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

CHIA-CHE TSAI, BHUSHAN JAIN, NAFEES AHMED ABDUL, AND DONALD E. PORTER

Chia-Che Tsai is a PhD student at Stony Brook University. His research involves restructuring operating system designs for less vulnerability and higher performance. He is also the main contributor to the Graphene library OS. chitsai@cs.stonybrook.edu

Bhushan Jain is a PhD student at the University of North Carolina at Chapel Hill. His research interests include virtualization security, memory isolation, and system security. bhushan@cs.unc.edu

Nafees Ahmed Abdul is a Software Engineer at Symbolic IO. He earned his master's degree from Stony Brook University. He is broadly interested in file systems and storage technologies. nabdul@cs.stonybrook.edu

Donald E. Porter is an Assistant Professor of Computer Science at the University of North Carolina at Chapel Hill and, by courtesy, at Stony Brook University. His research aims to improve computer system efficiency and security. In addition to recent work on write-optimization in file systems, recent projects have developed lightweight guest operating systems for virtual environments, system security abstractions, and efficient data structures for caching. porter@cs.unc.edu

Application programming interfaces (APIs) specify how application developers interact with systems. As APIs evolve over the life of a system, the system developers have little in the way of empirical techniques to guide decisions such as deprecating an API. We propose metrics for evaluating the importance of system APIs, as well as the relative maturity of a prototype system that claims partial compatibility with another system. Using these metrics, we study Linux APIs—such as system calls, `ioctl` opcodes, pseudo-files, and libc functions—yielding insights for developers and researchers.

System developers routinely make design choices based on what they believe to be the common and uncommon behaviors of a system. Imagine a developer who is building a prototype system designed to run Linux applications. This developer will prioritize the implementation tasks based on what he or she believes to be important, which may be heavily skewed toward the developer's preferred workloads.

In general, developers struggle to evaluate the impact of adding or removing APIs on backward-compatibility with existing applications, primarily because of a lack of metrics. The state-of-the-art is *bug-for-bug compatibility*, which requires all behaviors (even undefined or undocumented behaviors) of a system to be identical to its predecessor. For instance, deprecating or retiring an API in Linux requires a lengthy process of repeatedly warning application developers to adopt a replacement API and confirming that no applications are broken by the change—a process that can take many years.

In evaluating compatibility or completeness of a prototype system, a common metric is a simple count of supported system APIs. For instance, the Graphene library OS [1] offers support for 143 out of 318 Linux x86-64 system calls. Although system call counts are easy to measure, this metric fails to capture essential aspects of compatibility, such as the fraction of applications or users that could plausibly use the system. In order to indicate general usefulness, a good compatibility metric should relate to the application usage patterns of end users, factoring in both common and uncommon cases.

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 article summarizes a study of Linux APIs; a longer version is published in EuroSys 2016 [2]. This study contributes a data set and analysis tool that can answer several practical questions about API usage and compatibility. For instance, if a developer were to add one additional API to a given system prototype, which API would most increase the range of supported applications? Or if a given system API is optimized, what widely used applications would likely benefit? 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.

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. In general, compatibility treated as a binary property (i.e., bug-for-bug compatibility) loses important information when evaluating a prototype that is almost certainly incomplete. Metrics such as the count of supported system APIs are noisy at best and give no guidance as to which APIs are the most important.

One way to understand the completeness of a system or the importance of an API is to measure the impact on end users. In other words, if a system supports a set of APIs, how many applications chosen by users can run on the system? Or if an API were not supported, how many users would notice its absence? To answer these questions, we must consider both the difference in API usage among applications, and application popularity among users. We measure the former by analyzing application binaries and determine the latter from the installation statistics collected by the Debian and Ubuntu Popularity Contests [3, 4].

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 emulated APIs, and assume an application will work on a target system if the APIs used by the application (or API footprint) is implemented on the system (i.e., we assume implemented APIs work as expected). The popularity of applications is measured by *installations*, which are collections of applications installed by users on physical machines, virtual machines, containers, or partitions in multi-boot systems.

