Noname manuscript No.
(will be inserted by the editor)

Analyzing a Decade of Linux System Calls

Mojtaba Bagherzadeh · Nafiseh Kahani · Cor-Paul Bezemer · Ahmed E. Hassan · Juergen Dingel · James R. Cordy

Received: date / Accepted: date

Abstract Over the past 25 years, thousands of developers have contributed more than 18 million lines of code (LOC) to the Linux kernel. As the Linux kernel forms the central part of various operating systems that are used by millions of users, the kernel must be continuously adapted to changing demands and expectations of these users.

The Linux kernel provides its services to an application through system calls. The set of all system calls combined forms the essential Application Programming Interface (API) through which an application interacts with the kernel.

In this paper, we conduct an empirical study of the 8,770 changes that were made to Linux system calls during the last decade (i.e., from April 2005 to December 2014)

In particular, we study the size of the changes, and we manually identify the type of changes and bug fixes that were made.

Our analysis provides an overview of the evolution of the Linux system calls over the last decade. We find that there was a considerable amount of technical debt in the kernel, that was addressed by adding a number of sibling calls (i.e., 26% of all system calls). In addition, we find that by far, the `ptrace()` and signal handling system calls are the most difficult to maintain and fix.

Our study can be used by developers who want to improve the design and ensure the successful evolution of their own kernel APIs.

1 Introduction

Since its introduction in 1991, the Linux kernel has evolved into a project that plays a central role in the computing industry. In addition to its usage

School of Computing, Queen's University, Kingston, Ontario
Email: {mojtaba, kahani, bezemer, ahmed, dingel, cordy}@cs.queensu.ca

on desktop and server systems, the Linux kernel forms the foundation of the Android operating system that is used on almost 1.5 billion mobile devices [21].

Over the past 25 years, thousands of developers have contributed more than 18 million lines of code (LOC) to the Linux kernel. The kernel source code and its development process have been thoroughly analyzed by software engineering researchers (e.g., [1, 23, 24, 30, 31, 40, 42, 43, 48, 50–52, 60, 61]).

As the Linux kernel forms the central part of various operating systems, it must be continuously adapted to fulfill the changing demands and expectations of users [35]. As a result, many changes to the kernel are driven by changing or increasing demands from the users of the operating systems that use the kernel, or by hardware evolution and innovation. Analysis of the evolution of the Linux kernel can provide us with a window into how the demands of both operating system users and the computing industry have evolved.

The Linux kernel provides its services to an application through *system calls*. All system calls combined form the essential Application Programming Interface (API) through which an application interacts with the kernel. Even the simplest Linux application uses system calls to fulfill its goals. For example, the `ls` command exercises 20 system calls more than 100 times to list the contents of a directory.

Studying the evolution of the API of a system can lead to valuable insights for developers of the APIs of other systems. For example, Bogart et al. [4] interviewed developers of the `R`, `Eclipse` and `npm` ecosystems to understand the practices that are followed by each ecosystem for breaking an API, and found that each ecosystem has their own way of handling and communicating breaking API changes. Such knowledge can be leveraged by API developers to make decisions about the way in which their own system handles breaking API changes.

In this paper, we conduct an empirical study on the 8,770 changes that were made to the Linux system calls during the last decade, to sketch an overview of the changing landscape of the Linux kernel API. We are the first, to the best of our knowledge, to focus on system calls rather than on the Linux kernel as a whole. The main contributions of our study are:

1. An overview of the evolution of the Linux system calls over the last decade in terms of the size and type of changes that were made to the system calls.
2. A study of the type of bug fixes that were made to the system calls over the last decade.

Our study can be used by developers who want to improve the design and ensure the successful evolution of their own kernel APIs.

The outline of the rest of this paper is as follows. Section 2 gives background information about system calls. Section 3 discusses related work. Section 4 presents the methodology of our empirical study. Section 5, 6 and 7 discuss the results of our empirical study. Section 8 discusses the implications of our results. Section 9 discusses threats to the validity of our study. Section 10 concludes the paper.

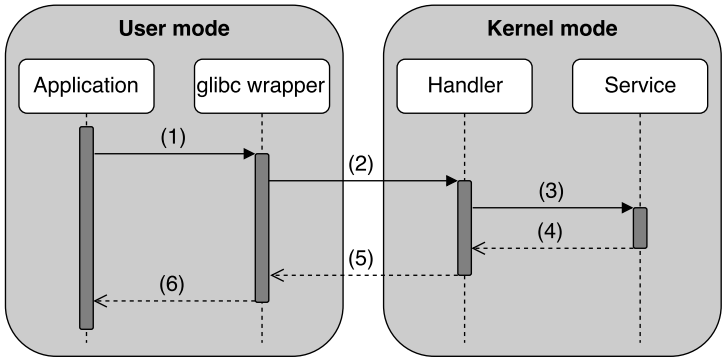

Fig. 1: The sequence of a system call.

2 System Calls

The kernel provides low-level services, e.g., network or file system-related, which need to be executed in kernel mode. In addition, the kernel enforces an isolated execution environment for each process. Therefore, it is necessary to continually make a context switch back-and-forth between the kernel and user mode. *System calls* are the primary method through which user space processes call kernel services.

Figure 1 outlines the (simplified) execution sequence of a system call. In this paper, we consider a system call as the combination of the handler and service in the kernel mode. While it is possible to invoke a system call directly from a process, in most cases the call goes through a wrapper function (step 1) in the C standard library (i.e, `glibc` [20]). Thus we focus on explaining the sequence for library calls here.

The `glibc` wrapper function traps the kernel into kernel mode and invokes the system call handler (step 2). The system call handler is a kernel function that retrieves the system call parameters from the appropriate registers and calls the required kernel services (step 3). Finally, the required kernel services are executed and the result is returned to the application (steps 4-6). There are two ways to trap into kernel mode, which we briefly discuss below.

2.1 The Old-Fashioned Way

Every available Linux system call has a unique identifier number [32]. In the old-fashioned way (i.e., before Linux kernel 2.5), the `glibc` wrapper function copies the identifier number of the system call that it wraps into the `%eax` register and copies the parameters into the other registers. The wrapper function then sends an interrupt, which causes the kernel to switch into kernel mode and read the `%eax` register to identify the appropriate service that must be called (step 2).

2.2 The Modern Way

Hayward reported that the old-fashioned way of executing system calls was slow on Intel Pentium 4 processors [26]. To solve this problem, an alternative way of executing system calls was added to the kernel. The alternative way uses Intel's `SYSENTER` and `SYSEXIT` (or AMD's `SYSCALL` and `SYSRET`) instructions [7]. These instructions allow fast entry and exit to and from the kernel without the use of expensive interrupts.

In the remainder of the paper, we present our empirical study of the system calls over the last decade.

3 Related work

In this section, we discuss prior related work. In particular, we first discuss related work on API evolution, then we discuss related work on evolution of the Linux kernel.

3.1 API Evolution

There have been many studies on API evolution. In this section, we give an overview of the most important prior work. The main contribution of our work in comparison to prior work on API evolution is that we are the first, to the best of our knowledge, to deliver an in-depth study of the evolution of the kernel API of an operating system with a very long maintenance history.

3.1.1 Refactoring in APIs

Dig and Johnson [12, 13] studied breaking changes in an API. They confirmed that refactoring plays an important role in API evolution. In particular, they found that more than 80% of breaking API changes are refactorings. Their conclusion is that many of these refactorings can and should be automated. In a large-scale study, Xavier et al. [68] showed that almost 28% of the API changes are breaking, which emphasizes the need for automation of API refactoring.

Several tools were proposed to automate API evolution (e.g., [27, 53, 69]). For example, the CatchUp! tool [27] uses the existing refactoring support of modern IDEs to record and replay API evolution. The Diff-CatchUp tool [69] uses differences between APIs to automatically suggest plausible replacements in the code that uses the API.

3.1.2 The effect of API evolution on developers

Robbes et al. [55] studied how developers in the Pharo ecosystem react to deprecation in the API. Hora et al. [28] later extended Robbes et al.'s study by studying the reaction of developers to all types of API changes in the Pharo

ecosystem. Hora et al. found that developers often do not react to API changes that are not because of deprecation. One of the main reasons is that developers are not notified of such API changes, while the use of a deprecated method yields a warning message. In addition, both Robbes et al. and Hora et al. found that it takes relatively long (i.e., a median adoption time of 14 days for deprecation changes and 34 days for all changes) to react to an API change.

Linares-Vásquez et al. [37] studied how the fault and change-proness of the Android API affects mobile apps that use this API. They found that Android apps that depend on fault and change-prone APIs are less successful. McDonnell et al. [46] found that the adoption time of changed API usage in the Android ecosystem is around 14 months.

In our study, we found that it can take very long (i.e., years) for system call changes to 'ripple through' to different system architectures (see Section 5.1), which indicates that even within one system, it takes time for API changes to be applied throughout the system.

3.2 Evolution of the Linux Kernel

The evolution of several aspects of the Linux kernel has been empirically studied over the years. We are the first, to the best of our knowledge, to study the evolution of system calls in depth, despite their importance. In this section, we discuss the most relevant related work.

Evolution of the Linux kernel as a whole: Godfrey and Tu [23, 24] conduct a quantitative study of the evolution of 96 versions of the Linux kernel. They find that the Linux kernel code grows at a geometric rate and follows Lehman's laws of software evolution [35]. In addition, Godfrey and Tu find that code cloning is a common practice in the Linux kernel, which is confirmed by Livieri et al. [40]. Izurieta and Bieman [31] re-analyze the evolution of the Linux kernel and conclude that the growth rate is similar to that of industrial systems.

Israeli et al. [30] study 810 versions of the Linux kernel, released over a period of 14 years, and find that the development follows several of Lehman's laws that are related to growth and stability of the development process. In addition, Israeli et al. find that the average complexity of functions is decreasing due to the addition of a large number of small functions.

Merlo et al. [48] define four metrics to study the similarity of 365 Linux kernel versions. They find that code removal is much higher in consecutive releases of unstable releases than for consecutive releases of stable releases. In a different study, Antoniol et al. [1] study code clones in the Linux kernel and find that code duplication remains stable across releases.

