibility system, called *Cycada* (formerly known as *Cider*) [7] – whose designs exposed, and were drastically impacted by, the changes in the ways applications on these platforms are using system abstractions.

• *Pebbles* [52] provides data protection at the level of application-level objects, such as emails or bank accounts. This level of protection is made possible by the new ways in which Android applications use storage abstractions. Rather than using traditional unstructured POSIX file system abstractions, Android applications instead store data almost exclusively in highly structured storage (SQLite). In *Pebbles*, we leverage this structured information to transparently and accurately reconstruct application-level objects so the OS can provide protection at their level. Without applications' almost exclusive reliance on structured storage, *Pebbles* would likely not be possible.

• *Cycada* [7] is a binary compatibility framework for Android and iOS applications. Given that both Android and iOS implement similar POSIX functionality, we initially thought that building *Cycada* would be relatively straightforward compared to previous Windows-UNIX compatibility efforts. However, achieving compatibility even between Android and iOS turned out to be a herculean task. A main obstacle was the extensive use of POSIX's `ioctl` extension API, which is highly platform-specific and loosely defined. To address this challenge, we elevated the level of abstraction at which we constructed binary compatibility from POSIX to newer, high-level abstractions used by applications, such as graphics and sound libraries. With this approach, and intuitively assuming that most applications leverage these abstractions, we were able to translate between well-defined interfaces and run unmodified iOS applications on Android.

Our experience with these systems led to anecdotal observations about changes in how modern applications use specific system abstractions. Other prior studies also suggest an evolution of specific POSIX abstractions [30, 58]. However, no prior work offers a rigorous characterization of these changes across the broad range of system abstractions standardized in POSIX and across multiple OSes.

Study Goals. Our goal in this study is to offer a rigorous characterization of how standardized system abstractions are being used by modern workloads, and whether they are

being replaced by other abstractions. Broadly speaking, we are interested in questions such as:

- Which POSIX abstractions are still being used and relevant for today's application workloads?
- Which abstractions are being replaced by new and non-standard abstractions?
- Are there any specific limitations of traditional abstractions that motivate these transitions?
- Are the replacement abstractions similar in the various OSes, or are the abstractions diverging?
- Are there any blatantly missing abstractions in POSIX, which modern workloads appear to require? If so, how are the gaps currently being filled?
- Are any traditional abstractions being used in ways for which they were not intended? If so, what are the performance or security implications of these uses?

We believe that the answers to these questions are relevant to a wide audience, including: *researchers*, who can design and optimize their systems by leveraging current, broadly applicable trends in application workloads, as illustrated by the preceding motivating examples; *application developers*, who may take advantage of new and more powerful abstractions available in modern OSes; *standard bodies*, such as OpenGroup, who may wish to reconsider certain, obsolete aspects of their standard in light of the new trends; and *educators*, who may wish to refresh their courses with coverage of the new, prominent OS abstractions that are replacing traditional ones [8].

Study Scope. We answer the preceding questions in the context of three popular, consumer-oriented OSes and workloads: Android 4.3 Jellybean, OS X 10.10.5 Yosemite, and Ubuntu 12.04 Precise Pangolin. Our OS choices stems not only from their popularity, but also from their diversity: Android is a relatively new, mobile OS; OS X is a desktop OS that hosts a large corpus of modern applications; and Ubuntu is a more traditional desktop OS, offering us a baseline to study evolution of abstractions. As workloads, we use real applications downloaded from the corresponding consumer-oriented repositories or app markets (Section 4.2). For a more complete view of POSIX's state, this study could be extended to other types of workloads, including server-side, embedded, and high-performance computing workloads. Our consumer-oriented study establishes the necessary tools, methodologies, questions, and initial answers to support such broader studies in the future. Our public release of tools and workloads facilitates such future studies.

3. POSIX Background

POSIX refers to a family of standards maintained by the Austin Group [53]. This family of standards describes a set of fundamental services needed for portable application development in UNIX-based OSes [32]. The latest POSIX revision, published in 2013 [56], differs only marginally from the first drafts of the standard, published in the late 80's and early 90's. It covers topics such as directory structure, command-line interpreters and utilities, environment variables, and system service functions and subroutines. Table 1 lists all major POSIX revisions with a short description of each, according to the Austin Open Group [53] and C/UNIX standards defined in the GNU/Linux manual.

Revision	Brief Description
POSIX.1-1990	Initial release including core services.
POSIX.2-1992	Describing commands and utilities.
POSIX.1b-1993	Describing real-time facilities.
POSIX.1c-1995	Describing POSIX threads interface.
POSIX.1-1996	Composed of POSIX.1-1990,POSIX.1b, and POSIX.1c.
POSIX.1d-1999	Describing additional real-time extensions.
POSIX.1g-2000	Describing networking APIs (including sockets).
POSIX.1j-2000	Describing advanced real-time extensions.
POSIX.1q-2000	Describing tracing extensions.
POSIX.1-2001	Composed of POSIX.1-1996, POSIX.2, SUSv2, POSIX.1d, POSIX.1g, and POSIX.1j.
POSIX.1-2004	Incorporated two technical corrigendum in POSIX.1-2001 fixing issues related to base definitions.
POSIX.1-2008	Adding remaining parts of POSIX.1-2001
POSIX.1-2013	Incorporating one technical corrigenda in POSIX.1-2008 fixing issues related to base definitions.

Table 1: POSIX revisions. Lists all major revisions and a brief description of each amendment.

POSIX defines 1,177 C functions and 14 global variables [56] that are intended to facilitate application portability at the source code level, and to codify a fundamental set of OS abstractions. The OpenGroup collates these APIs into 6 broad categories shown in [57]. These categories are: *signals*, *streams*, *IPC*, *realtime*, *threads*, and *sockets*. Not all of these functions are related to OS services (system calls). For example, on Android, of the 821 POSIX functions implemented, only 343 are related to system calls implemented by Linux kernel that Android is built on. The rest are utility functions fully implemented in user-space (e.g., `memcpy`, `strlen`, and `atoi`).

We focus our measurements on the system service functions and subroutines which are specified using the C programming language. We refine the official POSIX API classification into 14 more fine-grained categories. These categories provide meaningful insights into the types of functionality defined by POSIX, and aid our analysis of the evolution of these abstractions. Table 3 lists these categories, examples of prominent functions in each category, and the total number of interfaces implemented by various OSes. In Section 5.1, we discuss in detail the POSIX im-

plementations in bionic `libc` (Android), `glibc` (Ubuntu), and `libSystem.dylib` (a collection of constituent OS X libraries).

4. Methodology

Our study involves two types of experiments with real, client-side applications on the three OSes: *dynamic experiments* and *static analysis*. Dynamic experiments let us obtain detailed and precise POSIX usage patterns, but we can only run them at limited scale (e.g., 45 popular applications in our Android study). Static analysis lets us generalize trends at large scale (e.g., 1.1M applications in our Android study), but conclusions are not as precise.

In support of these studies, we developed *libtrack*, a tool that traces the use of a given native C library from modern applications. While *libtrack* is general and can trace the usage of arbitrary native libraries, in this paper we exclusively use it to track POSIX C standard library implementations in the OSes we study. *libtrack* implements two modules: (1) a dynamic module, which collects and analyzes traces of calls to a given C standard library produced by running applications; and (2) a static module, which analyzes arbitrary native libraries and binaries for links (i.e., dynamic relocations) to the given C standard library. Section 4.1 describes our *libtrack* implementation and Section 4.2 details our methodology of using it.

4.1 *libtrack*

Dynamic Module. *libtrack*'s dynamic module traces all invocations of native POSIX functions for every thread in the system. At a high level, for each POSIX function implemented in the C standard library of the OS, *libtrack* interposes a special "wrapper" function with the same name. Then, once a native POSIX function is called, *libtrack* logs the time of the invocation and a backtrace identifying the path by which the application invoked the POSIX function. It also measures the time spent executing the POSIX function, excluding any time spent in our wrapper function. *libtrack* then analyzes these traces to construct a call graph and derive statistics and measurements of POSIX API usage.

Interposing on `libc` calls (a particular example of a C POSIX standard library) is challenging, especially when support from `libc` is required to perform the tracking and logging functionality. We wished to run our experiments on actual user devices; this precluded the use of x86 dynamic instrumentation tools like PIN [33]. To trace `libc` invocations, along with their parameters and stack traces, *libtrack* interposes wrapper stubs that invoke the functions exported by `libc`. Several steps are involved:

• *Step 1: libtrack* gathers a list of all `libc` entry points exported in the symbol table for dynamic linking and their offsets within the "TEXT" segment of the original `libc` library. For each of these functions, *libtrack* takes advantage of ELF visibility attributes and marks each symbol's visibility as "HIDDEN" to avoid recursion (explained in Step 4).

• *Step 2:* With each function now hidden in the original `libc`, it is impossible to use `dlsym` to dynamically load them. Thus, *libtrack* creates a static lookup table that maps symbol names to offsets, using the data gathered in Step 1.

• *Step 3:* For each `libc` function, *libtrack* creates a wrapper stub function, which uses `dlopen` to ensure that the original `libc` has been loaded, and then invokes the lookup function created in Step 2. Using the offset returned by the lookup function, the wrapper stub can easily invoke the original `libc` function. The collection of these wrapper stubs will be compiled into a replacement, or *wrapped* `libc`.