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

API importance indicates how indispensable a given API is to at least one application on a randomly selected installation. Intuitively, if an API is used by no packages or installations, the API importance will be zero and its absence will cause no negative effects. If an API is only used by packages A, B, and C, the API importance will be the probability that either A, B, or C is chosen in an installation, which can be determined from package installation statistics. We consider an API to be *important* to an installation as long as one installed package requires the API. For instance, the `reboot` system call has almost 100% importance, but on most systems, this API is used only by the `/sbin/reboot` binary.

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

Weighted completeness indicates the fraction of installed applications that a prototype system can support on a randomly selected installation. Intuitively, a system with bug-for-bug compatibility will be able to support *all* applications and have 100% weighted completeness. If a system only supports packages A, B, and C, the weighted completeness will be the weighted fraction of A, B, and C over the average number of installed packages.

Data Collection

This study focuses on Ubuntu/Debian Linux as a baseline for comparison. We use static analysis to identify the API footprint of all applications and use installation statistics to evaluate the popularity of each application.

The methodology for measuring API importance and weighted completeness is summarized as follows:

1. For each available package, collect the API footprint of the package by disassembling all the binaries. The API footprint includes the APIs called by an executable or called by a library through function calls from the executable.
2. Calculate the API importance of each API based on the packages that use the API as well as the popularity of these packages.
3. For a target system, identify a list of supported APIs of the target system either from the system's source or as provided by the developers of the system.
4. Based on the API footprint of the packages, list the supported and unsupported packages for the target system.
5. Finally, weigh the list of supported packages based on their popularity and calculate the weighted completeness of the target system.

The Scope of This Study

Types of system APIs: We study various types of APIs defined in x86-64 Linux 3.19:

- 318 defined system call numbers.
- Opcodes for vectored system calls (635 `ioctl` opcodes, 18 `fcntl` opcodes, and 44 `prctl` opcodes).
- Pseudo-files in `proc`, `dev`, and `sys` file systems. In our application sample, we found 5,846 unique, hard-coded paths.
- 1,274 global functions in GNU libc 2.21 (libc.so only).

Sample of applications: 30,976 packages downloaded through `APT` on Ubuntu Linux 15.04.

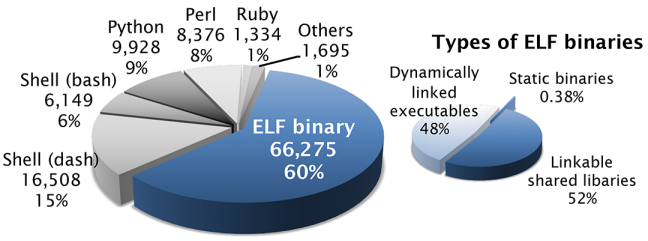

Figure 1: Types of applications and ELF binaries studied

Figure 1 shows the types of applications in these packages. We focused primarily on ELF binaries, which account for the largest fraction (60%) of Linux applications. For interpreted languages such as shell languages (21%) or Python (9%), we assume the API footprint of the applications is covered by the API footprint of the interpreter.

Package installation statistics: 2,935,744 installations collected in the Ubuntu and Debian Popularity Contests.

API Importance of Linux System Calls

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

- Which system calls are the most important to support in a new system or have highest costs to replace?
- Which system calls are candidates for deprecation?
- Which system calls are not supported by the OS but are still attempted by applications?

There are 318 system call numbers defined in x86-64 Linux 3.19. Figure 2 shows the distribution of system calls by API importance, ordered from the most important (near 100%) to the least important (0%)—similar to an inverted CDF.

Our study shows that over two-thirds (224 of 318) of Linux system calls have nearly 100% API importance. These system calls are indispensable for users—required by at least one application on every installation. Therefore, changing or removing these system calls would be highly disruptive.

On the other hand, we found 44 system calls with API importance above zero but less than 10%. In some cases, there are more popular alternatives with overlapping functionality. For instance, API importance for the System V message queue system calls (e.g., 100% for `msgget`) is higher than for POSIX message queue system calls (e.g., 5% for `mq_open`), although Linux supports both. This is attributable to System V message queues being more portable to other UNIX systems. Sometimes comparable functionality is provided by pseudo-files. For instance, the information returned by the system call `query_module` is also available by reading the pseudo-files `/proc/modules` and `/proc/kallsyms`.

