Evolution of the Unix System Architecture

An Exploratory Case Study

Spinellis, Diomidis; Avgeriou, Paris

DOI
10.1109/TSE.2019.2892149

Publication date
2021

Document Version
Final published version

Published in
IEEE Transactions on Software Engineering

Citation (APA)
Spinellis, D., & Avgeriou, P. (2021). Evolution of the Unix System Architecture: An Exploratory Case Study. *IEEE Transactions on Software Engineering*, *47*(6), 1134-1163. Article 8704965. https://doi.org/10.1109/TSE.2019.2892149

Evolution of the Unix System Architecture: An Exploratory Case Study

Diomidis Spinellis, *Senior Member, IEEE* and Paris Avgeriou, *Senior Member, IEEE*

Abstract—Unix has evolved for almost five decades, shaping modern operating systems, key software technologies, and development practices. Studying the evolution of this remarkable system from an architectural perspective can provide insights on how to manage the growth of large, complex, and long-lived software systems. Along main Unix releases leading to the FreeBSD lineage we examine core architectural design decisions, the number of features, and code complexity, based on the analysis of source code, reference documentation, and related publications. We report that the growth in size has been uniform, with some notable outliers, while cyclomatic complexity has been religiously safeguarded. A large number of Unix-defining design decisions were implemented right from the very early beginning, with most of them still playing a major role. Unix continues to evolve from an architectural perspective, but the rate of architectural innovation has slowed down over the system's lifetime. Architectural technical debt has accrued in the forms of functionality duplication and unused facilities, but in terms of cyclomatic complexity it is systematically being paid back through what appears to be a self-correcting process. Some unsung architectural forces that shaped Unix are the emphasis on conventions over rigid enforcement, the drive for portability, a sophisticated ecosystem of other operating systems and development organizations, and the emergence of a federated architecture, often through the adoption of third-party subsystems. These findings have led us to form an initial theory on the architecture evolution of large, complex operating system software.

Index Terms—Unix, software architecture, software evolution, architecture design decisions, operating systems

1 Introduction

UNIX[1] has a long and celebrated history. Its evolution spans five decades and is a result of the work by thousands of developers, including several distinguished pioneers. As an operating system, it has left an undeniable mark on the history of computing, while it has influenced tremendously the current state of the art in software, network, and hardware engineering.

Studying the evolution of operating system software is not just significant from a historical perspective; it can provide valuable insights into evolvability best practices and anti-patterns, for large, complex, and long-lived systems. Unix is a unique case among all operating systems, both due to its longevity, and its impact on the operating systems that followed. The evolution of a system of this size, complexity, and age can shed light on how similar systems can sustainably grow without the perils of software aging, such as soaring technical debt or uncontrolled architectural decay.

In this paper we study the evolution of Unix along the FreeBSD lineage from a software architecture perspective. While there have been studies on how Unix evolved (see Section 2), these have mostly focused at the source code level and were limited to the kernel. On the contrary, we turn our attention to the system architecture and study a) the core architectural design decisions across the main releases, and b) the evolution in the number of the system's features (obtained from the Unix reference documentation) and in the code's complexity. The former entails qualitative analysis, while the latter quantitative. These analyses subsequently lead to forming an initial theory on the architecture evolution of large and complex operating systems, regarding their form, pace, driving forces, as well as the accumulation of architectural technical debt.

The rest of the paper is structured as follows: In Section 2 we present related work, whereas in Section 3 we elaborate on the case study design. In Sections 4 and 5 we present the qualitative results (main architectural design decisions), and the quantitative results (evolution of size and complexity) respectively. Next, in Section 6 we discuss the main findings, and in Section 7 the threats to this study's validity. Finally, in Section 8 we conclude the paper with a summary and discussion of our findings.

2 Related Work

The work reported here covers mainly two areas: a) software evolution in general, which has been intensely studied, and b) the evolution of Unix in particular, where related work is more thin on the ground.

1. UNIX® is a registered trademark of The Open Group. For the sake of simplicity, in this paper we use the word "Unix" to refer both to UNIX systems developed at Bell Labs and to Unix-like systems, such as FreeBSD, that descended from them.

- *D. Spinellis is with the Athens University of Economics and Business, Athina 104 34, Greece. E-mail: dds@aueb.gr.*
- *P. Avgeriou is with the University of Groningen, Groningen 9712, Netherlands. E-mail: paris@cs.rug.nl.*

Manuscript received 19 May 2018; revised 18 Dec. 2018; accepted 28 Dec. 2018. Date of publication 2 May 2019; date of current version 14 June 2021.
(Corresponding author: Diomidis Spinellis.)
Recommended for acceptance by R. Mirandola.
Digital Object Identifier no. 10.1109/TSE.2019.2892149

2.1 Software Evolution

There have been several studies on the longitudinal evolution of large systems. The seminal work of Lehman [1] and its subsequent refinements attempted to establish laws of software evolution, not unlike those of biological evolution. Those laws have been the subject of much discussion and research work [2]: their validity has been long debated, their nature and scope have been iteratively refined by many researchers, while several studies have examined whether the laws hold for particular cases. The phenomenon of software evolution has also been studied under different terms, such as Software Aging [3], Software Decay [4], and more recently Technical Debt [5].

One of the most popular ways to study software evolution focuses on the growth of the source code. Hatton et al. conducted the largest study to date on software growth rate; specifically they studied the growth rate of over 404 million lines of both open source and proprietary software and concluded that code doubles about every 42 months [6]. Similarly, a large study on 6000 open source systems by Koch [7] revealed that while the mean growth is linear, there is a significant percentage of systems with super-linear growth.

Several papers examine the evolution of open source software from diverse angles [8]. Many take a quantitative approach, using statistics to determine relationships between various attributes, such as modularity and complexity [9], growth and change rate [10], complexity and cumulative change [11], or even the contributions and collaborations of the user community through social network analysis [12]. Some papers examine evolution of systems written in C in terms of modularity and complexity, and are thus directly relevant to this work. An early study of the Linux kernel growth by Godfrey and Tu [13] argued that the kernel's super-linear growth rate could be attributed to the linear growth of several subsystems; this is related to our finding (Section 6.3) that the accumulation of large subsystems plays an important role in the modern evolution of Unix. A subsequent study on the same topic [14] also looked at the issue of code complexity and found that "the average complexity per function, and the distribution of complexities of the different functions, are improving with time." Roughly similar trends, along with what appears to be a self-correcting process, are shown in a study of Unix programming practices [15].

There has also been a significant number of studies on the evolution of operating systems, particularly Linux. MacCormack et al. [16] studied Linux in terms of its structure and compared it with the first and an evolved version of Mozilla; the results emphasize the modularity of Linux and how Mozilla evolved from a less to a more modular structure (compared to Linux) in a matter of years. In addition to the aforementioned Linux study by Godfrey and Tu [13], which mostly measured lines of code of the operating system and its major subsystems, a subsequent study of the Linux kernel, conducted by Israeli and Feitelson [17], aimed at characterizing the operating system according to Lehman's laws of evolution. They used a number of quantitative metrics in addition to lines of code, such as number of system calls and cyclomatic complexity [18]. They were able to confirm several of Lehman's laws, while one of the interesting findings is that complexity decreases over time. In a follow-up study, Feitelson studied the Linux kernel evolution lifecycle [19], summarizing it as a linear piece-wise model with increasing slopes.

In addition to studying software evolution at the level of source code, a number of studies have focused on the architecture level. Behnamghader et al. [20] proposed a method for architecture recovery and subsequently used this method to study 23 open source systems examining the architectural changes during long periods of system evolution. Other approaches have also looked at architecture evolution, but using source code artifacts, such as classes and packages, as first-class entities. For example, D'Ambros et al. [21] proposed architectural metrics derived from source code analysis, and subsequently visualized those metrics to illustrate different aspects of the evolution of both the code and the architecture. Similarly, Wettel and Lanza [22] focused on the visualization of 'coarse-grained' characteristics of software evolution (packages and classes) as well as 'fine-grained' ones (methods). A final example is the work of Bouwers et al. [23], who proposed an architecture metric for architecture partitioning into components based on the evolution of numerous open source and proprietary systems.

Compared to the discussed related work, our work has the following differences: a) we focus on Unix; b) we analyze the architecture evolution not at the component level but at the level of architecture decisions, the seven key Unix feature types (user commands, system calls, libraries etc.), as well as the form and pace of architecture evolution, architectural technical debt, and notable architectural characteristics; c) we use data sources that span 48 years and 30 system releases.

2.2 Work on the Design and Evolution of Unix

The importance of Unix and its pedigree, rooted first in industrial (AT&T Bell Labs) and then in academic (University of California at Berkeley) research, has endowed it with numerous publications that detail the system's structure and evolution. These cover snapshots, subsystems, or specific periods.

Bell Labs staff published tens of papers on Unix and its applications as technical reports [24], [25], [26], [27], [28], [29], [30], [31], [32], [33]. Most of these were also distributed with each Unix release as *Volume 2—Supplementary Documents*, of the accompanying *Unix Programmer's Manual*. Two issues of *The Bell System Technical Journal*, which appeared in 1978 and 1984, were entirely devoted to Unix; these were later also published in book form [34], [35]. Bell Labs staff also published in outlets covering more diverse topics [36], [37], [38], [39], [40], [41]. This tradition of open publication was continued by staff and alumni of Berkeley's Computer Science Research Group (CSRG), as well as other systems researchers and developers [42], [43], [44], [45], [46], [47], [48], [49], [50]. These papers and many others provide rich insights regarding the functionality and evolution of specific facilities as well as the whole system.

Of particular importance to this study are: the CACM paper introducing the features, ideas, and design of Unix [36]; Ritchie's retrospective, detailing the system's strengths and weaknesses [51]; Thompson's overview of the implementation of Unix [52]; Rosler's paper on the evolution of C [53]; the study of portability as a design-shaping force [54]; and a subsequent report by Ritchie on the evolution of Unix, focusing on the filesystem, process control, I/O redirection,

and high-level languages [55]. More recent articles have covered the restoration and curation of historical artefacts, such as early editions of Unix [56], [57] and repositories of them [58], or their subsequent study [15], [59].

Another category of material related to this study is books detailing the internal workings of Unix and thereby also parts of its architecture. The thing that started it all is a slim two volume set prepared in 1977 by John Lions as teaching material for his operating systems course at the University of New South Wales. The first volume contains a line-by-line listing of the Sixth Edition Unix kernel, while the second volume is a source code commentary explaining the functionality of each listed element. Confusion regarding the associated intellectual property rights resulted in it circulating for two decades in *samizdat* photocopies or digital scans, before legal hurdles were lifted to allow its formal publication [60].

A decade later, Maurice Bach published a book covering in abstract terms, without reference to specific source code elements, the design of the Unix kernel, with an emphasis on System V Release 2 [61]. The book, based on material the author prepared for a course he taught at AT&T Bell Laboratories, covers most important data structures and algorithms. Meantime, on the West Coast, researchers who had worked on the Berkeley versions of Unix, published another book detailing the design of BSD Unix [62]. This work was expanded and updated at regular intervals to cover new editions of BSD Unix [63] and then its FreeBSD descendant [64], [65].

In this area we also mention Organick's high-level architecture analysis of the MULTICS operating system [66]—a system much larger and considerably more ambitious than several early versions of Unix. This is relevant, because AT&T Bell Labs was developing the system together with MIT and General Electric. When AT&T pulled out from the development of MULTICS, the Bell Labs team was left without a system on which to experiment with operating system design and, also, with valuable lessons learned from the MULTICS project.

This paper is not directly comparable to the work summarized here, but it builds on it (see Section 3.3) and on empirical data to study the evolution of Unix over a half-century period.

3 Case Study Design

The case study as an empirical method is used for investigating a phenomenon in its real life context [67]. The main reason for selecting to perform a case study rather than other types of empirical studies, is that we want an in-depth understanding of how and why architecture evolution phenomena occurred within the Unix ecosystem. This case study has been designed and is presented according to the guidelines of Runeson et al. [67].

3.1 Objectives and Research Questions

The goal of this study, stated here using the Goal-Question-Metric (GQM) approach [68], is to "*analyze* the Unix operating system *for the purpose of* evaluation and characterization of its architecture evolution *with respect to* its main architecture design decisions, size and complexity *from the point of view of* software developers *in the context of* the Unix ecosystem". The aforementioned goal can be achieved by answering the following research questions.

- RQ1 What are the main architectural design decisions along the major releases of the system?
- RQ2 How did complexity and the number of features evolve along the main releases of the system?

The first question aims at investigating the architecture's evolution from a *qualitative* perspective. An architecture is the set of main design decisions [69], [70]. Therefore, we study architecture evolution by identifying the major design decisions that were introduced along a number of the most significant releases (see Section 4). Such design decisions are mainly: (a) architecture components, including their interfaces, such as the kernel, shells, and libraries; (b) architecture connectors such as pipes and C header files; (c) architecture patterns [71] that were applied in the system, such as layering and reflection; and (d) the principles that guide the system architecture, such as modularity and separation of concerns. Architecture components and connectors, patterns, and principles constitute some of the key architecture decisions of software systems [69], [72]. We also report other types of decisions that cannot be classified in these categories, e.g., naming conventions. Every design decision is accompanied by a rationale, which is the most important section in decision documentation [70].

The second question looks also at the evolution of the Unix architecture, but from a *quantitative* point of view. Specifically we look at how metrics of size and complexity evolve over time; these metrics concern system features (e.g., number of user commands or system calls), as determined by the Unix reference documentation (for more details see Section 5.1). This gives us a complementary perspective to the qualitative results, as we can discern overall trends across decades rather than notable architecture changes in individual releases. Eventually, we combine the quantitative and qualitative results during our discussion (see Section 6) in order to derive findings and conclusions.

We note that in such quantitative analyses, it is common to also measure cohesion and coupling. However, in the case of Unix, this would require substantially more manual work for each revision of Unix. Namely, it would entail: a) the development of custom tools to analyze PDP-7 and PDP-11 assembly as well as early dialects of C; b) the configuration of analysis tools for the file layout and linking policies of each revision. Therefore, this is considered as out of scope for this work, but it does constitute appealing future work.

The answers to both research questions are interesting beyond the case of Unix. Thus, they will be used as raw data to form an initial theory on the architecture of large and complex operating systems (see Section 6).

3.2 Case Selection and Units of Analysis

The case study of this paper is characterized as single-case and embedded [67]: the Unix operating system is the case, while the different versions are the units of analysis. Our study starts with the unnamed 1970 PDP-7 version that became Unix, followed by the so-called "Research" editions that came out of Bell Labs, then continues with the Berkeley Software Distributions (BSD), and finishes with versions of the FreeBSD operating system distribution that carries on its development until today (see Fig. 7 in Section 6). We could not study Unix

TABLE 1
Units of Analysis: Key Releases, Dates and Size Metrics

Release	Date			Lines of Code		
				Kernel	Library	Programs
Research PDP7			1970	2,489	0	9,095
Research V1	3	Nov	1971	4,768	?	?
Research V2	12	Jun	1972	?	1,075	16,968
Research V3	15	Feb	1973	?	?	?
Research V4	30	Nov	1973	7,141	?	?
Research V5		Jun	1974	8,778	5,634	53,428
Research V6		May	1975	12,347	7,092	137,723
Research V7		Jan	1979	19,710	14,251	290,142
Bell 32V	28	Aug	1979	16,572	14,224	297,688
BSD 3	22	Mar	1980	25,096	4,637	545,942
BSD 4	16	Nov	1980	35,616	20,522	674,912
BSD 4.1c/2	2	Apr	1983	85,312	32,817	1,003,134
BSD 4.2	1	Jan	1985	91,309	31,296	1,265,337
BSD 4.3	4	Mar	1987	127,725	40,740	2,402,062
BSD 4.3/Tahoe	6	Jan	1990	218,783	150,883	2,552,789
BSD 4.3/Reno	2	Jan	1991	357,466	125,267	2,894,582
BSD 4.3/Net 2	20	Aug	1991	295,677	265,316	2,405,218
386BSD 0.0	4	Mar	1992	92,565	176,680	777,114
386BSD 0.1	15	Jul	1992	129,884	176,387	2,604,563
386BSD p/k	20	Jun	1993	210,828	176,810	2,894,027
FreeBSD 1.0	28	Oct	1993	233,262	143,091	2,218,098
FreeBSD 2.0	22	Nov	1994	381,206	262,920	2,932,865
BSD 4.4	25	Jul	1995	730,422	246,184	6,362,709
BSD 4.4/Lite2	25	Jul	1995	648,069	250,281	5,348,342
FreeBSD 3.0.0	21	Jan	1999	957,625	395,846	5,024,207
FreeBSD 4.0.0	20	Mar	2000	1,371,122	450,225	6,442,184
FreeBSD 5.0.0	16	Jan	2003	2,180,639	613,034	7,919,274
FreeBSD 6.0.0	3	Nov	2005	2,796,311	567,130	9,183,131
FreeBSD 7.0.0	24	Feb	2008	3,561,595	632,643	10,238,166
FreeBSD 8.0.0	20	Nov	2009	4,099,266	746,689	10,747,631
FreeBSD 9.0.0	2	Jan	2012	5,371,628	761,459	15,135,803
FreeBSD 10.0.0	16	Jan	2014	6,599,640	699,317	17,780,699
FreeBSD 11.0.0	22	Sep	2016	8,518,968	733,620	21,529,326

versions that derive from the Research editions via AT&T System V, such as Solaris, AIX, and HPUX, because most of the corresponding code remains proprietary and inaccessible. We chose not to study the evolution of Research editions into *Plan 9* [41], due to the system's limited adoption and lack of packaged release distributions. Other systems deriving from the BSD source code base are NetBSD, which focuses on widespread architecture portability, especially among embedded devices, and OpenBSD, which focuses on security. Although these projects differ in terms of vision and technologies, all frequently exchange among them code and ideas. This paper examines the architectural evolution in the popular FreeBSD line, to capitalize on the first author's inside knowledge of FreeBSD and on the system's excellent published design documentation [64], [65].