• *Step 4:* Many `libc` functions require globally visible data symbols, such as `environ`. In order to avoid duplicating these symbols, *libtrack* ensures that the original `libc` library is loaded by the dynamic linker prior to any other library in the system. This is done through the `LD_PRELOAD` environment variable used by the statically linked `init` binary. Because all the function symbols were hidden in Step 1, dynamically linked binaries will find `libc` functions in the wrapped `libc` generated by *libtrack*, but will use data symbols from the original, preloaded `libc`.

• *Step 5:* A single tracing function in the wrapped `libc` can be used by each `libc` wrapper stub. The stub function can pass the symbol name, arguments, and a pointer to the original `libc` function to the tracing function. The tracing function can dynamically use `libc` functionality through the lookup table generated in Step 2. By replacing the original `libc` with a wrapped version created by *libtrack*, we can track all invocations of POSIX functions by every thread of every application dynamically linking to `libc`.

Static Module. *libtrack* also contains a static module, which is a simple utility to help identify application linkage to POSIX functions of C standard libraries. Given a repository of Android APKs or a repository of Ubuntu packages, *libtrack*'s static module first searches each APK or package for native libraries. Then, it decompiles native libraries and scans the dynamic symbol tables for relocations to POSIX symbols. Dynamic links to POSIX APIs are indexed per application (or per package), and are finally merged to produce aggregate statistics of POSIX linkage on a repository of Android APKs (or on a repository of Ubuntu packages).

Tracing Limitations. There were significant challenges in attempting to trace the full POSIX in both the static and dynamic studies across multiple OSes. This motivated us to constrain tracing to subsets of POSIX in each OS and for each study type. We give complete listings of the functions we trace for each setting at `https://columbia.github.io/libtrack/limitations` and only overview the omissions here.

• For the *static study* we trace: 790 out of 821 C POSIX functions implemented in Android; 1,085 out of 1,115 C POSIX functions implemented in Ubuntu; and we do not run static analysis studies on OS X due to the lack of a large-scale snapshot of the Mac App Store, as noted in Section 4.2.

The only functions we omitted from the static studies were those defined as preprocessor macros and static inlines (31 functions in Android and 30 in Ubuntu), which are not exported in the symbol tables hence they cannot be discovered by *libtrack*. Examples include `htons`, `FD_SET`, and `va_arg`. None of our conclusions about unused POSIX functions refer to these functions, therefore these omissions have no effect on our static studies.

• For the *dynamic study* we trace: 372 out of the 821 C POSIX functions implemented in Android; 462 out of the 1,115 C POSIX functions implemented in Ubuntu; and 897 out of the 1,177 C POSIX functions implemented in OS X. In addition to omitting functions defined as preprocessor macros and static inlines, we omitted functions that were too expensive to trace dynamically because they were invoked too frequently, or were user-space only utility functions that did not make use of OS facilities. The tracing cost was particularly an issue in the context of Android on a resource-constrained tablet device. For Android and Ubuntu, these functions were all string and math-related utility functions, and pthread locking functions (e.g., `pthread_mutex_lock`). For Ubuntu only, we omitted some additional user-space only functions on which *libtrack* failed due to implementation limitations (e.g., `basename`, `sigsetjmp`). For OS X, we were able to trace string and math-related utility functions but had to omit some file system and IPC functions (e.g., `openat`, `mq_open`) due to implementation limitations of our tool. On each OS, most of the omitted functions (93% for Android, 91% for Ubuntu, and 74% for OS X) are user-space utilities, and not system functions, hence we do not believe that their omission has significant qualitative impact on our dynamic studies.

4.2 Workloads

Using *libtrack*, we perform both dynamic and static experiments. We use different workloads for each experiment type, which we describe in this section. All workloads are centered around consumer-oriented applications and do not reflect POSIX's standing in other types of workloads, such as server-side or high-performance computing workloads, as noted in Section 2. Our conclusions must therefore be viewed in light of this limitation.

Dynamic experiments. We drive dynamic experiments by interacting with popular Android, OS X, and Ubuntu applications (apps). We select apps from the official market places for each OS: Google Play (45 apps), Apple AppStore (10 apps), and Ubuntu Software Center (45 apps). We choose apps based on the number of installs across nine categories, selecting 5 apps from each category: social, productivity, games, communication, music, video, travel, shopping, and photography. We interact manually with these applications by performing typical operations, such as refreshing an inbox or sending an email with an email application. Table 2 shows a few examples of applications from our Android dataset, along with examples of actions we perform on them.

Category	Application	Installs	Operations
Social	Facebook	500M-1000M	post, check-in, chat
	Twitter	100-500M	tweet, follow, favorite
Productivity	Dropbox	100-500M	upload, share files
	Adobe	100-500M	open, edit files
Games	Angry Birds	100-500M	play 3 minutes
	Candy Crush	100-500M	play 3 minutes
Communication	Skype	100-500M	video call, chat
	Chrome	100-500M	browse, bookmark
Music, Audio	Shazam	100-500M	search songs, lyrics
	Pandora	100-500M	listen, rate songs
Media, Video	Youtube	500M-1000M	browse, watch videos
	Google Movies	100-500M	watch trailers, rate
Travel, Location	Maps	500M-1000M	query locations
	Google Earth	50-100M	search location
Shopping	Groupon	10-50M	search, start deals
	Amazon	10-50M	search, add items
Photography	PhotoGrid	10-50M	crop pics, create grid
	Aviary	10-50M	add effects to pictures
Total: 45 popular applications across 9 categories (5 apps per category).			

Table 2: Android applications and sample workloads. Applications were chosen based on Google Play popularity.

For the Android, OS X, and Ubuntu studies, we use the following devices: ASUS Nexus-7 tablet with stock Android 4.3 Jelly Bean ROM; MacBook Air laptop (4-core Intel CPU @2.0 GHz, 4GB RAM) running OS X Yosemite; and Dell XPS laptop (4-core Intel CPU @1.7GHz, 8GB RAM) running Ubuntu 12.04 Precise Pangolin.

Static experiments. We drive static experiments of POSIX usage at large scale by downloading over a million consumer applications and checking these applications, and associated libraries, for linkage to POSIX functions of C standard libraries. For Android, we download 1.1 million free Android apps from a Dec. 4, 2014 snapshot of Google Play [59] available on the Internet Archive [34]. For Ubuntu, we download 71,199 packages available for Ubuntu 12.04 on Dec. 4, 2014, using aptitude package manager with the sources list installed in our university's cluster. We do not run static experiments for OS X apps because no large-scale snapshot of the Mac App Store is currently available. Our static experiments focus on measuring linkage of POSIX functions implemented in C standard libraries; static analysis of Java libraries (e.g., for Android apps) or other types of libraries is outside our scope.

5. Results

We organize the results from our study in a sequence of questions akin to those in Section 2. The answer to each question informs the investigation of the subsequent question. We begin with an initial question of which POSIX functions and abstraction families are being used and which are not being used by modern workloads.

5.1 Which Abstractions Are Used and Which Are Not Used by Modern Workloads?

To answer this question, we run a series of investigations using results from different kinds of experiments. First, we examine which abstractions are implemented and which are

Category	Funcs	Android	OS X	Ubuntu	Example	Category (cont'ed)	Funcs	Android	OS X	Ubuntu	Example
Algorithms	12	5	12	12	`bsearch`	Proc. / Signals	98	67	98	98	`fork, sigaction`
Args	19	7	19	19	`uname`	Strings	200	127	200	200	`strlen, strcmp`
Extensions	1	1	1	1	`ioctl`	Terminals	21	19	21	21	`isatty`
File Systems	180	138	180	172	`fopen,fread`	Threads	102	69	102	102	`pthread_create`
IPC	34	11	34	34	`pipe`	Time / Date	29	27	29	29	`time`
Math	285	242	285	285	`srand, logf`	Users / Groups	32	18	32	32	`getuid,getgid`
Memory	28	23	28	25	`[cm]alloc`	Misc	80	23	80	29	`sysconf`
Network	56	44	56	56	`send, recv`						
POSIX: 1,177 functions; **Android**: 821 functions (69.7%); **OS X**: 1,177 functions (100%); **Ubuntu**: 1,115 functions (94.7%).											

Table 3: POSIX compliance of modern OSes. Modern OSes are not fully POSIX compliant.

not in the three OSes under investigation (Android, OS X, and Ubuntu); this will tell us in a definite manner which abstractions are truly omitted by various OS implementations. Second, we use our static analysis of 1.1M apps and packages on Android and Ubuntu to examine which of the implemented abstractions are actually linked by applications or their libraries; this will give us a more accurate view into what abstractions are *not* used by modern workloads. Third, we use results from our dynamic experiments with 45 popular Android apps, 10 OS X apps, and 45 Ubuntu apps, to examine what abstractions are effectively invoked by modern workloads; this will tell us what the popular POSIX abstractions are.

Implemented Abstractions. Table 3 shows the categories, or abstraction subsystems, of POSIX functions implemented in the standard libraries of different OSes. We analyze the following libraries: bionic `libc` for Android (the first version available on Android 4.3), `glibc` for Ubuntu (version 2.13), and `libSystem` for OS X (version 1213). Android's and Ubuntu's implementations are incomplete: of the 1,177 POSIX functions, Android implements 821 (69.7%) and Ubuntu implements 1,115 (94.7%). In contrast, OS X implements the complete set of POSIX interfaces. We give a synoptic per-subsystem analysis of the POSIX implementations, focusing on the omissions:

• *IPC:* Android implements only a small subset (32%) of the POSIX IPC functions; it is the single, lowest-coverage subsystem in the Android implementation. It partially implements two traditional abstractions: pipes and semaphores; and it omits all functions related to shared memory and message queues. The reason for these omissions, discussed in detail later, is that Android replaces traditional IPC abstractions in favor of newer, platform-specific IPC abstractions. Although OS X and Ubuntu also come with their own IPC abstractions, these OSes implement the complete POSIX IPC subsystem.