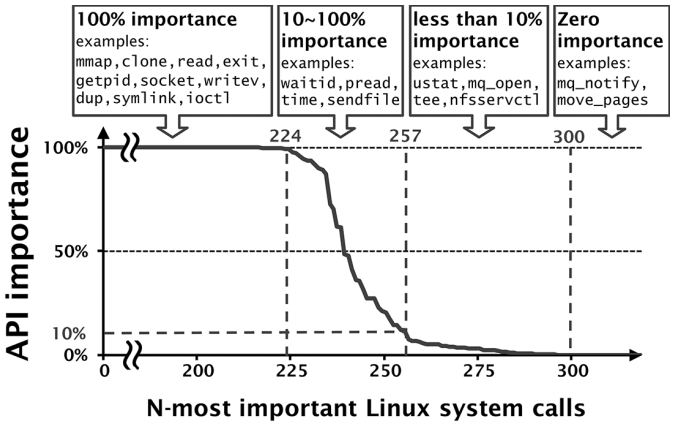

Figure 2: API importance of 318 Linux system calls in x86-64 Linux 3.19, ordered from the most to the least important

We also found five system calls—`uselib`, `nfsservctl`, `afs_syscall`, `vserver`, and `security`—that are officially retired but still have a non-zero API importance. These system calls are used because some applications still attempt the old calls for backward-compatibility with older kernels and, if these calls are unsupported, attempt newer API variants.

In total, 18 of 318 system calls defined in Linux 3.19 are not explicitly used by any application we studied. Eleven of these system calls are defined but retired and thus do not have an entry point in the kernel. Six system calls (`rt_tgsigqueueinfo`, `get_robust_list`, `remap_file_pages`, `mq_notify`, `lookup_dcookie`, and `move_pages`) are available but not used by any applications. These system calls are potential candidates for deprecation.

From "Hello World" to Every Application

For prototype systems or emulation layers, API importance and weighted completeness are useful for determining which APIs to implement first and for evaluating the progress of the prototypes. Our study shows an optimal path for adding system calls to a prototype system using a simple, greedy strategy of implementing the most important APIs first, which in turn maximizes weighted completeness.

Table 1 and Figure 3 demonstrate the optimal path of implementing Linux system calls, split into five stages, and the upper bound of weighted completeness that can be achieved. Essentially, one cannot run even the most simple application in Ubuntu/Debian Linux without at least 40 system calls. After this, the number of additional applications one can support by adding another system call increases steadily up to 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

Stage	System call examples	No. of system calls to support	Weighted completeness
I	mmap, vfork, exit, read, fcntl, kill, dup2	40	1.12%
II	mremap, ioctl, access, socket, poll, recvmsg	+41 (81)	10.68%
III	shutdown, symlink, alarm, listen, shmget, pread64	+64 (145)	50.09%
IV	flock, semget, ppoll, mount, brk, pause, clock_gettime	+57 (202)	90.61%
V	All remaining	+70 (272)	100%

Table 1: The optimal path of adding system calls in prototype systems, based on the order of API importance, to optimize the accumulated weighted completeness

calls. We do not provide a complete ordered list here in the interest of brevity, but this list is available as part of our released data set.

One of the uses of weighted completeness is to help guide the process of developing new prototype systems. Because 224 out of 318 system calls on Ubuntu/Debian Linux have 100% API importance, 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 support at least one application and have weighted completeness of more than 1%.

For simplicity, the optimal path we recommend only includes system calls, but one can construct a similar path including other types of APIs, such as vectored system calls, pseudo-files, and library functions.

Weighted Completeness of Linux Systems

We evaluate the weighted completeness of four systems or emulation layers: User-Mode-Linux [5], L4Linux [6], the FreeBSD's Linux emulation layer [7], and the Graphene library OS [1]. For each system, we identify the supported system calls by examining the defined system call tables in the source code. Figure 4 shows the weighted completeness of these systems based on the supported system calls. User-Mode-Linux (UML) and L4Linux both have over 90% weighted completeness, with more than 280 system calls implemented. FreeBSD's weighted completeness is 62.3% due to missing some less important system calls (e.g., `inotify_init`). Graphene's weighted completeness is only 0.42% due to missing scheduling control, but this is improved to 21.1% by adding two scheduling system calls.

