A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting

Chia-Che Tsai Bhushan Jain Nafees Ahmed Abdul Donald E. Porter
Stony Brook University
{chitsai,bpjain,nabdul,porter}@cs.stonybrook.edu

Abstract

This paper presents a study of Linux API usage across all applications and libraries in the Ubuntu Linux 15.04 distribution. We propose metrics for reasoning about the importance of various system APIs, including system calls, pseudo-files, and libc functions. Our metrics are designed for evaluating the relative maturity of a prototype system or compatibility layer, and this paper focuses on compatibility with Linux applications. This study uses a combination of static analysis to understand API usage and survey data to weight the relative importance of applications to end users.

This paper yields several insights for developers and researchers, which are useful for assessing the complexity and security of Linux APIs. For example, every Ubuntu installation requires 224 system calls, 208 `ioctl`, `fcntl`, and `prctl` codes and hundreds of pseudo files. For each API type, a significant number of APIs are rarely used, if ever. Moreover, several security-relevant API changes, such as replacing `access` with `faccessat`, have met with slow adoption. Finally, hundreds of libc interfaces are effectively unused, yielding opportunities to improve security and efficiency by restructuring libc.

1. Introduction

Systems engineers and researchers routinely make design choices based on what they believe to be the common and uncommon behaviors of a system. For instance, one recent project optimized the `stat` and `open` system calls at the expense of `rename` and `chmod` [52]. In the case of a general-purpose OS, determining exactly what the common case is can be challenging. Thus, a developer's view of what APIs are important may be skewed heavily towards that developer's preferred workloads.

Similarly, developers struggle to evaluate the impact of a change that affects backward-compatibility, primarily because of a lack of metrics. Deprecating an API is often a lengthy process, wherein users are repeatedly warned and eventually some applications may still be broken. For example, a range of security problems arise from the ill-specified behavior of the `signal` system call [1, 55]. Despite 15 years of warnings to move to the more secure `sigaction` call, `signal` has not been removed from 32-bit x86 Linux, because many legacy applications use `signal`. Eliminating or replacing needless, problematic APIs can be good for security, efficiency, and maintainability of OSes, but in practice this is difficult for OS developers to do without tools to analyze API usage.

Many experimental operating systems add a rough Unix or Linux compatibility layer to increase the number of supported applications [13, 15, 27, 56]. Such systems generally support a fraction of Linux system calls, often just enough to run a few target workloads. One metric for compatibility or completeness of a new feature is the count of supported system APIs [16, 19, 42, 51]. System call counts do not accurately estimate the fraction of applications or users that could plausibly use the system. OS researchers would benefit from the ability to translate a set of supported system calls to the fraction of applications that can be directly supported without recompilation. Similarly, it is useful to know which additional APIs would enable the largest range of additional applications to run on the system. In order to indicate general usefulness, a good compatibility metric should factor in the fraction of users whose choice of applications can be completely supported on a system.

At the root of these problems is a lack of data sets and analysis of how system APIs are used in practice. System APIs are simply not equally important: some APIs are used by popular libraries and, thus, by essentially every application. Other APIs may be used only by applications that are rarely installed. Evaluating compatibility is fundamentally a measurement problem.

This paper bridges the gap between data that is easy for system builders to measure and the metrics they need, contributing a methodology and thorough study of API usage

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

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901341

in x86-64 Ubuntu/Debian Linux. Our study statically analyzes all executable and shared library binaries from all 30,976 packages in the Ubuntu/Debian Linux repository, in order to identify the system API "footprint" of each binary. This paper combines the footprint data with data about how frequently each package is installed, which is measured from the Ubuntu and Debian "popularity contest" survey data [2, 7]. By combining these data sources, the paper contributes metrics which weigh the usage of each system API by estimated usage in real-world installations.

This paper contributes a data set and analysis tool that can answer several practical questions for systems researchers. For instance: in a given prototype, which missing APIs would increase the range of supported applications? Or, if a given system API is optimized, what widely-used applications would likely benefit? We expect that the ability to match evaluation workloads to modified or supported system APIs will be particularly useful. Similarly, this data and toolset can help OS maintainers evaluate the impact of an API change on applications, and can help users evaluate whether a prototype system is suitable for their needs.

The contributions of this work are as follows:

- An approach to measuring platform compatibility, suitable for evaluating the relative completeness of prototype systems. Rather than considering compatibility a binary property ("will something break?"), we use a fractional metric ("how many programs will not break?") which is better suited to measuring the progress of a prototype.
- A comprehensive data set of current API usage in Ubuntu Linux 15.04.
- Analysis and a range of insights into current API usage patterns. For instance, we identify an efficient path to implementing a new Linux compatibility layer, maximizing the additional applications per system call. We also identify that usage of many APIs is similarily distributed: some are widely used, and there is a sharp drop with a very long tail of rarely or never-used APIs. As an example, nearly 40% of libc APIs are used by less than one percent of applications on a typical installation.

2. Some APIs Are More Equal Than Others

We started this study from a research perspective, in search of a better way to evaluate the completeness of system prototypes with a Unix compatibility layer. In general, compatibility is treated as a binary property (e.g., bug-for-bug compatibility), which loses important information when evaluating a prototype that is almost certainly incomplete. Papers often appeal to noisy indicators that the prototype probably covers all important use cases, such as the number of total supported system or library calls, as well as the variety of supported applications.

These metrics are easy to quantify, but problematic. Simply put, not all APIs are equally important: some are indispensable (e.g., `read` and `write`), whereas others are very rarely used (e.g., `preadv` and `delete_module`). A simple count of system calls is easily skewed by system calls that are variations on a theme (e.g., `setuid`, `seteuid`, and `setresuid`). Moreover, some system calls, such as `ioctl`, export widely varying operations—some used by by *all* applications and many that are essentially never used (§3.3). Thus, a system with "partial support" for `ioctl` is just as likely to support all or none of the Linux applications distributed with Ubuntu.

This paper considers system APIs ("APIs") broadly: this includes system calls, as well as any other means by which OS kernel functionality is requested, such as a pseudo-file system (`/proc`). This paper also considers libraries like libc, which are typically responsible for exporting an API, like POSIX, as well as the primary way application developers interact with the OS kernel.

One of the ways to understand the importance of a given interface is to measure its impact on end-users. In other words, if a given interface were not supported, how many users would notice its absence? Or, if a prototype added a given interface, how many more users would be able to use the system? To answer these questions, we must consider both the difference in API usage among applications, and the popularity of applications among end-users. We measure the former by analyzing application binaries, and determine the latter from installation statistics collected by Debian and Ubuntu [2, 7]. An **installation** is a single system installation, and can be a physical machine, a virtual machine, a partition in a multi-boot system, or a chroot environment created by `debootstrap`. Our data is drawn from over 2.9 million installations (2,745,304 Ubuntu and 187,795 Debian).

We introduce two new metrics: one for each API, and one for a whole system. For each API, we measure how disruptive its absence would be to applications and end users—a metric we call **API importance**. For a system, we compute a weighted percentage we call **weighted completeness**. For simplicity, we define a **system** as a set of implemented or translated APIs, and assume an application will work on a target system if the application's API footprint is implemented on the system. These metrics can be applied to all system APIs, or a subset of APIs, such as system calls or standard library functions.

This paper focuses on Ubuntu/Debian Linux, as it is a well-managed Linux distribution with a wide array of supported software, which also collects package installation statistics. The default package installer on Ubuntu/Debian Linux is `APT`. A **package** is the smallest granularity of installation, typically matching a common library or application. A package may include multiple executables, libraries, and configuration files. Packages also track dependencies, such as a package containing Python scripts depending on the Python interpreter. Ubuntu/Debian Linux installation statistics are collected at package granularity and collect

several types of statistics. This study is based on data of how many Ubuntu or Debian installations installed a given target package.

For each binary in a package—either as a standalone executable or shared library—we use static analysis to identify all possible APIs the binary could call, or the **API footprint**. The APIs can be called from the binaries directly, or indirectly through calling functions exported by other shared libraries. A package's API footprint is the union of the API footprints of each of its standalone executables. We weight the API footprint of each package by its installation frequency to approximate the overall importance of each API. Although our initial focus was on evaluating research, our resulting metric and data analysis provide insights for the larger community, such as trends in API usage.