An overview of the major releases that comprise the units of analysis in this study appears in Table 1.[L1,2] For the Research editions the release date is derived from the corresponding manual date; in the remaining cases from the timestamp of the newest file. Cases where the code associated with specific releases has not been preserved are marked with a question mark. Details about that release were obtained by studying its manual, which, thankfully, is available for all versions of Unix. Further quantitative data for each one of these versions appear in Fig. 3 in Section 5.1.

3.3 Data Collection

In order to answer the research questions, we collected both qualitative and quantitative data. More specifically, for both RQ1 and RQ2 we used two data collection techniques [73]: documentation analysis on a number of documents (qualitative data) as well as static analysis of the source code (quantitative data). For the latter we examined the source code for each of the Unix releases, obtained from the Unix history repository [58].[3] For the former we used the following documents.

- The documentation (Unix Reference Manual pages) associated with each release [74]. In the cases where this was not available it was reconstructed from the source code markup.[4]
- Books and research papers described in Section 2.2.
- Recollections of Unix pioneers [75], [76], [77], [78], [79], [80], [81].

The use of multiple data sources, allowed us to perform data source triangulation, i.e., we were able to confirm the findings from different types of data sources. More details are given on Section 7.

A large part of our study is based on a data set of the Unix reference documentation and its visualization in the form of timelines [74]. This documentation is available from the First Research Edition onwards in what is known as "Volume I" of the *Unix Programmer's Manual* [82]. Note that Volume II [83] contains supplementary documents, which provide an in-depth treatment of specific tools and topics, such as the shell [84], the C programming language [85], the *lint* program checker [86], the *tbl* table formatter [87], and so on. Fortunately, the Unix documentation is maintained in electronic format (as *troff* [33] files) together with the system's source code. For releases where the source code has been lost, (denoted by a question mark in Table 1) scanned copies of the manual are still available.

To answer RQ1, a data set of all the system's architectural design decisions for every available release was created, based on the documentation. The data format and their collection process are described in reference [74]. The corresponding data and generation scripts are available online.[5]

To answer RQ2 we collected data primarily through source code analysis (to measure complexity) and document analysis on the Unix reference manuals (to measure feature set size).

3.4 Data Analysis

Quantitative data are analyzed through simple descriptive statistics and illustrated through histograms and scatter plots. To calculate the cyclomatic complexity [18] at the component level, we looked at the mean value over all functions comprising the corresponding component. This follows recently published results indicating that the mean and median rather than the sum are better defect predictors [88].

2. Footnotes prefixed by I (IN) document the derivation of numbers and tables through correspondingly numbered listings appearing in the supplementary online material, which can be found alongside this paper at https://doi.org/10.1109/TSE.2019.2892149.

3. DOI: 10.5281/zenodo.2525587
4. DOI: 10.5281/zenodo.2525571 DOI: 10.5281/zenodo.2525574
5. DOI: 10.5281/zenodo.2525613 DOI: 10.5281/zenodo.2525612

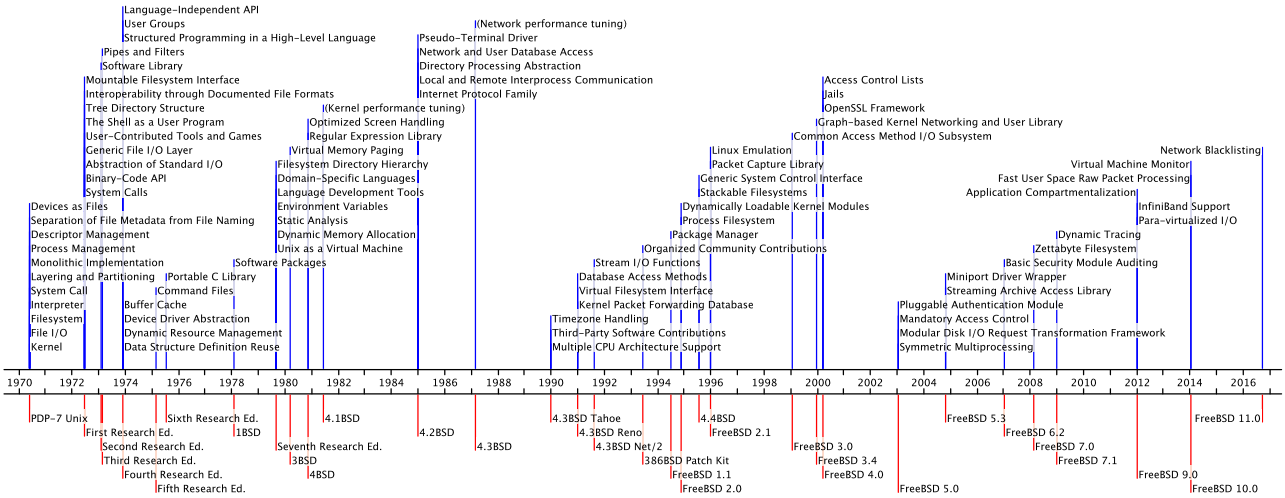

Fig. 1. Timeline of Unix's major releases and architectural design decisions.

In order to analyze qualitative data we have performed coding using Constant Comparison [89]. Specifically we performed Constant Comparison iteratively, refining the codes with their relationships in each iteration. The codes correspond to the architecture design decisions that were deemed worth reporting per major release of the system; in that sense the codes were not pre-formed but post-formed (i.e., they were created during the coding process). As is common in Constant Comparison (see reference [89]), we focused on unifying explanations for the various studied decisions, in particular why those decisions were made and how.

In the following sections we attempt to answer the research questions: Section 4 focuses on RQ1 by showing the evolution of the architecture design decisions in the major Unix releases; Section 5 tackles RQ2 by presenting the evolution of Unix size and complexity.

4 Qualitative Results

To answer the first research question, we examine the main architectural design decisions in major releases of the system (see Table 1). Each sub-section first introduces the Unix release and subsequently provides a short discussion of the principal design decisions for that release, such as components (e.g., commands, routines etc.), connectors (e.g., system calls, sockets etc.), patterns (e.g., Layers, Pipes and Filters, Reflection etc.), and principles (e.g., modularity, virtualization, low coupling etc.).

We used the detailed interactive timelines described in reference [74] to note when each feature appeared and when features disappeared. Hyperlinks from the timelines to the documentation allowed us to assess the type and importance of each new feature. Fig. 1 summarizes the nine online timeline diagrams into a single timeline of the principal design decisions of Unix.

In the following text, when documentation regarding a particular architectural design decision appeared in a given version of Unix then a reference is made to the corresponding "manual page" using the conventional *name*(SECTION) format. for example *ls*(I) refers to the *ls* command in Section I of the Unix reference manual.[6] In other cases, our text may refer to Unix source code, using a footnote such as this.[S1] This can be used to find and access the associated release, file, and line through a correspondingly numbered note provided in the supplementary online material.

4.1 PDP-7 Unix

Unix was originally written (as an unnamed system) in PDP-7 assembly language. A recently found and restored artifact from mid-1970s [59], allows us to examine its structure and techniques employed in its construction. The following design decisions stand out. Many of these survive until today.

Kernel Despite the system's diminutive size of 13,691[L2] lines, there is a clear separation between an operating system kernel that offers a few tens of services and user-level commands. The kernel loads and executes user-level commands, provides the file abstraction, virtualizes the hardware interfaces, and establishes ownership of files.

Layering and Partitioning The system is structured into two layers: the kernel and the commands. Following the Layers pattern [71, p. 33], the commands call the kernel, but the kernel does not depend on the commands. Furthermore, the commands adhere to the principle of low coupling: the code of each command is not coupled to code in other commands. This partitioning is established through a file naming convention: file names starting with the same sequence (e.g., *ed* for *ed1.s* and *ed2.s*) belong to the same partition.

System Call The transfer of control between the user programs and the kernel is implemented through special connectors: system calls. The kernel source code files define 35[L3] labels whose name starts with a period. These are the names of system call entry points. A subset of 28[L4] labels are grouped in a table,[S2] which allows them to be called from user programs using the `sys` instruction (some labels, such as those for low-level disk access, are not exported.) In the Second Edition manual we find the system calls documented in a dedicated section (II) of the manual.

6. Note that Roman section numbering (I–VIII) was employed from the First to the Sixth Research Edition. We follow the same convention in our references to these editions.

Listing 1. The inode definition in PDP-7 Unix

```
ii: .=.+1
inode:
 i.flags: .=.+1
 i.dskps: .=.+7
 i.uid: .=.+1
 i.nlks: .=.+1
 i.size: .=.+1
 i.uniq: .=.+1
  .= inode+12
```

Interpreter At least two system commands *ind* (indentation) and *lcase* (lower case conversion) are written in a (relatively) high-level language, namely B. This is implemented with a threaded code interpreter [90].

Monolithic Implementation The kernel is structured as nine assembly-language files (*s1.s – s9.s*) lacking easily discernible decomposition and partitioning. The same is also observed for the editor *ed*, which consists of two similarly named files (*ed1.s* and *ed2.s*).

Process Management The kernel can create an independently scheduled copy of a running process through the easily-implementable *fork* system call,[S3] which is named and modeled after Melvin Conway's *fork* and *join* proposed multitasking primitives [91]. The replacement of the running process copy with another program through the *exec* system call was not implemented at the time. Instead, the shell overlays the running code by reading the code of the other process from the disk, and then transfers execution to its entry point with a jump instruction [55], [59].

By the First Research Edition, the process management interface had evolved into four system calls that define the way processes: are created—*fork*(II), have their code loaded—*exec*(II), are terminated—*exit*(II), and are monitored for termination—*wait*(II). This basic model has been standardized under POSIX [92] and survives until today. The split of a new process creation from the loading of the corresponding code may seem like a peculiar architectural choice, because its benefits (the ability to create an identical sibling of an existing process) are small; typically a call to *fork* is immediately followed by one to *exec*. The reason behind this choice seems to be historical. Given the existence of the *fork* system call, it was easier to add an *exec* call than to create from scratch a call that would combine the two.

Descriptor Management The kernel provides I/O functionality, such as read and write, through special connectors, the file descriptor handles; these are small integers that map I/O calls to the underlying file or device. The kernel *fget*[S4] and *fput*[S5] routines provide a bare-bones interface for obtaining and disposing file descriptors to other kernel system calls (e.g., *creat*,[S6] *open*,[S7] *seek*[S8]).

Separation of File Metadata from File Naming The PDP-7 kernel separates a file's metadata (user-id, size, disk block locations, number of links) from the file's directory name by introducing the concept of a file information node (*inode*; see Listing 1). A function (*namei*[S9]) can obtain the inode associated with a path name, while other functions (*iget*[S10] and *iput*[S11]) deal with open files through their inodes. This elegant connector simplifies many file administration tasks.

Devices as Files The kernel follows the virtualization principle by abstracting devices, such as the console, the second terminal, and the paper tape drive, into files that are accessible via the file system's *system* directory[S12] (`/dev` in later versions). This type of binding, allows arbitrary programs to communicate with any device.

File I/O A simple yet powerful interface, based on the system calls *open*,[S13] *read*,[S14] *write*,[S15] *seek*,[S16] *tell*,[S17] and *close*,[S18] allows programs to access files as a flat sequence of bytes in both sequential and random access fashion. This interface has survived until today, both as Unix system calls and as the I/O API in popular programming languages.

Filesystem Four system calls allow the manipulation of files within the filesystem: *creat*,[S19] *rename*,[S20] *link*,[S21] and *unlink*.[S22] All have survived until today. The functionality of the *creat* system call has been usurped in a generalized form by *open*.[S23] Furthermore, the system calls *rename*, *link*, and *unlink* were extended in 2008 with siblings that work on file descriptors in order to avoid race conditions.[S24] Note that this problem could not have been foreseen, because directory support did not exist at the time.

4.2 First Research Edition

The First Research Edition (November 3, 1971[7]) was a rewrite of the PDP-7 Unix targeting the PDP-11 processor. The following architectural design decisions are visible in this edition. Note that, the shell-related design decisions may have also been available in the PDP-7 edition, but the corresponding shell does not seem to have survived in order to study their implementation.

System Calls Although the first edition Unix was a complete rewrite of PDP-7 Unix, it retained a large number of the defined system calls, thus establishing the core architecture of the Unix system call interface. Specifically, from the 28[L6] system calls implemented in the PDP-7 version[S25] and the 34[L7] calls implemented in the First Edition,[S26] 18[L8] are common between the two:[L9] *chdir*, *chmod*, *chown*, *close*, *creat*, *exit*, *fork*, *getuid*, *link*, *open*, *read*, *rele*, *seek*, *setuid*, *tell*, *time*, *unlink*, *write*. More impressively, from the 34[L10] system calls implemented in the First edition, 18[L11] have also survived in the modern FreeBSD-11.0.1 version:[S27,L12] *chdir*, *chmod*, *chown*, *close*, *creat*, *fork*, *fstat*, *getuid*, *link*, *mkdir*, *mount*, *open*, *read*, *setuid*, *stat*, *unlink*, *wait*, *write*.

Binary-Code API At the CPU level, system calls are typically dispatched through a memory address vector containing the location of the code implementing each call. At the programming level, system calls are referred to by names, such as *open* or *exec*. Rather than dynamically allocating system call names to entries in this table, the First Edition established a numbering scheme to place system calls in stable positions within the table. This allows Unix systems to maintain binary API compatibility of compiled programs between successive releases and even between different implementations, such as Linux, without requiring expensive adaptation layers. As can be seen from system calls defined in the 1971 First Edition[S28] (Listing 2[L13]) and the corresponding calls defined in the 2016 FreeBSD-11.0.1[S29] (Listing 3[L14]) the established numbering scheme persists until today.

7. The dates provided here are given by Salus [77, p. 43].

Listing 2. System calls 0–10 defined in the 1971 First Edition Unix

```
sysrele / 0
sysexit / 1
sysfork / 2
sysread / 3
syswrite / 4
sysopen / 5
sysclose / 6
syswait / 7
syscreat / 8
syslink / 9
sysunlink / 10
```

Listing 3. System calls 0–10 defined in the 2016 FreeBSD-11.0.1

```
 0    { int nosys(void); } syscall nosys_args int
 1    { void sys_exit(int rval); } exit
          sys_exit_args void
 2    { int fork(void); }
 3    { ssize_t read(int fd, void *buf,
          size_t nbyte); }
 4    { ssize_t write(int fd, const void *buf,
          size_t nbyte); }
 5    { int open(char *path, int flags, int mode); }
 6    { int close(int fd); }
 7    { int wait4(int pid, int *status,
         int options, struct rusage *rusage); }
 8    { int creat(char *path, int mode); }
 9    { int link(char *path, char *link); }
10    { int unlink(char *path); }
```

Abstraction of Standard I/O The First Edition shell offers the ability to associate user-specified files in the place of the program's standard input and standard output, through the corresponding I/O redirection symbols (< and >). This follows the virtualization principle by abstracting a program's standard I/O away from the terminal, allowing programs to operate on arbitrary files. The design decision is implemented by closing the default input or output file descriptor (typically associated with the terminal) and opening it again to associate it with the specified file.[S30]

Generic File I/O Layer Over a number of successive releases we see the evolution of a layer between the *read* and *write* system calls and the device drivers [50]. This handles read (*readi*[S31]) and write (*writei*[S32]) through an inode, read/write functionality common to both (*rdwr*[S33]), as well as the mapping of data to disk blocks (*bmap*[S34]).

User-Contributed Tools and Games The First Edition manual contains a section (VI) documenting "User Maintained Programs". Amazingly, this happened decades before open source operating system distributions, such as Debian and FreeBSD, started organizing third-party code contributions in the form of so-called "packages" or "ports". Operating systems by definition host user-written code. The architectural significance of this First Edition design decision is that the user-maintained components are documented in the system's manual, and are installed in a system-wide visible directory (typically `/usr/bin`—user binaries) rather than in the authors' home directories. This method supports a lightweight method for users to contribute code to the system, which can later mature to become an officially supported part of it.