For prototype developers, the proposed optimal path can maximize weighted completeness on a limited development budget,

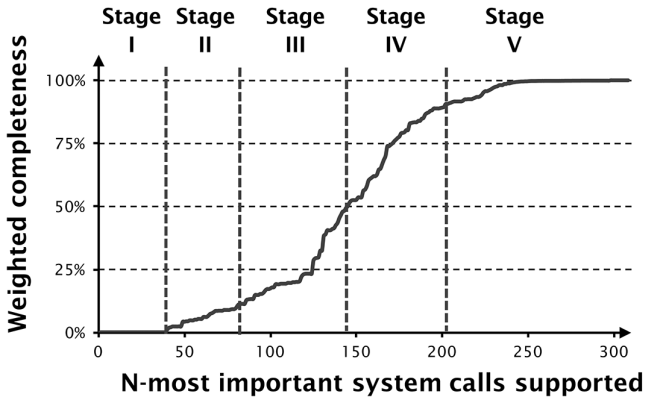

Figure 3: Accumulated weighted completeness when *N*-most important system calls are supported in the prototype system

especially for systems that implement a smaller fraction of APIs, such as Graphene or FreeBSD's Linux emulation layer. For existing prototypes, our study can identify the most important APIs that are missing, but can boost the weighted completeness most, witness the dramatic improvement (0.42% to 21.1%) on Graphene.

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 (*Opcode*). These *vectored system calls* significantly expand the system API, which we also consider in evaluating compatibility.

Among all the vectored system calls, `ioctl` represents the largest expansion of the Linux system APIs. There are 635 `ioctl` opcodes defined in the Linux 3.19 kernel source alone, and other kernel modules and drivers developed by third parties can define additional opcodes. Figure 5 shows the API importance of `ioctl` system call opcodes defined in Linux 3.19, ordered from the most important to the least important.

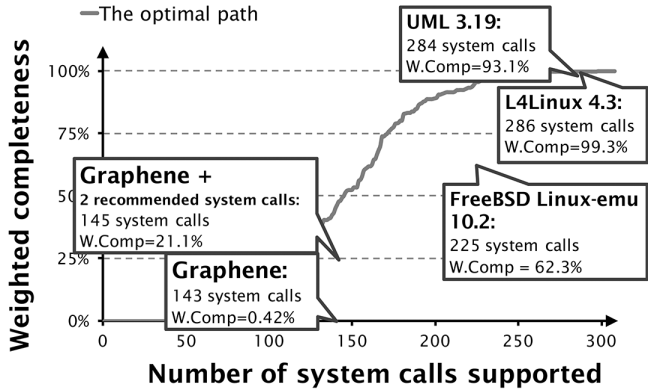

Figure 4: Weighted completeness of four systems or emulation layers with partial compatibility to Linux

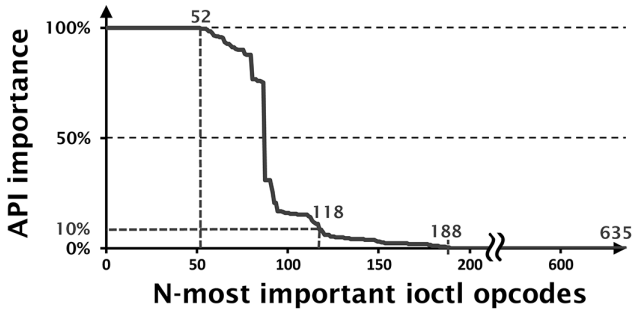

Figure 5: API importance of 635 `ioctl` opcodes defined in Linux 3.19 source, ordered from the most to the least important

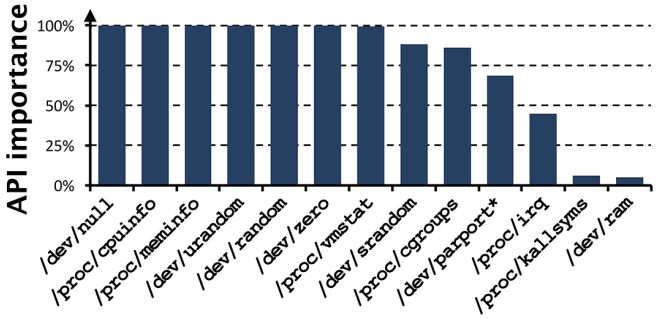

Figure 6: API importance of selected pseudo-files and devices under `/proc` and `/dev`

We observe a different trend for the API importance of `ioctl` opcodes from the system calls. Among the 635 `ioctl` opcodes defined in the Linux kernel source, fewer than one-tenth (52 opcodes) are indispensable on every installation, whereas more than two-thirds (447 opcodes) are never used by any application in Ubuntu/Debian Linux. In other words, when system developers implement `ioctl` opcodes in their prototype systems, it is more productive to focus on the opcodes that have higher API importance and implement the rest as needed for specific applications.