Lotufo et al. [42] and Passos et al. [52] explain that the Linux kernels offers its features and configuration options as an explicit variability model. Lotufo et al. study the evolution of this model and conclude that in the case of Linux, the evolution was smooth. Passos et al. present several evolution patterns for variability models that are extracted from a case study of the Linux kernel.

Li et al. [36] and Tan et al. [62] studied bug characteristics in open source projects, including Linux. Both studies found that semantic bugs, i.e., bugs that require domain knowledge to be solved, are by far the most common.

Atlidakis et al. [2] study the usage of POSIX in Android, Mac OS and Ubuntu Linux. POSIX is a set of standards and abstractions for operating system design, such as the design of the shell scripting language, file structure and environment variables. Atlidakis et al. find that while new abstractions are taking form, these abstractions are not converging into a new standard, which increases fragmentation across Linux-based operating systems.

The Unix operating system and the Linux kernel are very similar. Spinellis [60] discusses the Unix GitHub repository, which contains 44 years of Unix evolution. In addition, Spinellis et al. [61] study the evolution of C programming practices using the Unix operating system. They find, for example, that Unix developers evolved their coding style in tandem with advancements in hardware technology.

Evolution of a specific part of the Linux kernel: Tsai et al. [66] study the usage of the Linux API across all applications and libraries in the Ubuntu Linux 15.04 distribution. They identify important APIs by calculating the probability that an installation includes at least one application that requires the given API. Tsai et al. find that many APIs are not used in practice.

Palix et al. [51] use static analysis techniques to study the fault rate of parts of the kernel in versions 1.0 through 2.4.1 of Linux. The fault rate expresses the number of faults compared to the amount of code. Although kernel drivers comprise a large part of the kernel code and contain the majority of the faults, the fault rate of drivers is lower than the fault rate of the architecture-specific code and the file systems. In addition, Palix et al. find that while faults are continually being introduced, the overall quality of the kernel code is improving.

Padioleau et al. [50] study co-evolution of the Linux kernel and kernel drivers in versions 2.2 through 2.6 of Linux. From one version to the next one, co-evolution can account for up to 35% of the changed source code.

Lu et al. [43] study 8 years of Linux file system changes through 5,079 commits, of which 1,800 are bug fixes. They find that the number of bug fixes does not decrease over time. In addition, they show that semantic bugs, which require an understanding of file system semantics to find or fix, are the dominant bug category (over 50% of all bugs).

4 Methodology

This section introduces our approach for collecting and analyzing the evolution of system calls over the last decade. Figure 2 gives an overview of the steps of our data collection, and Figure 3 gives an overview of our empirical study.

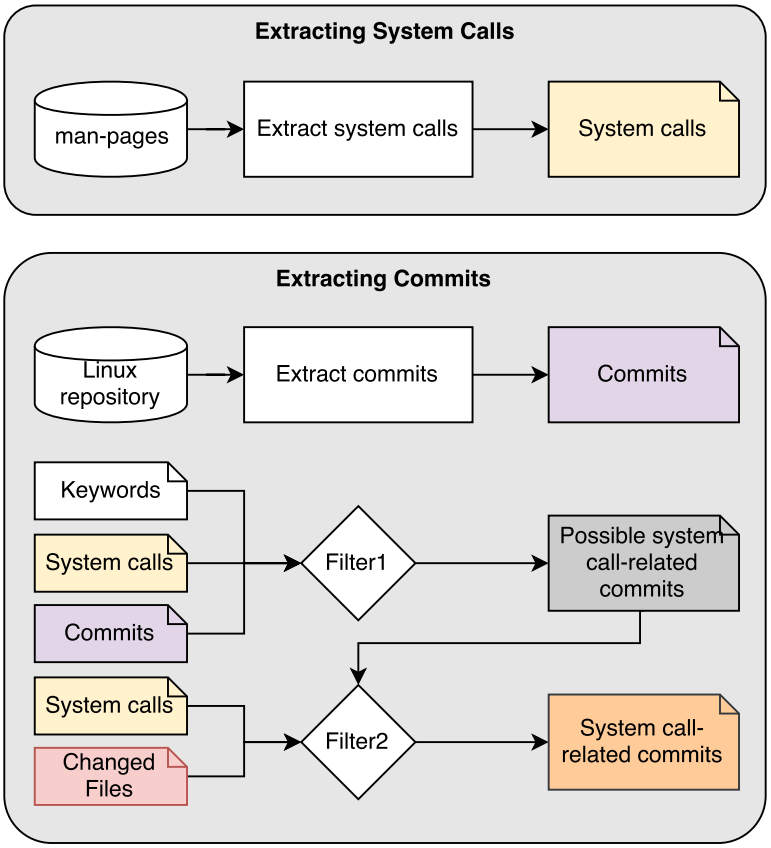

Fig. 2: An overview of our data collection.

4.1 Collecting Data

4.1.1 Extracting System Calls

We extract the list of existing system calls from the system call tables of each supported architecture (e.g., the Intel `x86` and AMD64 `x86_64`-architectures). For example, we study the `syscall*.tbl` and `syscall*.S` files, respectively for the `x86` and `s390`-architectures, in the `arch` file system in Linux. There exist 393 system calls as of Linux kernel 3.7. However, since not all architectures support all system calls, the number of available system calls differs per architecture [45]. 6 out of 393 system calls were removed from the kernel since earlier versions and 17 out of 393 system calls are architecture-specific, e.g., the `ppc_rtas` system call for PowerPC. Throughout this paper, we will study the full list of 393 system calls.

Two of the authors manually classified all existing system calls into the system call categories that are proposed by Mauerer [45]. After both authors independently finished the classification, they compared the classifications dis-

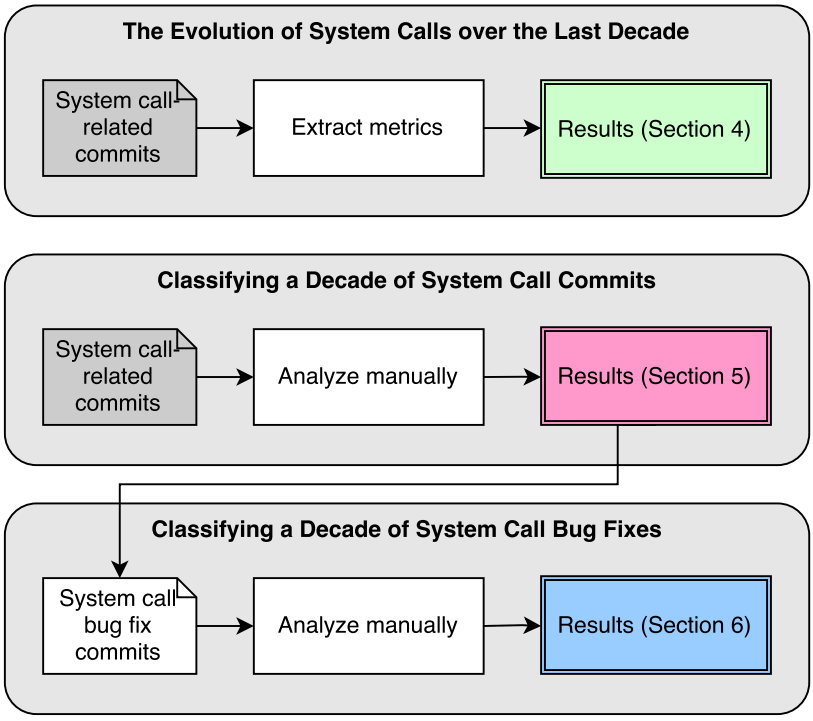

Fig. 3: An overview of our empirical study.

cussed and resolved differences. Table 1 shows the number of system calls for each system call category.

4.1.2 Extracting Commits

To study changes that are made to the system calls, we extract system call-related commits from the official Linux kernel git repository [65]. We consider a commit to be system call-related when it changes (1) code of the service provided by the system call, or (2) system call handler code. We extract commits, using the `--no-merges` option to avoid duplicate commits, between April 16, 2005 (i.e., the creation of the git repository) and December 31, 2014. We did not include commits from 2015, because we started our study in September 2015. Therefore, we did not have access to a set of all commits of 2015 at that time. We extract all commits of which the commit message matches one or more of the following keywords: the names of all system calls, and the terms "system call", "syscall" and "vdso"[1]. The keywords were identified based on our experience and study of documentation on making changes in the Linux kernel [39, 45].

[1] A vDSO is a shared object that can be accessed in the kernel and user mode without switching context. Hence, vDSOs are often employed by system calls [8].

Table 1: An explanation of the system call categories (ordered by the number of calls in each category). The last five columns show the number of system calls that were added to and removed from each category over the last decade.

				Number of system calls				
Code	**Category**	**Example**	**Total**	**New**	**NewF[1]**	**Sibling**	**Arch[2]**	**Rem[3]**
FS	File system & I/O	Reading and writing a file.	147	37	9	28	-	2
PM	Process management	Creating, cloning or debugging a process.	71	9	5	4	2	-
IPC	IPC[4] & network	Sharing memory between processes.	51	9	5	4	-	-
MM	Memory management	Mapping pages in memory.	27	7	7	-	3	2
SH	Signal handling	Killing a process.	24	3	1	2	-	2
TO	Time operations	Setting and querying the time.	23	4	4	-	-	-
SI	System info & settings	Retrieving information about the system.	21	1	-	1	-	-
SC	Scheduling	Thread prioritization.	14	2	2	-	-	-
SEC	Security & capabilities	Performing security checks.	8	3	3	-	-	-
MO	Modules	Loading a module.	6	1	-	1	-	-
All system calls			393	76	36	40	5	6

[1]NewF = the number of added system calls that provide new functionality
[2]Arch = the number of added system calls that are architecture-specific
[3]Rem = the number of removed system calls
[4]IPC = Interprocess Communication

We extracted 88,178 commits in total using our keywords. Not all of these commits are system call-related, because (1) system calls can be used in other parts of the kernel, and (2) some system calls have names that are common English words (e.g., `write()`). To extract commits that are truly system call-related, we applied a set of heuristics that are based on the location of the changed file(s). For example, we ignore commits that are extracted by the `read` keyword that do not change a file in the `/fs` (file system) folder. A full description of the heuristics that we applied is available in our online appendix [3]. After applying our heuristics, we have a set of 12,328 commits.