2.1 API Importance: A Metric for Individual APIs

System developers can benefit from an importance metric for APIs, which can in turn guide optimization efforts, deprecation decisions, and porting efforts. Reflecting the fact that users install and use different software packages, we define API importance as the probability that an API will be indispensable to at least one application on a randomly selected installation. We want a metric that decreases as one identifies and removes instances of a deprecated API, and a metric that will remain high for an indispensable API, even if only one ubiquitous application uses the API.

> Definition: **API Importance.**
> For a given API, the probability that an installation includes at least one application requiring the given API.

Intuitively, if an API is used by no packages or installations, the API importance will be *zero*, causing no negative effects if removed. We assume all packages installed in an OS installation are indispensable. As long as an API is used by at least one package, the API is considered *important* for the installation. Appendix A.1 includes a formal definition of API importance.

2.2 Weighted Completeness: A System-Wide Metric

We also measure compatibility at the granularity of an OS, which we call weighted completeness. Weighted completeness is the fraction of applications that are likely to work, weighted by the likelihood that these applications will be installed on a system.

The goal of weighted completeness is to measure the degree to which a new OS prototype or translation layer is compatible with a baseline OS. In this study, the baseline OS is Ubuntu/Debian Linux.

> Definition: **Weighted Completeness.**
> For a target system, the fraction of applications supported, weighted by the popularity of these applications.

The methodology for measuring the weighted completeness of a target system's API subset is summarized as follows:

1. Start with a list of supported APIs of the target system, either identified from the system's source, or as provided by the developers of the system.
2. Based on the API footprints of packages, the framework generates a list of supported and unsupported packages.
3. The framework then considers the dependencies of packages. If a supported package depends on an unsupported package, both packages are marked as unsupported.
4. Finally, the framework weighs the list of supported packages based on package installation statistics. As with API importance, we measure the effected package that is most installed; weighted completeness instead calculates the expected fraction of packages in a typical installation that will work on the target system.

We note that this model of a typical installation is useful in reducing the metric to a single number, but also does not capture the distribution of installations. This limitation is the result of the available package installation statistics, which do not include correlations among installed packages. This limitation requires us to assume that package installations are independent, except when `APT` identifies a dependency. For example, if packages *foo* and *bar* are both reported as being installed once, we cannot tell if they were on the same installation, or if two different installations. If foo and bar both use an obscure system API, we assume that two installations would be affected if the obscure API were removed. If foo depends on bar, we assume the installations overlap. Appendix A.2 formally defines weighted completeness.

2.3 Data Collection via Static Analysis

We use static binary analysis to identify the system call footprint of a binary. This approach has the advantages of not requiring source code or test cases. Dynamic system call logging using a tool like `strace` is simpler, but can miss input-dependent behavior. A limitation of our static analysis is that we must assume the disassembled binary matches the expected instruction stream at runtime. In other words, we assume that the binary isn't deliberately obfuscating itself, such as by jumping into the middle of an instruction (from the perspective of the disassembler). To mitigate this, we spot check that static analysis returns as superset of `strace` results.

We note that, in our experience, things like the system call number or even operation codes are fairly straightforward to identify from a binary. These tend to be fixed scalars in the binary, whereas other arguments, such as the contents of a write buffer, are input at runtime. We assume that binaries can issue system calls directly with inline system call instructions, or can call system calls through a library, such as libc. Our static analysis identifies system call instructions and constructs a whole-program call graph.

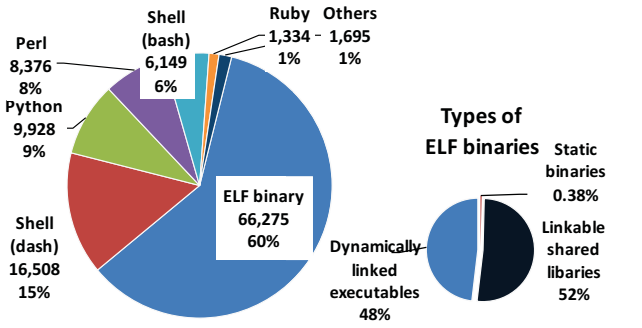

Figure 1. Percentage of ELF inaries and applications written in interpreted languages among all executables in the Ubuntu/Debian Linux repository, categorized by interpreters. ELF binaries include static binaries, shared libraries and dynamically-linked executables. Interpreters are detected by *shebangs* of the files. Higher is more important.

Our study focuses primarily on ELF binaries, which account for the largest fraction of Linux applications (Figure 1). For interpreted languages, such as Python or shell scripts, we assume that the system call footprint of the interpreter and major supporting libraries over-approximates the expected system call footprint of the applications. Libraries that are dynamically loaded, such as application modules or language native interface (e.g.,JNI, Perl XS) are not considered in our study.

2.4 Limitations

Popularity Contest Dataset. The analysis in this paper is limited by the Ubuntu/Debian Linux's package installer, `APT`, and their package installation statistics. Because most packages in Ubuntu/Debian Linux are open-source, our observations on Linux API usage may have a bias toward open-source development patterns. Commercial applications that are purchased and distributed through other means are not included in this survey data, although data from other sources could, in principle, be incorporated into the analysis if additional data were available.

We assume that the package installation statistics provided by Ubuntu/Debian Linux are representative. The popularity contest dataset is reasonably large (336,195 systems), but reporting is opt-in.

The data does not show how often these packages are actually used, only how often they are installed. Finally, this data set does not include sufficient historical data to compare changes to the API usage over time.

Static Analysis. Because our study only analyzes precompiled binaries, some compile-time customizations may be missed. Applications that are already ported using macro like `#ifdef LINUX` will be considered dependent to Linux-specific APIs, even though the application can be recompiled for other systems.

Our static analysis tool only identifies whether an API is potentially used, not how frequently the API is used during the execution. Thus, it is not sufficient to draw inferences about performance.

We assume that, once a given API (e.g., `write`) is supported and works for a reasonable sample of applications, handling missed edge cases should be straightforward engineering that is unlikely to invalidate the experimental results of the project. That said, in cases where an input can yield significantly different behavior, e.g., the path given to `open`, we measure the API importance of these arguments. Verifying bug-for-bug compatibility generally requires techniques largely orthogonal to the ones used in this study, and thus this is beyond the scope of this work.

We do not do inter-procedural data-flow analysis. As a result, we were unable to identify system call numbers for 2,454 call sites (4% of the relevant call sites) across all binaries in the repository. As a result, some system call usage values may be underestimated, and may go up with a more sophisticated static analysis.

Metrics. The proposed metrics are intended to be simple numbers for easy comparison. But this coarseness loses some nuance. For instance, our metrics cannot distinguish between APIs that are critical to a small population, such as those that offer functionality that cannot be provided any other way, versus APIs that are rarely used because the software is unimportant. Similarly, these metrics alone cannot differentiate a new API that is not yet widely adopted from an old API with declining usage.

3. A Study of Modern Linux API Usage

This section presents measurements of API usage, as well as several trends in how APIs are used. Of particular note is that the OS interface required by essentially all applications is substantially larger than the roughly 300 Linux system calls—the required interface also includes several vectored system call operations, such as `ioctl`, and special filesystem interfaces like `/sys` and `/proc`. We also note that a number of system calls and other APIs are so rarely used that they can be deprecated with little disruption or effort.

This section first examines the use of system calls in Linux applications. Section 3.2 analyzes the most efficient path to add system calls to a prototype, outlining a path from the minimal footprint for "hello world", up through the most demanding application (qemu), maximizing the number of supported applications at each step. Section 3.3 analyzes the importance of operations under vectored system calls, such as `ioctl`. Section 3.4 evaluates the API importance of pseudo-files, such as those under `/proc`. Finally, Section 3.5 examines current usage patterns for libc. Throughout the section, we identify several points at which APIs could be gainfully restricted, removed, or refactored, as well as identifying points where unexpected APIs can be

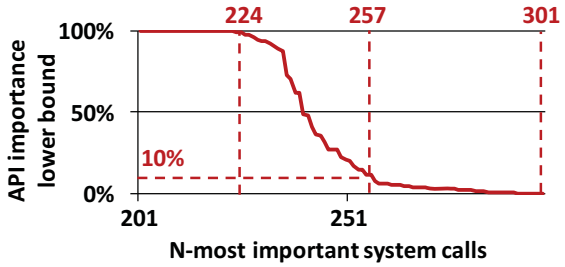

Figure 2. The trend of API importance as N-most important system calls among total 320 system calls of Ubuntu Linux 15.04 with Linux kernel 3.19. Higher is more important; 100% indicates all installations include software that make the system call.

essential to performance or functionality. We highlight key insights and recommendations in boxes.