For the opcodes of other two vectored system calls, `fcntl` and `prctl`, we observe that the fraction of their opcodes being indispensable on every installations is higher than the `ioctl` opcodes. In Linux 3.19, 18 `fcntl` opcodes and 44 `prctl` opcodes are defined. Among them, 11 `fcntl` opcodes and nine `prctl` opcodes have 100% API importance.

Pseudo-Files and Devices

In addition to the main system call table, Linux exports many additional APIs through pseudo-file systems, often mounted at `/proc`, `/dev`, and `/sys`. These are called pseudo-files because they are not backed by any physical storage but instead export the contents of kernel data structures to applications or administrators. Although many of these pseudo-files are used on the command line or in scripts input by an administrator, there is also routine use of pseudo-files in applications.

We use static analysis to find hard-coded pseudo-file paths in application binaries. Our approach does not capture the cases where paths of pseudo-files are passed in as input to the applications, such as `dd if=/dev/zero`. However, we observe that when pseudo-files are widely used as alternative system APIs, their paths tend to be hard-coded in the binary as a string or string pattern, such as `/proc/%d/cmdline`, where `%d` can be any process ID. Our analysis captures hard-coded paths and string patterns.

Easy extensibility is an appeal of using pseudo-files as system APIs; as a result, their count can be an order-of-magnitude larger than the number of system calls or opcodes. We found 12,039 binaries that access pseudo-files and devices. Of these, 5846 unique paths were found hard-coded in these binaries. Figure 6 shows the API importance of several common paths for pseudo-files and devices.

We find that several files, such as `/dev/null`, are widely used through hard-coded paths in applications, even though there might be simpler alternatives. For instance, among 12,039 binaries that use a hard-coded path, 3324 are hard-coded to access `/dev/null`. Although `/dev/null` is convenient for use on the command line and in scripts, it is surprising that such a significant number of applications write to this pseudo-file rather than eliding the `write` system call.

Because many pseudo-files are accessed from the command line, 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 permissions on pseudo-files and devices tend to be set liberally enough to leak a significant amount of information [8]. For files used by a single application, an abstraction like a fine-grained capability [9] might better capture the security requirement.

Standard System Library Functions

Functions defined in standard system libraries, such as libc, are also system APIs. It is a common practice that developers tend to use library functions as more portable, user-friendly wrappers of the kernel APIs instead of directly calling these kernel APIs. For instance, GNU libc [10] exports functions for using locks and condition variables, which internally use the more subtle `futex` system call.

We study 1274 global function symbols exported by GNU libc 2.21 (`libc.so` only; other libraries, such as `libm.so` and `libpthread.so` are not included). Among these functions, 42.8% have an API importance of 100%, 50.6% have a API importance of less than 50%, and 39.7% have an API importance of less than 1%, including ones that are never used. This result implies that

processes load a significant amount of unnecessary code into their address space. By splitting libc into sub-libraries based on API importance and common linking patterns, systems could realize a non-trivial space savings and reduce the attack surface for code reuse attacks.

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 that 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.

Unweighted API Usage in Applications

Weighing metrics based on installations is important for understanding the impact of API changes on end users. By removing this weight, however, we can also observe trends in how APIs are used by application developers.

Unweighted API Importance
For a given API, the probability an application (package) uses that API, regardless of its installation probability

Unweighted API importance shows the preference of application developers for an API over its variants and the effort to communicate with all relevant application developers if an API is changed.

One common reason for system developers to design API variants is to replace APIs that are prone to security problems. For instance, many of the `set*id` system calls (e.g., `setuid`) have subtle semantic differences across UNIX platforms. Chen et al. [11] conclude that `setresuid` is the most secure choice for having the clearest semantics across all UNIX flavors. Another example is that file-accessing system calls (e.g., `access`) can be exploited through time-of-check-to-time-of-use (TOCTTOU) attacks, and their variants (e.g., `faccessat`) can be used to resist these attacks.