• *FS:* Android implements 76% of the FS POSIX subsystem, while OS X and Ubuntu cover the entire subsystem. Android omits in particular all of POSIX's asynchronous I/O functions (`aio_*`) and database support functions (e.g., `dbm_*`). This may appear surprising given that the Android OS explicitly pushes toward asynchrony and structured storage [4, 5]. As we show later on, Android departs from POSIX interfaces for such functionality and develops its own abstractions for asynchrony and structured storage.

• *Threads:* Android implements 67% of threading interfaces, omitting for example `pthread_barrier*` and `pthread_spin*` utilities, and partially omitting `pthread_mutexatr*` utilities. OS X and Ubuntu implement the complete threading subsystem.

• *Memory:* All OSes largely implement the memory subsystem, with one exception: neither Android nor Ubuntu implement *typed memory* objects. POSIX typed memory objects are being substituted by new memory sharing and memory mapped files schemas.

• *Other:* Neither Android nor Ubuntu implement the POSIX tracing standard. Since tracing functionality required for debugging and logging is vital for the ecosystem of modern applications, a plethora of tools exist (e.g., [3, 11, 28]) that go beyond the capabilities of POSIX tracing subsystem.

Overall, the omission of some core POSIX interfaces in modern OSes, and the phasing out of traditional IPC, async I/O, and DB calls in Android, provide us with initial insights into which POSIX abstractions are exhibiting limitations or are unnecessary for modern workloads. We investigate the forces driving these transitions in subsequent sections.

Linked Abstractions. We next examine which of the implemented POSIX functions are actually ever linked (and therefore potentially used) by modern applications. Contrary to the previous section, our results in this section do not capture 31 Android and 30 Ubuntu C POSIX functions defined as preprocessor macros and static inlines, a limitation explained earlier in Section 4.1. Figure 1 shows the number of Ubuntu packages and Android applications that dynamically link to POSIX functions of the respective C standard libraries. The results come from our large-scale static studies of (a) 1.1 million Android apps downloaded from the Internet's Archive Dec. 4, 2014 snapshot of Google Play; and (b) 71,199 packages available for Ubuntu 12.04, using aptitude package manager with the sources list maintained in

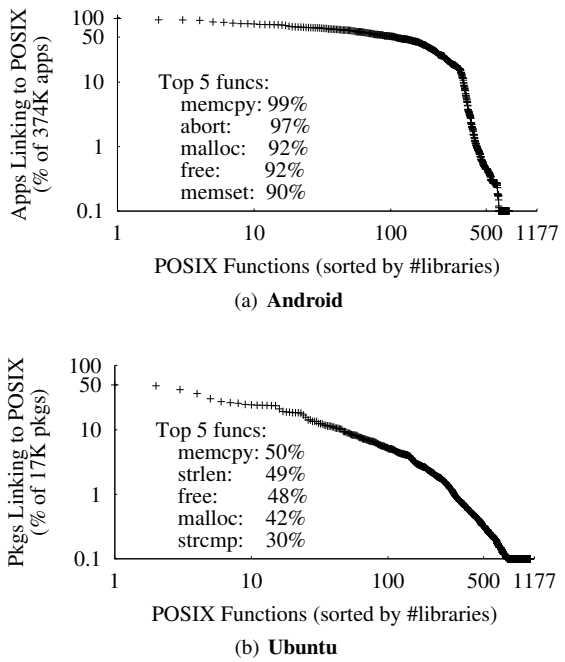

Figure 1: POSIX function linkage (logscale both axis). Static analysis of (a) 374,463 Android apps with native libs and (b) 17,989 Ubuntu packages. Only a fraction of POSIX functions are ever linked.

our university's cluster. For OS X, we have not run this type of analysis. Recall that, as stated earlier in 4.2, our static experiments account only for linkage to native C standard libraries and analysis of other types of libraries (e.g., Java for Android) is outside the scope of our experiments.

Overall, in Android, of the 821 POSIX functions implemented, 114 of them are never dynamically linked from any native library, and approximately half (364 functions) are dynamically linked from 1% or fewer of the apps. Furthermore, our static analysis of Ubuntu packages shows that desktop Linux has a similar, albeit less definitive, trend with Android: phasing out traditional IPC and FS POSIX abstractions. Focusing on a few rarely linked subsystems that are of particular relevance for subsequent discussions, we remark:

• *IPC:* In Android, we observe particularly low linkage for named pipe functions (`mkfifo`). This decreases the already narrow POSIX IPC support in Android. Similarly, in Ubuntu, we observe that message queues (`mq*`) – a set of abstractions not implemented in Android – are being linked by less than 0.3% of Ubuntu packages.

• *FS:* In Android, we observe that file lock functions are extremely rarely linked: `flockfile` and `funlockfile` are linked from only 0.7% of the apps. Also, async I/O calls (`aio*` not implemented in Android) are being linked by less than 0.1% of Ubuntu packages.

Dynamically Invoked Abstractions. Although linkage is a definite way to identify unused functions, it is only a speculative way to infer usage. Therefore, we next examine the actual usage of POSIX functions by quantifying runtime invocations from our dynamic experiments described in Section 4.2. Table 4 shows the top 30 most frequently invoked POSIX calls, along with the CPU times dedicated to executing these functions across the apps studied in each OS. The results are normalized separately on each OS and the calls are sorted in descending order.

• *Memory:* The memory subsystem contains some of the most heavily invoked functions, many of which are among the top-10 across all OSes. User-space utilities for memory handling (e.g., `memset`, `memcpy`, and `memcmp`) and system-call backed POSIX calls for (de)allocation of memory arenas (i.e., `[cm]alloc` and `free`) are the most popular interfaces of this subsystem. Also, the function `mprotect` is ranked 14th in Android. This function is usually invoked with argument `PROT_READ | PROT_EXEC` to set up a not-writable JIT compiler cache. This cache, as evidenced in our stack traces, is then utilized via `setjmp` (ranked 27th in Android). The memory subsystem is also among the most expensive subsystems in terms of CPU consumption of its invocations, as discussed further in Section 5.2. This popularity and cost of memory calls are due to the proliferation of high-level frameworks across all OSes [6, 14, 54] and high-level programming languages [10, 12].

• *Threads:* POSIX user-space threads are also very popular across all OSes. In Android and OS X, Thread Local Storage (TLS) operations are extensively used. `pthread_getspecific`, a function that retrieves the value bound to a TLS key, is the second most popular call in Android and OS X. This user-space routine is used to retrieve TLS keys that help applications map between high-level threads and low-level native `pthreads`. In Ubuntu, conditional variables are heavily used (e.g., `pthread_cond_signal` range). The ubiquity of threading operations is again due to high-level frameworks and programming languages, which make extensive use of multi-threading, particularly to obtain asynchrony, as shown in Section 5.5.

• *IPC* We observe across all three OSes that no POSIX calls belonging to traditional POSIX IPC are among the frequently invoked operations. As discussed earlier, Android departs from the traditional IPC in favor of higher-level IPC abstractions, namely *Binder*, the core Android message passing system. Similarly, OS X supports higher-level IPC primitives built atop *Mach IPC*, which diverged from POSIX since its inception. Finally, Ubuntu also provides applications with message passing capabilities based on D-Bus.

• *FS:* Similarly to IPC, FS abstractions are invoked, but are not comparatively as popular in terms of invocations. More important, most an analysis of the stack traces of these calls reveals that most of the FS invocations, such as file read and write, mainly serve higher-level storage abstractions, such as SQLite in Android and CoreData in OS X. The rationale behind this departure from the traditional FS (and IPC) primitives in favor of higher-level abstractions is further described in subsequent sections.