3.1 Spot the Most Valuable System Calls

We begin by looking at the API importance of each system call, in order to answer the following questions:

- Which system calls are the most important to support when implementing a new system, or have high costs to replace, if desired?
- Which system calls are very rarely used and candidates for deprecation?
- Which system calls are not supported by the OS, but still attempted by applications?

There are 320 system calls defined in x86-64 Linux 3.19 (as listed in `unistd.h`). Figure 2 shows the distribution of system calls by importance. The Figure is ordered by most important (at 100%) to least important (around 0%)—similar to inverted CDF. The figure highlights several points of interest on this line.

Over two-thirds (*224 of 320*) of system calls on Linux are indispensable: required by at least one application on every installation. Among the rest, 33 system calls are important on more than ten percent of the installations. 44 system calls have very low API importance: less than ten percent of the installations include at least one application that uses these system calls.

Our study also shows the contributors to an API's importance. For instance, Table 1 lists system calls that are only called by one or two particular libraries (e.g., libc). These system calls are wrapped by library APIs, so applications depend on them only because the libraries do. To eliminate the usage of these system calls, developers only have to pay minimum efforts to re-implement the wrappers in libraries.

Among the 44 system calls with a API importance above zero but less than ten percent, some are cases where a more popular alternative is available. For instance, Linux supports both POSIX and System V message queues. The five APIs for POSIX message queues have a lower API importance than System V message queues. We believe this

System Calls	Imp.	Packages
`clock_settime`, `iopl`, `ioperm`, `signalfd4`	100%	libc
`mbind`	36.0%	libnuma, libopenblasp
`addkey`	27.2%	libkeyutils
`keyctl`	27.2%	pam_keyutil, libkeyutils
`requestkey`	14.4%	libkeyutils
`preadv`, `pwritev`	11.7%	libc

Table 1. System calls which are only directly used by particular libraries, and their API importance ("Imp."). Only system calls with API importance larger than ten percent are shown. These system calls are wrapped by library APIs, thus they are easy to deprecate by modifying the libraries.

System Calls	Imp.	Packages
`seccomp`, `sched_setattr`, `sched_getattr`	1%	coop-computing-tools
`kexec_load`	1%	kexec-tools
`clock_adjtime`	4%	systemd
`renameat2`	4%	systemd, coop-computing-tools
`mq_timedsend`, `mq_getsetattr`	1%	qemu-user
`io_getevent`	1%	ioping, zfs-fuse
`getcpu`	4%	valgrind, rt-tests

Table 2. System calls with usage dominated by particular package(s), and their API importance ("Imp."). This table excludes system calls that are officially retired.

is attributable to System V message queues being more portable to other UNIX systems. Similarly, we observed that `epoll_wait` (100%) has a higher API importance than `epoll_pwait` (3%), even though `epoll_pwait` is commonly considered more robust for the same purpose—waiting on file descriptor events. Table 2 lists system calls used by only one or two packages—generally special-purpose utilities, such as `kexec_load`, which is used by `kexec-tools`).

In some cases, system calls are effectively offloaded to a file in `/proc` or `/sys`. For instance, some of the information that was formerly available via `query_module` can be obtained from `/proc/modules`, `/proc/kallsyms` and the files under the directory `/sys/module`. Similarly, the information that can be obtained from the `sysfs` system call is now available in `/proc/filesystems`.

We also found five system calls `uselib`, `nfsservctl`, `afs_syscall`, `vserver` and `security` system calls that are officially retired, but still have a low, but non-zero, API importance. For instance `nfsservctl` is removed from Linux kernel 3.1 but still has API importance of seven percent, because it is tried by NFS utilities such as `exportfs`. These utilities still attempt the old calls for backward-compatibility with older kernels.

Unused System Calls	Reason for Disuse
set_thread_area, tuxcall, create_module, and 6 more.	Officially retired.
sysfs	replaced by /proc/filesystems.
rt_tgsigqueueinfo, get_robust_list	Unused by applications.
remap_file_pages	No non-sequential ordered mapping; repeated calls to mmap preferred.
mq_notify	Unused: Asynchronous message delivery.
lookup_dcookie	Unused: for profiling.
restart_syscall	Transparent to applications.
move_pages	Unused: for NUMA usage.

Table 3. Unused system calls and explanation for disuse.

Among 43 least-used system calls, some are replaceable by alternatives with higher API importance; 5 are officially retired but still tried by few applications. System developers could use this data to identify relevant applications, accelerating replacement of these system calls.

In total, 18 of 320 system calls in Linux 3.19 are not used by any application in the Ubuntu/Debian Linux repository. We list these system calls in Table 3. In addition to the issues discussed above, Ten of these system calls do not have an entry point, but are still defined in the Linux headers. Five of the unused system calls such as rt_tgsigqueueinfo, get_robust_list, remap_file_pages, mq_notify, lookup_dcookie provide an interface that is not used by the applications. These system calls can be potential candidates for deprecation. However, even though restart_syscall is not used by any application, it is internally used by the kernel.

In addition to ten already retired system calls, we found seven other candidate system calls for deprecation or in need of more exposure to developers.

3.2 From "Hello World" to Qemu

Figure 3 shows the optimal path of adding system calls to a prototype system, using a simple, greedy strategy of implementing the N-most important APIs, which in turns maximizes weighted completeness. In other words, the leftmost points on the graph are the most important APIs, but the y coordinate only increases once enough system calls are supported that a simple program, such as "hello world" can execute. Similar to a CDF, this line continues up to 100% of

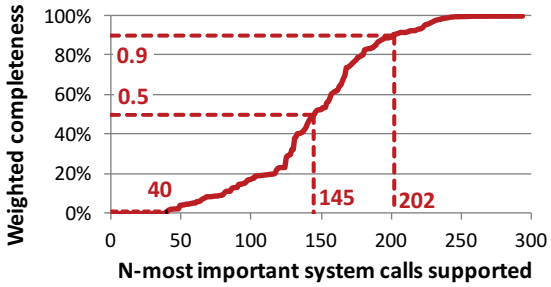

Figure 3. Accumulated weighted completeness when N top-ranked system calls are implemented in the OS. Higher is more compliant.

Ubuntu applications. The graph highlights several points of interest in this curve.

Essentially, one cannot run even the most simple programs without at least 40 system calls. After this, the number of additional applications one can support by adding another system call increases steadily until an inflection point at 125 system calls, or supporting extended attributes on files, where weighted completeness jumps to 25%. To support roughly half of Ubuntu/Debian Linux applications, one must have 145 system calls, and the curve plateaus around 202 system calls. On the most extreme end, qemu's MIPS emulator (on an x86-64 host) requires 270 system calls [18]. A weighted completeness of 100% implies that all Linux applications ever used are supported by the prototype.

Table 4 breaks down the recommended development phases by rough categories of required system calls. We do not provide a complete ordered list here in the interest of brevity, but this list is available as part of our dataset, at http://oscar.cs.stonybrook.edu/api-compat-study.

A goal of weighted completeness is to help guide the process of developing new system prototypes. Section 3.1 showed that 224 out of 320 system calls on Ubuntu/Debian Linux have 100% API importance. In other words, if one of these 224 calls is missing, at least one application on a typical system will not work. Weighted completeness, however, is more forgiving, as it tries to capture the fraction of a typical installation that could work. Only 40 system calls are needed to have weighted completeness more than 1%.

It takes the most effort to support first and last 10% of any installation (0–10% and 90–100% weighted completeness). The gain in functionality is precipitous when adding the 81st–202nd system calls.

For simplicity, Table 4 only includes system calls, but one can construct a similar path including other APIs, such as vectored system calls, pseudo-files and library APIs. For example, developers need not implement every operation

Stage	Sample System Calls	#	Weighted Completeness
I	`mmap`, `vfork`, `exit`, `read`, `gettid`, `fcntl`, `getcwd` `sched_yield`, `kill`, `dup2`	40	1.12 %
II	`mremap`, `ioctl`, `access`, `socket`, `poll`, `recvmsg`, `dup`, `unlink`, `wait4`, `select`, `chdir`, `pipe`	+41 (81)	10.68 %
III	`sigaltstack`, `shutdown`, `symlink`, `alarm`, `listen`, `pread64`, `getxattr`, `shmget`, `epoll_wait`, `chroot`, `sync`, `getrusage`	+64 (145)	50.09 %
IV	`flock`, `semget`, `ppoll`, `mount`, `brk`, `pause`, `clock_gettime`, `getpgid`, `settimeofday`, `capset`, `reboot`, `unshare`, `tkill`	+57 (202)	90.61 %
V	All remaining	+70 (272)	100 %

Table 4. Five stages of implementing system calls based on the API importance ranking. For each stage, a set of system calls is listed, with the work needed to accomplish (# of system calls) and the weighted completeness that can be reached.

of `ioctl`, `fcntl` and `prctl` during the early stage of developing a system prototype.