TABLE 2
Documented File Formats and Their Users in the First and Second (*) Research Edition

Format	Description	Clients
a.out	Assembler and linker output	as, ld, strip, nm, un
Archive	Object code libraries	ar, ld
Core	Crashed program image	Kernel, db
Directory	Filesystem directories	du, find, ls, ln, mkdir, rmdir
Filesystem	Filesystem format	check, dump,* mkfs, restor*
Password	User accounts and passwords	chown, find, getpw,* login,* ls, passwd*
Tape*	DECtape file format	mt,* tap*
utmp	Logged in users	init, login,* who,* write*
wtmp*	Users login history	acct, date, init, login, tacct, who

The First Edition user-contributed programs included programming languages (*basic*), games (*bj*—black jack, *chess*, *moo*, *ttt*—tic-tac-toe), tools (*das*—disassembler), peripheral interfacing (*dli*, *dpt*—load DEC paper tapes), and nowadays familiar utilities (*cal*, *sort*). Documenting the user-contributed software was enforced through an interesting technical measure: a scheduled (*cron*) job would remove software that lacked up-to-date manual pages [80]. Currently, section VI of the Unix manual documents games, while some tools documented in the First Edition are now standardized Unix user commands. Third parties can still contribute code to Unix distributions through their *ports* or *packages* mechanisms.

The Shell as a User Program The documentation of the password file—passwd(V)—details that each record's fifth field contains the program to use as the shell. This allows arbitrary components to be specified as the ones with which a logged in user will interact; an editor for clerical staff and games were given as examples [36].

Interoperability through Documented File Formats Section V of the First Edition manual documents nine file formats. These act as connectors, allowing diverse programs to interoperate through an external coupling mechanism by reading and writing the corresponding files. Two more were added in the Second Edition, and more continued to be added in future editions. File formats used by more than one program are listed in Table 2. The files demonstrate two of the system's architectural principles: using flat files rather than elaborate file structures, and adhering to conventions (use of documented formats) rather than implementing complex enforcement mechanisms (e.g., APIs).

Tree Directory Structure Two system calls, *mkdir*(II) and *chdir*(II), provide the interface used for creating a new file directory and for establishing a directory as the current one. Other elements required for creating a tree directory structure are established by convention, which minimizes architectural complexity. Specifically, directories are plain files containing file entries in a known documented format—*directory*(V). In addition, two directory entries with special names, "`.`" and

"`..`", point to the current and parent directory respectively. A number of commands provide the required user-level support: *chdir*(I), *find*(I), *ln*(I), *ls*(I), *stat*(I), *mkdir*(I), *mv*(I), *rm*(I), and *rmdir*(I). They perform administrative chores and enforce restrictions by operating directly on the directory data.

Mountable Filesystem Interface Two system calls, *mount*(I) and *umount*(I), and two administrator programs with the same name provide an interface for connecting storage units containing filesystems to abritrary points of the directory structure. Its existence supports a single tree-structured name space for all files, hiding from users and programs the complexity and ugliness of "drives" or "devices". It also guided by example the philosophy of using a single consistent naming scheme for all files, which proved important as the system evolved.

4.3 Second Research Edition

The Second Edition (June 12, 1972) source code has only survived as a few system utility program fragments, which were recovered from a subset of a disk dump's DECtapes. Fortunately, this edition's manual survived as a printed document and provided the basis for this section's observations of architectural evolution.

Software Library The Second Edition manual contains a section (III) documenting 23 "subroutines", with a considerably wider scope than the few documented in the First Edition. These components mainly consist of a floating point math emulator, trigonometric, logarithmic, and conversion math functions, buffered I/O, memory management, sorting, and string processing. More than half (14) of them have survived as functions with the same name and functionality in the modern C library: *atan*(III), *atof*(III), *atoi*(III), *ctime*(III), *cos*(III), *exp*(III), *getc*(III), *hypot*(III), *itoa*(III), *log*(III), *putc*(III), *qsort*(III), *sin*(III), and *sqrt*(III). Their survival showcases the power of well-chosen abstractions.

4.4 Third Research Edition

The Third Edition (February 1973) is available through its manual pages—14,982[L15] lines of *troff* code—and the C compiler—2,751[L16] lines of C code.

Pipes and Filters This pattern was introduced in the Third Edition [77, p. 50], but the corresponding kernel assembly code has not survived. Even in the C source code of the Fourth Edition kernel, the pipe system call is only a stub redirecting to the `nosys` system call entry point. It seems probable that the corresponding system call was implemented in the assembly version of the kernel, which coexisted with it, and the C version had not caught up. However, the Third Edition manual documents the pipe system call[S35] and the construction of pipelines through the shell.[S36] (The syntax used for pipelines was at the time different from the current one.) Furthermore, the interface to diverse commands was changed overnight to allow them to run as filters, i.e., receive input from another process through their standard input stream and provide their output to another process through their standard output stream [75]. For example, the *cat*, *od*, *pr*, and *sort* commands are documented in the Second Edition manual with a mandatory input file argument. In the corresponding Third Edition manual pages, the file argument is optional—when missing the commands process their standard input. Moreover, the documentation of numerous commands—*crypt*(I), *hyphen*(I), *od*(I), *opr*(I), *ov*(I), *pr*(I), *sort*(I)—explicitly states that they can be used as a filter.

NAME

pipe – create a pipe

SYNOPSIS

(pipe = 42.)
sys pipe
(read file descriptor in r0)
(write file descriptor in r1)

pipe(fildes)
int fildes[2];

Fig. 2. The *pipe*(II) system call interface documented both for assembly language (using registers *r0* and *r1*) and for C callers.

4.5 Fourth Research Edition

The Fourth Edition (November 1973) is available through its manual pages—18,975[L17] lines of *troff* code—and the kernel—7,141[L18] lines of which just 768[L19] are written in PDP-11 assembly and the rest are written in C. Interestingly, the kernel exhibits a division of effort on architectural boundaries: Ken Thompson (*ken*) appears to have worked more on the main part of the kernel,[S37] while Dennis Ritchie (*dmr*) appears to have mainly worked on device drivers.[S38]

Structured Programming in a High-Level Language The rewriting of the system kernel from PDP-11 assembly language in a high-level language that later became C (at the time it was known as "new B") imposed discipline in the scoping of identifiers. This increased the kernel's modularity by allowing the definition of small (on average about 17.9[L20] lines long) functions. Thus, the Fourth Edition kernel defines 105[L21] C functions and 50[L22] assembly language symbols. Contrast these numbers with the 200[L23] (global) symbols defined in the PDP-7 kernel and the 248[L24] symbols defined in the First Edition (PDP-11) kernel.

User Groups The kernel introduces user groups and two system calls to manage them: *getgid*(II) and *setgid*(II). A few commands such as *chmod*(I) and *ls*(I) are correspondingly adjusted, and file permissions are extended to include group ones in addition to the existing 'owner' and 'others' settings. Despite its spartan interface, the concept is extremely powerful. Coupled with group ownership of files (which include devices mapped to the filesystem name space), permissions associated with a file's group, and the ability to have programs assume the identity of a specified group, it allows the administrative control of resource access according to a user's group and action. For example, appropriate group permissions can provide all operators tape and disk drive access for backup purposes, without requiring a complex access control list to be associated with each corresponding device. The concept is an elegant case of solving a problem by adding another level of indirection.

Language-Independent API The gradual implementation of the system in a high-level language necessitated an API that would be compatible with both assembly language code and code written in C. Consequently, the system calls are provided and documented through an API that is callable from both languages—an example can be seen in Fig. 2. Such mechanisms supporting language coexistence under the same roof were later extended to cover Fortran and Pascal, and nowadays serve diverse languages ranging from Java and Go to JavaScript and Python.

Data Structure Definition Reuse The kernel contains in its top level directory 12[L25] C header files that are used in

165[L26] instances by 35[L27] kernel source code files. (Regarding header adoption by user-space programs the—closest available—Fifth Edition source code has 17[L28] instances of header file use in 13[L29] files.) Header files provide a shared mechanism for communicating through reused data structures, something that in the past was performed simply by copying the data structure's definition from a manual into the code of each program. The use of header files allows the evolution of data structures by the addition of fields and changes to their types. This in turn can be used to promote portability, through the use of types that are appropriate for each CPU architecture.

Dynamic Resource Management Two routines, *malloc*[S39] and *mfree*,[S40] are introduced to manage the dynamic allocation and release of main memory blocks for in-memory processes and of continuous disk swap area blocks for swapped-out processes. Through these routines both allocations reuse the same underlying data structure, a *map*. Each of the two maps (*coremap*[S41] and *swapmap*[S42]) is an array of structures containing the position and size of each allocated block [60, p. 5–1].

Device Driver Abstraction The manual documents in section IV 16 "special files", which are located under the `/dev` directory. These correspond to diverse devices, including the *cat*(IV) phototypesetter interface, the *da*(IV) voice response unit, the *dc*(IV) data-phone interface, the *kl*(IV) console typewriter, the *pc*(IV) paper tape reader/punch, the *tm*(IV) magnetic tape interface, and various disk drive types. These files are implemented by device drivers.[S43]

At the kernel level each character device driver provides through the `cdevsw` table what we would call today an object-oriented interface with five methods:[S44] `d_open`, `d_close`, `d_read`, `d_write`, and `d_sgtyy`. Block devices provide through the similar `bdevsw` interface three functions: `d_open`, `d_close`, and `d_strategy`. These functions have mostly obvious semantics, transforming hardware-agnostic I/O requests into the protocol required by the corresponding devices. The `d_strategy` function is responsible for queuing read and write requests and the `d_sgtty` function for getting and setting a terminal's speed and processing flags. This standardized interface hides device-specific hardware intricacies from the rest of the kernel and from user level programs, thus virtualizing the underlying devices. In a departure from this modular interface, the interrupt functions associated with the devices are directly hard-coded in the interrupt table.[S45]

Remarkably, both the `cdevsw` and the `bdevsw` interface (renamed into `devsw`), extended with a few more functions still exist in modern versions of Unix, demonstrating the design's enduring relevance and utility.[S46,S47]

Buffer Cache The buffer cache[S48] stores in main memory a copy of data read from or written to secondary storage. This bridges the performance gap between the high-latency secondary storage and the lower-latency main memory. Offered as a service to all block device I/O, it improves the performance of both kernel and user-process disk I/O, at the expense of complicating the maintenance of consistent disk structures.

The buffer cache is another pattern that has persisted through time to the current version FreeBSD-11, even down to the names of three buffer structure flags.[S49,S50]

4.6 Fifth Research Edition

The surviving Fifth Edition (June 1974) is only missing the source markup of the manual pages. This edition was officially made available to universities for educational use [93, p. 8].

Command Files Already from the Second Edition the shell documents its ability to run with the name of a file containing commands as an argument. In the Fifth Edition we see four files containing such sequences of commands. These are used to configure the system at boot time,[S51] to update the C-compiler's archive containing nonce-language expression template tables,[S52] to compile, link, and install diverse system files,[S53,S54] and to create the manual's table of contents and index.[S55] At just 69[L30] lines the amount of code embedded into these files is very modest. However, this use marks the beginning of scripting in Unix, which will later become a dominant paradigm.

4.7 Sixth Research Edition

The Sixth Edition (May 1975), is the first that became widely available outside Bell Labs through licenses to commercial and government users. John Lions studied and documented the kernel's structure as material for teaching two operating systems courses at the University of New South Wales in Australia in 1977 [60].

Portable C Library A library[S56] of routines implemented in the C programming language is provided with the explicit goal to improve portability among the three operating systems on which the language was made available: PDP 11 Unix, Honeywell 6000 GCOS, and IBM 370 OS. The library implements in C, functionality that was at the time coded in assembly language, such as the formatted printing[S57,S58] and dynamic memory allocation.[S59,S60] In the Sixth Edition release it seems that both the portable library and the original routines coexisted, and that Unix tools relied on the assembly language routines. Some routines—e.g., *printf*(III)—were offered as plug-compatible alternatives, while other functionality (e.g., memory allocation) was provided using different interfaces. Over time the portable C library influenced the design and implementation of the Unix C library. The modern standard C library defines both routines stemming from the original assembly language implementation and those, such as *system* (3), that were introduced by the portable library version.

4.8 Seventh Research Edition

The Seventh Edition (January 1979), includes many new influential commands, and is the version that was widely ported to other processor architectures.

Unix as a Virtual Machine Early problems in porting programs written in C between diverse operating systems [54, p. 2025] convinced Dennis Ritchie that it would be easier to port the operating system between diverse hosts than to port the application programs between operating systems [81]. An Interdata 8/32 computer was purchased and used to prove this point. The project involved

- the implementation of a C compiler whose code generation part could be adapted for various CPU architectures [94];
- the extension of the C programming language to aid the portability of code written in it;

- the abstraction through libraries and header file definitions of elements that varied between different machines; and
- the identification, revision, and isolation of the kernel's machine dependent parts from the bulk (95 percent) of the code that could remain the same across all systems [54].

Dynamic Memory Allocation A main memory allocator, *malloc*(3) is offered as part of the C library. It allows programs to dynamically allocate memory space for storing data, rather than reserve fixed amounts of space. The void filled by it, is evident by its rapid and widespread adoption. It is directly used by 26 user-mode programs (out of about 160) and also in the implementation of other library functions, namely by the standard I/O and by the multiple precision arithmetic libraries.

Static Analysis A dedicated program, *lint*(1) [25], is offered to check C code for issues that are not caught by the C compiler. It performs strict type checking, detects potential portability problems, and identifies error-prone or wasteful constructs. Static program fault analysis was, and still is, a resource-demanding and imprecise task. Implementing it as a separate program frees the compiler from its demands and also provides an isolated experimentation venue that cannot easily disrupt the day-to-day development of production code.

Environment Variables The kernel,[S61] the shell,[S62] and the C library[S63] act in concert to support *environment variables*—*environ*(5). These allow an array of arbitrary strings (by convention key-value pairs) to be passed down the process invocation tree, thus establishing a simple, low-overhead, open-ended, one-directional, parameter-passing connector. Environment variables appear in the shell as ordinary variables, and can be accessed in C code with a single function call—*getenv*(3). By being part of a process's operating-system context data, they are inherited down to arbitrary levels of process invocation, without requiring any coordination with intermediate layers.

An important environment variable is PATH, which specifies a list of directory paths where the shell looks for executable programs. Changing these paths allows end-users to extend or substitute the programs supplied by the operating system. End-user operating-system configuration was later extended to other areas, including the location of manual pages (MANPATH) and dynamically linked libraries (ID_RUN_PATH) as well as the filesystem hirearchy—*chroot*(2). This line of evolution culminated into the modern operating-system-level virtualization systems, such as Linux kernel name spaces and control groups, FreeBSD Jails, and Solaris Zones.

Language Development Tools The lexical analyzer generator, *lex*(1) [29], introduced in the Seventh Edition complements the parser generator, *yacc*(1) [28], already present in the Sixth Edition. Together these two offer the basis for constructing programming language front ends [95]. This significantly simplifies the implementation of a programming language parser to a task achievable by a competent programmer rather than an expert on automata theory. The utility of this approach is exemplified by the existence of twelve tools whose grammar is written in *yacc*: *awk*(1), *bc*(1), *cpp*(1), *egrep*(1), *eqn*(1), *lex*(1), *m4*(1), *make*(1), *pcc*(1),[S64] *neqn*(1), and *struct*(1). Through the availability of compiler tools, the implementation of many complex facilities is abstracted into the development of a domain-specific language which acts as a platform for solving the corresponding problem.

Domain-Specific Languages Aided by the availability of compiler tools, several tools based on *little* or *domain-specific languages* [96], [97], [98] support a variety of generic processing tasks in a way that allows end-users to write specialized code in order to achieve their particular goals. Tools introduced in the Seventh Edition include the Bourne shell [84], [99]—*sh*(1), *awk*(1) for processing field-oriented records [100], *sed*(1) for manipulating plain-text files [101], *find*(1) for filesystem hierarchy traversals, *expr*(1) and *bc*(1) for evaluating expressions, *egrep*(1) for finding lines that match an extended regular expression, *m4*(1) for performing macro processing [102], and *make*(1) for maintaining program dependencies [103]. Some of the languages, such as those employed by *find*, *expr*, and *egrep* are fairly basic, and code written in them rarely spans more than a single line. The rest are more sophisticated, and some have been occasionally (mis-)used to build large applications.

Filesystem Directory Hierarchy The documented layout—*hier*(7)—for the filesystem hierarchy specifies the role and contents of 51[L31] directories. The structure has remained mostly stable over the years. It has even been adopted and standardized by the Linux community, in the form of the *Filesystem Hierarchy Standard*. The documented structure offers another example of establishing flexible conventions over implementing rigid enforcement mechanisms. It also demonstrates the formalization of a structure that evolved organically over the years. Although the directory hierarchy changed a lot before the Seventh Edition, as the Unix team experimented with various layouts, it stabilized after it was documented and evolved only gradually. An example of an early change is that the `/usr` directory was initially used for user programs, but was later repurposed to denote a general purpose directory, typically residing on a large mounted filesystem. Significant developments after the Seventh Edition include the addition of the `/home` directory for user files and the `/var` directory for system files that change while the system is running. Importantly, the documented hierarchy contains all parts of a self-hosted system: source code, development tools, libraries, and documentation.