As a final filtering step, we manually went through the commits to remove all commits that were not related to system calls. After the final step, our data set contains 8,770 system call-related commits, covering all versions between 2.6.12-rc2 and 3.19-rc2 of the Linux kernel.

We manually verified the change history of two randomly-selected system calls (`reboot()` and `fork()`) and found that our keyword-matching missed respectively 0% and 4% of the commits for those system calls.

4.2 Analysis

Figure 3 shows the steps taken in our empirical study. Our empirical study consists of the following steps:

1. A quantitative analysis of system call-related commits (Section 5).
2. A manual classification and qualitative analysis of:
   (a) System call-related commits (Section 6).
   (b) System call bug fix commits (Section 7).

In the remainder of this paper, we present the results of each step of our empirical study in detail.

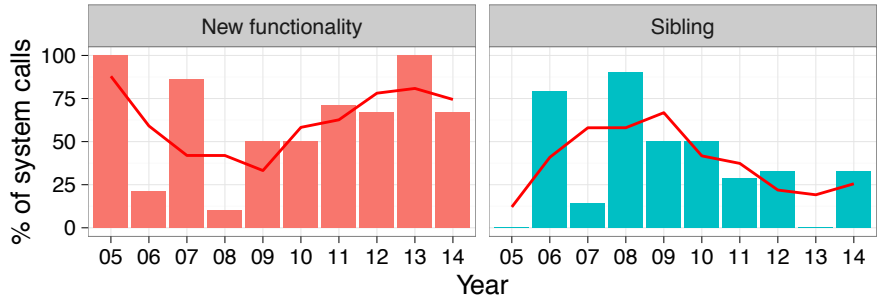

Fig. 4: The percentage of new system calls that provide new functionality and the number of sibling system calls. The red line is a LOESS regression fit line [19].

Table 2: System calls that were removed in the last decade.

System call	Removed	Reason
`set_zone_reclaim()`	2006	Had a flawed design and was therefore never made accessible in user mode and ultimately removed.
`perfctr()`	2010	Replaced by the `perf_open` system call.
`nfsservctl()`	2011	Replaced by a set of files that can be used to control the `nfsd` filesystem [33].
`remap_file_pages()`	2014	Had few users in practice and required 600+ lines of non-trivial code in the kernel [6].
`fast_syscall_xtensa()`	2014	Had issues when called with invalid arguments and was not used by anybody [18].
`fast_spill_registers()`	2014	Had issues when called with invalid arguments and was not used by anybody [18].

5 The Evolution of System Calls over the Last Decade

Motivation: We first conducted a study on the number and size of the Linux system calls. We studied the evolution of system calls over the last decade in three dimensions: (1) the number of system calls, (2) the size of a commit (in terms of the number of lines of code and the number of files that are changed), and (3) the number of developers who work on the system calls. We detail the approach and findings for each dimension below.

5.1 The Number of System Calls

Approach: We began with a quantitative study of the number of system calls that were added and removed from the kernel over the last decade, and the number of commits that are required to do so. We identified these commits through a manual classification process which is described in Section 6.

A new system call must be activated on a system architecture before it can be used on that architecture. Therefore, we studied the integration delay

for a system call on the supported system architectures to get a better understanding of whether all architectures are equally supported by the Linux kernel.

Finally, we conducted a qualitative study on the system calls that were added during the last decade. In this qualitative study, we first focused on sibling calls and then on the functionality that is added by new system calls.

Findings: **76 system calls were added to and 6 system calls were removed from the kernel, through 482 of the 8,870 (5%) system call-related commits.** Table 2 shows the system calls that were removed in the last decade, together with the reason for removing them. Removing a system call is always done in one or two commits. However, to add a system call, an average of 6.3 commits, 352 days and 5.4 developers are needed. The relatively large number of commits needed to add a system call demonstrates that adding a system call to the kernel is time-consuming and complex. To add a new system call, first, a service function is implemented which provides the main interface to the kernel. Then, the system call is activated (*wired up*) for the 31 system architectures that are currently supported by the Linux kernel. A system call can be activated by assigning a unique number to the call and adding its name and number to the system call table. Each supported architecture has its own system call table. Depending on the functionality of the system call, the system call may require an architecture-specific implementation.

A new system call is usually not activated for all architectures at the same time. We manually studied the 76 added system calls and observed that there may be a delay in activating a system call in architectures that ranges from a day to several years. For example, the `accept4` system call, which was introduced in November 2008, was activated on the same day for the `Sparc64`-architecture, in August 2010 for the `ARM`-architecture and in 2013 for the `Xtensa`-architecture (even though the kernel has supported this architecture since 2005 [70]).

40 out of 76 (53%) new system calls were sibling calls, which provide functionality that is similar to that of an existing system call. Table 1 shows the number of new system and sibling calls for each category over the last decade. A sibling call is a system call that is similar in functionality, and often in name, to another system call. In most cases, sibling calls are a repayment of technical debt in the Linux kernel API, i.e., the introduction of a sibling call indicates that the original system call was not designed with the required extensions in mind.

Additionally, our study shows that 102 of the 393 (26%) currently existing system calls are sibling calls. We identify six types of sibling calls, which can be distinguished by the name of the sibling call. Table 3 shows the number of sibling calls of each type and the pattern through which each type can be identified. In the following paragraphs, we explain each type of sibling call.

1. *Parameter extension-sibling calls*: The *parameter extension*-sibling calls are wrapper functions for the original system call. These sibling calls can be

recognized by the number [1..4] after the system call name, which indicates the number of arguments that the sibling call takes. This type of sibling call emerged from extended knowledge of how the original system call is used in practice. To prevent endless extension of the list of system calls with siblings for every new argument (e.g., the `dup()`, `dup2()` and `dup3()` system calls), the `flags` and a flexible structure pointer argument were introduced [38]. The bits of the `flags` argument can be used to select behaviour in the system call. The usage of the `flags` argument was officially included in the guidelines for adding a system call in 2015 [17], but the `flags` argument was used much earlier, e.g., in 2006 [16]. The flexible structure pointer allows to pass a `struct` object with the function arguments enclosed, which can be extended as required.
While the `flags` and flexible `struct` arguments introduce some additional complexity inside the system call (i.e., to handle the arguments), the impact on the kernel interface itself and dependent applications is smaller than when new arguments are continually added to the system calls. Figure 4 shows that the percentage of new system calls that are sibling calls has considerably decreased over the past few years, which suggests that the strategy of using a flexible `flags` and `struct` argument is effective for avoiding the introduction of new sibling calls. An example of a system call that uses the `struct` argument is the `perf_event_open()` system call.

2. *Architecture-sibling calls*: The *architecture*-sibling calls add support for 32 and 64-bit arguments to the original system call. For example, the `truncate64()` system call supports truncating larger files than the `truncate()` system call does.
3. *Working directory-sibling calls*: The `working directory`-sibling calls were all added in 2006 to implement a virtual current working directory, which is necessary for, e.g., a multi-threaded backup [15]. The difference between an original system call and its *working directory*-sibling is the way in which the parameters are treated. For example, the `open()` and `openat()` system calls have the following signatures:

```
int open(const char *pathname, int flags)
int openat(int dirfd, const char *pathname, int flags)
```

If `pathname` contains a relative path, the `open()` system call will interpret the path relative to the current working directory of the calling process. The `openat()` system call will interpret the path relative to the directory referred to by file descriptor `dirfd`. Aside from how the parameters are treated, the `open()` and `openat()` system calls provide the same functionality.

4. *Backwards compatibility-sibling calls*: In six cases, a system call was replaced by a newer version. The old version was renamed (e.g., from `vm86()` to `vm86old()`) to provide backwards compatibility. The `glibc` wrapper function takes care of the backwards compatibility, so that developers who

wish to remain using the old system call do not need to change their applications.

5. *Real time-sibling calls*: The eight *real time*-siblings add support for real-time operations to the system call. The main difference between the original system call and its *real time*-sibling is that the sibling can handle larger signal sets as argument.
6. *Other sibling calls*: There exist 30 sibling calls that cannot easily be grouped based on their naming scheme. An example of these sibling calls is the `waitpid()` system call, which suspends execution of the calling process until the child process specified by the `pid` argument terminates. The `waitpid()` system call is a sibling of the `wait4()` system call, which suspends the execution of the calling process until one of its children terminates.

Table 3: The number of sibling calls of each type.

Type of sibling	**Pattern**	**# of calls**	**Example**
Parameter extension	**[1..4]*	12	`dup()`, `dup2()`
Architecture	**[32|64]*	32	`truncate()`, `truncate64()`
Working directory	**at*	14	`open()`, `openat()`
Backwards compatibility	**old*	6	`vm86()`, `vm86old()`
Real time	*rt**	8	`sigreturn()`, `rt_sigreturn()`
Others	-	30	`waitpid()`, `wait4()`
Total number		102	

Table 4: The number of new system calls in the last decade that provide new functionality, grouped by functionality and ordered by the number of system calls.

Functionality	**# of system calls**	**Example**
Monitoring	8	`inotify()`, `getcpu()`
Synchronization	7	`eventfd()`, `signalfd()`
Hardware-specific	6	`cacheflush()`, `move_pages()`
Message passing	5	`process_vm_readv()`, `tee()`
Security	3	`bpf()`, `seccomp()`
Other[1]	7	`setns()`, `clock_adjtime()`
Total number	36	

[1]The "other" functionality group contains system calls that provide functionality which could not be grouped.