3.3 Vectored System Call Opcodes

Some system calls, such as `ioctl`, `fcntl`, and `prctl`, essentially export a secondary system call table, using the first argument as an operation code. These *vectored* system calls significantly expand the system API, dramatically increasing the effort to realize full API compatibility. It is also difficult to enforce robust security policies on these interfaces, as the arguments to each operation are highly variable.

The main expansion is from `ioctl`. Linux defines 635 operation codes, and Linux kernel modules and drivers can define additional operations. In the case of `ioctl`, we observe that there are 52 operations with the 100% API importance (Figure 4), each of which are as important as the 226 most important system calls. Of these 52 operations, 47 are frequently used operations for TTY console (e.g., `TCGETS`) or generic operations on IO devices (e.g., `FIONREAD`).

On the narrow end, `fcntl` and `prctl` have 18 and 44 operations, respectively, in Linux kernel 3.19. Unlike `ioctl`, `fcntl` and `prctl` are not extensible by modules or drivers, and their operations tend to have higher API importance (Figure 5). For `fcntl`, eleven out of eighteen `fcntl` operations in Linux 3.19 have API importance at around 100%. For `prctl`, only nine out of 44 operations

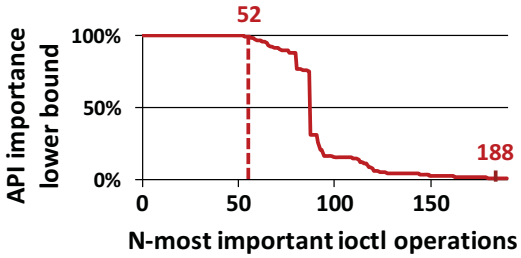

Figure 4. 188 ioctl operations with highest API importance. Roughly 447 infrequent operations on the tail are omitted. Higher is more important; 100% indicates all installations include software that request the operations.

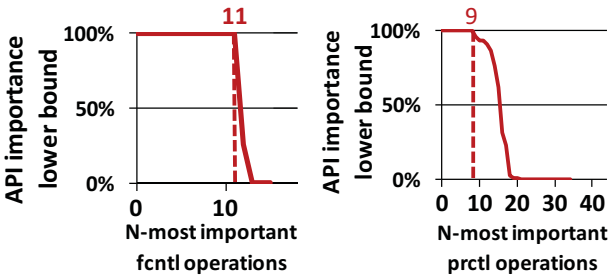

Figure 5. Ranking of API importance among `fcntl` and `prctl` codes. Higher is more important; 100% indicates all installations include software that request the operations.

have API importance around 100%, and only eighteen has API importance larger than 20%.

Thus, developers of a new system prototype should support these 47 most important `ioctl` operations, about half of the `fcntl` opcodes, and only 9–20 `prcntl` operations.

> In building a prototype system, a relatively small set of operations in vectored system calls are essential.

Compared to system calls, `ioctl` has a much longer tail of infrequently used operations. Out of 635 `ioctl` operation codes defined by modules or drivers hosted in Linux kernel 3.19, only 188 have API importance more than one percent, and for only 280 we can find usage of the operations in at least one application binary. Those unused operations are good targets for deprecation, in the interest of reducing the system attack surface.

> `ioctl` system call has a very long tail of unused operations, which may create system security risks.

3.4 Pseudo-Files and Devices

In addition to the main system call table, Linux exports many additional APIs through pseudo-file systems, such as `/proc`, `/dev`, and `/sys`. These are called pseudo-file systems because they are not backed by disk, but rather export

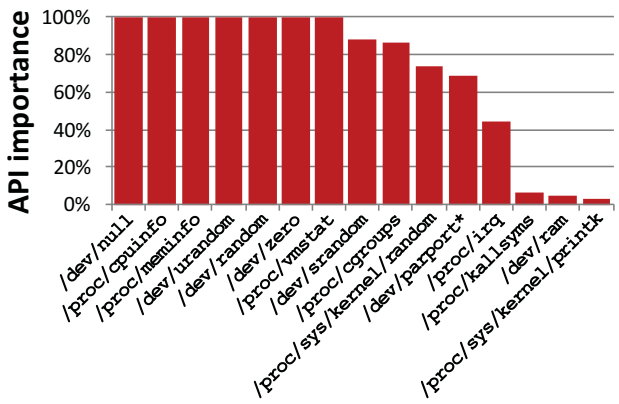

Figure 6. API importance distribution over files under `/dev` and `/proc`. Higher is more important; 100% indicates all installations include software that accesses the file.

the contents of kernel data structures to an application or administrator as if they were stored in a file. These pseudo-file systems are a convenient location to export tuning parameters, statistics, and other subsystem-specific or device-specific APIs. Although many of these pseudo-files are used on the command line or in scripts by an administrator, a few are routinely used by applications. In order to fully understand usage patterns of the Linux kernel, pseudo-files must also be considered.

We apply static analysis to find cases where the binary is hard-coded to use a pseudo-file. Our analysis cannot capture cases where a path to one of these file systems is passed as an input to the application, such as `dd if=/dev/zero`. However, when a pseudo-file is widely-used as a replacement for a system call, these paths tend to be hard-coded in the binary as a string or string pattern. A common pattern we observed was `sprintf(``/proc/%d/cmdline'', pid)`; our analysis captured these patterns as well. We also do not differentiate types of access in this study, such as separating read versus write of a pseudo-file; rather we only consider whether the file is accessed or not. Thus, our analysis is limited to strings stored in the binary, but we believe this captures an important usage pattern.

Figure 6 shows the API importance of common pseudo-files under `/dev` and `/proc`. These files are ordered from highest API importance; the long tail of files used rarely or directly by administrators is omitted.

Some files are essential, such as `/dev/null` and `/proc/cpuinfo`. These files are widely used in binaries and scripts. Among 12,039 binaries that use a hard-coded path, 3,324 access `/dev/null` and 439 access `/proc/cpuinfo`. However, it is plausible to provide the same functionality in simpler ways. For instance, `/proc/cpuinfo` provides a formatted wrapper for the `cpuinfo` instruction, which one could export directly to userspace using virtualization hardware, similar to Dune [17]. Similarly, `/dev/zero` or `/dev/null` are convenient for use on the command line, but it is surprising that a significant number of applications issue `read` or `write` system calls, rather than simply zeroing a buffer or skipping the `write` (e.g., `grub-install`). Thus, in implementing a Linux compatibility layer, a small number of pseudo-files are essential, and perhaps others could be eliminated with modest application changes.

APIs as pseudo-files or pseudo-devices also have a large subset of infrequently used or unused APIs. Many of them are designed to support one specific application or user. For example, `/dev/kvm` is only intended for `qemu` to communicate with the kernel portion of the KVM hypervisor. Similarly `/proc/kallsyms` is used primarily to export debugging information to kernel developers.

Because so many files in `/proc` are accessed from the command line or by only a single application, it is hard to conclude that any should be deprecated. Nonetheless, these files represent large and complex APIs that create an important attack surface to defend. As noted in other studies, the permission on `/proc` tend to be set liberally enough to leak a significant amount of information [34]. For files used by a single application, an abstraction like a fine-grained capability [48] might better capture this need. For files used primarily by the administrator, carefully setting directory permissions should be sufficient.

> A few pseudo-files are essential and must be implemented by any Linux-emulator. Most serve a specific purpose, and might benefit from stricter enforcement of the principle of least privilege.

In the case of the `/dev` file system, the most commonly used files are pseudo-devices, such as accessing the virtual terminal (`/dev/tty`, `/dev/console`, and `/dev/pts`), or other functionality such as the random number generator (`/dev/urandom`). Even among pseudo-devices, features such as accessing one's standard in and out, or a process's TTY via the `/dev/` interface are not heavily used.

Intuitively, one would not expect many device paths to be hard-coded, and most direct interactions with a device would be done using administrative tools. For instance, we see that some applications do hard-code paths like `/dev/hda` (commonly used for an IDE hard drive), yet an increasing number of systems have a root hard drive using SATA, which would consequently be named `/dev/sda`. Thus, although applications may use paths like `/dev/hda` as a default device path, modern systems are sufficiently varied that these generally need to be searched at runtime.