4.9 First and Second Berkeley Software Distributions

The first Berkeley Software Distribution (BSD), released in early 1978, contained the Berkeley Pascal System [104], the *ex* line editor [105], and a number of tools. The Second Berkeley Software Distribution (2BSD), included the full-screen editor *vi* [106], the associated terminal capability database and management library *termcap*, and many more tools, such as the *csh* shell [107].

Software Packages The two Berkeley distributions introduced to the user community third-party software packages targeting Unix. Over the years packages proliferated and got distributed, initially through USENET [108, pp. 958–959] newsgroups and later over the internet in the form of *ports* to a specific operating system distribution. The established filesystem directory hierarchy, provided a template for laying out the source code, the documentation, and the manual

pages. In addition, the use of *make*(1) provided a common way for expressing compilation and deployment rules. In total 2BSD came with 32[L32] *makefiles*.

4.10 3BSD

The 3BSD release, which came out in late 1979, extended *Unix 32V*, a direct port of the Seventh Edition Unix to the DEC/VAX architecture, with support for virtual memory and the 2BSD additions.

Virtual Memory Paging The virtual memory (VM) [42] subsystem is a major component introduced in the 3BSD kernel, comprising 17 percent of the main kernel code (2808 out of 16039 C source code lines).[S65] It is delineated from the rest of the kernel code, by having its eleven source code files begin with a unique prefix (`vm`), a method followed by other elements in future releases.

The VM system primarily supports allocating memory to processes when no free main memory is available by swapping out suitable VM pages to disk (paging). In addition, new forms of the *read*(2), *write*(2), and *fork*(2) system calls are provided, which utilize VM for performing the I/O—*vread*(2) and *vwrite*(2)—and for reusing a process's memory space—*vfork*(2). This violation of abstraction proved to be a short-lived experiment. Subsequent releases abstracted the use of VM by the common I/O routines—*read*(2) and *write*(2)—removing the need to call separate routines in order to benefit from VM capabilities. Furthermore, the use of *vfork*(2) is discouraged in modern FreeBSD versions.

4.11 4BSD

The 4BSD release (October 1980) was developed by the newly established Computer Systems Research Group working on a contract for the Defense Advanced Research Projects Agency (DARPA). The contract aimed at standardizing, at the operating system level through the adoption of Unix, the computing environment used by DARPA's research centers [77, p. 159–160]. The release included a 1k block filesystem, support for the VAX-11/750, enhanced email, job control, and new signal semantics that addressed existing race conditions—the so-called reliable signals [109, pp. 270–283].

Regular Expression Library Regular expressions feature prominently in numerous Unix tools, such as *ed*(1), *grep*(1), *egrep*(1), *awk*(1), *sed*(1), and *expr*(1). Consequently, supporting the corresponding functionality as part of the C library—*regex*(3)—is an obvious design decision. The provision of the regular expression library in 4BSD is an important enhancement, foreshadowing the widespread support for regular expressions in most modern programming languages.

However, the regular expression library's development took time and its adoption was lackluster. Initially, each tool had its own regular expression engine.[S66,S67,S68,S69,S70] The reason for this may have been incompatibilities between the regular expressions processed by diverse tools, primitive support for libraries, or tool owners too fond of their own regular expression implementation to demand a common library [110]. Even when the library was provided, few programs adopted it. In 4BSD only a single program, *more*(1), made use of the library.[S71] By 4.3BSD (1986) just two more (new) programs were using it: *dbx*(1) and *rdist*(1). The reason for this slow-paced adoption may be that BSD Unix developers did not feel owners of the tools and code developed at Bell Labs. By the release of FreeBSD 11.0 (2016) the situation had changed, and four of the tools referred in the preceding paragraph—*ed*(1), *grep*(1), *sed*(1), and *expr*(1)—were rewritten as open source software, and used the contemporary version of the regular expression library.

Optimized Screen Handling The *curses*(3) library addresses the problem of placing characters on arbitrary positions of diverse incompatible terminal displays over a low-bandwidth connection. Cursor-addressable displays used to require different, incompatible, escape sequences for performing tasks such as clearing the screen, using a highlighted font, or placing the cursor in a specific screen position. Moreover, refreshing an entire screen sized 80×24 characters over a 300–1200 baud serial terminal connection can take a long time. Consequently, for the sake of efficiency, usable screen content must be preserved and only content differences should be sent down the line. To solve these problems the *curses* library abstracts the character escape sequences required to manipulate cursor-addressable terminals into a set of C library routines and a database—*termcap*(5)—that stores the sequences associated with each terminal type. The library also optimizes the display's updates by mirroring its content in internal data structures and using those data to minimize the transmitted data.

Modern command-line interfaces work on terminal emulators running on bitmap displays with high-bandwidth connections, and almost all emulators are standardized to follow the X Window System *xterm* escape sequences. Thus, none of the original requirements associated with the *curses* library hold today. However, the library is still used to maintain the functionality of programs designed for completely different hardware.

4.12 4.2BSD

The 4.2BSD release (September 1983) was based on a design described in an architecture manual written by Bill Joy and his colleagues [111]. It included many important features delivered in 4.1BSD and three more informal interim releases [78]: 4.1BSD (performance improvements); 4.1aBSD (TCP/IP networking and networking tools); 4.1bBSD (Berkeley Fast Filesystem [43] and symbolic links); and 4.1cBSD (new signal code). Compared to the pre-releases, the final release improved networking support and added new signal facilities and disk quotas.

Internet Protocol Family Support for the internet protocol family was arguably one of the most influential Unix design decisions that appeared in the second decade of the system's life. With 6,586[L33] lines of code implementing five protocols (ARP, IP, TCP, UDP, AND ICMP) the effort appears very modest by today's standards. This protocol stack was widely used as a reference implementation in routers and other operating systems. From an architectural perspective, the decision to implement this functionality in the kernel rather than as a user space program (as was e.g., the case for the KA9Q implementation [112]) may have contributed to the performance, standardization, and widespread adoption of these protocols and the corresponding implementation.

Local and Remote Interprocess Communication Both local and remote bidirectional interprocess communication (IPS) between arbitrary processes is established through the, now

TABLE 3
Uses of the Socket API in 4.2BSD

System call	Uses
bind	23
connect	15
accept	13
select	12
listen	11
sendto	10
shutdown	9
recvfrom	8
getsockname	6
recv	2
send	2
sendmsg	1
getsockopt	0
recvmsg	0
socketpair	0

ubiquitous, *socket*(2) API for setting up and accepting network connections. In earlier versions IPS was mainly implemented through the *pipe*(2) system call realization of the corresponding pattern. This establishes only a one-directional communication path between processes of a common ancestor.

In contrast to the parsimony of earlier Unix system call additions the new API is based on a plethora of new system calls (Table 3[L34]). There are arguments to be made for and against shoehorning new facilities on existing system calls, as could be done in this case by reusing the *open*(2), *read*(2), *write*(2), and *close*(2) API. Reusing or improving an existing API reduces the system's API surface and the associated learning curve, but can also negatively affect the compatibility of existing code, runtime performance, and the API's ease of use. Certainly however, the exhibited profligacy marks a departure of architectural style from the parsimony of earlier Unix editions.

The *sockets* API is used in the 4.2BSD release by two library functions—*rcmd*(3X) and *rexec*(3X); eleven system daemons—*comsat*(8C), *ftpd*(8C), *gettable*(8C), *implogd*(8C), *rexecd*(8C), *rlogind*(8C), *af*(8C), *rshd*(8C), *rwhod*(8C), *telnetd* (8C), and *tftpd*(8C); and eight user-mode programs—*ftp*(1C), *rlogin*(1C), *rsh*(1C), *talk*(1C), *telnet*(1C), *tftp*(1C), *whois*(1C), and *sendmail*(1C). Based on the, sometimes small, number of the provided system calls uses in client code (Table 3) one could claim that the provided API was over-engineered.

Also, in retrospect, the abstraction from the TCP protocol to stream sockets and from the UDP protocol to datagram sockets was another instance of over-engineering, because for decades mainstream systems maintained a one-to-one relationship between the two protocols and the corresponding abstractions. However, the proliferation and evolution of networking protocols at that time forced the networking stack's designers to adopt the abstraction as a precaution for other evolutionary avenues. This is expressed in a *Caveat* section in the *inet*(4F) manual page.

> "The Internet protocol support is subject to change as the Internet protocols develop. Users should not depend on details of the current implementation, but rather the services exported."

Directory Processing Abstraction Three new system calls—*mkdir*(2), *rename*(2), *rmdir*(2)—and the *directory*(3) access library comprising the *opendir*(3), *readdir*(3), *telldir*(3), *seekdir* (3), and *closedir*(3) functions—individually documented in 4.3BSD—abstract the processing of directory entries. Before the introduction of this feature, directory operations were performed by directly accessing and manipulating the contents of the corresponding disk structures. This abstraction promotes innovation in filesystem design, such as the long file names introducted with the Berkeley Fast Filesystem [43].

Network and User Database Access A series of library functions provide an interface for accessing entries stored in the filesystem table—*getfsent*(3X), the user group file—*getgrent*(3), the hosts database—*gethostent*(3N), the networks database—*getnetent*(3N), the protocols database—*getprotoent*(3N), the user details file—*getpwent*(3), and the network services database—*getservent*(3N). Abstracting this functionality into reusable libraries reduces code duplication, errors, incompatibilities, and also makes programs using this functionality easy to adapt in the future. For example, in modern FreeBSD the same routines can be configured—via the *nsswitch.conf*(5) file—to provide data: from local files (as was the case in the original implementations), from a local key/value database, from the Internet Domain Name System, from NIS/YP servers, or from a caching daemon.

Pseudo-Terminal Driver The pseudo-terminal driver—*pty* (4)—allows the creation of software-controlled terminal-like devices. These appear as a pair of master-slave devices. A process, such as a shell or an editor, attached to the slave end has the illusion of working on a physical terminal. However its I/O does not come from an actual terminal, but from another process controlling the master end. Through this connector mechanism arbitrary user processes can create virtual terminals that can be used by other processes without any prior arrangement or adjustment. The facility is used in 4.2BSD by the remote login daemon—*rlogind*(8), the (similar) telnet daemon—*telnetd*(8), and the terminal session typescript program—*script*(1).

4.13 4.3BSD Tahoe

The 4.3BSD Tahoe release (June 1988) supported the CCI Power 6/32 minicomputer (code-name *Tahoe*) and improved TCP algorithms.

Multiple CPU Architecture Support The kernel is split into machine-dependent and machine-independent parts. The machine-dependent parts support the original VAX[S72] architecture and the new Tahoe[S73] architecture. The split places code for interfaces,[S74] system configuration,[S75] and booting[S76] into separate directories. In total from the 218,783[L35] lines of the system's kernel source code, 104,279[L36] lines reside in the `vax` directories and 42,112[L37] reside in the `tahoe` directories. Thus, a significant part of the kernel code (72,392[L38] lines) appears to be portable between different processor architectures.

Third-Party Software Contributions The system contains 59[L39] files comprising 25,739[L40] lines that are marked as "`software contributed to Berkeley`". These come from individuals (Arthur Olson, Chris Torek, and Rick Adams), as well as corporations (Computer Consoles Inc, Excelan Inc, Harris Corp, Sun Microsystems, Inc, and Symmetric Computer Systems). Although the size of the

contributions is modest, the phenomenon is important, because it marks the beginning of growing the system through what evolved to become an open source software community. By the next release (4.3BSD Reno) the third party software contributions had swelled to 896[L41] files, 218,455[L42] lines, from about 70[L43] entities.

Timezone Handling The release incorporates a public domain timezone handling package developed outside Berkeley [64, p. 9].[S77] The package stores the timezone change rules into a database, allowing it to be updated independently from the code that interprets those rules. This allows end-users to individually configure their local timezone, and administrators to easily update the database as new rules come into effect. This is the approach now followed by the majority of Unix systems.

4.14 4.3BSD Reno

The 4.3BSD Reno release (June 1990) supported virtual filesystem implementations through the *vnode* interface, Hewlett-Packard 9000/300 workstations, and OSI networking. It also incorporated a new virtual memory system adapted from Carnegie-Mellon's MACH microkernel operating system, a Network File System (NFS) implementation done at the University of Guelph,[S78] and an automounter daemon. Considerable code was released by Berkeley with a license allowing its easy redistribution and reuse.

Kernel Packet Forwarding Database The kernel provides a special network socket domain, PF_ROUTE, that user-level programs can use to query and manipulate its network packet routing database—*route*(4). The kernel uses this database to act as a router by forwarding packets between network interfaces, while user-level programs, such as *routed*(8) and *XNSrouted*(8), implement routing protocols by communicating with other hosts over the network. Following elegantly the separation of concerns principle, this minimizes the amount of complex code that must be maintained within the kernel, while avoiding the context switching overhead of a user-mode routing program.

Virtual Filesystem Interface Disk operations that were performed on inodes are virtualized through an object-oriented interface of *vnode* operations (`vnodeops`).[S79] A structure of function pointers provides access to storage, with functions such as `open`, `read`, and `write`, as well as to file naming with functions such as `mkdir`, `rename`, and `readdir`. The two groups were split in 4.4BSD in order to simplify the implementation of different storage methods, such as a log-structured filesystem [64, p. 244]. This interface is used to implement three filesystems: UFS, the original Unix filesystem; MFS, a filesystem storing files in virtual memory; and NFS, a filesystem operating over network connections.

Database Access Methods The *db*(3) library and API allow programs to store and retrieve key-value pairs in a file or memory–resident lightweight database [113]. Elements can be stored using btree, hashed, or flat-file data structures. An object-oriented interface, implemented through function pointers, provides get, put, delete, and sequential access methods. In contrast to the lethargic adoption of the regular expression library added in 4BSD, the provided functionality is reused by tens of programs, with the corresponding `db.h` header file included 105[L44] times in FreeBSD 11.1.

4.15 4.3BSD Net/2

The 4.3BSD networking release 2 (June 1991) came with a (what is now called) open source reimplementation of almost all important utilities and libraries that used to require an AT&T license. It also included a kernel that had been cleaned from AT&T source code, requiring just six additional files to make a fully-functioning system. This was the version used by Bill Jolitz to create a compiled bootable Unix system for Intel 386–based PCs.

Stream I/O Functions The *funopen*(3) family of functions allow C programs to access arbitrary data through the widely-used *stdio*(3) interface. The object-oriented constructor-like functions take as arguments *read*, *write*, *seek*, and *close* function pointers and return an opaque `FILE` pointer that supports all the usual operations, such as *getchar*(3) and *printf*(3). This elegant interface can be used for providing stream-like access to compressed—*zopen*(3), application-protocol—*fetch*(3), or encrypted data. However, the specific interface, the few library functions that build on it, and its GNU library equivalent—*funopencookie*(3), which was added in FreeBSD 11, have not seen significant adoption.

4.16 4.4BSD

The 4.4BSD release (June 1994) came out following two years of litigation and settlement talks regarding the alleged use of proprietary AT&T material. As a result of the negotiation this release removed three files that were included in the *Net/2* release, added Unix System Laboratories (USL) copyrights to about 70 files, and made minor changes to a few others. The release included additional work done on the system, such as support for the portal filesystem.

Stackable Filesystems The creation of a *vnode* stack allows a new filesystem type to use an existing one's operations. The simplest use of this concept is the null filesystem—*mount_null*(8), which allows an existing filesystem's sub-tree to appear in an arbitrary place of the global filesystem name space. This concept was extended in 4.4BSD/Lite1 with the union filesystem—*mount_union*(8), which allows the translucent addition of one (e.g., writable) filesystem on top of another (e.g., a CD-ROM).

Generic System Control Interface The generic system control interface—*sysctl*(3,8)—provides a library function—*sysctl*(3)—and an administrator utility—*sysctl*(8)—for examining or setting the kernel's state, later documented through its internal interface—*sysctl*(9). This interface replaces the original method that involved directly accessing the kernel's memory space through a special file—`/dev/kmem`. The *sysctl* facility offers significant portability, efficiency, security, and maintainability benefits compared to the `/dev/kmem` access method it replaces [64, pp. 612–614].

However, offering a standalone hierarchical view of the kernel space through the commonly-adopted "management information base" (MIB) abstraction, it sits at odds and competes with alternative architectural concepts, namely the provision of interfaces through the Unix hierarchical filesystem, and kernel interfacing through system calls, as could be done through the *kernfs*, *procfs*, and *fdesc* filesystems [63, p. 238]. Kirk McKusick, a principal BSD designer and developer, in an email to this paper's authors explained this choice stating that BSD users resoundingly found the *sysctl* facility a far more convenient way for remote system management compared to

the hierarchical filesystem access method. He added that the *sysctl* interface is also considerably more efficient.

4.17 386BSD Patch Kit

386BSD was a derivative of the BSD Networking 2 Release targeting the Intel 386 architecture developed by Lynne and William Jolitz [45]. The 386BSD patch kit contains 171 commits associated with patches made to 386BSD 0.1 by a group of volunteers from mid-1992 to mid-1993.