9 sibling calls provide a batch execution of the original system call. These sibling calls allow the user to repeatedly execute a system call without having to make the expensive context switch between execution modes. For example, the `sendmmsg()` system call allows the user to send multiple messages, rather than the `sendmsg()` system call which allows to send only one at a time. The first batch sibling call in our studied dataset was added in 2009 [9]. However, earlier examples of batch sibling calls exist. For example, the `writev()` and `readv()` sibling calls were introduced as batch sibling calls for the `write()` and `read()` system calls in `4.2BSD` in 1983 [34].

Most functionality that is added by new system calls during the last decade is to support monitoring and synchronization. We grouped the new functionality system calls from the last decade based on the type of newly-added functionality. Table 4 shows the functionality groups and the number of system calls that provide such functionality.

> *Not all architectures directly support all new system calls. 53% of the new system calls that were added in the last decade are sibling calls, which are a repayment of technical debt in the kernel. To avoid having to add additional sibling calls in the future, flexible* `flags` *and* `struct` *arguments are currently being used.*

5.2 The Size of System Call Commits

Approach: We conducted a quantitative study on the number of lines of code and the number of files that were changed by system call-related commits and kernel commits. We extracted the system call-related commits as described in Section 4. We extracted the kernel commits by retrieving all 508,980 commits that are made in the study period (including the system call-related commits) from the git repository of the Linux kernel.

We used the Wilcoxon signed-rank test to compare commits that are made to the system calls with commits that are made to the kernel. The Wilcoxon signed-rank test is a non-parametrical statistical test, of which the null hypothesis is that the two input distributions are identical. If the p-value of the Wilcoxon test was smaller than 0.05, we rejected the null hypothesis and concluded that the input distributions are significantly different. To quantify the difference between two distributions, we calculated Cliff's delta effect size [41]. Cliff's delta returns a real number d between -1 and 1. The absolute value of the returned number is used to assess the magnitude of the effect size. We used the following thresholds for d, which are provided by Romano et al. [56]:

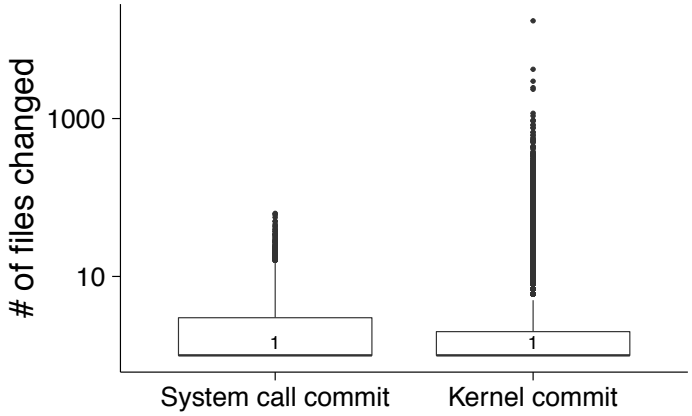

Fig. 5: The number of files that are changed in a system call-related commit and in a kernel commit. The number in the boxplot is the median value. Note that the axis is in logarithmic scale.

$$\text{Effect size} = \begin{cases} negligible, & \text{if } |d| \leq 0.147. \\ small, & \text{if } 0.147 < |d| \leq 0.33. \\ medium, & \text{if } 0.33 < |d| \leq 0.474. \\ large, & \text{if } 0.474 < |d| \leq 1. \end{cases}$$

Findings: **There were on average 25 lines of code (LOC) that were added to the system calls per day.** Compared to the average number of LOC that are added to the Linux kernel (approximately 3,500 in 2012 [54]), the growth of the system call code is relatively slow. The slow growth implies the following: (1) the system calls are relatively stable and (2) it is feasible for system call developers to keep track of the daily commits and evolution of the system calls.

The commits that are made to system calls are slightly more scattered than kernel commits. Figure 5 shows the distribution of the number of files that are changed in a system call-related commit and in a kernel commit. We found that 58% of the system call-related commits make changes to one file, while 75% of the commits makes changes to at most two files. The Wilcoxon signed-rank test shows that the difference between the number of files that are changed in a system-call related commit and a kernel commit is significant, but with an effect size of 0.05, which is negligible.

With an average growth of 25 LOC per day, the system calls are relatively stable.

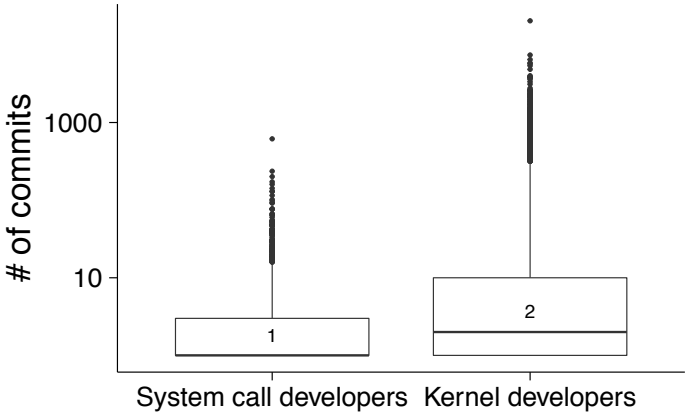

Fig. 6: The number of commits that are made by each developer in the last decade. The number in the boxplot is the median value. Note that the axis is in logarithmic scale.

5.3 The Developers of System Calls

Approach: We conducted a quantitative study on the system call developers and kernel developers. We computed the skewness of the distribution of the number of commits made by a developer to study whether the contributions are equally spread over the community. The skewness of a distribution captures the symmetry of that distribution around the mean and median. A positive skew means that most developers contribute a small number of commits, while a negative skew means that most developers contribute a large number of commits.

Findings: **The majority of developers who worked on the system calls during the last decade provided a single fix or extension.** Figure 6 shows the number of commits that were made by each developer to the system calls and the Linux kernel during the last decade. The median number of commits made by a developer was 1, compared to 2 for the Linux kernel. In both cases, the distribution of the number of commits was heavily right-skewed (i.e., a skewness of 16.27 and 37.11 for the system calls and kernel respectively). The high skewness indicates that while there is a small group of very active developers, the majority of system call developers commit only once.

There exists a small group of very active system call developers.

6 Classifying a Decade of System Call Commits

Motivation: In the previous section we studied how much system calls changed over the last decade. However, we did not study in depth why and how system calls evolved. In this section, we classified system call-related changes from the last decade based on the driver for committing them.
Approach: The first two authors manually and independently classified the commit messages of all 8,770 system call-related commits that were extracted from 2005 to 2015 into one or more of the following commit categories:

1. **Add/remove:** The commit was made to add or remove one or more system calls.
2. **Bug fix:** The commit was made to fix a bug.
3. **Improvement:** The commit was made to make an improvement.
4. **Restructuring:** The commit was made to conduct code restructuring, such as cleaning up comments or refactoring.

After classifying all commits, the first author identified 686 out of 8,770 (8%) conflicting classifications. To resolve these conflicts, the first two authors discussed the differences until an agreement was reached.

Table 5: The system calls with the most commits during the last decade.

System call	ALL	Restructuring[1]	Bug fix	Improvement
`ptrace()`	743	46%	35%	21%
`signal()`	714	53%	33%	18%
`ioctl()`	438	44%	32%	25%
`futex()`	257	35%	43%	23%
`ipc()`	253	51%	23%	30%
`mmap()`	213	30%	43%	31%
`perf_event_open()`	199	10%	46%	45%
`readdir()`	169	46%	41%	14%
`splice()`	166	30%	40%	25%

[1]Note that since a commit can be classified into multiple categories, the percentages in a row for a system call may not add up to 100%.

Findings: **8,288 of the 8,770 commits (95%) were made to maintain, improve and fix bugs in system calls. 4,498 (50%) of these commits were made to only 25 (6%) of the 393 system calls.** Table 5 shows the system calls for which more than 150 commits were made during the last decade. The `ptrace()` and `signal()` system calls required by far the most commits. The high number of bug fixes demonstrates the complexity of the `ptrace()` and `signal()` system calls, which is caused by their conceptual complexity and dependence on the underlying architecture. The `ptrace()` system call is used in debuggers or system call tracing applications. The `signal()` system call is used to install a new signal handler. As both the `ptrace()` and `signal()` system calls trigger exceptional cases in context switching, such as

Table 6: The number of commits per commit category, ordered by the number of commits.

Commit category	# of commits	% of commits
Restructuring	3,164	35
Bug fix	3,247	36
Improvement	2,131	24
Add/remove	482	5
Total # of classifications[1]	9,024	100

[1]Note that this number is higher than the total number of studied commits, as we classified some commits into multiple categories.

the system call restart mechanism [5, 63], their internals are complex by nature. The complexity of the `ptrace()` and `signal()` system call (especially when they are interacting) is acknowledged on the Linux kernel mailing list by one of the main kernel developers [67].

35% of the system call-related commits were made to conduct restructuring on the code. Table 6 shows the number of commits per commit category. The large portion of restructuring commits emphasizes the importance of refactoring to keep the source code of the system calls clean and manageable. The restructuring activities consist of writing helper functions, cleaning dead or duplicate code, merging code, generalizing functionality, relocating files and formatting code.

36% of the system call-related commits were made to fix bugs. Figure 7 shows the trends of the number of commits per commit category for the last 10 years. The number of bug fixes has been steadily increasing over the last decade. Lu et al. [43] observed a similar number of bug fixes (approximately 35% of the commits) and a similar trend when studying eight years of Linux file system-related commits. In Section 7 we study the bug fixes in more detail.

Restructuring of the `ptrace()` and signal handling system calls caused restructuring peaks in 2008 and 2012. As shown in Figure 7, there were peaks in the number of restructuring commits in 2008 and 2012. In 2008, the peaks were caused by restructuring to the `ptrace()` and signal handling system calls.

The `ptrace()` system call is highly dependent on the underlying system architecture [57]. Hence, applications that rely on the `ptrace()` system call, such as `gdb` and `strace`, are not easily portable. In 2008, 101 out of 485 (21%) restructuring commits were made to make the `ptrace()` system call less dependent on the underlying system architecture.