3.5 Reorganizing System Library APIs

In addition to studying kernel interfaces, we also analyze the API importance of APIs defined in core system libraries, such as libc. Most programmers don't directly use the APIs exported by the kernel, but instead program to more user-friendly APIs in libc and other libraries. For instance, GNU

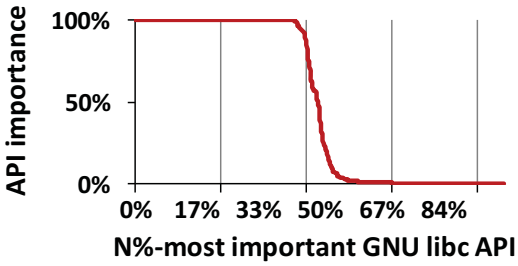

Figure 7. API importance distribution over the set of GNU Library C. Higher is more important; 100% indicates all installations include software that use the libc API.

libc [5] exports APIs for using locks and condition variables, which internally use the subtle `futex` system call [28].

Figure 7 shows the API importance of the global function symbols exported by libc—1,274 in total. Among these APIs, 42.8% have a API importance of 100%, 50.6% have a API importance of less than 50%, and 39.7% have a API importance of less than one percent, including some that are not used at all. In other words, about 40% of the APIs inside libc are either not used or only used by few applications.

This result implies that most processes are loading a significant amount of unnecessary code into their address space. By splitting libc into several sub-libraries, based on API importance and common linking patterns, systems could realize a non-trivial space savings.

There are several reasons to avoid loading extra code into an application. First, there are code reuse attacks, such as return-oriented programming (ROP) [47], that rely on the ability to find particular code snippets, called gadgets. Littering a process with extra gadgets offers needless assistance to an attacker. Similarly, when important and unimportant APIs are on the same page, memory is wasted. Finally, the space overhead of large, unused jump tables is significant. In GNU libc 2.21, `libc-2.21.so` essentially has 1274 relocation entries, occupying 30,576 bytes of virtual memory. By sorting the relocation table according to API usage, most libc instances could load only first few pages of relocation tables, and leave the remaining relocation entries for lazy loading.

We analyzed the space savings of a GNU libc 2.21 which removed any APIs with API importance lower than 90%. In total, libc would retain 889 APIs and the size would be reduced to 63% of its original size. The probability an application would need a missing function and load it from another library is less than 9.3%(equivalent to 90.7% weighted completeness for the stripped libc). Further decomposition is also possible, such as placing APIs that are commonly accessed by the same application into the same sub-library.

> Decomposing or re-ordering library API can lower memory costs in typical application processes.

System Calls	Libraries
`access`, `arch_prctl`, `mprotect`	ld.so
`clone`, `execve`, `getuid`, `gettid`, `kill`, `getrlimit`, `setresuid`	libc
`close`, `exit`, `exit_group`, `getcwd`, `getdents`, `getpid`, `lseek`, `lstat`, `mmap`, `munmap`, `madvise`, `mprotect`, `mremap`, `newfsstat`, `read`	libc, ld.so
`rt_sigreturn`, `set_robust_list`, `set_tid_address`	libpthread
`rt_sigprocmask`	librt
`futex`	libc, ld.so, libpthread

Table 5. Ubiquitous system call usage caused by initialization or finalization of libc family.

Effects of standard libraries on API importance. Libc and the dynamic linker (`ld.so`) also contribute to the system call footprint of every dynamically-linked executable. This has a marked effect on the API importance of some system calls. The APIs used to initialize a program are listed in Table 5. In several cases, such as `set_tid_address`, however, libc or libpthread may be the only binaries using these interfaces directly, indicating that changes to some important system interfaces would only require changes in one or two low-level libraries.

> GNU Library C and the dynamic linker can have a first-order effect on the API importance of some system calls.

4. Linux Systems and Emulation Layers

This section uses weighted completeness to evaluate systems or emulation layers with partial Linux compatibility. We also evaluate several libc variants for their degree of completeness against the APIs exported by GNU libc 2.21.

4.1 Weighted Completeness of Linux Systems

To evaluate the weighted completeness of Linux systems or emulation layers, the prerequisite is to identify the supported APIs of the target systems. Due to the complexity of Linux APIs and system implementation, it is hard to automate the process of identification. However, OS developers are mostly able to maintain such a list based on the internal knowledge.

We evaluate the weighted completeness of four Linux-compatible systems or emulation layers: User-Mode-Linux [25], L4Linux [32], FreeBSD emulation layer [26], and Graphene library OS [51]. For each system, we explore techniques to help identifying the supported system calls, based on how the system is built. For example, User-Mode-Linux and L4Linux are built by modifying the Linux source code, or adding a new architecture to Linux. These systems will define architecture-specific system call tables, and reimplement `sys_*` functions in the Linux source that are origi-

Systems	#	Suggested APIs to add	W.Comp.
UML 3.19	284	`name_to_handle_at`, `iopl`, `ioperm`, `perf_event_open`	93.1%
L4Linux 4.3	286	`quotactl`, `migrate_pages`, `kexec_load`	99.3%
FreeBSD-emu 10.2	225	`inotify*`, `splice`, `umount2`, `timerfd*`	62.3%
Graphene	143	`sched_setscheduler`, `sched_setparam`	0.42%
Graphene¶	145	`statfs`, `utimes`, `getxattr`, `fallocate`, `eventfd2`	21.1%

Table 6. Weighted completeness of several Linux systems or emulation layers. For each system, we manually identify the number of supported system calls ("#"), and calculate the weighted completeness ("W.Comp.") . Based on API importance, we suggest the most important APIs to add. (*: system call family. ¶: Graphene after adding two more system calls.)

nally aliases to `sys_ni_syscall` (a function that returns `-ENOSYS`). Other systems, like FreeBSD and Graphene, are built from scratch, and often maintain their own system call table structures, where unsupported systems calls are redirected to dummy callbacks.

Table 6 shows weighted completeness, considering only system calls. The results also identify the most important system calls that the developers should consider adding. User-Mode-Linux and L4Linux both have a weighted completeness over 90%, with more than 280 system calls implemented. FreeBSD's weighted completeness is 62.3% because it is missing some less important system calls such as `inotify_init` and `timerfd_create`. Graphene's weighted completeness is only 0.42%. We observe that the primary culprit is scheduling control; by adding two scheduling system calls, Graphene's weighted completeness would be 21.1%.

4.2 Weighted Completeness of Libc

This study also uses weighted completeness to evaluate the compatibility of several libc variants — eglibc [4], uClibc [8], musl [6] and dietlibc [3] — against GNU libc, listed in Table 7. We observe that, if simply matching exported API symbols, only eglibc is directly compatible to GNU libc. Both uClibc and musl have a low weighted completeness, because GNU libc's headers replace a number of APIs with safer variants at compile time, using macros. For example, GNU libc replaces `printf` with `__printf_chk`, which performs an additional check for stack overflow. After normalizing for this compile-time API replacement, both uClibc and musl are at over 40% weighted completeness. In contrast, dietlibc is still not compatible with most binaries

Libc variants	#	Unsupported (samples)	W.Comp.	W.Comp. (normalized)
eglibc 2.19	2198	None	100%	100%
uClibc 0.9.33	1867	`__uflow`, `__overflow`	1.1%	41.9%
musl 1.1.14	1890	`secure_getenv`, `random_r`	1.1%	43.2%
dietlibc 0.33	962	`memalign`, `stpcpy`, `__cxa_finalize`	0%	0%

Table 7. Weighted completeness of libc variants. For each variant, we calculate weighted completeness based on symbols directly retrieved from the binaries, and the symbols after reversing variant-specific replacement (e.g.,`printf` becomes `__printf_chk`).

linked against GNU libc — if no other approach is taken to improve its compatibility. The reason of low weighted completeness is that dietlibc does not implement many ubiquitously used GNU libc APIs such as `memalign` (used by 8887 packages) and `__cxa_finalize` (used by 7443 packages).

5. Unweighted API Importance

API importance is weighted by the number of installations of applications that use the API. As a result, one ubiquitous application can cause the API importance of an API it uses to be close to 100%. This section observes trends for APIs with multiple variants, using an additional unweighted API importance metric. We remove the weighting by installation frequency to focus on trends in developer behavior.