Organized Community Contributions The patch kit functionality adds to the Unix architecture a mechanism for accepting and distributing contributions coming from a decentralized team of individuals. Unix was first distributed with an open source license through the 4.3BSD networking release 1 (Net/1) in November 1988. This was a subset of the code that did not include material requiring an AT&T license. It was released to help vendors create standalone networking products without incurring the AT&T binary license costs, but did not include all the material required to run the system. This was later addressed by the 386BSD version. However, none of the two releases offered a way to manage third-party contributions. This essential characteristic of an open source *project* (as opposed to open source *software*) was formed more than four years after the release of 4.3BSD Net/1. Patch kit elements contain their changes in Unix "context diff" format, and can therefore be applied automatically to the 386BSD distribution. Each patch is accompanied by a metadata file listing its title, author, description, and prerequisites.

4.18 Overview of FreeBSD Releases

The FreeBSD Project started in early 1993 with the release of FreeBSD 1.0 to address difficulties in maintaining 386/BSD through patches and working with its author to secure the future of 386/BSD [114]. The focus of the project was to support the PC architecture, appealing to a large, not necessarily highly technically sophisticated audience [64, p. 11]. For legal reasons associated with the settlement of the USL case, while FreeBSD versions up to 1.1.5.1 were derived from the BSD Networking 2 Release, later ones were derived from the 4.4BSD-Lite Release 2 with 386/BSD additions.

4.19 FreeBSD 1.1

Package Manager The software ports facility[S80] provides a mechanism to compile and install third-party packages and their dependencies. It was first documented—*ports* (7)—in FreeBSD 2.2.6 and is available and growing on modern FreeBSD systems. It handles the modifications (patches) required for making a software package work under FreeBSD, the installation of required dependencies, and the installation and de-installation of the corresponding package. The loose coupling of packages to the operating system and the automatic handling of dependencies, allow the FreeBSD system to grow in functionality in diverse directions without excessively burdening its core.

4.20 FreeBSD 2.0

Process Filesystem The `/proc` filesystem—*procfs*(5)—provides a two-level view of running processes in the form of files appearing in the filesystem hierarchy [115]. It was originally introduced in a different form in 4.4BSD/Lite1.[S81] At its top is a list of directories corresponding to running processes. Each process directory contains files allowing the monitoring and control of the process's status and state, such as its CPU registers, memory, and resource use. The architectural significance of the process directory is that it supplies an alternative interface to functionality typically provided through system calls such as *ptrace*(2), and (for application within a process context) *getrlimit*(2) and *getrusage*(2).

Dynamically Loadable Kernel Modules The loadable kernel module facility—*lkm*(4)—allows the dynamic loading and unloading of kernel code at runtime. It has been replaced in FreeBSD 3.0.0 with the similar *dynamic kernel linker* facility—*kldload*(8), *kldstat*(8), *kldunload*(8)—to support the dynamic linking of kernel code at boot time [116, pp. 636–637]. Loadable kernel modules allow the provision of significant functionality to the kernel, such as device drivers, filesystems, emulators, and system calls. This reduces the kernel's default memory footprint and attack surface. The recent (11.1) version of FreeBSD provides 992[L45] loadable kernel modules.

4.21 FreeBSD 2.1

Linux Emulation Although the Linux kernel was developed independently from the Unix systems examined here, it follows the Unix system call API down to its numbering scheme. Nevertheless, some of its system calls are not directly supported by FreeBSD, while others have subtle differences in their interface specification. In addition, its executable file format differs from the FreeBSD one. A set of kernel files[S82] allows FreeBSD to load and run executable programs compiled for the GNU/Linux operating system. This is accomplished by suitably marking the corresponding process in order to emulate the behavior of Linux-specific system calls.

Packet Capture Library The efficient capturing and monitoring of network packets is an important diagnostic facility. The packet capture library *pcap*[S83]—documented in FreeBSD 8.0 as *pcap*(3)—together with the *tcpdump*(1) program allow the specification of packets to be captured, the compilation of the corresponding filter into a virtual machine program that can be dynamically injected for execution into the operating system kernel, and the retrieval and analysis of the captured packets. Developed by an independent group, the library abstracts diverse packet capture mechanisms into a portable interface.

4.22 FreeBSD 3.0

Common Access Method I/O Subsystem The common access method (CAM) I/O subsystem abstracts operations to storage devices based on a (draft) ANSI standard. It started by supporting SCSI and CD-ROM disks, but by the release of FreeBSD 9.0 it evolved to also cover the commonly used ATA and SATA disk drives [65, Section 8.1]. Its three layers comprise (from kernel to the device) the device-specific (e.g., SATA drive) peripheral access, the scheduling and dispatch of I/O commands, and the routing of commands to devices through the host bus adapter.

4.23 FreeBSD 3.4

Graph-based Kernel Networking and User Library The *netgraph*(4) subsystem allows the implementation of sophisticated networking protocols by following a data-flow model. Diverse

network packet processing nodes are connected through hook functions into a graph data structure by means of an object-oriented interface. *Netgraph* nodes can implement protocols, such as the point-to-point protocol—PPP, *ng_ppp*(8)—or provide utility functions, such as the Berkeley packet filter—BPF, *ng_bpf*(8). The FreeBSD 11.1 *netgraph* subsystem[S84] contains 177[L46] files offering *netgraph* functionality through 90,471[L47] lines of code.

4.24 FreeBSD 4.0

OpenSSL Framework The OpenSSL—secure sockets layer (SSL) and transport layer security (TLS) framework[S85]—provides two C libraries and a user command, *openssl*, that allow programs and users to work with these protocols. In addition, the framework's elements expose a variety of symmetric and public key encryption algorithms, message digest functions, and certificate handling operations. The framework's size is considerable, comprising 1,127[L48] files and 227,118[L49] lines of code. From an architecture perspective the framework's incorporation is notable due to its size, its development method (it was implemented by a separate team), and the fact that it brings on board its own command interface method, namely the provision of 22[L50] sub-commands accessible from the *openssl*(1) command.

Jails The *jail* (2,8) system call and command allow the system's administrator to isolate a set of processes in a confined environment, restricting the operations the processes can perform [117]. This extends the *chroot*(2) system call, which can offer a process a restricted view of the filesystem name space, to cover the virtualization of networking, interprocess communication, and filesystem mounting. Jails thus allow administrators to run processes with complex or brittle requirements in separate virtualized container environments, such as those provided by Docker [118]. Jails also allow cloud-service providers to host many clients with full administrative access to their (virtual) host on the same server. Such clients cannot run their own operating system, as they might be able to do under a full-blown hypervisor, but the service is very efficient in terms of resource utilization. In terms of architecture, jails provide an additional lightweight level of virtualization on top of the operating system.

Access Control Lists A library—*acl*(3)—provides an interface for extending the traditional Unix user/group/all read/write/execute discretionary access control model with access control lists (ACLS). Later releases add support for ACLs in the UFS filesystem[S86] and for the finer-grained permissions of NFSv4.[S87] In the current FreeBSD 11.1 version ACLS allow the specification of more than a dozen permissions for an arbitrary number of principals (users or groups).

4.25 FreeBSD 5.0

Symmetric Multiprocessing The kernel's code can run on multiple processors or CPU cores by synchronizing access to shared resources through a hierarchically ordered set of locking primitives [64, p. 93]. A large part of this extensive change involves the addition of 3,764[L51] mutex-based thread synchronization calls, which exist in 7.3[L52] percent of the kernel's 4,873[L53] source code files.

Modular Disk I/O Request Transformation Framework The GEOM modular disk I/O request transformation framework—*geom*(3, 4, 8)—allows storage subsystem requests to be transformed in order to support disk partitioning, aggregation, encryption, journaling, and I/O statistics collection. It is designed around a scheme where each functionality (e.g., striping) is implemented in a separate class. Object instances with provider and consumer interfaces are connected in a directed acyclic graph, which forms the transformation layers.

Mandatory Access Control The mandatory access control framework—*mac*(4)—supports fine-grained control of a system's security policies through diverse pluggable policy modules. Examples of supported policies include multilevel security [119], low-watermark, Biba [120], and process partition. The kernel associates policy-agnostic labels with filesystem objects, network interfaces, terminals, and users. This then allows the relevant kernel subsystems (filesystem, network, IPS, process management, VM) to obtain access control permissions from the framework, and inform it regarding objects' life cycle events [65, Section 5.10].

Pluggable Authentication Module The pluggable authentication module (PAM) architecture provides a way to implement and abstract diverse low-level user authentication methods, while presenting client programs with a single high-level API—*pam*(3). In addition to authentication, the library supports account, session, and password management. Retrofitting the Unix authentication system with PAM simplifies the introduction of sophisticated authentication methods, such those using one-time passwords and directory access, through the installation of corresponding modules.

4.26 FreeBSD 5.3

Streaming Archive Access Library More than a dozen ways to package multiple files into a single one have become widespread over the past half century. Typically each format is associated with corresponding packaging and compression programs, such as *ar*(1), *tar*(1), *cpio*(1), *gzip*(1), *compress*(1), or *bzip2*(1). The *archive*(3) access library consolidates these disparate formats. It allows programs using it to read and write most common archive formats, interfacing between an archive's files and those resident on a filesystem.

Miniport Driver Wrapper A kernel facility and an application program allow the use of network interface hardware device drivers conforming to the Microsoft Windows Network Driver Interface Specification (NDIS) miniport API to be used under FreeBSD. Thus, binary (compiled code) components developed for a radically different operating system can become FreeBSD device drivers. This mechanism addresses the difficulty of building or obtaining FreeBSD-specific device drivers for network interfaces.

4.27 FreeBSD 6.2

Basic Security Module Auditing The Basic Security Module Auditing (BSM) facility comprises kernel changes, system calls—*audit*(2), a library *libbsm*(3), configuration files—e.g., *audit_control*(5), a binary file format—*audit.log*(5), and support programs—*praudit*(1), *auditreduce*(1), *audit*(8), *auditd*(8)—to generate and process streams of records that are required for security auditing. The audited events include both kernel-level events, such as filesystem or network accesses, and application-level events, such as a user's authentication [65, Section 5.11].

4.28 FreeBSD 7.0

Zettabyte Filesystem The Zettabyte filesystem (ZFS) is an evolution of the 4.4BSD log-structured filesystem derived from Sun's OpenSolaris code base. It is based on the concept of checkpoints, which allow the filesystem to move from one consistent state to another [48]. Furthermore, by utilizing the availability of abundant memory and processing power resources in modern servers, it ensures data integrity through end-to-end checksums, it offers various levels of software RAID, and it provides massive (zettabyte-sized) scalability through (potentially hybrid) device pooling [65, Chapter 10]. The filesystem's code is organized around a layered architecture of considerable size, starting at 80,107[L54] lines in FreeBSD 7.0 and growing to 205,899[L55] lines in FreeBSD 11.1.

4.29 FreeBSD 7.1

Dynamic Tracing The DTrace facility, brought over from Sun's Solaris, builds on the reflection architectural pattern [71, p. 193] to allow the monitoring of the system's operation through thousands of probes. The probes can be configured and monitored through programs written in the D domain-specific language [49]. The *dtrace*(1) command executes these programs to enable the specified probes and report the collected details. DTrace has two important advantages over alternative approaches, such as system call tracing, kernel counter statistics, or profiling. First, it can monitor the whole application stack, including function boundaries, networking, scheduling, filesystems, system calls, and application code. Second, by installing only the required probes through dynamic code patching its performance impact on production systems is negligible when no data are collected.

4.30 FreeBSD 9.0

Para-virtualized I/O A set of devices conforming to the VirtIO specification—*virtio*(4)—allow efficient I/O in cases where FreeBSD runs on top of a hypervisor. The provided network, storage, and memory interfaces can eliminate the cost of emulating legacy hardware and of memory copying between the hypervisor and the guest operating system.

InfiniBand Support InfiniBand is a computer network communications standard offering high-speed (10–300 Gb/s) and low-latency (140–2600 ns). These design decisions make it attractive in high-performance computing applications as well as in other areas requiring fast interconnects between computers or between computers and storage systems. The technology is complex and demanding. Therefore, a group named the OpenFabrics Alliance is developing a cross-platform InfiniBand stack for diverse operating systems and distributing it as open source software. FreeBSD's InfiniBand support incorporates this large (325,818[L56] lines) code base.[S88]

Application Compartmentalization The *capsicum*(4) operating system capability and sandbox framework allows applications and libraries to be compartmentalized into isolated components in order to reduce the impact of security vulnerabilities. It works by allowing applications to enter a reduced capability mode, and by offering a system call API to restrict an application's access to global name spaces, such as the filesystem. For example, a potentially vulnerable application's processing part can be given only the right to write to a file previously opened by another part of the application that has retained ambient authority.

4.31 FreeBSD 10.0

Virtual Machine Monitor The *bhyve*(4,8) virtual machine monitor allows a FreeBSD system to host instances of other unmodified operating systems running on top of it. Supported operating systems include FreeBSD, NetBSD, OpenBSD, GNU/Linux, Windows, and SmartOS. At 30,391[L57] lines of code (rising to 62,906[L58] lines in FreeBSD 11.1) it is a modest implementation effort, relying heavily on the virtualization support offered by modern CPUS and supporting hardware.

Fast User Space Raw Packet Processing The *netmap*(4) framework [121] provides an API through which user space applications can access and inject packets associated with network interfaces, the system's network stack, or the *vale*(4) virtual switch. Through direct synchronized access to the kernel's corresponding ring buffers, applications avoid the overhead of data copying and can thus process millions of packets per second. This allows FreeBSD systems to implement network devices such as routers, switches, and firewalls [65, Section 13.8].

4.32 FreeBSD 11.0

Network Blacklisting The *blacklistd*(8) daemon listens from other networking daemons for notifications regarding successful or failed connection attempts. The *blacklistd.conf*(5) configuration file specifies the conditions under which the blacklist daemon will setup the system's packet filter to block connections associated with ports on which abusive behavior has been detected. The *libblacklist*(3) library implements the protocol for communicating between the daemons.

5 Quantitative Results

From 1970 until today the system's source code grew by three orders of magnitude, from 13 thousand to more than ten million lines of code. Is this growth rate reflected in terms of the number of features? What types of features are responsible for the main growth and what does their growth rate look like? What are the outliers and how can they be potentially explained? Is the size growth accompanied by a growth in code complexity? In order to answer these questions, as aforementioned in Section 3.3, we used the system's reference documentation as well as source code analysis.

5.1 Feature Growth

We analyzed the system's reference documentation rather than other categories of features (e.g., the system partitions as shown in Fig. 5 or 6), because its structure has remained essentially unchanged. Specifically, throughout its lifetime, the Unix reference documentation is divided into nine sections. In this study we ignore two: Section 6, which has evolved to document a few games, and Section 7, which documents miscellaneous elements ranging from the ASCII character set, to email addresses, to formatting macros, to the system's directory hierarchy. The remaining seven sections are listed in the first two columns of Table 4. The evolution in the number of their features is illustrated in Fig. 3.

As we can see in the corresponding figure, over the past half century the Unix system grew in similar proportion in the number of all feature types. Some outliers can be explained by constraints or choices associated with particular releases. For example, the decrease in the number of commands in 386BSD is probably due to the fact that this release

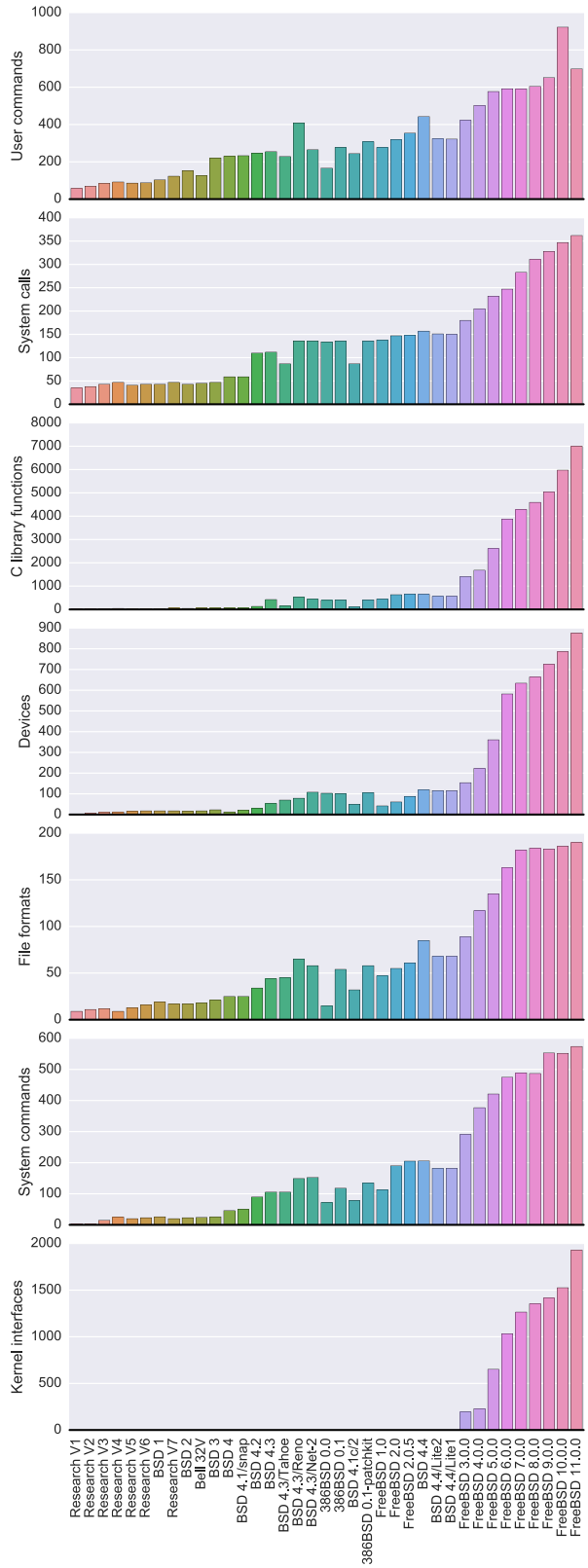

Fig. 3. Evolution in the number of feature types across key releases.

shoehorned a system that was distributed through tapes to run on VAX minicomputers into one that was distributed through floppy disks to run on PCS. Over the same period the system got released as open source software, which resulted in releases that purged items containing proprietary code. These were then gradually reimplemented or replaced with open source alternatives. The temporarily high number of user commands in 4.3BSD Reno stems from the inclusion of diverse user-contributed programs, such as Emacs, USENET News, and the X-Window System. These were later distributed as separate packages.