Popular POSIX interface in Android

POSIX function	Invocations	Time
memset	25% (66.0M)	14% (100s)
pthr_getspec	20% (52.8M)	12% (81.5s)
memcpy	20% (52.1M)	13% (87.8s)
free	9% (24.9M)	8% (57.7s)
malloc	8% (20.9M)	9% (61.1s)
pthr_self	3% (10.1M)	0% (0.0s)
memcmp	3% (8.6M)	2% (16.3s)
memmove	1% (3.6M)	1% (6.8s)
clock_gettime	1% (3.3M)	0% (0.0s)
realloc	0.8% (2.1M)	1% (8.0s)
pthr_once	0.6% (1.7M)	0.7% (4.6s)
gettimeofday	0.5% (1.4M)	0.6% (4.2s)
ioctl	0.5% (1.3M)	16% (108s)
mprotect	0.4% (1.1M)	3% (26.1s)
fread	0.4% (1.0M)	0.8% (5.3s)
write	0.4% (926K)	2% (16.1s)
pthr_cond_sig	0.3% (697K)	0.7% (4.9s)
memchr	0.3% (686K)	0.6% (3.9s)
pthr_cond_brd	0.2% (608K)	0.6% (4.0s)
getpid	0.2% (593K)	0.2% (1.6s)
bsearch	0.2% (493K)	0.2% (1.5s)
pth_mtx_destr	0.2% (455K)	0.1% (0.4s)
calloc	0.2% (393K)	0.2% (1.3s)
getuid	0.2% (393K)	0.1% (1.0s)
recv	0.1% (367K)	0.9% (6.1s)
fclose	0.1% (298K)	0.1% (0.6s)
setjmp	0.1% (287K)	0.3% (1.9s)
pthr_mtx_init	0.1% (275K)	0.1% (0.4s)
read	0.1% (234K)	0.9% (5.8s)
pthr_equal	0.1% (183K)	0.1% (0.4s)
Total:	259.6M	674.7s

Popular POSIX interface in OS X

POSIX function	Invocations	Time
malloc	27% (6.9M)	36% (31.7s)
pthr_getspec	19% (4.9M)	7% (6.2s)
calloc	18% (4.6M)	30% (26.7s)
pthr_self	11% (3.0M)	4% (3.7s)
pthr_equal	11% (2.9M)	3% (3.5s)
pthr_once	5% (1.4M)	2% (1.9s)
gettimeofday	1% (386K)	2% (1.8s)
realloc	1.0% (250K)	2% (2.1s)
bsearch	1.0% (242K)	2% (2.1s)
memcpy	0.5% (115K)	0.3% (0.3s)
pthr_mattrset	0.4% (94K)	0.2% (0.2s)
pthr_mattrinit	0.4% (94K)	0.2% (0.2s)
pthr_mattrdest	0.4% (92K)	0.2% (0.2s)
pthr_setspec	0.3% (65K)	0.1% (0.1s)
getenv	0.1% (31K)	0.1% (0.1s)
recv	0.1% (29K)	0.1% (0.0s)
send	0.1% (29K)	0.1% (0.0s)
nanosleep	0.1% (16K)	0.8% (0.7s)
fcntl	0.1% (16K)	0.1% (0.1s)
fgets	0.1% (16K)	0.3% (0.2s)
mmap	0.1% (14K)	0.4% (0.3s)
munmap	0.1% (12K)	0.2% (0.1s)
qsort	0.1% (11K)	0.1% (.0s)
geteuid	0.1% (11K)	0.1% (.0s)
setjmp	0.1% (10K)	0.1% (.0s)
read	0.1% (9K)	0.2% (0.2s)
pread	0.1% (7K)	0.1% (0.1s)
close	0.1% (6K)	0.2% (0.2s)
write	0.1% (6K)	0.1% (0.1s)
getuid	0.1% (2K)	0.1% (0.0s)
Total:	25M	87s

Popular POSIX interface in Ubuntu

POSIX function	Invocations	Time
memcpy	24% (23.7M)	17% (9.7s)
free	20% (19.1M)	14% (8.2s)
malloc	18% (17.3M)	16% (9.0s)
memcmp	6% (6.1M)	3% (2.2s)
memmove	4% (4.8M)	3% (2.0s)
memset	4% (4.6M)	3% (2.1s)
realloc	3% (3.5M)	3% (2.1s)
calloc	2% (2.8M)	3% (1.7s)
gettimeofday	1% (1.3M)	0% (0.0s)
pthr_cnd_sig	0.9% (869K)	3% (1.7s)
pthr_cnd_twait	0.8% (801K)	0.0% (0.0s)
getpid	0.8% (767K)	0.7% (0.4s)
pthr_cnd_brd	0.8% (748K)	0.7% (0.4s)
pthr_equal	0.8% (735K)	0.4% (0.2s)
sched_yield	0.8% (725K)	0.0% (0.0s)
ffs	0.7% (679K)	0.5% (0.3s)
read	0.7% (643K)	1% (1.1s)
nanosleep	0.6% (603K)	0.0% (0.0s)
poll	0.6% (602K)	3% (1.8s)
memchr	0.6% (584K)	0.4% (0.2s)
fread	0.6% (532K)	0.5% (0.3s)
ioctl	0.4% (425K)	0.7% (0.4s)
recvmsg	0.4% (386K)	3% (1.7s)
fgets	0.3% (320K)	0.3% (0.2s)
write	0.3% (276K)	0.9% (0.5s)
recv	0.3% (272K)	1% (0.6s)
send	0.2% (231K)	2% (1.4s)
pthr_cnd_wait	0.2% (200K)	0.0% (.0s)
sem_wait	0.2% (179K)	0.0% (.0s)
select	0.2% (160K)	0.9% (0.5s)
Total:	95M	56s

Table 4: Popular POSIX Interfaces in Android, OS X, and Ubuntu. Shows which POSIX functions are popular across the three OSes, the frequncy of their invocations, and the total CPU time dedicated into executing these functions. The results are sorted in descending order of number of invocations, normalized separately in each OS. We perform our expriments using the following devices: ASUS Nexus-7 tablet with stock Android 4.3 Jelly Bean ROM; MacBook Air laptop (4-core Intel CPU @2.0 GHz, 4GB RAM) running OS X Yosemite; and Dell XPS laptop (4-core Intel CPU @1.7GHz, 8GB RAM) running Ubuntu 12.04.

• *Other:* Other heavily invoked POSIX calls belong to the Time POSIX subsystem. For example, `gettimeofday` is ranked 12th in Android, 7th in OS X, and 9th in Ubuntu. This system call is essential for implementing timers to support periodic tasks, such as garbage collection.

• *Extension APIs (*`ioctl`*):* Finally, we observe the unexpected popularity of `ioctl`, an extension API that lets applications and libraries bypass well-defined abstractions and interact directly with the kernel. `ioctl` is the 13th most frequently invoked function in Android, the 23rd in Ubuntu, and 42nd most popular function in OS X. `ioctl` also comes with considerable CPU cost. On Android, it is the POSIX call that consumes the most CPU, 16% of the total CPU time dedicated to POSIX calls. This observation surprised us, and motivated us to investigate further *why* this particular call is being used so frequently, and why it is so expensive.

5.2 Why Is IOCTL Invoked So Often?

`ioctl` invocations are surprisingly popular and expensive across all three OSes. In Android, `ioctl` is the 13th most frequently invoked POSIX function and the single most expensive function in CPU time. In OS X, it is the 42nd most invoked function and the 14th most expensive one. And in Ubuntu it is the 23nd most invoked function and the 22nd most expensive one. In this section, we reveal why it is invoked so often and why it consumes so much CPU. We start with a more in-depth profiling of invocations and CPU consumption of various POSIX subsystems (abstractions).

Figure 2 shows the profiling results for the 45 Android applications in our workload. For each application, it shows the breakdown of POSIX invocations (top) and of total CPU time (bottom), split on various POSIX subsystems. The results are normalized separately for each application, and the total invocations and CPU time are shown above each set of bars. In general, more complicated apps lead to a higher number of hits on the POSIX API. Some of the apps with particularly high numbers of total invocations include Google Earth, Google Chrome, and Twitter. Memory and threading POSIX subsystems account for the vast majority of POSIX invocations across all applications (top). In comparison to these two subsystems, `ioctl`'s invocations are much fewer, however they are hardly negligible particularly if one looks at the CPU times breakdowns (bottom). In CPU consumption, `ioctl` becomes one of the most expensive component in POSIX, even compared to entire subsystems. Games, Media & Video, and Photography apps in-

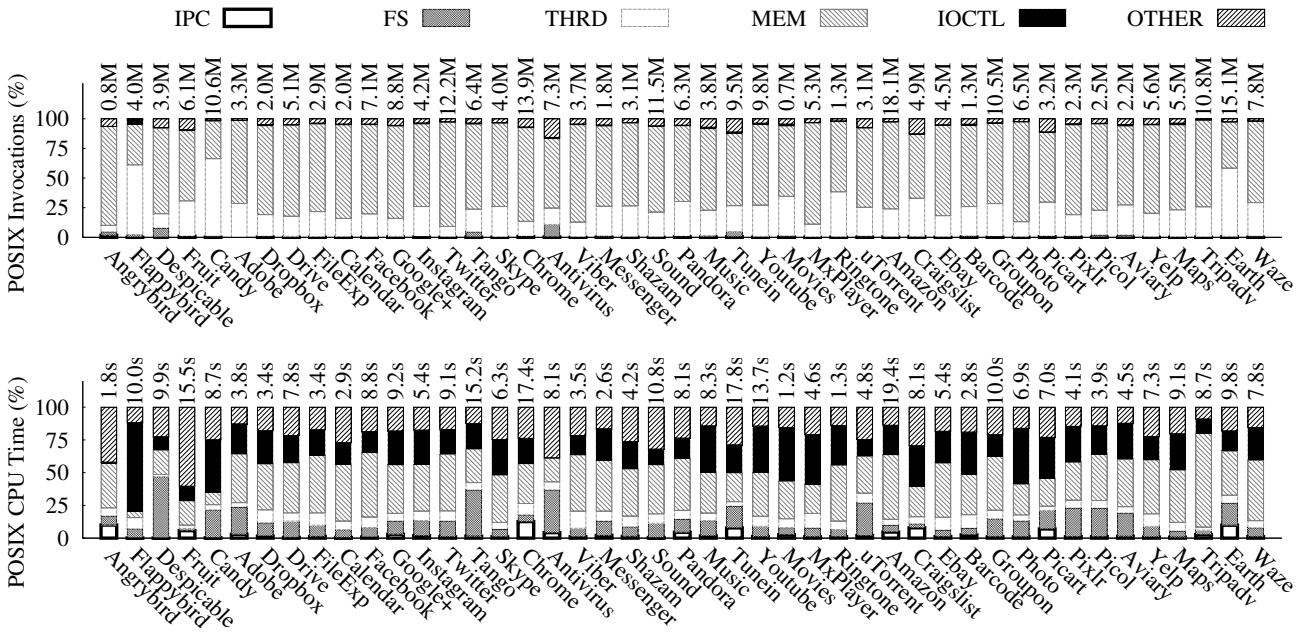

Figure 2: Distribution of POSIX Invocations and CPU Consumption for Various Subsystems. Shows the distribution of POSIX invocations (top subfigure) and CPU consumption (bottom subfigure) for various subsystems, across 45 popular Android applications with which we interact using an ASUS Nexus-7 tablet with stock Android 4.3 Jelly Bean ROM. The results are normalized per application, and the totals appear on top of each set of bars in millions of invocations and seconds respectively. Notably, `ioctl` has a high CPU consumption across most apps, although it is not very popular in terms of invocations.

volve `ioctl` operations with the largest proportional cost. Aside from `ioctl`, the memory subsystem is also a major CPU consumer, in accordance to its popularity.