140 out of 485 (29%) restructuring commits were made in 2008 to signal handling system calls. Most of these commits are made to unify the source code of the 32-bit, 64-bit and real-time versions of the signal handling system calls. For example, the source code of the 32 and 64-bit version of the `signal()` system call was merged into one file. Within that file, the preprocessors `#ifdef`

`CONFIG_X86_64` and `#ifdef CONFIG_X86_32` are used to execute the required version of the system call.

In 2012, 164 out of 448 (37%) restructuring commits were made to signal handling system calls. The majority of those 171 commits were made to change the `sigaltstack()`, `sigprocmask()`, `sigsuspend()`, `sigaction()`, `rt_sigprocmask()`, `rt_sigpending()`, `rt_sigqueueinfo()` and `rt_sigaction()` from architecture-specific into generic system calls.

In 2009, 45 commits were made to improve the robustness of system calls towards a reported security issue. Vulnerability report `CVE-2009-0029` [49] describes a security issue in which system calls of the `s390`, `PowerPC`, `sparc64` and `MIPS` 64-bit architecture rely on the user-mode application to do sign extension when using 32-bit arguments in a 64-bit register. The system calls did not verify that the sign extension was done correctly, allowing malicious users to crash the kernel or gain privileges through a crafted system call. 45 commits were required to make the architectures in question robust to this security issue.

The `perf_event_open()` system call had 87 changes in 2009, of which 28 were bug fixes. The `perf_event_open()` system call was added in December 2008 and provides an abstraction for accessing performance counters. Since this abstraction is architecture-specific, the implementation of the abstraction is complex and bug-prone, which is demonstrated by the high number of bug fixes for this system call in 2009.

In 2014, there is an increase in bug fixes related to file system & I/O calls. However, we were unable to identify a specific system call as the culprit for the increase.

Bug fixes had the smallest commit size. We calculate the size of a commit by counting the number of lines of code that are added and removed by the commit. Figure 8 shows the distributions of the commit size for the commits in the four commit categories. The median commit size of a bug fix is 9. The largest commits are made for improvements (median commit size 23) and restructuring (median commit size 24).

> *The vast majority of the development effort goes to a small group of system calls. Especially the `ptrace()` and `signal()` system calls require a large amount of restructuring.*

7 Classifying a Decade of Bug Fixes for System Calls

Motivation: In Section 6, we observed that one-third of the system call-related commits were made to fix one or more bugs. In this section, we studied these bug fixes in more depth to gain an insight into what type of bugs were prevalent in system calls during the last decade. Such insights can help system call developers to better understand what are the most bug-prone and therefore, more difficult to develop, system calls.

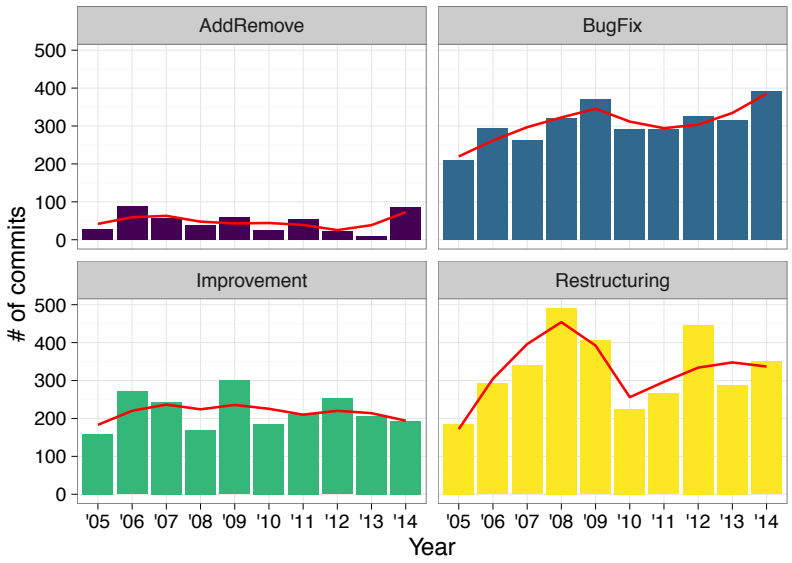

Fig. 7: The number of commits made in each commit category over the last decade. The red line is a LOESS regression fit line.

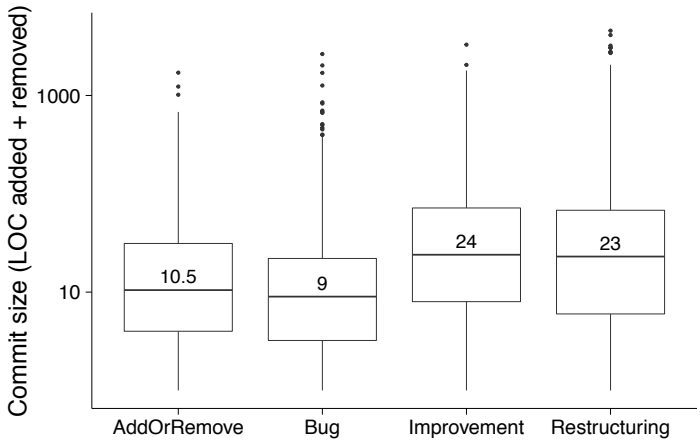

Fig. 8: The size of the commits that were made in each commit category over the last decade. The number in the boxplot is the median value. Note that the commit size is in logarithmic scale.

Approach: To conduct a qualitative study of the bug fixes for system calls, we classified all bug fix commits into bug fix categories, based on the type of bug that the commit was fixing. We used the same process for classifying the bug fix commits as we did for the commits in Section 6. The first two authors manually and independently classified the commit message of all 3,067 bug fix commits into one or more of the following bug fix categories:

1. **Compatibility:** Compatibility-related bugs are caused by compatibility issues between architectures (e.g., 32-bit versus 64-bit).
2. **Concurrency:** Concurrency-related bugs are caused by issues with atomicity, execution order, synchronization or locking, and lead to problems such as deadlock or race conditions [44].
3. **Error code:** Error code-related bugs are caused by returning the wrong error code or handling a returned error code incorrectly.
4. **Memory:** Memory-related bugs are caused by incorrect usage of the memory, thereby introducing an issue such as a memory leak.
5. **Semantic:** Semantic bugs are bugs in the implementation of the system call-specific behaviour, such as the logic of the service provided by the system call.

We used the same bug fix categories as Lu et al. in their study of bug fixes for Linux file systems [43], but we added the compatibility category as we find during our study that compatibility bugs are a recurring issue for system calls. After classifying the bug fix commits, we identified 62 out of 3,247 (2%) conflicting classifications. These conflicts were resolved in the same way as described as in Section 6.

In addition, we calculated the number of bug fixes per system call by dividing the number of bug fixes for all system calls in a system call category (see Table 1) by the number of system calls in that category. For example, there were 69 compatibility-related bug fixes and 147 system calls in the file system & I/O category. Hence, there were 0.47 compatibility-related bug-fixes per file system & I/O system call.

Finally, we calculated the normalized static entropy [25] of each system call to express its bugginess as a value between 0 and 1. The entropy helps us to understand how the bug fixes are spread over the study period. A system call has an entropy of 0 when all its bug fixes are made in the same year. Likewise, a system call has an entropy of 1 when its bug fixes are evenly spread over the studied years. We calculate the normalized static entropy for each system call as follows:

$$entropy_s = -\sum_{k=1}^{n}(p_k * log_n p_k)$$

where n is the number of studied years and p_k is the probability of having a bug fix in year k for system call s. For example, if there were 10 bug fixes in total in 10 years for a system call s, of which 2 were made in 2005 and 8 in 2012, the entropy of s is 0.22:

$$\begin{aligned} entropy_s &= -\frac{2}{10} * log_{10}\frac{2}{10} + \frac{8}{10} * log_{10}\frac{8}{10} \\ &= 0.22 \end{aligned}$$

Findings: **Developers make mistakes in the seemingly trivial activation process of a system call.** The steps that are required to activate a

system call, such as assigning the unique number and updating the system call table, are performed manually. As a result, developers make mistakes in the activation process. For example, a mistake in the system call number of the `migrate_pages()` system call caused the `migrate_pages()`, `select6()` and `ppoll()` system calls to malfunction [47]. Similar typographical errors caused problems in other cases [10, 11, 14, 58, 59, 64].

Clearly there is value in automating the activation process of a system call. In particular, automation can help developers to adhere to the DRY-principle (Don't Repeat Yourself) [29], as the activation process is the same for all system calls. Currently, the only automated assistance is given by the `linux/scripts/checksyscalls.sh` script, which lists the missing system calls for a specific architecture, as compared to the `i386`-architecture.

Memory management system calls have the highest bug fix entropy. Table 7 shows the median entropy of a system call for each system call category. Memory management system calls have the highest median entropy (0.74), which indicates that most memory management system calls had bug fixes throughout the study period. There are six system calls that have an entropy of 0.95 or higher: the `ptrace()`, `mbind()`, `signal()`, `umount()`, `symlink()`, `fork()` and `fcntl()` system calls. The entropy of the `ptrace()` and `signal()` system calls emphasizes their complexity, as the number of bug fixes made for those system calls is very high as well (see Table 5). The high number of bug fixes and entropy indicate that every year a large number of bug fixes is made to these system calls.

Table 7: The median entropy of a system call for each system call category, ordered by median entropy.

Category	Median entropy
Memory management	0.74
File system & I/O	0.45
Process management	0.45
System info & settings	0.44
IPC & network	0.43
Security & capabilities	0.38
Time operations	0.30
Signal handling	0.30
Scheduling	0.30
Modules	0.00

58% of the bug fix commits were made to fix semantic bugs. Table 8 shows that semantic bugs are by far the most common for system calls. This observation is in line with Lu et al.'s [43] findings for Linux file systems.

The portion of bug fixes that fix memory-related bugs remained constant throughout the last decade. Figure 9 shows the portion of all bug fixes that fit in a specific bug category over the last decade. Interestingly, the proportions remain relatively constant over the years. In their study on bugs

Table 8: The number of commits per bug fix category, ordered by the number of commits.