Turning our attention to specific feature types we see that growth has not been uniform across them; there is evidence of interesting trends for which we can hypothesize specific reasons. Growth in user and system administration commands as well as file formats has been relatively uniform. This can be expected, if we regard an operating system as a platform hosting (an expanding set) of programs and files.

The evolution in the number of system calls tells a more interesting story. There are two periods of relative stability. One over the Research Unix editions, which can be understood if one considers that its developers took pride in demonstrating "that a powerful operating system for interactive use need not be expensive either in equipment or in human effort" [36]. Consequently, they avoided bloating the kernel with functionality of marginal utility. A subsequent rise in the number of system calls followed by stability can be seen over the Berkeley releases. The rise can be attributed to research targetting specific areas: networking, filesystems, and interactive use. The subsequent stability marks a consolidation phase where the developed interfaces are used by an expanding number of user and system administration commands. The continuous rise in the number of system calls over FreeBSD releases can be attributed to a community keen on operating system innovation, and, maybe, one in which a large number of volunteer developers are eager to leave their mark on the kernel.

The evolution of C library functions tells a similar story. A restrained timidity over the Research Unix editions resulted in a core set of functions, most of which were later standardized as the C programming language library. Berkeley releases broke that tradition by introducing many new functions to accommodate newly provided functionality. That period's attitude seems to be that if some functionality is generally useful, then it should be made available as a library. This established a tradition for providing library interfaces to access the system's files and to package into libraries complex functionality, such as regular expression matching and embedded database support. Having broken the taboo of limiting the C library to a core set of portable functions, the rise in the provided functionality continued with the FreeBSD releases, resulting in a substantially larger number of library functions. Modern frameworks, such as .NET, Jakarta EE, and Python, have followed this lead by providing extensive support for diverse functionality.

Changes in the number of supported devices were probably driven by external factors, namely availability of such devices, demand for using them, and resources for implementing their driver code. The drop in the number of devices from 386BSD to FreeBSD 1.0 stems from the cleanup of obsolete non-working device drivers: from the *acc*(4) local/distant host DARPA IMP interface to the *vx*(4) dialup communications multiplexor.

Documentation for the kernel APIS (Unix Reference Manual Section 9) was only introduced in the late 1990s, so there is less to observe in the corresponding Figure. The initial rise probably stems from a vigorous effort to document existing interfaces, while subsequent growth may have been organic.

TABLE 4
Number of Documented Features in Current Operating Systems

Section	Description	FreeBSD	macOS	OpenBSD	Solaris	Ubuntu	Windows
1	User commands	700[L59]	1,352 [L60]	404[L61]	2,169[L62]	1,916 [L63]	N/A
2	System calls	370[L64]	252 [L65]	270[L66]	236[L67]	462 [L68]	} 6,001[L69]
3	C library functions	7,280[L70]	10,186 [L71]	5,094[L72]	7,693[L73]	3,726 [L74]	
4	Supported devices	907[L75]	46 [L76]	964[L77]	402[L78]	4,693 [L79]	N/A
5	File formats	191[L80]	192 [L81]	124[L82]	245[L83]	234 [L84]	N/A
8	Administration commands	579[L85]	661 [L86]	373[L87]	894[L88]	735 [L89]	461[L90]
9	Kernel interfaces	1,936[L91]	N/A	928[L92]	1,534[L93]	7,434 [L94]	N/A

To judge in context the evolution of supported features, the remaining columns of Table 4 list the number of documented feature types in diverse current operating systems: FreeBSD 11.1.0, Apple macOS 10.13.3, OpenBSD 6.3, Oracle Solaris 11.3, Ubuntu Linux 16.04.5 LTS, and Microsoft Windows 10 (build 16,299). The numbers were obtained as follows: for FreeBSD and OpenBSD by processing the source code and Makefiles;[L95] for Solaris by processing the indices of Oracle's on-line reference library;[8,L96,L97] for Windows by processing the source code of the Windows Server documentation[9] and the HTML markup of the Windows UAP umbrella library index;[10,L98] for Ubuntu and macOS by counting the number of manual page files or processing the kernel's source code in servers offered by the Travis CI continuous integration platform[11,L99] through a small project constructed for this purpose.[12] To keep the figures comparable we tried to provide numbers that reflect server rather than desktop installations.

As is evident from the table, the number of feature types is similar in magnitude across systems with different histories, architectures, or evolutionary paths. Where marked differences exist these can be readily explained. For example, in the case of device drivers, the differences stem either from the use of standardized hardware (macOS) or from widespread adoption (Ubuntu Linux). Also, because the Windows API does not clearly distinguish between kernel interfaces and user-level utility functions, the entry points of its API appear in the table spanning the rows for system calls and for C library functions. None of the systems exhibits the economy evident in, say, the 1979 Seventh Edition Unix. We interpret this as a sign that requirements from a modern operating system drive the corresponding essential complexity (and sometimes the accidental complexity). The observed quantitative rise in supported feature types is not coincidental, but a response to environmental pressures.

5.2 Cyclomatic Complexity

We also looked at the cyclomatic complexity evolution of the system's two major partitions the kernel-space code (C source code files—those with a `.c` suffix—nowadays residing under the `sys` directory), and the user space code. For the latter, we further distinguish between the libraries shared among multiple programs (C files in `lib`), and the user, administrator and system commands (all other C files). The reason for this distinction is that libraries are reused by other programs and therefore required to be more maintainable (have lower complexity).

Fig. 4 shows the cyclomatic complexity evolution trends over time for the three aforementioned types. In broad terms this follows a steep rise followed by a gradual decline. A possible explanation for the rise could be that improved technology (e.g., 9,600 baud glass terminals replacing 110 baud teletypewriters) might allow the adoption of more complex program structures [15]. The curve's steepness could be explained by the rapid introduction of these technologies, which enjoy the exponential growth benefits associated with Moore's Law [122]. The gradual fall could correspondingly be attributed to corrections addressing excessive complexity, implemented by adding better new code or by refactoring existing code. The reason behind such changes could be to satisfy the implicit quality requirements associated with the construction of a large and sophisticated software artefact [123]. This hypothesis is corroborated by the fact that the cyclomatic complexity of the three areas follows their relative criticality and importance. Is is lower for the kernel where a fault can bring down a complete system, as well as for the libraries where a problem can affect many programs. In fact the curves for the kernel and the libraries are surprisingly similar, especially after the mid-1990s. In contrast, it is higher for user, administrator and system commands where the code is isolated in separate processes and where problems typically affect only a single command. However, in all three cases the mean cyclomatic complexity at the end of the studied period is around 6, which is considered overall rather low [124, pp. 342–344] for such a complex long-lived system.

An enabling factor for battling cyclomatic complexity may be advances in CPU clock speeds and in compiler technology, such as the inter-procedural analysis offered by GCC and later LLVM [125]. These have allowed developers who crammed code into a single function, in order to avoid the performance penalty of function calls, to write smaller, more modular functions.

To put the evolution of cyclomatic complexity into perspective, the bottom part of Fig. 4 illustrates the corresponding complexity evolution of the GNU coreutils, the GNU C library, and the Linux kernel, juxtaposed with that of the Unix commands, library, and kernel respectively. We note that the measured periods are not identical, as the top part starts from the mid-70s, while the bottom part starts from 1995 or slightly earlier; we therefore leave out the first two decades of Unix in our comparison. The resemblance in each

8. https://docs.oracle.com/cd/E53394_01/
9. https://github.com/MicrosoftDocs/windowsserverdocs/
10. https://docs.microsoft.com/en-gb/windows/desktop/apiindex/windows-umbrella-libraries
11. https://travis-ci.org/dspinellis/documented-facilities/builds/459375741
12. https://github.com/dspinellis/documented-facilities

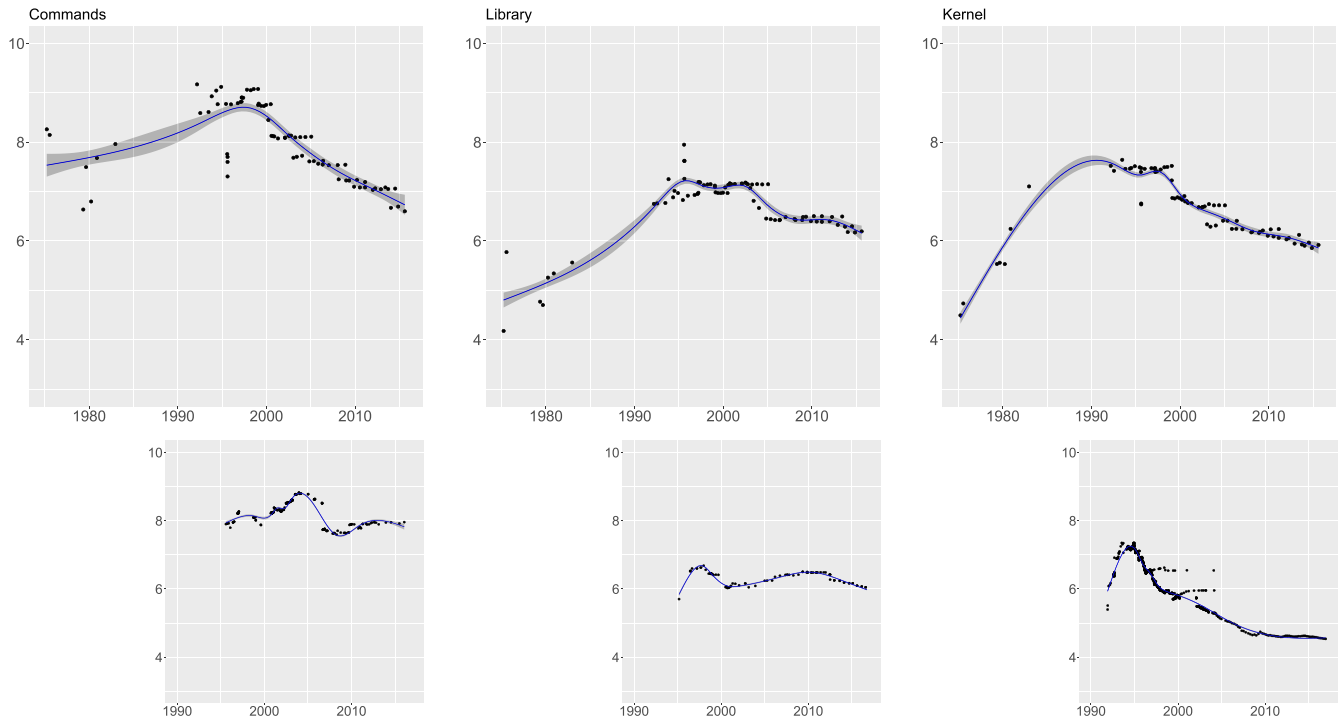

Fig. 4. Mean cyclomatic complexity of code over time. Top: for this study's Unix systems' user-space commands, C libraries, and kernel. Bottom: for the GNU coreutils user-space commands, the GNU C library, and the Linux kernel.

pair of curves is striking: the same initial incline and subsequent descent is observed.

We conjecture that the inverted U-curve in the GNU/Linux case is caused by reasons similar to Unix: steadily improving hardware capabilities throughout the 80s and 90s lead to the incline, followed by corrective actions to improve quality, as the complexity started to become overwhelming. It appears that the GNU/Linux community exhibits a similar maturity to that of FreeBSD [123], striving for code quality through re-working and refactoring the code. The actual cyclomatic complexity also fluctuates around the same figures: 7 to 9 for the commands, 6 to 7 for the libraries (after 1995), and 4.5 to 7.5 for the kernel.

There are however some pronounced differences as well. While the Unix commands had their complexity gradually reduced until the end of our measurements, reaching 6.5, the GNU user-space commands stabilized after 2010 at approx. 8. The reason behind this may be lower stability and maintainability requirements regarding individual commands compared to the monolithic kernel. Also, the peak in the two curves differs by about a decade, which indicates that the GNU/Linux community started to incorporate quality improvement guidelines and practices later than the Unix community. Moreover, the GNU C library had a second period of increasing complexity, albeit much more moderate. This may indicate again a creeping lack of attentiveness regarding design quality as the lessons of the preceding drive were forgotten, or be a side effect of the effort to adjust to a new version of C (C11). Finally, regarding the complexity of the kernel, while reaching its climax in the mid-90s in both cases, the Linux kernel complexity improved at a faster rate, dropping even lower than its starting point. This indicates a strong drive in the Linux community to refactor and remove technical debt, probably lead by key members in the kernel development team.

6 Towards an Initial Theory of Operating System Architecture Evolution

Our findings from the qualitative and quantitative analysis are interesting not just for the case of Unix, but for similar operating systems. Thus, they can form the basis to establish an initial theory on how the architecture of operating systems evolves. Building theories in Software Engineering has been argued, among others, as a necessary means to analytically generalize results, thus going beyond individual findings [126]. To build this theory, we follow the first four steps, as prescribed by Sjøberg et al. [127]. We thus we derive: *constructs*, which are the main entities of the theory; *propositions*, which establish relations between the constructs; *explanations*, which shed further light into the propositions; and *scope* which determines where the theory applies. The fifth step, which entails testing the theory through further empirical studies is regarded as future work.

The *constructs* in our case include the main concepts from the research questions, i.e., architecture decisions, evolution, system lifetime, features, size and complexity. They also extend to technical debt, conventions, portability, software ecosystems and third-party systems. This set of constructs is grounded in the collected data as described in Section 3.3 and comprises the main concepts that were derived during the data analysis (particularly Constant Comparison—see Section 3.4). Accordingly, the *scope* includes large, complex and long-lived operating systems.

The derived propositions and explanations are elaborated in the following sub-sections, grouped into those concerning: a) the form and pace of architectural evolution, b) the accumulation of architectural technical debt, and c) forces for architectural evolution. Each proposition is formulated as one sentence (in italics) and briefly elaborated, followed by a paragraph with the explanation.