To understand why `ioctl` is so expensive, we look at the distribution of CPU times for individual `ioctl` invocations across applications in each OS. In Android, we found that `ioctl` calls range from nanoseconds up to milliseconds, with an average of 67 $u$sec and a standard deviation (stdev) of 347 $u$sec. On average, `ioctl` is relatively expensive, compared to a global average of 2 $u$sec per POSIX call and even to a global average of 10 $u$sec per system-call-backed POSIX call. In OS X, the average `ioctl` cost is 513 $u$sec with a standard deviation of 8 $u$sec, and in Ubuntu the average `ioctl` cost is 922 $n$sec and its stdev 867 $n$sec.

We next ask what kind of functionality relies on `ioctl` in the three OSes. To gain initial insight, we develop three trivial apps for Android, which avoid the complexities of real apps, and profile their `ioctl` invocations and CPU times. The apps are: (1) a command-line Java app that prints a message to stdout without interacting with Android framework (*bench1*); (2) an app built atop Android framework and includes one activity printing a message in the screen (*bench2*); and (3) an app that extends the previous activity with a single push-button (*bench3*). Figure 3 shows for each app a breakdown of the number of invocations (top) and CPU time (bottom), split by POSIX subsystems. The command-line app (bench1) contains hardly any traces of `ioctl` in either invocations or CPU time. As soon as we add one trivial UI element to the app (bench2), `ioctl`'s cost becomes dominant over the cost of complete POSIX subsystems. The two specific libraries that invoke `ioctl` in this case are the `libnvrm` and `libnvrm_graphics` graphics libraries. These userspace graphics libraries resort to `ioctl` to bypass the POSIX API, and directly talk to NVDIA's proprietary drivers in the Linux kernel.

To gain a comprehensive view of all the kinds of functionalities that resort to `ioctl` across the three OSes, we look at the stack traces of each `ioctl` invocation to identify those libraries that invoke `ioctl` frequently. We classify these libraries based on their the type of functionality they implement. Table 5 shows the top three libraries that account for most of the invocations in the three OSes. In Android, graphics libraries (`libnvrm` and `libnvrm_graphics`) lead to the lion's share of `ioctl` invocations. Next comes *Binder*, Android's core IPC mechanism, whose functionality is split between a Linux kernel module and a userspace library. In OS X, the majority of `ioctl` invocations come from network libraries. In Ubuntu, graphics libraries trigger approximately half of `ioctl` invocations and the remaining part is mainly due to libraries providing network functionality.

We next ask *why* are these libraries relying on `ioctl`? Could it be that POSIX is missing some abstractions needed by these libraries, hence the libraries must resort to unstruc-

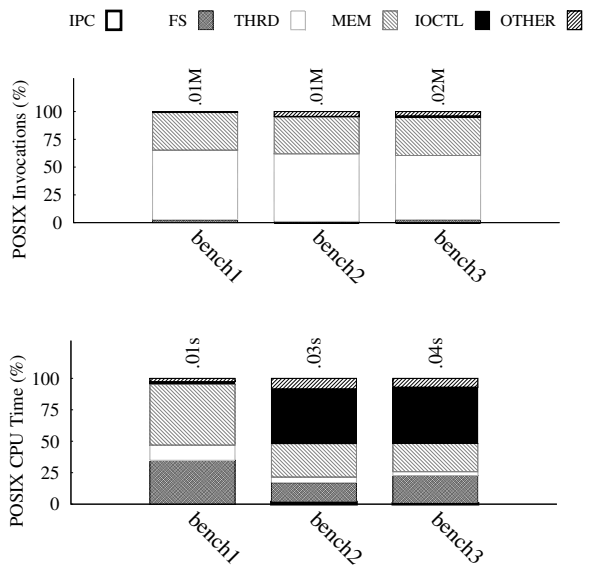

Figure 3: Distribution of POSIX Invocations and CPU Consumption for Three Micro-benchmark Apps. Shows the distribution of POSIX invocations (top) and CPU consumption (bottom) for various subsystems, across 3 micro-benchmark apps. The first benchmark app includes no GUI elements. The next two, include an activity and an extended activity that both lead to expensive graphics-related `ioctl` operations.

OS	1st Lib	2nd Lib	3rd Lib	Invoc.
Android	Graphics (74%) (e.g., `libnvrm`)	Binder (24%) (e.g.,`libbinder`)	Other (1%)	1.3M
OS X	Network (99%) (e.g.,`net.dylib`)	Loader (0.6%) (e.g.,`dyld`)	-	682
Ubuntu	Graphics (52%) (e.g., `libgtk`)	Network (47%) (e.g.,`libQtNet`)	Other (1%)	0.4M

Table 5: Top Libraries that Invoke IOCTL. Shows the percentage of `ioctl` invocations the top-3 libraries lead to. Graphics libraries and network functionality are the main sources of `ioctl` invocations.

tured, extension APIs to implement their functionality? We address this question next.

5.3 Does POSIX Lack Abstractions Needed by Modern Workloads?

Taking hints from Table 5, we investigate graphics and networking functionality in modern OSes, which rely significantly on `ioctl` signaling that such abstractions are missing in POSIX. In the following section, we additionally discuss IPC and storage abstractions, which are examples of abstractions that are replacing older abstractions and whose implementation also relies quite heavily on extension APIs.

Graphics. POSIX explicitly omits graphics as a point of standardization. As a result, there is no standard OS interface to graphics cards, and different platforms have chosen different ways to optimize this critical resource. The explicit omission of a graphics standard could be due to the emergence of OpenGL [35, 36], a popular cross-platform graphics API used in drawing and rendering. While OpenGL works well for applications, OS and graphics library designers have no well-defined interfaces to accomplish increasingly complicated operations. Because these operations leverage optimized, and often black-box, GPU hardware, the OS kernel must marshal opaque data from user space graphics APIs into vendor-specific hardware.

The lack of a standard kernel interface to GPUs has lead to sub-optimal performance and limited extensibility. Modern GPU hardware, and its associated drivers and libraries, are trending towards a general purpose computational resource, and POSIX is not in a position to standardize the use of this new, powerful paradigm. In the past 4 years, these problems have been studied in detail [38, 42, 45, 46, 50]. For example, the PTask API [45] defines a dataflow based programming model that allows the application programmer to utilize powerful GPU resources in a more intuitive way. This new OS abstraction results in massive performance gains by matching API semantics with OS capabilities.

The lack of a standard graphics OS abstraction also causes development and runtime problems. It is the primary reason `ioctl` is so frequently used in Android and Ubuntu. With no alternative, driver developers are forced to build their own structure around the only available system call, `ioctl`. Using opaque input and output tokens, the `ioctl` call can be general purpose, but it was never intended for the complicated, high-bandwidth interface GPUs require. Graphics library writers must either use ioctl as a funnel into which all GPU command and control is sent via opaque data blobs, or they must design a vendor-specific demux interface akin to the `syscall` system call. This is unfortunate and unnecessary work for both library and driver developers.

Interestingly, in OS X, graphics functionality does not account for any significant portion of `ioctl` invocations. The reason is that the driver model in OS X, `IOKit`, is more structured than in Android or Ubuntu. It uses a well-defined Mach IPC interface that can effectively marshal parameters and graphics data across the user-kernel boundary. This creates a re-usable, vendor-agnostic API, however, the current interface is designed around the same black-box hardware that runs on other platforms. Therefore, a standardized OS interface to graphics processors would likely have the same benefits on OS X as it would on Ubuntu or Android.

In summary, with no standard OS interface to GPUs, system designers for Android, Ubuntu, and OS X are forced to rely upon extension POSIX interfaces (`ioctl`) or to invent new interfaces. As a result, cross-platform system programming for GPUs remains a challenge to future system research. We illustrated this challenge based on past research on the *Cycada* iOS-Android binary compatibility system in Section 2.

Networking. Although POSIX has a set of 56 APIs to support network operations, Table 5 shows that developers reg-

ularly use `ioctl` to express additional customized network functionality in support of their systems. 99% of `ioctl`'s invocations in OS X and 47% in Ubuntu, were low-level socket operations. The particular use of `ioctl` appeared to be in an attempt to circumvent around POSIX restrictions on exactly what appears in headers files for operations like `[set|get]sockopt`. Unlike Graphics, which are explicitly omitted as a point of standardization from POSIX, the use of `ioctl` to perform network operations is surprising and indicates the fact that developers miss fully-standardized support to express desired network operations, and therefore resort to `ioctl`'s support.