Bug fix category	**# of commits**	**% of commits**
Semantic	1,922	58
Concurrency	521	16
Memory	339	10
Compatibility	285	9
Error code	221	7
Total # of classifications[1]	3,288	100

[1]Note that this number is higher than the total number of studied commits, as we classified some commits into multiple categories.

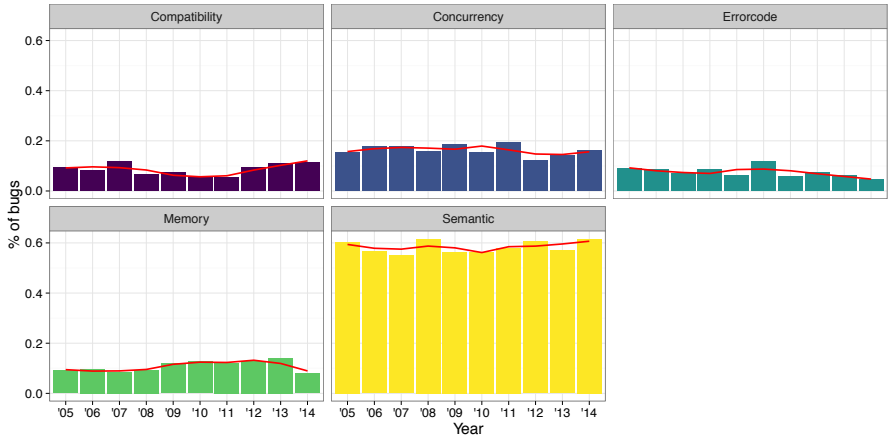

Fig. 9: The % of bugs made in each bug category over the last decade. The red line is a LOESS regression fit line.

in open source software, Li et al. [36] reported a downward trend for memory-related bugs from 1991 to 2006, due to the use of automated detection tools of memory-related bugs.

A manual review of the memory-related bug fixes for system calls shows that most fixes are for "simple" issues, such as memory leaks or null-pointer dereferences. The majority of the memory leaks are caused by (1) not properly releasing memory after entering a failure path and (2) not initializing a variable properly. The majority of the null-pointer dereferences are caused by not validating a returned value or function argument.

The downward trend for memory bugs does not appear to apply or continue for system calls, which suggests that existing tooling is not powerful enough to automatically detect memory-related bugs in system calls, or that such tooling is not used by inexperienced system call developers.

Signal handling system calls have the highest number of semantic (9.33) and compatibility-related (1.70) bug fixes per system call. The high number of compatibility-related bug fixes emphasizes the dependency

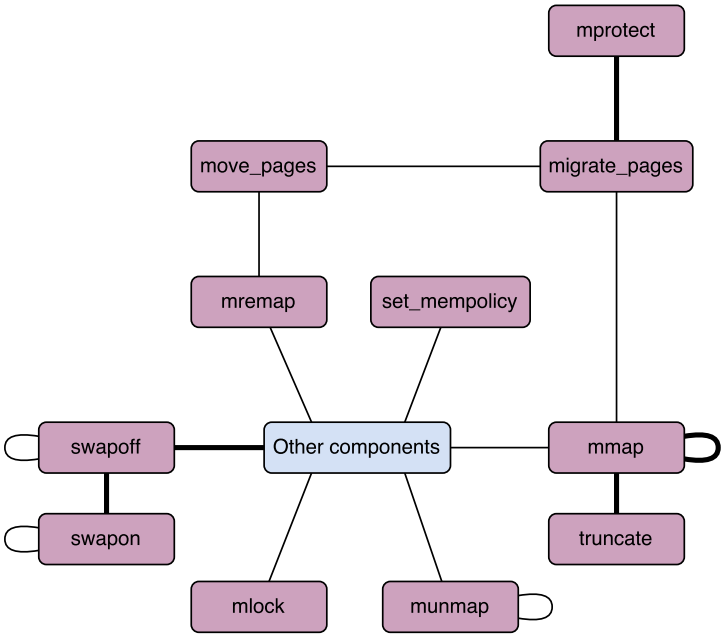

Fig. 10: All race conditions for memory management system calls (the purple nodes) that were fixed during the last decade. The thick lines indicate two fixed race conditions.

of signal handling system calls on the underlying system architecture. The majority of these bugs are related to the `signal()` system call.

Signal handling system calls have by far the highest number of semantic bugs per system call, which confirms the complexity of these system calls. A manual review shows that concepts such as the system call restart mechanism [5, 63], signal handling and the signal stack are challenging for developers to grasp. For example, when a system call is interrupted during its execution and queued, it needs to be restarted later. However, the system call should not always be completely restarted, which causes confusion with developers. There are 36 bug fixes that add support for handling exceptional cases in the system call restart mechanism. These bug fixes are often fixes for the same problem on different architectures.

Memory management system calls have the highest number (1.81) of concurrency-related bug fixes per system call. Most of these bug fixes address race conditions that occur between system calls. A race condition occurs when two system calls, or application code in kernel or user mode, are executed simultaneously and the output of the system is dependent on the execution order. As a result, race conditions can lead to non-deterministic behavior and should be avoided. To understand and prevent race conditions, developers must understand interactions between system calls and other parts of the system, which is difficult.

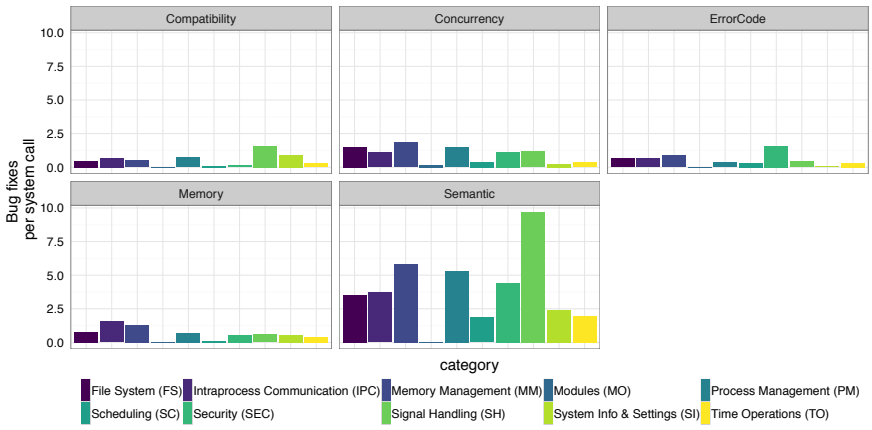

Fig. 11: The number of bug fixes per system call in each bug category over the last decade.

We extract all race conditions between memory management system calls that were fixed during the last decade and show them in Figure 10. Each node is a system call and a connection between two nodes indicates that a race condition between these system calls was fixed during the last decade. Figure 10 shows that the `mmap()` system call is the memory management system call with the most fixed race conditions during the last decade. The full graph of all fixed race conditions can be found online [3]. The graph of fixed race conditions is valuable for system developers, as it gives an overview of which interactions between a system call and the system may occur, and hence require extra testing.

> *Signal handling system calls have by far the highest number of semantic bug fixes per system call. Memory management system calls have the highest rate of concurrency-related bug fixes, and the highest bug fix entropy.*

8 Discussion

In this section, we discuss the implications of our findings. First, we discuss the generalizability of our findings through a study on the FreeBSD operating system. Then, we discuss the implications of our findings with regards to maintenance effort and the support of multiple architectures in an open source operating system. Finally, we give suggestions for opportunities for automation.

8.1 Generalizability of Our Findings

In this paper, we focused on the evolution of the Linux kernel API (i.e., the system calls). Studying the Linux kernel API is important because nowadays, many systems depend on this kernel API. For example, almost 1.5 million mobile devices run on the Android operating system [21], which is derived from Linux. To verify the generalizability of our findings, we performed a sanity check to ensure that the system call mechanism in other UNIX-based operating systems is similar, by studying the documentation[2] and source code[3] of the latest commit of the FreeBSD operating system.

There exist 447 system calls in FreeBSD. FreeBSD allows the activation of system calls through configurations, mainly to provide backwards compatibility by keeping old system calls activated. 386 system calls are activated by default in all configurations in BSD. Through a manual study, we found that 199 of the system calls on Linux and FreeBSD have the same signature. Approximately 164 system calls in FreeBSD have a different signature but provide very similar functionality as their Linux counterparts (either system calls or `glibc` methods).

The process of handling and maintaining system calls in FreeBSD and Linux is similar and includes managing system call tables and context switching. The needed steps to add a system call are similar. However, the process of managing the system call table and adding the system call number in the related header files is automated in FreeBSD, while it is manual in Linux.

The different types of sibling calls in FreeBSD are similar to those in Linux. For example, we observed the parameter extension sibling system calls such as `thr_kill()` and `thr_kill2()`, `pipe()` and `pipe2()`; working directory system calls such as `mkdirat()` and `mkdir()`; and batch operation system calls such as `mlock()` and `mlockll()`. There are no sibling calls of the backwards compatibility and real-time type in FreeBSD. However, we did observe other types of sibling system calls such as `rfork()` and `vfork()`. The existence of sibling calls in both FreeBSD and Linux suggests that UNIX-based operating systems deal with technical debt in a similar way.

Our findings on the FreeBSD operating system confirm that other UNIX-based operating systems use a system call mechanism that is similar to that of Linux. Therefore, we can safely assume that our findings can be generalized towards other UNIX-based operating systems.

8.2 Maintenance Effort

In Section 6 and 7, we studied the maintenance effort in terms of commits and bugs in system calls over the last decade. In this section, we compare our findings with similar studies on Linux [36, 62] and the Linux file system [43].

[2] `https://www.freebsd.org/`

[3] `https://github.com/freebsd/freebsd/commit/67f6441`

Lu et al. [43] classified 5,200 commits in 6 types of Linux file systems. While their and our classification resulted in mostly the same division of maintenance effort, we found a higher percentage of commits that were related to improvement of the studied system (i.e., 25% compared to 10% in Linux file systems). In addition, we found that 5% of the commits were done to add or remove system calls, which is naturally not necessary for file systems. Our findings show that more maintenance effort is spent on improving the reliability, performance and functionality of the existing system calls, thereby confirming the importance of the core component of Linux that the system calls form together.