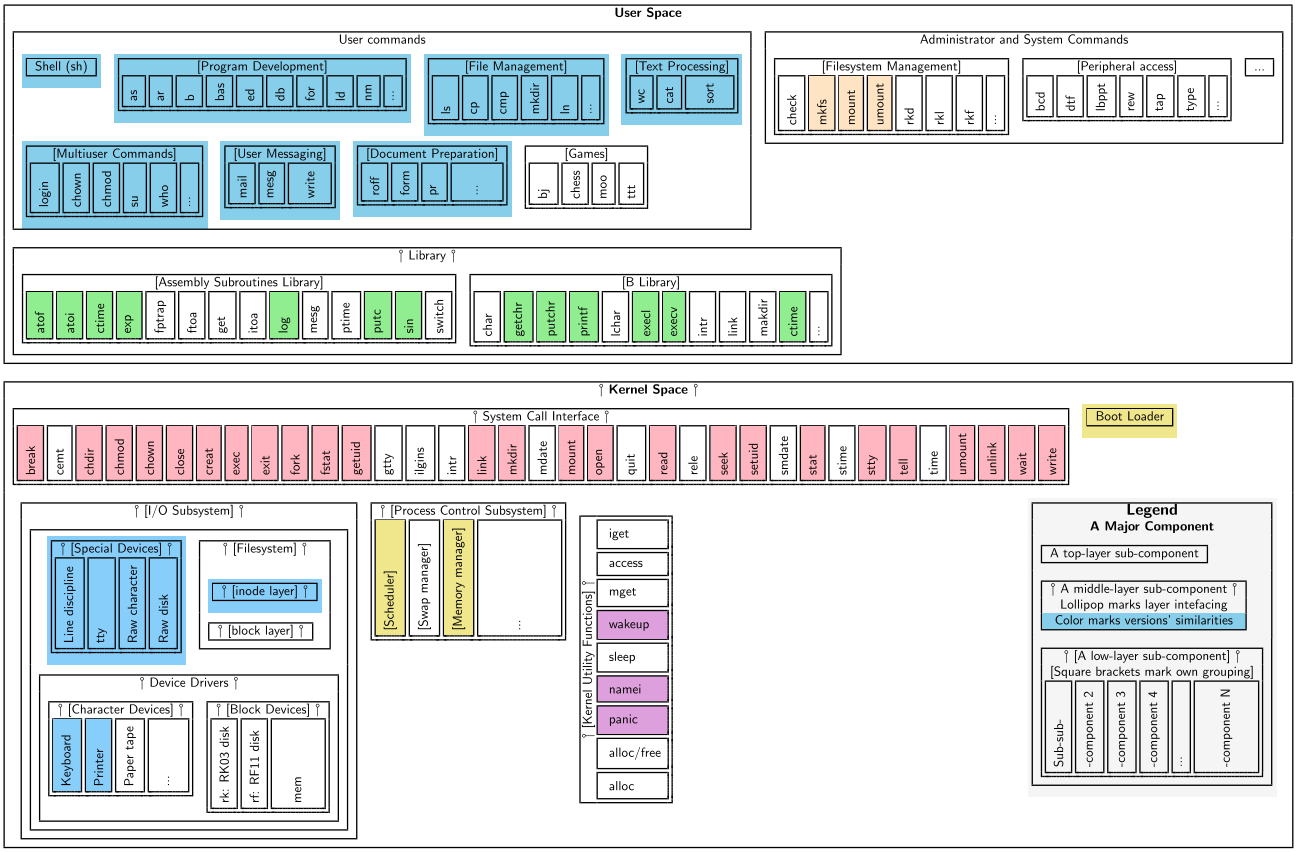

Fig. 5. High-level architecture of the first research edition (1972).

6.1 Form and Pace of Architectural Evolution

Proposition 1. *Many core architecture decisions are taken at the beginning of the system's lifetime.*

A surprising finding of our study was the large number of Unix-defining design decisions that were implemented right from the very early beginning. This can clearly be seen in the evolution timeline (Fig. 1). Despite the diminutive size of the PDP-7 and the First Research editions, they included the most important of the system calls still used today, the notion of devices as files, the abstraction of standard I/O, and a tree directory structure.

The *influence* of the early architectural decisions is also apparent if one compares the high level architecture (module view) of the First Edition architectural diagram (Fig. 5) with the system's current architecture (Fig. 6).[13] The system's first-level decomposition has remained essentially the same. The *permanence* of many early design decisions is illustrated through highlighted items in the two diagrams. Note that, as the current architecture diagram is drawn at a much coarser scale, many of the First Edition features appear in the current architecture grouped together under an entity with the same colour. For example, the *File I/O* system call box in Fig. 6 includes the *open*, *read*, *write*, *close* system calls depicted individually in Fig. 5, while the *math*, *stdio*, *stdlib*, and *time* parts of the C Standard library in Fig. 6 contain among others the colored library functions in Fig. 5.

13. We encourage readers to focus on the overall structure, because many may find the text labels illegible due to their small font size. Further details can be readily obtained by zooming in on the manuscript's digital version.

Explanation The developers of early Unix sought to "distill and simplify" [59] three powerful and influential operating systems: Multics, Project Genie, and CTSS [76], some of which had already suffered from the "second system syndrome" [128], [79, p. 463]. Consequently, the Unix developers' experience guided them to implement the system around a few key ideas with enduring value.

Proposition 2. *Most important architecture decisions survive over the system lifetime.*

The number of long-lived architectural design decisions in Unix is impressive. Of the 15,596^{L100} elements documented over the past half-century 12,043^{L101} (more than 75 percent) are still documented in the current edition of FreeBSD. Most deprecated commands offer functionality that is nowadays available through add-on packages (number factoring, form generation, voice synthesis, hyphenation, Fortran compilation) or deal with deprecated technology (GCOS and UUCP communication, DECtape handling). On the system call side, the few removed ones are mainly those that have been replaced by more general mechanisms. For example, the functionality of the Third Edition's signal handling calls—*cemt*(II), *fpe*(II), *ilgins*(II), and *intr*(II)—is nowadays provided by the single *sigaction*(2) call. In contrast, device drivers have seen a very high churn rate. This is to be expected due to big and visible changes in hardware device technologies; nobody nowadays uses punched card readers, paper tape punches, dataphones, or washing machine–sized 121MB RA80 disk units.

Moreover, we observed the longevity of not only explicit design decisions, but also implicit ones. Specifically, we saw

Fig. 6. High-level architecture of FreeBSD 11.0 (2017).

that implicit design decisions that are not part of a documented API can also survive over decades and even influence the design of other systems. For example, the virtual filesystem interface (Section 4.14) has been adopted by the Linux kernel [129, Chapter 13], while the device driver so-called strategy routine (Section 4.5) could also be found in the design of Linux device drivers [130, Section 14.4.3].

Explanation The longevity of architectural decisions is mainly due to the desire to maintain backward compatibility and the benefits derived from it. From as early as 1977 this was instituted through—initially informal and later formal—standardization. First, a committee sponsored by the AT&T Bell Laboratories Computer Technologies Area monitored and promoted the portability and evolution of the C programming language and associated libraries [53, p. 1687]. Later, Unix standardization was formalized through efforts such as POSIX [92], [131], [132] and the C language standards [133], [134], [135].

Proposition 3. *New architecture decisions are continuously made, further fueling architecture evolution.*

Despite the influence and permanence of the early architecture, the study also demonstrates that the Unix architecture continues to evolve significantly many years after the system's foundations have been cast into stone. For example, many important architectural design decisions of Unix, such as system portability, dynamic memory allocation, environment variables, language development tools, little languages, and static program analysis, first made their appearance in the Seventh Edition; ten years after the PDP-7 prototype was implemented. In more recent decades, Unix has continued to grow significantly in size and

complexity through the addition of large third-party subsystems (see Table 6) integrated to the system's core features.

Explanation The reason for the continuing evolution is, unsurprisingly, new requirements. These stem from the need to accommodate more sophisticated user programs, which appears to be mirrored in the rise of supported C library functions and system calls seen in Fig. 3, or support new hardware, which can be observed through the rising number of supported devices depicted in the same figure. Requirements can also arise from advances made by other operating systems—work aimed at keeping up with the Joneses, as it were.

Proposition 4. *The rate of architecture decisions declines over the system's lifetime.*

Despite evidence of continued architectural evolution, by looking at the elements listed in the evolution timeline (Fig. 1) it is also evident that the rate of it has slowed down over the system's lifetime in terms of new significant design decisions introduced. One can observe three major 'waves': the first comprising the Research Editions, which featured a significant number of major design decisions; the second and third in the 1990s and 2000s respectively, which featured fewer and fewer such significant design decisions.

Explanation Two plausible explanations can be given. First, architectural changes become more difficult as the system ages, due to the system's increased volume and complexity. For example, when pipes were introduced in the Third Research Edition, the few members of the Unix team worked overnight to convert most of the system's utilities into filters (see Section 4.4). Introducing such a change in a modern system would be orders of magnitude more difficult and complex. Second, due to the system's maturity, new major or even disruptive features are seldom required—to a large extent we are witnessing functional saturation.

6.2 Accumulation of Architectural Technical Debt

Proposition 5. *A major source of architecture technical debt is architecture decisions offering features that are either similar to existing ones or remain under-used.*

As one might expect in a system developed over half a century, our study also revealed symptoms of architectural technical debt. We use the definition of technical debt from a recent Dagstuhl seminar [136].

> "Technical debt consists of design or implementation constructs that are expedient in the short term, but set up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability."

Technical debt comes in many flavors; our proposition concerns two different types of technical debt that were predominantly observed in Unix.

The first type refers to adding functionality that is the same or similar to existing functionality, without removing the existing one or merging them into a single source. Retaining two or more competing facilities that provide analogous functionality hurts understandability and maintainability. Examples include:

- the proliferation of system calls that perform slightly different functions, such as the nine variants for reading data—*read*(2), *pread*(2), *readv*(2), *preadv*(2), *recv*(2), *recvfrom*(2), *recvmmsg*(2), *recvmsg*(2), *sctp_generic_recvmsg*(2)—and a similar number for writing data, or the 14 *...at* siblings of existing system calls—*bindat*(2), *connectat*(2), *fstatat*(2), *faccessat*(2), *linkat*(2), *mkdirat*(2), *mkfifoat*(2), *mknodat*(2), *openat*(2), *readlinkat*(2), *symlinkat*(2), *unlinkat*(2), *renameat*(2) [137];
- the support of multiple logging mechanisms: writing to plain files in `/var/log`, logging via *syslogd*(8), process accounting via *act*(2), and BSM auditing via *auditd*(8) (Section 4.27);
- the coexistence of the traditional user-group-others file permission settings, with access control lists (Section 4.24), and a separate mandatory access control framework (Section 4.25); and
- the coexistence of two multitasking primitives: threads and processes.

We found one striking example of this type of technical debt that relates to loss of conceptual integrity. An important innovation of the Unix operating system is the mapping of storage devices, terminals, communication links, and memory onto special files. According to the system's creators, this homogeneous treatment has three advantages: it makes the device I/O API similar to the file API; it allows ordinary programs to be used on special files by supplying their corresponding file names; and it reuses the existing file protection mechanism on special files [138, pp. 1909–1910]. Over time, competing approaches have breached the conceptual integrity of this approach by not using special files and thus losing the above advantages. For example, the monitoring and control of the system and its processes can be achieved following the special file approach, through the *procfs*(5) filesystem (Section 4.20). However, such functionality is also provided through system calls—*ptrace*(2), *getrusage*(2), *getrlimit*(2), and through the *dtrace*(1) system. Similarly, special files can be used to control the operating system's configuration, as is for example done through Linux's *sysfs* filesystem [129, 355–361]. However, most of this functionality is implemented through numerous system calls—e.g., *acct*(2), *adjtime*(2), *auditctl*(2), *getfsstat*(2), *gettimeofday*(2), *kenv*(2), *mincore*(2), *modfind*(2), *procctl*(2), *quotactl*(2), *settimeofday*(2), and also through the hierarchical but distinct *sysctl* interface (Section 4.16).

Explanation The main driver behind amassing similar competing features is a lack of ownership regarding the conceptual integrity of the whole system [79, p. 460]. As the evolution of Unix moves between groups and individuals, these may be more interested in leaving their mark through new functionality than in consolidating existing work and refactoring old code to work with incrementally improved features. This can be seen in Fig. 3, where the move from Bell Labs to Berkeley and then to FreeBSD is marked by an increase in the number of system calls. Furthermore, each generation of code stewards may be hesitant to radically change code of their predecessors. In addition, as the system is increasingly built by bringing together code developed by diverse teams to serve multiple projects,

it becomes very difficult to coordinate extensive refactoring changes.

The second type of technical debt has to do with complicated functionality that was offered but never quite used. This violates the YAGNIprinciple ('you aren't gonna need it') and incurs extra maintenance effort for functionality that is not actually in use. Removing this redundancy and cleaning up the system would remedy the technical debt. A typical example of this is the socket-based IPS with its large number of system calls (see Table 3).

Explanation This type of technical debt is almost always inadvertent: certain architectural decisions appear sound at a given time, but later become problematic because of changes in the technology or the application domain. For instance, the elaborate socket stream and datagram abstractions that were designed as part of the network protocol API in 4.2BSD (Section 4.12) were rendered irrelevant by the universal adoption of Internet protocols and the eclipse of competing technologies [139, p. 87]. However, the accompanying complexity still burdens the API. On the positive side, the networking API's generality allowed support for version 6 of the Internet Protocol to be introduced without requiring any new system calls.

Proposition 6. *The architecture technical debt is systematically paid back despite increasing system size and complexity.*

The evidence of technical debt we found in Unix is substantial and it does hurt the system's maintainability and evolvability. However, for a system of its size, complexity, and age, the technical debt of Unix is impressively limited. Usually the growth in size and complexity over a long period of time results in incurring technical debt at an increasing rate; thus most systems of similar size have become 'big balls of mud'. On the contrary, Unix has maintained comparatively high internal quality and does not manifest many architectural 'quick fixes' or 'workarounds'. Evidence of corrective action following the accumulation of technical debt is visible in Fig. 4, where increases in cyclomatic complexity are followed by a subsequent decrease.

Explanation One could argue that the system's high overall internal quality may be due to the dedication and exceptional talent of the developers who worked on the system, coupled with the lack of commercial pressure to follow shortcuts for the sake of expediency. However, when the quality of the FreeBSD Unix kernel is compared against that of three other systems (Linux, Solaris, and the Windows Research Kernel) they all appear to be at similar levels [123]. We therefore argue that the main reason for the low technical debt is a natural selection process: the size (eight million lines in the case of FreeBSD) and complexity of a modern operating system kernel as well as the reliability requirements [51, p. 1960] are such that sub-par quality is either weeded out or the corresponding system is abandoned. The way stringent reliability requirements force high internal quality can be observed in Fig. 4, where the mean cyclomatic complexity lowers as we move from stand-alone user commands, to the C library used by all of them, to the large monolithic kernel on which everything depends. A counter example is the case of Multics, which Thompson has characterized as overdesigned, overbuilt, and close to unusable [79, p. 463]; it never thrived.

6.3 Forces of Architectural Evolution

An architecture is driven by requirements but also by forces, such as technology, organization culture, or design philosophy. The following propositions concern such forces.

Proposition 7. *The preference for conventions instead of enforcement facilitates evolution by reducing effort and offering flexibility.*

The system's developers often established and followed lightweight conventions rather than implementing rigid enforcement mechanisms. In early editions, such conventions included the grouping of related files through their names, the setup of identifier name spaces through a prefix (Section 4.1), the processing of directories as files, the creation of a navigable tree directory structure through arbitrary file links, the adoption of simple text files as a common data format, and the use of documented file formats as a program coupling mechanism (Section 4.2). In the Seventh Research Edition (Section 4.8) the same principle was applied in the setup of environment variables as key-value pairs and the detailed documentation of the system's directory layout.

Explanation The practice of convention over enforcement minimized the system's implementation effort and promoted experimentation. Problems arising through undisciplined behavior were addressed when they truly became insurmountable [76]. This practice was a major contributing factor for the unusually rich functionality compared to their code size that early Unix systems provided. The approach's flexibility also allowed the effortless adaptation and morphing of the conventions to changing needs. We argue that, with good taste and some discipline, such an approach can yield superior outcomes than what will result from a rigid enforcement mechanism designed in advance for fuzzy requirements. Once more, agile, descriptive approaches thrive over prescriptive ones.

Proposition 8. *Portability, due to its inherent complexity, is a key driver of evolution.*

Another major force that has been driving the software architecture is portability. A key contribution of Unix was the implementation of an operating system that could be easily ported between different machine architectures. In the words of Johnson and Ritchie [54] the system should be "easily portable unchanged" between different hosts, but also "easy to change" so as "to take full advantage of machines much more powerful along many possible dimensions". The hard portability requirements between diverse hardware architectures and devices forced the system's designers to adopt numerous sophisticated methods of abstraction in order to tame the associated complexity.

Explanation Portability has driven architecture evolution mostly through the use of layers used to hide non-portable functionality behind portable abstractions [50]. Early on, the need for portability influenced the design of the system, the C programming language, the portable C library (Section 4.7), as well as header files (Section 4.8) and static analysis tools (Section 4.8) [54]. Furthermore, a portability approach adopted by Unix's designers was to define abstract machine models for C and Unix [54, pp. 2041–2046]. During the long evolution of Unix, many architecture decisions were made to facilitate portability, e.g., the introduction of the *vnode*

TABLE 5
Major FreeBSD Third-Party Influences

Source	Commits	LoC
TrustedBSD Project	1,215	413,339
NetBSD	1,166	2,665,223
OpenBSD	726	113,195
KAME	451	163,874
Semihalf sp.	330	214,289
DragonflyBSD	179	675,906
Linux	151	109,600
Qualcomm Atheros, Inc.	139	46,608
ABT Systems Ltd	133	8,704
Juniper Networks, Inc.	125	66,971
NetApp, Inc.	120	8,044
Illumos	97	56,618
OpenSolaris	95	125,503
Wheel Systems, Inc.	81	3,552
Yandex LLC	64	3,630
Apple, Inc.	58	13,378

TABLE 6
Major Third-Party Subsystems in FreeBSD 11.1

Subsystem	kLoC	LoC %
llvm	3,413	10.81
gcc	1,576	4.99
binutils	1,111	3.52
ntp	873	2.76
heimdal	756	2.39
openssl	661	2.09
subversion	558	1.77
gdb	488	1.54
groff	438	1.39
ofed	404	1.28
libstdc++	394	1.25
wpa	380	1.20
libarchive	310	0.98
sqlite3	281	0.89
ncurses	242	0.77
netbsd-tests	239	0.76
zfs	230	0.73
dtrace	205	0.65
sendmail	205	0.65
unbound	189	0.60
gcclibs	187	0.59
openssh	179	0.57
byacc	150	0.48
libc++	142	0.45
tcpdump	123	0.39
compiler-rt	121	0.38
ldns	115	0.36
tcsh	109	0.34
openbsm	102	0.32
elftoolchain	101	0.32

interface for abstracting diverse filesystems (Section 4.14), and the modern CAM (Section 4.22), *negraph* (Section 4.23), and GEOM (Section 4.25) stacks.