5.4 What POSIX Abstractions Are Being Replaced?

Continuing with hints from Table 5, we next discuss abstractions that exist in POSIX but appear to be replaced by new abstractions. We seek to understand what the replaced abstractions are, why they are being replaced, what the new replacement abstractions are, and whether the new abstractions are converging or diverging across the three OSes.

Inter-Process Communication. A central IPC abstraction in POSIX is the message queue API (`mq_*`). On all platforms we studied, applications used some alternate form of IPC. In fact, Android is missing the `mq_*` APIs altogether. IPC on all of these platforms has divergently evolved beyond (in some cases parallel to) POSIX.

• *IPC on Android: Binder.* In Android, Binder is the standard method of IPC. Binder APIs are exposed to apps through highly abstracted classes and interface definition files that give structure and meaning to all communication. Some of the insurmountable limitation of traditional POSIX IPC that urge for new IPC mechanism include: **(i)** Filesystem-based IPC mechanisms, e.g., named pipes, cannot be used in Android (and other similarly sandboxed systems) due to a global security policy that eliminates world-writable directories. **(ii)** Message queues and pipes cannot be used to pass file descriptors between processes. **(iii)** POSIX IPC primitives do not support the context management necessary for service handle discovery. **(iv)** Message queues have no support for authorization or credential management between senders and recipients. **(v)** There is no support for resource referencing and automatic release of in-kernel structures.

Binder overcomes POSIX IPC limitations and serves as the backbone of Android inter-process communication. Using a custom kernel module, Binder IPC supports file descriptor passing between processes, implements object reference counting, and uses a scalable multi-threaded model that allows a process to consume many simultaneous requests. In addition, Binder leverages its access to processes' address space and provides fast, single-copy transactions. When a message is sent from one process to another, the in-kernel Binder module allocates space in the destination process' address space and copies the message directly from the source process' address space.

Binder exposes IPC abstractions to higher layers of software in appropriately abstract APIs that easily support service discovery (through the `Service Manager`), and registration of RPCs and intent filtering. Android apps can focus on logical program flow and interact with Binder, and other processes, through what appear to be standard Java objects and methods, without the need to manage low-level IPC details. Because no existing API supported all the necessary functionality, Binder was implemented using `ioctl` as the singular kernel interface. Binder IPC is used in every Android application, and accounts for nearly 25% of the total measured POSIX CPU time which funnels through the `ioctl` call.

• *IPC on Linux: D-Bus and KD-Bus.* In Ubuntu, the D-Bus protocol [25] provides apps with high-level IPC abstractions. D-Bus describes an IPC messaging bus system that implements **(i)** a system daemon monitoring system-wide events, e.g., attaching or detaching a removable media device, and **(ii)** a per-user login session daemon for communication between applications within the same session. There are several implementations of D-Bus available in Ubuntu. The applications we inspect use mostly the *libdbus* implementation (38 out of 45 apps). This library is, at the low level, implemented using traditional Unix domain sockets, and it accounts for less than 1% of the total CPU time measured across our Ubuntu workloads.

Another recent evolution in IPC is called the Linux Kernel D-Bus, or *kdbus*. The kdbus system is gaining popularity in GNU/Linux OSes. It is an in-kernel implementation of D-Bus that uses Linux kernel features to overcome inherent limitations of user space D-Bus implementations. Specifically, it supports zero-copy message passing between processes. Also, as opposed to D-Bus, it is available at boot, and there is no need to wait for the D-Bus daemon to bootstrap. Thus, Linux security modules can directly leverage it.

• *IPC on OS X: Mach IPC.* IPC in OS X diverged from POSIX (and its Android/Linux contemporaries) since its inception. Apple's XNU kernel uses an optimized descendant of CMU's Mach IPC [16, 44, 61] as the backbone for inter-process communication. Mach comprises a flexible API that supports high-level concepts such as: object-based APIs abstracting communication channels as *ports*, real-time communication, shared memory regions, RPC, synchronization, and secure resource management. Although flexible and extensible, the complexity of Mach has led Apple to develop a simpler higher-level API called *XPC* [17]. Most apps use XPC APIs that integrate with other high-level APIs, such as Grand Central Dispatch [18], and `launchd` the Mach IPC based *init* program providing global IPC service discovery.

To highlight key differences in POSIX-style IPC and newer IPC mechanisms, we adapt a simple Android Binder benchmark found within the Android source code to measure both pipes and unix domain sockets as well as Binder transactions. We also use the MPMMTest application found

Android				OS X			Ubuntu	
Tx/Rx	Pipes	Unix	Binder	Pipes	Unix	Mach	Pipes	Unix
	avg ($u$sec)	avg ($u$sec)	avg ($u$sec)	avg ($u$sec)	avg ($u$sec)	avg ($u$sec)	avg ($u$sec)	avg ($u$sec)
32 bytes	40	54	115	6	11	19	18	18
128 bytes	44	56	114	7	51	19	20	10
1 page	62	73	93	8	54	12	21	20
10 pages	291	276	93	18	175	15	58	27
100 pages	2402	1898	94	378	1461	12	923	186

Table 6: Latency Comparison of Different IPC mechanisms on Android, OS X, and Ubuntu. Custom Binder benchmark from AOSP adapted for pipes and unix domain sockets. Mach IPC measurements use the MPMMTest from open source XNU. Android benchmarks use an ASUS Nexus-7 tablet with Android 4.3 Jelly Bean; OS X benchmarks use a MacBook Air laptop (4-core Intel CPU @2.0 GHz, 4GB RAM) running OS X Yosemite; Ubuntu benchmarks use a Dell XPS laptop (4-core Intel CPU @1.7GHz, 8GB RAM) running Ubuntu 12.04 Precise Pangolin. The results are averaged over 1000 repetitions and the time is reported in $u$sec. Both Binder and Mach IPC are nearly constant time in the size of the transaction.

in Apple's open source XNU [13]. We measure the latency of a round-trip message using several different message sizes, ranging from 32 bytes to 100 (4096 byte) pages. The results are summarized in Table 6. Both Binder and Mach IPC leverage fast, single-copy and zero-copy mechanisms respectively. Large messages in both Binder and Mach IPC are sent in near-constant time. In contrast, traditional POSIX mechanisms on all platforms suffer from large variation and scale linearly with message size.

In summary, driven by the common need for feature-rich IPC interfaces, and given the limitations of traditional POSIX IPC, different OSes have created similar but not converging, and non-standards adherent, IPC mechanisms.

File System. Several papers have already identified the migration of modern applications away from traditional file system abstractions into higher-level storage abstractions, such as relational databases or object-relational managers [30, 52]. The departure from the traditional POSIX file system into higher-level storage abstractions, i.e., *sqlite*, was the key insight that allowed *Pebbles* reconstruct application-level objects without input from applications. Besides the benefits of new higher-level storage abstractions, there are also significant performance overheads if the underlying file system, designed based on certain optimization decisions – such as lazy-copying of in-kernel file system caches – is used by applications in ways that violate these assumptions. Our measurements complement these previous works.

First, there is a clear trend of transitioning from regular unstructured data files into structured data. In Android, 35 of the 45 apps we checked depend on structured data stored in *sqlite*. Typical Android applications that depended on structured data included Media & Video, Accessories & Development, and Games applications. On the other hand, in Ubuntu, the transitioning to structured data is less definite. Only 12 of the 45 applications are using structured data – mainly the web-oriented apps. These results suggest that *Pebbles*'s design from past work [52], as discussed in Section 2, may extend to OS X but is unlikely to apply to Ubuntu.

Second, all the Android applications we checked issue file system-related POSIX calls through *libsqlite*. The use of this library causes five expensive `fsync` calls for every single INSERT/UPDATE/DELETE operation. Overall, the causes of the migration into higher-level storage abstractions have been discussed [48]: databases allow high-level, convenient access to structured data. The open question, however, remains: how should operating systems and file systems better support these new and significant abstractions?

Asynchronous I/O. Our experiments reveal another interesting evolutionary trend: the replacement of POSIX APIs for asynchronous I/O with new abstractions built on multi-threading abstractions. The nature and purpose of threads has been a debate in OS research for a long time [21, 37, 41]. POSIX makes no attempt to prioritize a threading model over an event-based model; it simply outlines APIs necessary for both. Our study revealed that while POSIX locking primitives are still extremely popular and useful, new trends in application abstractions are blurring the line between event and thread by combining high-level language semantics with pools of threads fed by event-based loops. This new abstraction or programming paradigm has evolved directly from the unique operating environment of mobile devices. The intimate nature of interacting with a computer via touch intuitively drives low-latency and fast response time requirements for application GUIs. While an event-based model may seem the obvious solution, event processing in the input or GUI context quickly leads to sub-optimal user experience, especially when display refresh rates require $< 16ms$ intra-frame processing time to maintain smooth and responsive GUIs. Dispatching event-driven work to a queue backed by a pool of pre-initialized threads has become a de facto standard programming model in Android, OS X, and Ubuntu. Although this paradigm appears in all the OSes we studied, the implementations are extremely divergent.