Li et al. [36] and Tan et al. [62] automatically classified bugs in open source systems, including Linux. In addition, Lu et al. [43] classified bugs in the Linux file system. All three studies found that semantic bugs are by far the most common, which is confirmed by our study.

An interesting observation is that Li et al. [36] and Tan et al. [62] found a downward trend in memory bugs in open source systems, while we did not observe such a downward trend, as explained in Section 7. One possible explanation is that the available tools for detecting memory bugs, such as `valgrind`[4], are not capable of crossing the boundary between kernel and user mode, which is required to debug system calls. Another possible explanation is that it is difficult to systematically test the Linux kernel in all possible configurations, as many of these configurations can be tested only on specific combinations of hardware. Hence, our recommendations for future researchers are that existing memory bug detection tools (1) should be made compatible with testing system calls, and (2) should better support testing different kernel configurations.

Several implications can be derived based on our findings with regards to the effort for maintaining system calls.

Implication 1: Compared to regular software systems, kernel APIs require an additional type of maintenance that involves adding and removing system calls. In Section 7, we found that this additional type of maintenance is susceptible to bugs. However, as shown by FreeBSD, most of the maintenance required to add and remove system calls can be automated. Therefore, we recommend that the process of adding and removing system calls is automated in Linux as well.

Implication 2: 11% of the system-call related changes are made to the system call handler mechanism. We observed that 89% of the system-call related changes are made to maintain the actual system call, while 11% of the changes are related to the system call handler mechanism, including context switching, `vDSO` and `vsyscall` (two mechanisms two accelerate system call execution). Our finding implies that approximately 11% of the maintenance effort of a kernel API is assigned to the infrastructure for providing the API.

[4] `http://valgrind.org/`

8.3 Supporting Multiple System Architectures

Because an operating system's kernel operates close to the hardware, a large part of the system-call related changes are architecture-dependent. We classified all changes that were made to files in (a subfolder of) the `/arch` folder as architecture-dependent, and we found that 41% of the changes were architecture-dependent. We found that there are several implications for an open source operating system that supports multiple architectures.

Implication 3: There are likely to exist inconsistencies between supported features across different architectures. As we found in Section 5, system calls are usually not activated in all architectures at the same time. In addition, as we found in Section 7, different bugs may exist across architectures. An additional burden of supporting multiple architectures is that system calls that are architecture-dependent need to be reimplemented for each architecture, which results in a considerable amount of code duplication.

Implication 4: Architecture-dependent code may prevent a large number of developers from contributing to that code. In Section 5, we found that a small group of developers works on system calls, as compared to the Linux operating system in general. In addition, we observed that 653 developers contributed to the architecture-dependent code, while 1,002 developers contributed to the architecture-independent code. However, a smaller number of contributors does not necessarily mean weak support for an architecture. The combination of low cost of a `x86`-server that runs Linux, and its similarity to a Unix-server, have made the combination of the `x86` server and Linux attractive to industry. Intel recognized this attraction, and contributed a large part of the code for `x86` support in Linux, thereby increasing the popularity of their own `x86` architecture [22]. Hence, despite the lower number of developers that work on architecture-dependent code, support for a particular architecture can still be strong in an open source operating system if a small group of developers is active.

The above implications show that future research should focus on propagating changes that are made to architecture-dependent code across other architectures. In addition, future research should focus on methods for checking functional consistency across architectures, to assist with the prevention and detection of architecture-dependent bugs.

8.4 Suggestions for Applying Automation in the Linux Kernel API Evolution

During our study, we encountered several cases in the evolution of the Linux kernel API where automation would have been possible. In the remainder of this section, we give our suggestions on where to apply automation.

8.4.1 Automated Testing

System call developers make heavy use of fuzz testing, a technique that calls system calls at random and in parallel, with random arguments. Two fuzz testing tools for system calls are Trinity[5] and Syzkaller[6], which detected respectively 42 and 422 bugs over the last years. The input of such fuzz testing tools is a set of system calls that are called during the fuzz test. Currently, system call developers specify this set based on their knowledge of and experience with interaction (e.g., sharing resources) between system calls. Such knowledge is difficult to gain, despite the small growth of the system call API per day (Section 5.2). However, as we showed in Section 7, we can use historical information to identify interactions between system calls.

Suggestion 1: Our race graph [3] can be used to guide the fuzz testing process to identify more bugs. One of the deliverables of our study is the race graph in which historical race bugs are visualized. This race graph can be used to identify interactions between system calls, but also to identify patterns of race bugs, which can in turn be used to guide the fuzz testing process. Many of these patterns can be identified using the system call names or by reading their `man`-pages. We extracted four main patterns from the race graph:

1. **Pattern 1: reader-writer or writer-writer.** This pattern describes the case in which two system calls read from and write to the same resource. Often, file system and memory management system calls are susceptible to such bugs. System calls that interact following this pattern can often be identified by their name (e.g., `recvmsg() & sendmsg()` and `readv() & writev()`). Other examples of system calls that follow this pattern are `mmap() & truncate()`, `read() & truncate()` and `move_pages() & mmap()`.
2. **Pattern 2: admin-admin.** Some system calls are used for administrative tasks in Linux. These system calls may race with each other during their execution. For example, `open() & close()`, `ptrace() & signal()`, `swapon() & swapoff()`, `inotify_rm_watch() & inotify_add_watch()`, `mount() & umount()` and `umount() & close()`.
3. **Pattern 3: (reader or writer)-admin.** This pattern occurred between administrative system calls and other system calls that try to read from or write to the resource that is targeted by the administrative system calls (e.g, `mprotect() & migrate_pages()`, `inotify_watch() & umount()` and `close() & write()`).
4. **Pattern 4: self-race.** Multiple instances of the same system call may race together (e.g., `mmap(), swapon(), and swapoff()`). This pattern may happen for any writer or administrative system call that is executed several times simultaneously.

The above patterns can be used to select system calls for fuzz testing, thereby leading to better results and increasing the efficiency of the existing

[5] `https://codemonkey.org.uk/projects/trinity/`

[6] `https://github.com/google/syzkaller`

tools. In addition, the patterns can be used for regression testing of system calls.

Suggestion 2: Existing automated testing tools should be extended to support system call testing. There exist several automated testing tools, such as `valgrind`, which are currently not capable of crossing the boundary between kernel and user mode. These tools should be extended to support system call testing.

8.4.2 Automated Refactoring

We found in Section 6 that 35% of the system call-related changes were done to restructure/refactor code. In our manual review, we found that the majority of these changes were done to reduce duplication in the code by extracting duplicate code into a helper function, or by replacing architecture-dependent code with architecture-independent code.

Suggestion 3: Automated refactoring tools should be used when restructuring the Linux kernel API. Dig and Johnson [12, 13] already stressed the importance of automating refactoring more than 10 years ago. However, we did not find evidence that automated refactoring tools are systematically applied to refactor the Linux kernel API. Hence, our suggestion is to use existing tools, such as code clone detectors, to assist with the restructuring of the Linux kernel API.

9 Threats to Validity

One of the threats to the *external validity* of our findings is generalization. In this paper we study Linux system calls in depth. Linux is one of the oldest, most well-developed open source projects and studies of Linux have led to numerous interesting findings (e.g., [1, 23, 24, 30, 31, 40, 42, 43, 48, 50–52, 60, 61]). As explained in the previous section, we are certain that our findings apply to other Unix-based operating systems. However, it is possible that system calls in non-Unix-based operating systems have different characteristics than Linux system calls. More research is needed to make claims about the further generalizability of our findings.

A second external threat comes from the quality of the classified commit messages. We use the commit messages to classify commits and understand the drivers of developers for performing such changes. Hence, we rely on the quality of the description of the commit (i.e., the commit message).

The most important threat to the *internal validity* of our results is the manual classification process that we used to classify commits and bug fixes. To mitigate this threat, two of the authors independently conducted the classification. The low percentage of conflicting classifications (i.e., 5% of the commits and 2% of the bugs) suggests that the classification task was not overly susceptible to subjectivity. We made all our classifications available in an online appendix [3].

We found that it is extremely difficult to fully automate the commit extraction process. Hence, we applied a set of heuristics to semi-automate the extraction of commits that are related to system calls, in combination with a final manual analysis step. We do not claim that we studied all system call-related commits. However, our studied data set is still considerably large with 8,770 system call-related commits. Hence, our findings and conclusions are not affected by the fact that we may not have studied *all* system call-related commits.

10 Conclusion

The Linux kernel provides its services to the application layer using so-called system calls. All system calls combined form the Application Programming Interface (API) of the kernel. Hence, system calls provide us with a window into the development process and design decisions that are made for the Linux kernel.

We conducted an empirical study of the changes that were made to the system calls during the last decade (i.e., from April 2005 to December 2014). The most important findings of our study are:

1. There is a considerable amount of technical debt in the kernel, that is addressed by an increase in the number of sibling system calls. Guidelines were introduced in 2015 to avoid technical debt of this type in the future by using more flexible function arguments for system calls.
2. There exists a small group of very active system call developers and the growth of the system calls is slow (i.e., 25 LOC per day).
3. The `ptrace()` and signal handling system calls are by far the most difficult ones to maintain and fix.

Our findings can be used by kernel API developers to learn about the challenges and problems that come with the long term maintenance of an kernel API, such as the long-lived Linux kernel API. In particular, we make two important suggestions in our paper. First, we suggest that historical information about the evolution of a kernel API should be used to guide the testing process. Second, we suggest that existing automated testing tools are extended, so that they can be used for testing system calls.

References

1. Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. Analyzing cloning evolution in the Linux kernel. *Information and Software Technology*, 44(13):755–765, 2002.
2. Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. Posix abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh*

European Conference on Computer Systems (EuroSys), pages 19:1–19:17. ACM, 2016.

3. Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E. Hassan, Juergen Dingel, and James R. Cordy. Analyzing a Decade of Linux System Calls: Online Appendix. `http://sailhome.cs.queensu.ca/replication/systemcalls_evolution/`, 2017. (Last visited: Apr 6, 2017).
4. Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM, 2016.
5. Jonathan Corbet. A new system call restart mechanism. `https://lwn.net/Articles/17744/`, 2002. (Last visited: Apr 6, 2017).
6. Jonathan Corbet. The possible demise of remap_file_pages(). `https://lwn.net/Articles/597632/`, 2014. (Last visited: Apr 6, 2017).
7. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference, A-Z. `https://goo.gl/1cFOvB`, 2016. (Last visited: Apr 6, 2017).
8. Matt Davis. Creating a vDSO: the Colonel's Other Chicken. `http://www.linuxjournal.com/content/creating-vdso-colonels-other-chicken`, 2012. (Last visited: Apr 6, 2017).
9. Arnaldo Carvalho de Melo. net: Introduce recvmmsg socket syscall. `https://github.com/torvalds/linux/commit/a2e2725541`, 2009. (Last visited: Apr 6, 2017).
10. Helge Deller. correctly wire up mq functions for compat case. `https://github.com/torvalds/linux/commit/fee707b45`, 2013. (Last visited: Apr 6, 2017).
11. Helge Deller. fix epoll pwait syscall on compat kernel. `https://github.com/torvalds/linux/commit/ab3e55b11`, 2014. (Last visited: Apr 6, 2017).
12. D. Dig and R. Johnson. The role of refactorings in API evolution. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pages 389–398. IEEE, Sept 2005.
13. Danny Dig and Ralph Johnson. How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.
14. Chase Douglas. Add compat truncate. `https://github.com/torvalds/linux/commit/dd90bbd5f`, 2009. (Last visited: Apr 6, 2017).
15. Ulrich Drepper. *at syscalls: Intro. `http://lwn.net/Articles/164584/`, 2005. (Last visited: Apr 6, 2017).
16. Ulrich Drepper. [PATCH] Implement AT_SYMLINK_FOLLOW flag for linkat. `https://github.com/torvalds/linux/commit/45c9b11a1`, 2006. (Last visited: Apr 6, 2017).

17. David Drysdale. Documentation: describe how to add a system call . https://github.com/torvalds/linux/commit/4983953d, 2015. (Last visited: Apr 6, 2017).
18. Max Filippov. xtensa: deprecate fast_xtensa and fast_spill_registers syscalls. https://github.com/torvalds/linux/commit/9184289, 2014. (Last visited: Apr 6, 2017).
19. John Fox and Sanford Weisberg. Nonparametric regression in R. https://socserv.socsci.mcmaster.ca/jfox/Books/Companion/appendix/Appendix-Nonparametric-Regression.pdf, 2010. (Last visited: Apr 6, 2017).
20. Free Software Foundation. The GNU C library. https://www.gnu.org/software/libc/, 2016. (Last visited: Apr 6, 2017).
21. Gartner. Gartner says tablet sales continue to be slow in 2015. http://www.gartner.com/newsroom/id/2954317, 2015. (Last visited: Apr 6, 2017).
22. Al Gillen and Jean S. Bozman. Running mission-critical workloads on enterprise linux x86 servers. *IDC Whitepaper*, 2013.
23. Michael W Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 131–142. IEEE, 2000.
24. Michael W. Godfrey and Qiang Tu. Growth, evolution, and structural change in open source software. In *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE)*, pages 103–106. ACM, 2001.
25. Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 78–88. IEEE, 2009.
26. Mike Hayward. LKML: Mike Hayward: Intel P6 vs P7 system call performance. https://lkml.org/lkml/2002/12/9/13, 2002. (Last visited: Apr 6, 2017).
27. Johannes Henkel and Amer Diwan. Catchup!: Capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 274–283, New York, NY, USA, 2005. ACM.
28. A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente. How do developers react to API evolution? the Pharo ecosystem case. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 251–260. IEEE, Sept 2015.
29. Andrew Hunt. *The pragmatic programmer*. Pearson Education India, 2000.
30. Ayelet Israeli and Dror G Feitelson. The Linux kernel as a case study in software evolution. *Journal of Systems and Software*, 83(3):485–501, 2010.
31. Clemente Izurieta and James Bieman. The evolution of FreeBSD and Linux. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering (ESEM)*, pages 204–211. ACM, 2006.

32. Michael Kerrisk. Linux programmer's manual: Linux system calls. http://www.man7.org/linux/man-pages/man2/syscalls.2.html, 2015. (Last visited: Apr 6, 2017).
33. Michael Kerrisk. Linux programmer's manual: nfsservctl. http://man7.org/linux/man-pages/man2/nfsservctl.2.html, 2015. (Last visited: Apr 6, 2017).
34. Michael Kerrisk. Linux programmer's manual: writev. http://www.man7.org/linux/man-pages/man2/writev.2.html, 2017. (Last visited: Apr 6, 2017).
35. M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept 1980.
36. Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 25–33. ACM, 2006.
37. Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. API change and fault proneness: A threat to the success of Android apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC-FSE)*, pages 477–487, New York, NY, USA, 2013. ACM.
38. Linux Kernel Documentation. Adding a new system call. https://www.kernel.org/doc/html/latest/process/adding-syscalls.html, 2005. (Last visited: Apr 6, 2017).
39. Linux Kernel Documentation. Submitting patches: the essential guide to getting your code into the kernel. https://www.kernel.org/doc/Documentation/process/submitting-patches.rst, 2016. (Last visited: Apr 6, 2017).
40. Simone Livieri, Yoshiki Higo, Makoto Matsushita, and Katsuro Inoue. Analysis of the Linux kernel evolution using code clone coverage. In *4th International Workshop on Mining Software Repositories (MSR)*, pages 22–22. IEEE, 2007.
41. Jeffrey D Long, Du Feng, and Norman Cliff. Ordinal analysis of behavioral data. *Handbook of psychology*, 2003.
42. Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. Evolution of the Linux kernel variability model. In *International Conference on Software Product Lines*, pages 136–150. Springer, 2010.
43. Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. *Transactions on Storage (TOS)*, 10(1):3:1–3:32, January 2014. ISSN 1553-3077.
44. Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.*, 42(2):329–339, 2008.
45. Wolfgang Mauerer. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

46. T. McDonnell, B. Ray, and M. Kim. An empirical study of API stability and adoption in the android ecosystem. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 70–79, Sept 2013.
47. Kyle McMartin. Reorder syscalls to match. `https://github.com/torvalds/linux/commit/1e67685b1`, 2007. (Last visited: Apr 6, 2017).
48. Ettore Merlo, Michel Dagenais, P Bachand, JS Sormani, Sara Gradara, and Giuliano Antoniol. Investigating large software system evolution: the Linux kernel. In *Proceedings of the 26th International Computer Software and Applications Conference (COMPSAC)*, pages 421–426. IEEE, 2002.
49. National Institute of Standards and Technology. National Vulnerability Database: CVE-2009-0029. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-0029`, 2009. (Last visited: Apr 6, 2017).
50. Yoann Padioleau, Julia L Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 59–71. ACM, 2006.
51. Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: ten years later. In *ACM SIGPLAN Notices*, volume 46, pages 305–318. ACM, 2011.
52. Leonardo Passos, Krzysztof Czarnecki, and Andrzej Wąsowski. Towards a catalog of variability evolution patterns: the Linux kernel case. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, pages 62–69. ACM, 2012.
53. Jeff H. Perkins. Automatically generating refactorings to support api evolution. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 111–114, New York, NY, USA, 2005. ACM.
54. Pingdom. Linux kernel development by the numbers. `http://royal.pingdom.com/2012/04/16/linux-kernel-development-numbers/`, 2012. (Last visited: Apr 6, 2017).
55. Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to API deprecation?: The case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 56:1–56:11. ACM, 2012.
56. Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices. In *Annual Meeting of the Southern Association for Institutional Research*, 2006.
57. Sandeep S. Process tracing using ptrace. `http://www.tldp.org/LDP/LGNET/81/sandeep.html`, 2002. (Last visited: Apr 6, 2017).
58. Haavard Skinnemoen. fix sys sync file range call convention. `https://github.com/torvalds/linux/commit/73d4393d1`, 2008. (Last visited: Apr 6, 2017).
59. Haavard Skinnemoen. Fix timerfd breakage on avr32. `https://github.com/torvalds/linux/commit/46a56c5a0`, 2008. (Last visited: Apr 6,

2017).
60. Diomidis Spinellis. A repository with 44 years of unix evolution. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*, pages 462–465. IEEE Press, 2015.
61. Diomidis Spinellis, Panos Louridas, and Maria Kechagia. The evolution of c programming practices: A study of the unix operating system 1973–2015. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 748–759. ACM, 2016.
62. Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.
63. Linus Torvalds. compatibility syscall layer (lets try again). `https://lwn.net/Articles/17746/`, 2002. (Last visited: Apr 6, 2017).
64. Linus Torvalds. renameat2 does not need (or have) a separate compat system. `https://github.com/torvalds/linux/commit/9abd09acd`, 2014. (Last visited: Apr 6, 2017).
65. Linus Torvalds. Linux git repository. `https://github.com/torvalds/linux/`, 2016. (Last visited: Apr 6, 2017).
66. Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern Linux API usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, pages 16:1–16:16. ACM, 2016.
67. Al Viro. [braindump][rfc] signals and syscall restarts. `https://lkml.org/lkml/2012/12/6/366`, 2012. (Last visited: Apr 6, 2017).
68. L. Xavier, A. Brito, A. Hora, and M. T. Valente. Historical and impact analysis of API breaking changes: A large-scale study. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147. IEEE, Feb 2017.
69. Z. Xing and E. Stroulia. API-evolution support with diff-catchup. *IEEE Transactions on Software Engineering (TSE)*, 33(12):818–836, Dec 2007.
70. Chris Zankel. [PATCH] xtensa: Architecture support for Tensilica Xtensa Part 1. `https://github.com/torvalds/linux/commit/8e1a6dd2`, 2005. (Last visited: Apr 6, 2017).