Once an API has been identified as having a security risk, and a more secure variant is developed, one might wish to know how many vulnerable packages are still in the wild, and how many have moved to less exploit-prone APIs. Similarly, one might want to know how many applications have not migrated away from a deprecated API, even if these applications are not widely used.

> Definition: **Unweighted API importance.**
> For a given API, the probability an application (package) uses that API, irrespective of probability of installation.

We begin by looking at the unweighted API importance of each system call. Figure 8 shows the distribution of system calls across packages. Recall that using API importance, over two-thirds of system calls on Linux are required by at least one application on every installation. Using unweighted API importance, Figure 8 suggests that only 40 system calls are used by all packages, and 130 system calls are used by at least 10% of packages. Over half of Linux system calls are used by less than 10% of packages.

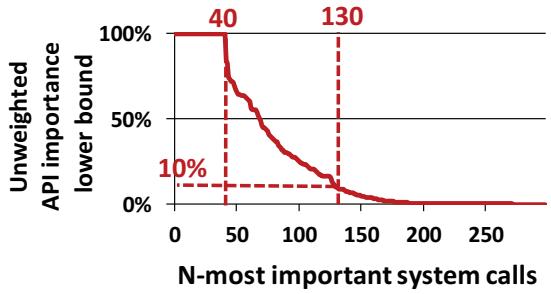

Figure 8. The trend of unweighted API importance in N-most important system calls among total 320 system calls of Ubuntu Linux 15.04 with Linux kernel 3.19.

One family of APIs prone to security problems are the `set*id` API family. Many of the `set*id` APIs have subtle semantic differences across different Unix variants. Chen et al. [22] conclude that `setresuid` has the clearest semantics across all Unix flavors. Table 8 shows the unweighted API importance of `set*id` and `get*id` system calls. Most packages have adopted the more clear and secure interface. System calls `setuid`, `setreuid`, and `setresuid` have unweighted API importance of 15.67%, 1.88% and 99.68% respectively. However, for `get*id` system calls, the unweighted API importance suggests that the `getres*id` system calls are only used by roughly 36% of packages.

Directory operations have a long history of exploitable race conditions [20, 21, 54], or time-of-check-to-time-of-use (TOCTTOU) vulnerabilities. In a privileged application, one system call (e.g., `access`) checks the user's permission, and a second call operates on the file. There are countermeasures that effectively walk the directory hierarchy in user space [50]. This approach replaces calls like `access` with `faccessat`, and similar variants. Table 8 shows the current unweighted API importance of `*at` system call variants and their older counterparts. We observed that the unweighted API importance of the race-prone `access` is still high (74.24%), whereas `faccessat` is only 0.63%. This suggests about 75% of the packages use the more vulnerable `access` system call instead of the more secure one.

In addition to security-related hints, unweighted API importance indicates whether obsolete APIs have been replaced by newer variants. For instance, `wait4` system call is considered obsolete [9], and the alternative `waitid` is preferred, as it more precisely specifies which child state changes to wait for. However, unweighted API importance of `wait4` and `waitid` is 60.56% and 0.24%, respectively. This indicates that 60% of the packages are still using the older `wait4` system call. Table 9 shows similar trend for some other system calls. Our dataset provides more opportunity for system developers to actively communicate with application developers, in order to speed up the process of retiring problematic APIs.

Insecure API	U.API Imp.	Secure API	U.API Imp.
Unclear vs. Well-defined ID Management Semantics			
`setuid`	15.67%	`setresuid`	99.68%
`setreuid`	1.88%		
`setgid`	12.07%	`setresgid`	99.68%
`setregid`	1.24%		
`getuid`	99.81%	`getresuid`	36.19%
`geteuid`	55.15%		
`getgid`	99.81%	`getresgid`	36.14%
`getegid`	48.87%		
Nonatomic vs. Atomic Directory operations			
`access`	74.24%	`faccessat`	0.63%
`mkdir`	52.07%	`mkdirat`	0.34%
`rename`	43.18%	`renameat`	0.30%
`readlink`	46.38%	`readlinkat`	0.50%
`chown`	24.59%	`fchownat`	0.23%
`chmod`	39.80%	`fchmodat`	0.13%

Table 8. The unweighted API importance ("U. API Imp.") of secure and insecure API variations.

Old API	U. API Imp.	New API	U. API Imp.
`getdents`	99.80%	`getdents64`	0.08%
`utime`	8.57%	`utimes`	17.90%
`fork`	0.07%	`clone`	99.86%
`vfork`	99.68%		
`tkill`	0.51%	`tgkill`	99.80%
`wait4`	60.56%	`waitid`	0.24%

Table 9. The unweighted API importance ("U. API Imp") of old (generally deprecated) and new (preferred) API variations. Higher is more important.

> Adoption of newer, preferred API variants is often slow, and kernel developers could benefit from an easy mechanism to identify relevant developers.

Some APIs are specific to a particular OS, such as Linux, and often have more portable variants. Table 10 shows the comparison between Linux-specific APIs and their generic variants. The results show most developers prefer portable or generic APIs more than Linux-specific APIs. Except `pipe2`, most API variants that are Linux-specific have unweighted API importance lower than 10 percent.

Finally, we consider system calls with multiple variants where one version has increased functionality. Table 11 shows the difference between these system calls. Interestingly, more developers chose the less powerful variants, such as using `select` over `pselect6`, or `dup2` over `dup3`. This indicates that more often than not, developers choose simplicity unless a task demands the functionality of a more powerful API variant.

Linux Specific	U. API Imp.	Portable / Generic	U. API Imp.
preadv	0.15%	readv	62.23%
pwritev	0.16%	writev	99.80%
accept4	0.93%	accept	29.35%
ppoll	3.90%	poll	71.07%
recvmmsg	0.11%	recvmsg	68.82%
sendmmsg	5.17%	sendmsg	42.49%
pipe2	40.33%	pipe	50.33%

Table 10. The unweighted API importance (U. API Imp.)of more API variants, and comparison between Linux-specific versions and more portable or generic versions. Higher is more important.

System Call	U. API Imp.	System Call	U. API Imp.
read	99.88%	pread64	27.23%
dup3	8.72%	dup2	99.75%
		dup	66.64%
recvmsg	68.82%	recvfrom	53.80%
sendmsg	42.49%	sendto	71.71%
select	61.53%	pselect6	4.13%
chdir	44.61%	fchdir	2.20%

Table 11. The unweighted API importance among similar API variants. Higher is more important.

Developers prefer the most portable or the simplest API option among variations of same system call.

6. Implications for System Developers

The statistics in Section 3 can inform decisions of application developers, library developers, and kernel developers. Similarly, the ability to easily generate a comprehensive data set of API footprints has several practical uses.

One practical benefit of this study is the ability to automatically identify a system call profile of every application distributed with Ubuntu/Debian Linux. In fact, we observed that the total 31,433 applications have 11,680 different system call footprint and 9,133 out of these applications have a unique system call footprint. We note that these numbers may vary with dynamic analyses, but the fact that one third of all Debian/Ubuntu applications have a unique system call footprint is interesting.

System call footprints have been explored previously for identifying malware or software compromises [36]. Linux has recently added seccomp, a Berkeley Packet Filter-based system call filtering framework [46]; generation of seccomp policies can be easily automated using our framework, reducing the system's attack surface in the event of an application compromise.

These tools can also help OS developers evaluate when it is safe to remove a deprecated interface, or when interfaces appear to be irrelevant to most users (e.g., remap_file_pages). In the case of an irrelevant interface, this may either indicate something is a candidate for deprecation (e.g., lookup_dcookie), or that a useful or important feature (e.g., faccessat) is not getting sufficient traction. Linux developers currently wait as long as six years to retire an interface, allowing ample time for application and library developers to change. Our dataset and methodology can allow more proactive outreach and more rapid system evolution.

The libc function call popularity can similarly help library developers to remove function calls that are not used (222 functions). Moreover, the function call importance distribution can also help reduce the library's memory footprint by organizing the in-memory layout by importance.

7. Implementation Details

This section provides additional implementation details of our analysis framework.

Our analysis is based on disassembling binaries inside each application package, using the standard objdump tool. This approach eliminates the need for source or recompilation, and can handle closed-source binaries. We implement a simple call-graph analysis to detect system calls reachable from the binary entry point (e_entry in ELF headers). We search all binaries, including libraries, for system call instructions (int $0x80, syscall or sysenter) or calling the syscall API of libc. We find that the majority of binaries — either shared libraries or executables — do not directly choose system calls, but rather use the GNU C library APIs. Among 66,275 studied binaries, only 7,259 executables and 2,752 shared libraries issue system calls.