• *Asynchrony in Android: Thread Pools and Event Loops.* Android defines several Java classes which abstract the concepts of creating, destroying, and managing threads (`ThreadPool`), looping on events (`Looper`), and asynchronous work dispatching (`ThreadPoolExecutor`). A Looper class can dispatch bits of work based on input events to a ThreadPoolExecutor that manages a set of threads. For

example, one can use Looper to asynchronously dispatch file downloads for processing into a ThreadPool.

• *Asynchrony in Ubuntu: Thread Pools and Event Loops.* Ubuntu applications can choose from a variety of libraries providing similar functionality, but through vastly different interfaces. The `libglib`, `libQtCore`, and `libnspr` all provide high-level thread abstractions based on the GNOME [27] desktop environment, and account for more than 13% of all measured POSIX CPU time. In particular, `libglib` provides the `GSource`, `GThreadPool`, and `GAsyncQueue` C-structure based APIs that perform conceptually similar functions to their Android Java counterparts. In addition, the `libQtCore` library implements threading abstractions for Qt-based Ubuntu applications.

• *Asynchrony in OS X: Grand Central Dispatch.* In OS X the APIs are, yet again, different. The vast majority of event-driven programming in OS X is done through Mach IPC, and corresponding kernel system calls comprise the majority of system CPU time [20]. Apple has written C, Objective-C, and Swift based APIs around event handling, thread pool management, and asynchronous task dispatch. Most of these APIs are enabled through Grand Central Dispatch [18] (GCD). GCD manages a pool of threads, and even defines POSIX alternatives to semaphores, memory barriers, and asynchronous I/O. Low-level objects such as `dispatch_queue_t` and `dispatch_source_t` are managed by functions such as `dispatch_async` and `dispatch_source_set_event_handler`. The GCD functionality is exported to even higher levels of abstraction in classes such as `NSOperation` [19]. Apple has even written a tutorial [15] for developers on how to migrate *away* from POSIX threads.

A fascinating practical result of exposing high-level event and thread management APIs to app developers is that the number of threads in the system increases. In our measurements we found that our idle Android tablet had 690 instantiated threads, our idle MacBook Air laptop had 580 instantiated threads, and our idle Ubuntu laptop had 292 instantiated threads. Particularly for Android, more than 100 instantiated threads were directly tied to custom Binder, Looper, and ThreadPool management APIs.

In summary, driven by the strong need for asynchrony and the event-based nature of modern GUI applications, different OSes have created similar but not converging and non-standard adherent threading support mechanisms.

5.5 Unintended Effects of POSIX Use

We end our study with some interesting findings about unintended side-effects due to lack of use of various POSIX corners. Section 5.1 revealed that some POSIX APIs, although implemented, are never actually used by real, consumer-oriented applications. We discuss two implications of this aspect here. First, it is a well-known fact that unused (dead) code in any program is a liability [40, 47, 51]. If a piece of code is unused, its testing, maintenance, and debugging becomes a lower priority for developers, increasing the chance

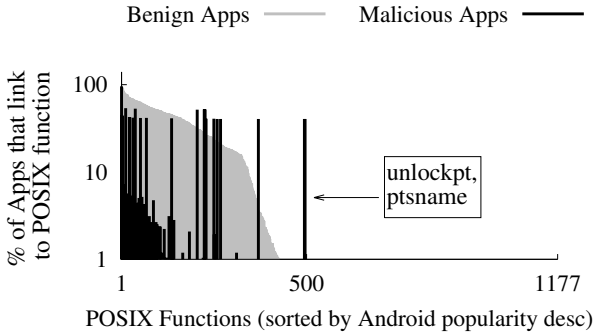

Figure 4: Popularity of POSIX in Benign vs. Malicious Apps.

of bugs to survive over long periods. Our study reveals that this observation applies to POSIX and its implementation in Android. We find that the `syslog` function, a POSIX logging utility, is implemented by Android's bionic libc, but the implementation is actually broken. Not only is there no syslog daemon process to which the `syslog` function would communicate, but the code also attempts to connect to a Unix domain socket named `/dev/kmsg` which in Android is actually a character device! (Incidentally, we found a similar side-effect in our prior work *Cycada* [7]: in iOS, the `select` system call, presumably also equally rarely used as in the OSes we study here, is buggy; its latency scales linearly with the number of file descriptors and the call fails to return when selecting on more than 250 descriptors.)

Second, it is well-known that malware developers often exploit interfaces known to be old, poorly maintained, and buggy. Our study reveals evidence of this aspect, as well. We examined the POSIX functions linked from 1,260 malicious Android applications available on a popular malware corpus [62]. Figure 4 compares the distribution of POSIX APIs in 45 popular (and hopefully benign) Android apps with that of POSIX APIs in the malicious apps. The POSIX calls on the x axis are sorted by their popularity in the benign apps. The spikes appearing toward the right-hand side of the graph show that malware often leverages APIs that are not linked by benign applications. For example, a set of pseudo-terminal functions are quite popular among the malicious apps (40.5%), but hardly linked in benign apps (0.45%). Such functions might be used by malware to open remote shells on compromised devices. Thus, unused corners of POSIX can be abused and should be well-understood to properly protect them.

6. Related Work and Discussion

Our results demonstrate a clear transition away from POSIX for four key OS subsystems: graphics, IPC, threading, and storage. The trend away from POSIX in each of these areas has followed different, and often divergent, paths on different platforms. The changes likely stem from POSIX's limitations in supporting modern workloads, which are increasingly graphical, latency-sensitive, and memory-constrained (particularly on mobile devices). To sustain the new workloads, and take full advantage of the new, pow-

erful hardware available in these devices, OS designers have evolved some POSIX APIs and concepts to fit newer paradigms, left other POSIX APIs in dusty corners of seldom used libraries, and invented new APIs outside the original bounds of POSIX.

All of these evolutions are natural and positive, however they have a downside: as the trend away from POSIX follows different and often divergent paths on different platforms, POSIX's cross-OS portability goal becomes even more distant. What should OS developers, researchers, standards bodies, and educators do about this trend? Should we allow this departure from POSIX to continue risking even the limited cross-OS portability we have today? Should we instead examine these trends to determine whether POSIX is due for an upgrade? Should educators evolve their courses to cover the new abstractions that are taking form in modern OSes, e.g., IPC, and if so, what new abstractions should they cover?

Several recent works have explored system designs that build upon this departure from the POSIX API to reap benefits of various kinds. Section 2 provided two examples from our own experience, but there are numerous other examples. Arrakis [43] proposes a new operating system design to circumvent the kernel on I/O operations, including changing the POSIX API, to attain superior performance. A POSIX version of Arrakis is also developed, with better performance than Linux, but not as good as the non-POSIX design. Flux [31] enables app migration across heterogeneous Android devices by leveraging higher-level Android APIs to avoid the need to capture low-level device state managed by device-specific `ioctl` extensions. OptFS [23] departs from traditional POSIX semantics for file system design, with a design that requires only minor modifications to POSIX, to yield performance improvements for some workloads that are an order of magnitude better. Furthermore, a plethora of large scale distributed file systems have moved away from POSIX to avoid performance penalties, including Ceph [60], HDFS [49], and GFS [26], the latter of which is justified in part due to its support of customized internal infrastructure and therefore lacks the need to support POSIX semantics. Our work is complementary to all of these works, providing the first rigorous characterization of evolution of POSIX abstractions across multiple OSes. Our results can be leveraged by developers, researchers, and standards bodies to adapt their applications and optimization and standardization efforts toward the new or changed abstractions.

Two recent studies [30, 58] make related observations about modern use of system-level APIs. A 2011 paper [30] observes that file system APIs are being used in unexpected ways in modern OSX workloads. One reason, the authors observe, is that system-level workloads are driven by high-level data management libraries. We make this and other observations at the broader scope of all system abstractions standardized in POSIX across multiple OSes.

A paper concurrent to ours [58] statically analyzes all packages in the Ubuntu 15.04 distribution with the goal of identifying frequently-used system APIs and developing metrics to evaluate research prototypes for compatibility with modern consumer-oriented workloads. While driven by different goals, our studies reach several common conclusions, including that large portions of POSIX are unused by modern workloads and that extension APIs, such as `ioctl`, are used extensively. From the standpoint of these common conclusions, each study adds complementary dimensions that mutually strengthen our findings. Their study covers a broader set of system APIs, considering `/proc` pseudo files and the many specific `ioctl`/`fcntl`/`prctl` codes in addition to standard POSIX functions in libc. Our study generalizes the observations to multiple OSes and uses dynamic experiments, in addition to static analysis, to reveal the performance implications of heavy extension API usage. Finally, our study identifies forces driving the evolution of several key abstractions, including graphics and IPC, and uniquely illustrates how new OS functionality is built on extensive use of extension APIs.

7. Conclusions

Perfect application portability across all UNIX-based OSes is clearly beyond the realm of possibility. However, maintaining at least a subset of harmonized abstractions is still a viable alternative for preserving some degree of uniformity within the UNIX-based OS ecosystem. Within the context of its limitations – such as the exclusive focus on consumer-oriented workloads – our study shows that POSIX abstractions are evolving in significant ways in three modern UNIX-based OSes (Android, OS X, and Ubuntu), and that changes are not converging to any new unified set of abstractions. Parts of the POSIX standard appear unnecessary for consumer-OS developers to implement, others include a set of obsolete abstractions whose implementations are sometimes buggy and abused by attackers, and a set of new abstractions necessary for modern workloads is completely missing. We believe that a new revision of the POSIX standard is due, and urge the research community to investigate what that standard should be. Our study provides a few examples of abstractions, graphics, IPC, storage, and networking, as starting points for re-standardization, and we recommend that any changes be informed by popular frameworks that are now taking a principal role in modern OSes.