Table 2 shows the difference in unweighted API importance among the secure and insecure API variants. In some cases, like `set*id` system calls, the secure API variants (e.g., `setresuid`) are well-adopted and have higher unweighted API importance than the insecure ones. However, in more cases, like `get*id` and `access`, the insecure API variants are more commonly used by application developers.

Besides security-related reasons, system developers may create API variants by retaining old APIs for backward-compatibility. For instance, the `wait4` system call is considered obsolete and will soon be replaced by `waitid` [12], but `wait4` still remains available in Linux. In some other cases, multiple API variants are retained because one variant is specific to a particular OS like Linux, and the other is more generic and portable. Table 3 shows the difference in unweighted API importance among these variants. In general, API variants designed to be portable (e.g., `writev`) are more commonly used by the application developers than Linux-specific ones (e.g., `pwritev`). However, older APIs (e.g., `wait4`) may still be widely used in applications, even though the new APIs are introduced to improve the application portability.

Insecure API	Usage	Secure API	Usage
`setuid`	15.67%	`setresuid`	99.68%
`setreuid`	1.88%		
`getuid`	99.81%	`getresuid`	36.19%
`geteuid`	55.15%		
`access`	74.24%	`faccessat`	0.63%
`rename`	43.18%	`renameat`	0.30%
`chmod`	39.80%	`fchmodat`	0.13%

Table 2: Usage (unweighted API importance) of secure and insecure API variants in applications

Old (Obsolete) APIs vs. New APIs

Old API	Usage	New API	Usage
`getdents`	99.80%	`getdents64`	0.08%
`tkill`	0.51%	`tgkill`	99.80%
`wait4`	60.56%	`waitid`	0.24%

Linux-Specific APIs vs. Portable APIs

Linux-specific API	Usage	Portable API	Usage
`preadv`	0.15%	`readv`	62.23%
`pwritev`	0.16%	`writev`	99.80%
`accept4`	0.93%	`accept`	29.35%
`recvmmsg`	0.11%	`recvmsg`	68.82%
`sendmmsg`	5.17%	`sendmsg`	42.49%

Table 3: Usage (unweighted API importance) of similar API variants in applications

Conclusion

In this study, we define new metrics for evaluating API usage and compatibility, and, based on these results, we 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 all-system calls defined in Linux, once one considers vectored system call opcodes and pseudo-files. We also show that a substantial range of system calls and other APIs are rarely used. Finally, we provide a method to evaluate partial support of APIs in prototype systems, and plot an optimal path for adding system calls. We expect that the data set will be of use to researchers and developers for further study, and the methodology can be applied to future releases and other operating systems.

The data set and analysis tool are available at: http://oscar.cs.stonybrook.edu/api-compat-study.

Acknowledgments

We thank Bianca Schroeder and William Jannen for their insightful comments on this work. This research was supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, CNS-1405641, CNS-1408695, CNS-1526707, and VMware. Bhushan Jain is supported by an IBM PhD Fellowship.

References

[1] 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)*, 2014.

[2] C. Tsai, B. Jain, N. Ahmed 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 ACM European Conference on Computer Systems (EuroSys),* 2016.

[3] Ubuntu popularity contest: http://popcon.ubuntu.com.

[4] Debian popularity contest: http://popcon.debian.org.

[5] J. Dike, *User Mode Linux* (Prentice Hall, 2006).

[6] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schonberg, "The Performance of μ-Kernel-Based Systems," *SIGOPS Operating System Review,* vol. 31, no. 5 (Dec. 1997), pp. 66–77.

[7] R. Divacky, "Linux Emulation in FreeBSD," master's thesis: http://www.freebsd.org/doc/en/articles/linux-emulation.

[8] S. Jana and V. Shmatikov, "Memento: Learning Secrets from Process Footprints," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[9] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: A Fast Capability System," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP),* 1999.

[10] The GNU C library: http://www.gnu.org/software/libc.

[11] H. Chen, D. Wagner, and D. Dean, "Setuid Demystified," in *Proceedings of the 11th USENIX Security Symposium,* 2002.

[12] `wait4(2)` Linux man page: http://linux.die.net/man/2/wait4.