Our call-graph analysis allows us to only select system calls that are actually used by the application, not all the system calls that appear in libc. Our analysis takes the following steps:

- For a target executable or a library, generate a call graph of internal function usage.
- For each library function that the executable relies on, identify the code in the library that is reachable from each entry point called by the executable.
- For each library function that calls another library call, recursively trace the call graph and aggregate the results.

Precisely determining all possible call-graphs from static analysis is challenging. Unlike other tools built on call-graphs, such as control flow integrity (CFI), our framework can tolerate the error caused by over-approximating the analysis results. For instance, programs sometimes make function call based on a function pointer passed as an argument by the caller of the function. Because the calling target is dynamic, it is difficult to determine at the call site. Rather, we track sites where the function pointers are assigned to a register, such as using the lea instruction with an address relative to the current program counter. This is an over-

Evaluation Criteria	Size
Source Lines of Code (Python)	3,105
Source Lines of Code (SQL)	2,423
Total Rows in Database	428,634,030

Table 12. Implementation of the analysis framework.

approximation because, rather than trace the data flow, we assuming that a function pointer assigned to a local variable will be called. This analysis could be more precise if it included a data flow component.

We also hard-code for a few common and problematic patterns. For instance, we generally assume that the registers that pass a system call number to a system call, or an opcode to a vectored system call, are not the result of arithmetic in the same function. We spot checked this assumption, but did not do the data flow analysis to detect this case.

Finally, the last mile of the analysis is to recursively aggregate footprint data. We insert all raw data into a `Postgresql` database, and use recursive SQL queries to generate the results. To scan through all 30,976 packages in the repository, collect the data, and generate the results takes roughly three days.

Our implementation is summarized in Table 12. We wrote 3,105 lines of code in Python and 2,423 lines of code in SQL (Postgresql). The database contains 48 tables with over 428 Million entries.

8. Related Work

Concurrent with our work, Atlidakis et al. [14] conducted a similar study of POSIX. A particular focus of the POSIX study is measuring fragmentation across different POSIX-compliant OSes (Android, OS X, and Ubuntu), as well as identifying points where higher-level frameworks are driving this fragmentation, such as the lack of a ubiquitous abstraction for graphics. Both studies identify long tails of unused or lightly-used functionality in OS APIs. The POSIX study factors in dynamic tracing, which can yield performance insights; our study uses installation metrics, which can yield insights about the impact of incompatibilities end-users. Our paper contributes complimentary insights, such as a metric and incremental path for completeness of an emulation layer, as well as analysis of the importance of less commonly-analyzed APIs, such as pseudo-files under `/proc`.

A number of previous studies have investigated how other portions of the Operating System interact, often at large scale. Kadav and Swift [35] studied the effective API the Linux kernel exports to device drivers, as well as device driver interaction with Linux—complementary to our study of how applications interact with the kernel or core libraries. Palix et al. study faults in all subsystems of the Linux kernel and identify the most fault-prone subsystems [38]. They find architecture-specific subsystems have highest fault rate, followed by file systems. Harter et al. [31] studied the interaction of a set of Mac OS X applications with the file system APIs—identifying a number of surprising I/O patterns. Our approach is complementary to these studies, with a focus on overall API usage across an entire Linux distribution.

A number of previous studies have drawn inferences about user and developer behavior using Debian and Ubuntu package metadata and popularity contest statistics. Debian packages have been analyzed to study the evolution of the software itself [29, 37, 45], to measure the popularity of application programming languages [10], to analyze dependencies between the packages [23], to identify trends in package sizes [12], the number of developers involved in developing and maintaining a package [44], and estimating the cost of development [11]. Jain et al. used popularity contest survey data to prioritize the implementation effort for new system security policies [33]. This study is unique in using this information to infer the relative importance of of system APIs to end users, based on frequency of application installation.

A number of previous projects develop techniques or tools to identify software incompatibilities, with the goal of avoiding subtle errors during integration of software components. The Linux Standard Base (LSB) [24] predicts whether an application can run on a given distribution based on the symbols imported by the application from system libraries. Other researchers have studied application compatibility across different versions of same library, creating rules for library developers to maintain the compatibility across versions [40]. Previous projects have also developed tools to verify backward compatibility of libraries, based on checking for any changes in library variable type definitions and function signatures [41]. Another variation of compatibility looks at integrating independently-developed components of a larger software project; solutions examine various attributes of the components' source code, such as recursive functions and strong coupling of different classes [49]. In these studies, compatibility is a binary property, reflecting a focus on correctness. Moreover, these studies are focused on the interface between the application and the libraries or distribution ecosystem. In contrast, this paper proposes a metric for relative completeness of a prototype system.

Identifying the system call footprint of an application is useful for a number of reasons; our work contributes data from studying trends in system API usage in a large set of application software. The system call footprint of an application can be extracted by static or dynamic analysis. The trade-off is that dynamic analysis is easier to implement quickly, but the results are input-dependent. Binary static analysis, as this paper uses, can be thwarted by obfuscated binaries, which can confuse the disassembler [57]. Static binary analysis has been used to automatically generate application-specific sandboxing policies [36]. Dynamic analysis has been used to compare system call sequences

of two applications as an indicator of potential intellectual property theft [53], to identify opportunities to batch system calls [43], to model power consumption on mobile devices [39], and to repackage applications to run on different systems [30]. These projects answer very different questions than ours, but could, in principle, benefit from the resulting data set.

9. Conclusion

Based on this study, we can draw several conclusions about the nature of Linux APIs. First, for any OS installation in our data set, the required API size is several times larger than the 320 system calls in Linux, once one considers `ioctl` opcodes and files under `/proc`. A solid two-thirds of system calls are indispensable. We show that a substantial range of system calls and other APIs are rarely or even never used. And the paper plots a rough guide for adding system calls to a Linux emulation layer or research prototype.

We expect that this data set will be of use to researchers and developers for further analysis, Our methodology and tools can be easily applied to future releases and other distributions. Our data set, tools, and other information are available at `http://oscar.cs.stonybrook.edu/api-compat-study`.

Acknowledgments

We thank the anonymous reviewers, William Jannen, and our shepherd, Bianca Schroeder, for their insightful comments on earlier drafts of the work. This research was supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, CNS-1405641, CNS-1408695, CNS-1526707, and VMware.

Appendix A Formal Definitions

A.1 API Importance

Definition: **API Importance.**
For a given API, the probability that an installation includes at least one application requiring the given API.

A system installation (`inst`) is a set of packages installed ($\{\texttt{pkg}_1, \texttt{pkg}_2, ..., \texttt{pkg}_k \in \texttt{Pkg}_{\texttt{all}}\}$). For each package (`pkg`) that can be installed by the installer, we analyze every executable included in the package ($\texttt{pkg} = \{\texttt{exe}_1, \texttt{exe}_2, ..., \texttt{exe}_j\}$), and generate the API footprint of the package as:

$$\texttt{Footprint}_{\texttt{pkg}} = \{\texttt{api} \in \texttt{API}_{\texttt{all}} \mid \begin{array}{l}\exists \texttt{exe} \in \texttt{pkg}, \\ \texttt{exe has usage of api}\end{array}\}$$

For a target API, API importance is calculated as the probability that any installation includes at least one package that uses the API; i.e., the API belongs to the footprint of at least one package. Using Ubuntu/Debian Linux's package installation statistics, one can calculate the probability that a specific package is installed as:

$$Pr\{\texttt{pkg} \in \texttt{Inst}\} = \frac{\text{\# of installations including pkg}}{\text{total \# of installations}}$$

Assuming the packages that use an API are $\texttt{Dependents}_{\texttt{api}} = \{\texttt{pkg} | \texttt{api} \in \texttt{Footprint}_{\texttt{pkg}}\}$. API importance is the probability that at least one package from $\texttt{Dependents}_{\texttt{api}}$ is installed on a random installation, which is calculated as follows:

$$\begin{aligned}&\texttt{Importance(api)} = Pr\{\texttt{Dependent}_{\texttt{api}} \bigcap \texttt{Inst} \neq \emptyset\} \\ &= 1 - Pr\{\forall \texttt{pkg} \in \texttt{Dependent}_{\texttt{api}}, \texttt{pkg} \notin \texttt{Inst}\} \\ &= 1 - \prod_{\texttt{pkg} \in \texttt{Dependent}_{\texttt{api}}} Pr\{\texttt{pkg} \notin \texttt{Inst}\} \\ &= 1 - \prod_{\texttt{pkg} \in \texttt{Dependent}_{\texttt{api}}} (1 - \frac{\text{\# of installations including pkg}}{\text{total \# of installations}})\end{aligned}$$

A.2 Weighted Completeness — A System-Wide Metric

Definition: **Weighted Completeness.**
For a target system, the fraction of applications supported, weighted by the popularity of these applications.