8. Acknowledgments

Michael Swift provided valuable feedback on earlier drafts of this paper. Andrei Papancea and Benjamin Hanser helped with running application workloads and obtaining measurements. This work was supported in part by DARPA Contract FA8650-11-C-7190, NSF grants CNS-1351089, CNS-1162447, CNS-1422909, and CCF-1162021, a Google Research Award, and a Microsoft Research Faculty Fellowship.

References

[1] SQLite. https://www.sqlite.org/. Accessed: 03/08/2016.

[2] Adobe Systems Inc. PhoneGap — Home. http://phonegap.com/. Accessed: 3/19/2015.

[3] Android.com. Android Developer, Debugging. http://developer.android.com/tools/debugging/index.html. Accessed: 10/18/2015.

[4] Android.com. Android Developer, Operations to Multiple Threads. http://developer.android.com/training/multiple-threads/index.html. Accessed: 10/18/2015.

[5] Android.com. Android Developer, Saving Data. http://developer.android.com/training/basics/data-storage/index.html. Accessed: 10/18/2015.

[6] Android.com. ART and Dalvik. https://source.android.com/devices/tech/dalvik/. Accessed: 10/18/2015.

[7] J. Andrus, A. V. Hof, N. AlDuaij, C. Dall, N. Viennot, and J. Nieh. Cider: Native Execution of iOS Apps on Android. In *Proceedings of the* 19^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, pages 367–382, Salt Lake City, Utah, USA, Mar. 2014.

[8] J. Andrus and J. Nieh. Teaching Operating Systems Using Android. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE 2012)*, pages 613–618, Raleigh, NC, USA, Mar. 2012.

[9] Appcelerator Inc. Enterprise Mobile Application Development Platform — The Appcelerator Platform. http://www.appcelerator.com/platform/appcelerator-platform/. Accessed: 3/19/2015.

[10] Apple Inc. Developer, Objective-C. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html. Accessed: 10/18/2015.

[11] Apple Inc. Developer support, debugging. https://developer.apple.com/support/debugging. Accessed: 10/18/2015.

[12] Apple Inc. Developer, Swift. https://developer.apple.com/swift/. Accessed: 10/18/2015.

[13] Apple, Inc. OS X 10.11.2 - Source. http://opensource.apple.com/tarballs/xnu/xnu-3248.20.55.tar.gz. Accessed: 3/20/2016.

[14] Apple Inc. OS X Frameworks. https://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/OSX_Technology_Overview/SystemFrameworks/SystemFrameworks.html. Accessed: 10/18/2015.

[15] Apple, Inc. Migrating Away from Threads. https://developer.apple.com/library/ios/documentation/General/Conceptual/ConcurrencyProgrammingGuide/ThreadMigration/ThreadMigration.html, Dec. 2012. Accessed: 03/25/2015.

[16] Apple, Inc. Mach Overview. https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html, Aug. 2013. Accessed: 03/22/2015.

[17] Apple, Inc. Creating XPC Services. https://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingXPCServices.html, July 2014. Accessed: 03/22/2015.

[18] Apple, Inc. Grand Central Dispatch (GCD) Reference. https://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html, Sept. 2014. Accessed: 03/19/2015.

[19] Apple, Inc. NSOperation Class Reference. https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSOperation_class/index.html, Sept. 2014. Accessed: 03/25/2015.

[20] Apple Kernel Engineer. Personal communication, Mar. 2015.

[21] R. V. Behren, J. Condit, and E. Brewer. Why Events Are a Bad Idea (for High-concurrency Servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, pages 4–4, Lihue, HI, USA, May 2003.

[22] Box. Box Platform Developer Documentation. https://developers.box.com/sdks/. Accessed: 3/19/2015.

[23] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP 2013)*, pages 228–243, Farminton, PA, USA, Nov. 2013.

[24] Corona Labs. Cross-Platform Mobile App Development for iOS, Android - Corona Labs. https://coronalabs.com/. Accessed: 3/19/2015.

[25] freedesktop.org. D-Bus. http://www.freedesktop.org/wiki/Software/dbus/. Accessed: 10/18/2015.

[26] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 29–43, Bolton Landing, NY, USA, Oct. 2003.

[27] GNOME.org. gnome. http://www.gnome.org/. Accessed: 01/14/2014.

[28] gnome.org. Gnome Developer, Debugging. https://developer.gnome.org/gtk3/stable/gtk-running.html#interactive-debugging. Accessed: 10/18/2015.

[29] Google Inc. Binder — Android Developers. http://developer.android.com/reference/android/os/Binder.html, Mar. 2015. Accessed: 03/19/2015.

[30] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the* 23^{rd} *ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 10:1–10:39, Cascais, Portugal, Oct. 2011.

[31] A. V. Hof, H. Jamjoom, J. Nieh, and D. Williams. Flux: Multi-Surface Computing in Android. In *Proceedings of the* 10^{th} *European Conference on Computer Systems (EuroSys 2015)*, pages 24:1–24:17, Bordeaux, France, Apr. 2015.

[32] IEEE and The Open Group. POSIX.1 Backgrounder. http://www.opengroup.org/austin/papers/backgrounder.html, 2003. Accessed: 01/14/2015.

[33] Intel. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pintool. Accessed: 01/14/2014.

[34] Internet Archive. Android Apps. https://archive.org/details/android_apps&tab=about. Accessed: 03/15/2016.

[35] Khronos Group. OpenGL Specification, Feb. 2011. http://www.khronos.org/opengl/.

[36] Khronos Group. History of OpenGL - OpenGL.org. https://www.opengl.org/wiki/History_of_OpenGL, Jan. 2015. Accessed: 03/22/2015.

[37] H. C. Lauer and R. M. Needham. On the Duality of Operating System Structures. *SIGOPS Operating Systems Review*, 13(2):63–76, Apr. 1979.

[38] K. Menychtas, K. Shen, and M. L. Scott. Enabling OS Research by Inferring Interactions in the Black-box GPU Stack. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC 2013)*, pages 291–296, San Jose, CA, USA, June 2013.

[39] Morning Tec. Stella SDK. http://www.stellasdk.org/. Accessed: 3/19/2015.

[40] NSA. Guidance for Addressing Malicious Code Risk. https://www.nsa.gov/ia/_files/guidance_for_addressing_malicious_code_risk.pdf, Sept. 2007. Accessed: 03/24/2015.

[41] J. K. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference (USENIX ATC 1996), Jan. 1996.

[42] S. Panneerselvam and M. M. Swift. Operating Systems Should Manage Accelerators. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar 2012)*, Berkeley, CA, USA, June 2012.

[43] S. Peter, J. Li, , I. Zhang, D. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI 2014)*, pages 1–16, Broomfield, CO, USA, Oct. 2014.

[44] R. Rashid, R. Baron, A. Forin, R. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi. Mach: A Foundation for Open Systems. In *In Proceedings of the 2nd Workshop on Workstation Operating Systems. IEEE (WWOS-II)*, pages 109–112, Pacific Grove, CA, USA, Sept. 1989.

[45] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 233–248, Cascais, Portugal, Oct. 2011.

[46] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP 2013)*, pages 752–757, Farminton, PA, USA, Nov. 2013.

[47] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), Sept. 1975.

[48] M. Seltzer and N. Murphy. Hierarchical File Systems Are Dead. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS 2009)*, pages 1–1, Monte Veritá, Switzerland, May 2009.

[49] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST 2010)*, pages 1–10, Incline Village, NV, USA, May 2010.

[50] M. Silberstein, B. Ford, I. Keidar, and Emmett. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, pages 485–498, Houston, TX, USA, Mar. 2013.

[51] R. Smith. A Contemporary Look at Saltzer and Schroeder's 1975 Design Principles. *IEEE Security and Privacy*, 10(6), Nov. 2012.

[52] R. Spahn, J. Bell, M. Z. Lee, S. Bhamidipati, R. Geambasu, and G. Kaiser. Pebbles: Fine-grained Data Management Abstractions for Modern Operating Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI 2014)*, pages 113–129, Broomfield, CO, USA, Oct. 2014.

[53] The Austin Group. The Austin Group. http://www.opengroup.org/austin/. Accessed: 03/25/2014.

[54] The GTK+ Project. GTK+. http://www.gtk.org/. Accessed: 10/18/2015.

[55] The Open Group. Last POSIX Revision. http://pubs.opengroup.org/onlinepubs/9699919799. Accessed: 03/19/2015.

[56] The Open Group. The Open Group Base Specifications Issue 7: Base Definitions. http://pubs.opengroup.org/onlinepubs/9699919799/. Accessed: 01/14/2014.

[57] The Open Group. The Open Group Base Specifications Issue 7: System Interfaces. http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html. Accessed: 01/14/2014.

[58] C.-C. Tsai, B. J. Nafees, A. Abdul, and D. E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support When You' re Supporting. In *Proceedings of the* 11^{th} *European Conference on Computer Systems (EuroSys 2016)*, London, United Kingdom, Apr. 2015.

[59] N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2014)*, pages 221–233, Austin, TX, USA, June 2014.

[60] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on*

Operating Systems Design and Implementation (OSDI 2006), pages 310–320, Seattle, WA, USA, Nov. 2006.

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

[62] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP 2012)*, pages 95–109, Washington, DC, USA, May 2012.