Weighted completeness is used to evaluate the relative compatibility on a system that supports a set of APIs ($\texttt{API}_{\texttt{Supported}}$). For a package on the system, we define it as supported if if every API that the package uses is in the supported API set. In other words, a package is supported if it is a member of the following set:

$$\texttt{Pkg}_{\texttt{Supported}} = \{\texttt{pkg} | \texttt{Footprint}_{\texttt{pkg}} \subseteq \texttt{API}_{\texttt{Supported}}\}$$

Using weighted completeness, one can estimate the fraction of packages in an installation that end-users can expect a target system to support. For any installation that is an arbitrary subset of available packages ($\texttt{Inst} = \{\texttt{pkg}_1, \texttt{pkg}_2, ..., \texttt{pkg}_k\} \subseteq \texttt{Pkg}_{\texttt{all}}$), weighted completeness is the expected value of the fraction in any installation (`Inst`) that overlaps with the supported packages ($\texttt{Pkg}_{\texttt{Supported}}$):

$$\texttt{WeightedCompleteness}(\texttt{API}_{\texttt{Supported}}) = E(\frac{|\texttt{Pkg}_{\texttt{Supported}} \bigcap \texttt{Inst}|}{|\texttt{Inst}|})$$

where $E(\texttt{X})$ is the expected value of `X`.

Because we do not know which packages are installed together, except in the presence of explicit dependencies, we assume package installation events are independent. Thus, the approximated value of weighted completeness is:

$$\begin{aligned}&\frac{E(|\texttt{Pkg}_{\texttt{Supported}} \bigcap \texttt{Inst}|)}{E(|\texttt{Inst}|)} \\ &\sim \frac{\sum_{\texttt{pkg} \in \texttt{Pkg}_{\texttt{Supported}}} (\frac{\text{\# of installations including pkg}}{\text{total \# of installations}})}{\sum_{\texttt{pkg} \in \texttt{Pkg}_{\texttt{all}}} (\frac{\text{\# of installations including pkg}}{\text{total \# of installations}})}\end{aligned}$$

References

[1] Cert c coding standards—signals. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=3903. Accessed on 3/21/2016.

[2] Debian popularity contest. http://popcon.debian.org/by_inst. Accessed on 3/21/2016.

[3] *diet libc*: A libc optimized for small size. https://www.fefe.de/dietlibc/. Accessed 3/21/2016.

[4] The embedded GNU Libc. http://www.eglibc.org/. Accessed on 3/21/2016.

[5] The GNU C library. https://www.gnu.org/software/libc/. Accessed on 3/21/2016.

[6] *musl* libc. http://www.musl-libc.org/. Accessed on 3/21/2016.

[7] Ubuntu popularity contest. http://popcon.ubuntu.com/by_inst. Accessed on 3/21/2016.

[8] *uClibc*. https://www.uclibc.org/. Accessed on 3/21/2016.

[9] wait4 man page. http://man7.org/linux/man-pages/man2/wait4.2.html. Accessed on 3/21/2016.

[10] J. J. Amor, J. M. Gonzalez-Barahona, G. Robles, and I. Herraiz. Measuring libre software using debian 3.1 (sarge) as a case study: Preliminary results. *UPGRADE - The European Journal for the Informatics Professional*, VI(3):13–16, 06 2005.

[11] J. J. Amor, G. Robles, and J. M. González-Barahona. Measuring Woody: The size of Debian 3.0. *CoRR*, abs/cs/0506067, 2005.

[12] J. J. Amor, G. Robles, J. M. González-Barahona, and I. Herraiz. From pigs to stripes: A travel through Debian. In *Proceedings of the DebConf5 (Debian Annual Developers Meeting)*, Helsinki, Finland, 07 2005.

[13] J. Appavoo, M. A. Auslander, D. Da Silva, D. Edelsohn, O. Krieger, M. Ostrowski, B. S. Rosenburg, R. W. Wisniewski, and J. Xenidis. Providing a Linux API on the scalable K42 kernel. In *Proceedings of the USENIX Annual Technical Conference*, pages 323–336, 2003.

[14] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2016.

[15] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2010.

[16] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[17] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–348, 2012.

[18] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–46, 2005.

[19] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dos. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2010.

[20] N. Borisov, R. Johnson, N. Sastry, and D. Wagner. Fixing races for fun and profit: How to abuse atime. In *Proceedings of the USENIX Security Symposium*, pages 303–314, 2005.

[21] X. Cai, Y. Gui, and R. Johnson. Exploiting unix file-system races via algorithmic complexity attacks. *Oakland*, pages 27–41, 2009.

[22] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proceedings of the USENIX Security Symposium*, pages 171–190, 2002.

[23] O. F. de Sousa, M. A. de Menezes, and T. J. P. Penna. Analysis of the package dependency on Debian GNU/Linux. *Journal of Computational Interdisciplinary Sciences*, 1(2):127–133, 03 2009.

[24] S. Denis. Linux distributions and applications analysis during linux standard base development. *Proceedings of the Spring/-Summer Young Researchers. Colloquium on Software Engineering*, 2, 2008.

[25] J. Dike. *User Mode Linux*. Prentice Hall, 2006.

[26] R. Divacky. Linux emulation in FreeBSD. https://www.freebsd.org/doc/en/articles/linux-emulation/, 03 2015. Accessed on 3/21/2016.

[27] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[28] H. Franke, R. Russel, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, 2002.

[29] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[30] P. J. Guo and D. Engler. CDE: Using system call interposition to automatically create portable software packages. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[31] 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 ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 71–83, 2011.

[32] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of μ-kernel-based systems. *SIGOPS Operating System Review*, 31(5):66–77, Oct. 1997.

[33] B. Jain, C.-C. Tsai, J. John, and D. E. Porter. Practical techniques to obviate setuid-to-root binaries. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 8:1–8:14, 2014.

[34] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 143–157, 2012.

[35] A. Kadav and M. M. Swift. Understanding modern device drivers. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 87–98, 2012.

[36] L. Lam and T.-c. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In E. Jonsson, A. Valdes, and M. Almgren, editors, *Recent Advances in Intrusion Detection*, volume 3224 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2004.

[37] R. Nguyen and R. Holt. Life and death of software packages: An evolutionary study of debian. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '12, pages 192–204, 2012.

[38] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–318, 2011.

[39] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 153–168. ACM, 2011.

[40] S. Pavel and S. Denis. Binary compatibility of shared libraries implemented in C++ on GNU/Linux systems. *Proceedings of the Spring/Summer Young Researchers. Colloquium on Software Engineering*, 3, 2009.

[41] A. Ponomarenko and V. Rubanov. Automatic backward compatibility analysis of software component binary interfaces. In *IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, volume 3, pages 167–173, June 2011.

[42] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 161–176, 2009.

[43] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. System call clustering: A profile directed optimization technique. Technical report, The University of Arizona, May 2003.

[44] G. Robles and J. M. González-Barahona. From toy story to toy history: A deep analysis of Debian GNU/Linux, 2003.

[45] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, and J. J. Amor. Mining large software compilations over time: Another perspective of software evolution. In *Proceedings of the International Workshop on Mining Software Repositories*, MSR, pages 3–9, 2006.

[46] SECure COMPuting with Filters (seccomp). `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`. Accessed on 3/12/2016.

[47] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, Oct. 2007.

[48] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, 1999.

[49] H. Singh and A. Kaur. Component compatibility in component based development. *International Journal of Computer Science and Mobile Computing*, 3:535–541, 06 2014.

[50] D. Tsafrir, T. Hertz, D. Wagner, and D. D. Silva. Portably preventing file race attacks with user-mode path resolution. Technical report, IBM Research Report, 2008.

[51] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 9:1–9:14, 2014.

[52] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter. How to Get More Value from your File System Directory Cache. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2015.

[53] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 149–158, 2009.

[54] J. Wei and C. Pu. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[55] M. Zalewski. Delivering signals for fun and profit. 2001.

[56] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 19–19, 2006.

[57] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the USENIX Security Symposium*, pages 337–352, 2013.