Proposition 9. *A sophisticated ecosystem of other operating systems and third parties constantly shapes the architecture evolution.*

At the organizational level, the architecture evolution of Unix systems in general and the FreeBSD lineage studied here in particular has been influenced by technology developed by other related systems and organizations. Fig. 7[14] depicts how diverse Unix variants and releases cross-pollinated one another through the adoption of code. In addition, the ideas behind Unix have influenced even more operating systems that were independently developed, including Android, GNU/Linux, Microsoft Windows, Minix [140], MS-DOS, QNX, and Z/OS. Some of this influence was applied through formal standardization via the POSIX effort [92], [131], [132] and the Single UNIX Specification.

An early influencer of Unix was DARPA, which funded CSRG to produce 4BSD (see Section 4.11). This undertaking's success brought increased scrutiny, criticism regarding the system's performance, and, as a response, a systematically tuned kernel released as 4.1BSD [78].

Further acknowledged third party software contributions can be traced back to 4.3BSD Tahoe (see Section 4.13). More details regarding influences from diverse systems and organizations can be derived by looking at individual configuration management system code commits. Since 1994 commit messages in the systems studied here have often included an "`Obtained From:`" header, which allowed us to track direct influences via the adoption of code. In total we found[L104] 7,685 such commits, from 1,283 sources, totaling 7,742,678 code lines. Sources with more than 50 commits each, are listed in Table 5.[L102,L103] We see that FreeBSD has been mainly influenced by its close siblings, such as NetBSD, OpenBSD, and DragonflyBSD, as well as closely affiliated projects, such as the TrustedBSD and KAME projects. Furthermore, influencers also include companies using FreeBSD, such as Semihalf, Juniper, NetApp, Yandex, Wheel Systems, and Apple. Finally, we also see influence from systems that are less closely related to FreeBSD, such as Linux, Illumos, and OpenSolaris. Additional third-party influence comes from wholesale-integrated components (Table 6) described in the next proposition.

The architecture evolution of Unix was also influenced over time through many non-technical decisions and developments. Chief among them were those associated a) with source code availability, which initially promoted third-party contributions and later led to organizations built around open source software development, b) the development of competing versions (Fig. 7) and (mostly uninspired) efforts to combine them, and c) the movement of people between organizations [79, p. 454], which resulted in a cross-pollination of ideas.

Explanation A significant part of developing operating systems takes place among a family of systems derived from the same source code base (Fig. 7) or influenced by the same key ideas (first column of Table 5). To remain compatible and competitive with other operating systems, the FreeBSD team routinely imports code from them (Table 5). Furthermore, its permissive distribution license allows FreeBSD elements to be easily reused in derived systems and in related development efforts, such as Apple's macOS.

Non-technical factors were influencing Unix right from its birth. In the 1970s AT&T was still operating under a 1956 "consent decree" [141]. Under its terms, the Bell Labs

14. Based on a diagram by Eraserhead1, Infinity0, and Sav_vas, licensed under Cc BY-SA 3.0, via Wikimedia Commons.

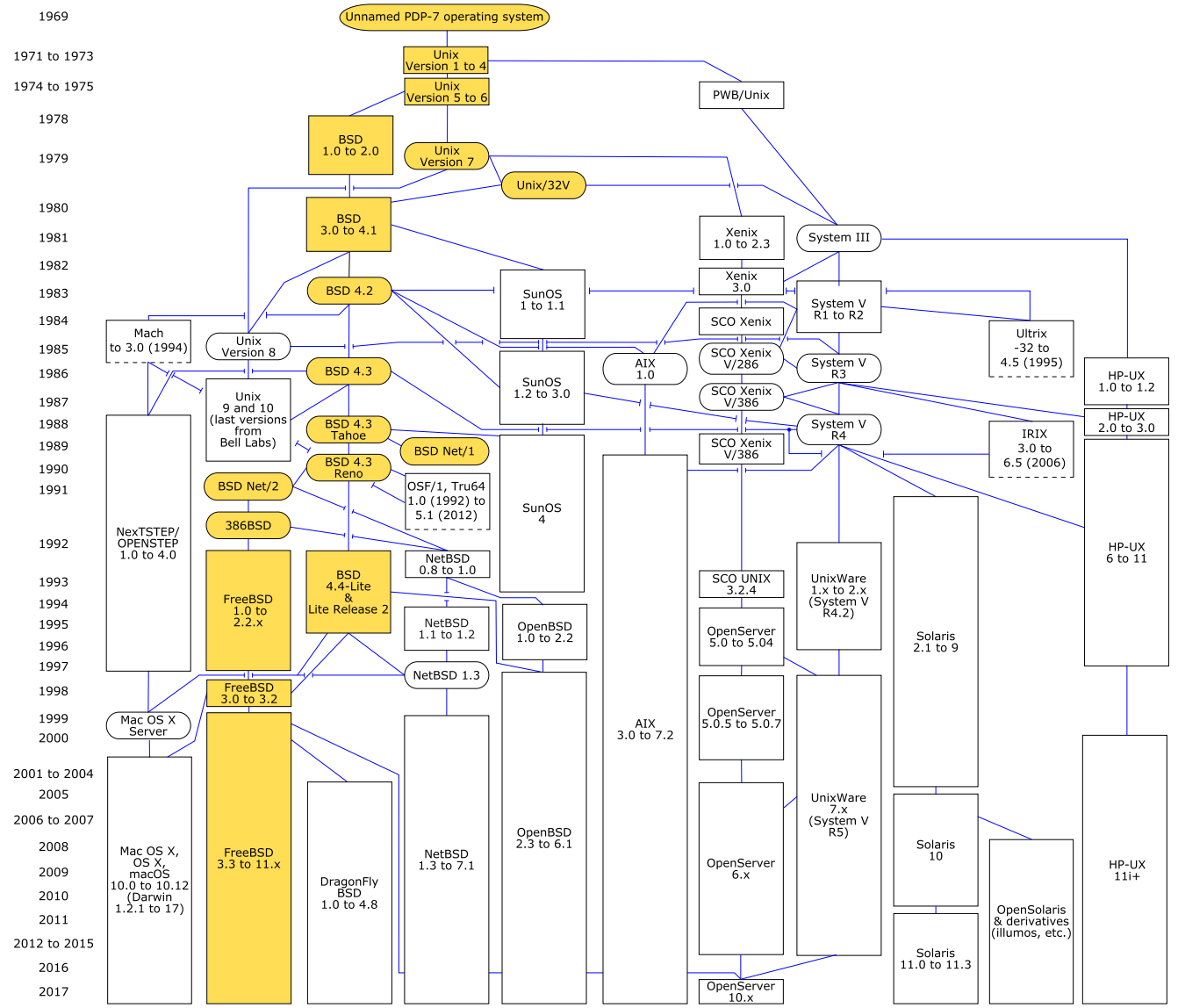

Fig. 7. A simplified diagram of Unix variants and releases related through code. The highlighted elements form this study's examined lineage.

owners, Western Electric and AT&T, were prohibited from manufacturing and offering non-telecommunications equipment and services. Consequently, AT&T could not market or license Unix for profit; Unix was initially liberally licensed royalty-free through simple letter agreements [77, p. 60], and its source code became widely available. This allowed staff at universities around the world to study its code and contribute improvements. AT&T's legal restrictions also left ample room for the development of competing versions of Unix from organizations such as USG (AT&T's Unix Support Group), Microsoft (XENIX), Berkeley (BSD), and tens of hardware vendors [77, p. 209–210]. Many companies lacked resident experts to act as 'arbiters of taste' [77, p. 211] in the place of the original Unix developers. As a result, companies involved in the so-called 'Unix wars' [77, p. 225] between competing implementations were often aggressively and indiscriminately piling up features, which were haphazardly 'taped together' [77, p. 211]. Then, in the 1980s and 1990s AT&T's licensing terms became more intricate and restrictive, limiting the availability of Unix source code [142], which was carefully guarded as a trade secret [93, p. 20]. These restrictions led Berkeley's CSRG and others to work on open source implementations of Unix, and the emergence of a structure that was conductive to open source development.

Proposition 10. *The adoption of third-party subsystems facilitates evolution through reusability but incurs technical debt.*

Another observed force has been the adoption of many large subsystems, which are developed by independent efforts and periodically integrated into the released versions. Table 6[L105] lists current ones whose size exceeds one hundred thousand lines of code (including documentation and tests). With the exception of DTrace and ZFS, which are deeply integrated within the FreeBSD source code tree, the other 90[L106] subsystems reside in two separate directories[S89,S90] and can be easily upgraded as new upstream versions are released. In contrast to the FreeBSD *ports*(7), the subsystems in these directories form an integral part of the operating system, and are typically required for its construction and operation. Many of these subsystems offer functionality that was in the past developed within the system's boundaries. This practice

outsources the development of key system parts, leaving to the FreeBSD core team the responsibility for choosing among alternative implementations, such as the choice between the GCC or LLVM as the compiler infrastructure.

Explanation The reasoning behind adopting third-party subsystems is simple: the increasing size and complexity of these subsystems entails substantial effort savings for the FreeBSD and multiple other operating system distributions, such as GNU/Linux and macOS, that reuse them. On the other hand, a downside of this approach is that the third-party subsystems are developed to utilize only the least common denominator functionality of all operating systems that host them. Consequently, each operating system that adopts them also inherits some technical debt: providing functionality that might be required by some third-party packages requires the coordinated addition of this facility by all operating systems where the third-party software runs. This makes it more likely for each third party tool to duplicate some required functionality (resulting in redundancy) in a slightly different manner (damaging understandability).

Proposition 11. *Large subsystems form their own architecture, independently of the architecture of the encompassing system.*

We have observed a strong force towards federating the architecture. Many large subsystems, such as the Graph-based Kernel Networking and User Library (*netgraph*—Section 4.23), OpenSSL Framework (SSL—Section 4.24), Mandatory Access Control (MAC—Section 4.25), Pluggable Authentication Module (PAM—Section 4.25), Modular Disk I/O Request Transformation Framework (GEOM—Section 4.25), Basic Security Module Auditing (BSM—Section 4.27), Zettabyte Filesystem (ZFS—Section 4.28), and Dynamic Tracing (*DTrace*—Section 4.29), have their own architecture, with distinct principles, layers, components, plug-in mechanisms, subcommands, design patterns, and conventions. Some of these constituent structures can be observed in Fig. 6.

Explanation The main reason for this phenomenon is that the size and complexity of Unix may have grown way beyond the point by which it can be maintained as a monolith (see Table 1). In addition, many subsystems are now independently developed by third parties (see Table 6). This makes it difficult to coordinate their architecture with that of the FreeBSD core.

7 Threats to Validity

Our study is subject to limitations that can be categorized into construct validity, external validity, and reliability following the guidelines of Runeson et al. [67]. Internal validity is not a concern for this study because we did not examine causal relations [67].

7.1 Construct Validity

This type of validity concerns to what extent the studied items really represent what the researchers aim at according to the research questions [67]. In our case, the research questions inquire about the main architectural design decisions of Unix over time, as well as the evolution of the system's size and complexity. Regarding the former, we classified as architectural design decisions some of the most significant architectural components, connectors, patterns, and principles [69], [72]. To mitigate a potential mis-interpretation of architecture design decisions, the first author independently performed the constant comparison, and the second author controlled the coded design decisions in a second iteration. In case of disagreement, the two authors discussed until a consensus was reached; several architectural design decisions were removed as a result of this process.

Another potential risk regards whether we were exhaustive during data collection: i.e., whether we may have missed any significant architectural design decision and at the same time whether all reported architectural design decisions are significant. This risk cannot be completely mitigated as the significance of architecture design decisions is to a large extent subjective. However, our data source triangulation did help in spotting those architectural design decisions that were given attention by more than one data source: decisions derived from the code and the Unix documentation that were also prominently discussed in books and recollections of Unix pioneers, were given priority in our selection process. Furthermore, even if we cannot claim exhaustiveness, we used an extensive amount of data sources to increase the chances of reaching correct decisions.

Regarding the quantitative results, the size of the system is measured in terms of number of features (e.g., user commands or system calls), and complexity is measured in terms of cyclomatic complexity. While these may not be unique ways to measure size and complexity, they are certainly valid ones [124]. Moreover, both the architectural feature data set and the tool used for measuring cyclomatic complexity are based on published peer-reviewed research [15], [74] thus partially mitigating threats associated with the validity of the measurement instrument.

7.2 Reliability

This type of validity concerns to what extent the data collection and analysis depend on the actual researchers. This risk has been partially mitigated as the coding was performed iteratively by the first author, with the second author controlling the results. However we need to acknowledge that the first author has decades of Unix experience. While this has been instrumental in understanding the details of the object of study and subsequently performing the coding, it may have introduced a certain bias on selecting the architectural design decisions (an expert may not be able to look at the system objectively and may be biased regarding the importance of the different design decisions). Again, data source triangulation has helped to partially deal with this bias, as we made sure that all selected architectural design decisions were described in more than one data source—typically documentation and source code. Moreover the reliability of the study is strengthened by being open and explicit about the process of data collection and analysis, and publishing online or in this paper's supplement all used tools and data.

7.3 External Validity

This type of validity concerns whether the findings can be generalized to other cases and contexts [67]. This study is rather unique in the sense that it does not aim at providing a general conclusion about a population (i.e., category of

systems or an application domain). In addition, the history of Unix is exceptional, with numerous stakeholders and environments influencing its development, therefore the validity of extending any findings to other systems is debatable. Consequently, we do not claim that either our qualitative or our quantitative findings should also hold for other large operating systems. However Unix has been the dominant operating system for decades, and its development has strongly influenced subsequent widely-used operating systems, such as GNU/Linux, macOS, and Android. In that sense, particularly the qualitative results regarding the architecture design decisions of Unix are relevant for other operating systems, because they provide many of the significant design decisions and accompanying rationale.

8 Conclusion

We looked closely into the evolution of Unix from an architectural perspective by examining 30 core releases from the PDP-7 Research Edition to FreeBSD 11. We triangulated data sources (source code, documentation, research papers and books, pioneers' recollections) to extract valid and up-to-date data. We have procured and produced a wealth of data and made it available to the community [58], [74] for further studies.

Our analysis yielded both qualitative and quantitative results. The qualitative examination allowed us to establish a timeline with the most important milestones that shaped the Unix architecture; those milestones are detailed as components, connectors, patterns and principles as well as other key architecture decisions. We also discussed the rationale of those decisions and how they affected future developments. Through the quantitative analysis we showed the trends on size growth for the seven principal feature types (user commands, system calls, libraries etc.), as well as complexity. We found a uniform growth in size but also some outliers, for which we conjectured corresponding explanations. We discovered that cyclomatic complexity grew at first, but was subsequently reduced especially for the library and the kernel, where code quality matters the most. Finally, we put the Unix evolution in context. First, by comparing the number of current FreeBSD features with that of five other current operating systems, we found a similar magnitude, indicative of their essential complexity. Second, by contrasting the cyclomatic complexity with the GNU coreutils, C library and the Linux kernel, we observed overall an inverted U-curve with some marked differences.

Based on the results, we ventured on generalizing them by developing an initial theory on the architecture evolution of operating systems; the theory is comprised of eleven propositions and their corresponding explanations. Numerous early design decisions survive the test of time and are still visible decades after their introduction. Nevertheless, innovation continues uninterruptedly to accommodate changes in computing technology and networking, although with a slower pace as decades go by. Furthermore, architectural technical debt creeps in mostly by retaining two or more functionally-equivalent facilities, but also by offering complicated under-used functionality that adds maintenance effort without much actual value. However, architectural technical debt does not reach critical levels, as its remediation is systematic despite increasing size and complexity. Moreover, the philosophy of lightweight informal mechanisms instead of formal prescriptive ones, the drive for portability, and an intricate ecosystem of other operating systems and third parties are factors that shape the architectural evolution of large, long-lived operating systems. Nevertheless, given the current size and complexity of Unix, its evolution can only be sustained through the adoption of third-party subsystems, while many large subsystems have formed an architecture of their own.

Looking forward, progress in hardware and applications will continue to exert evolutionary pressure on Unix's architecture on several fronts. Flash storage and universal memory computing change how secondary storage is used and addressed; CPUs with tens of cores require support for finer-grained parallelism; GPU computing calls for appropriate high-level abstractions; deep learning methods change the nature of computation by elevating data into its main determinant; security and privacy demand fresh approaches both at the data center and at the edges; mobile and IOT devices impose demanding constraints on computing resources, power, and real-time performance. In addition, the operating system's large code base and the backward compatibility requirements of existing applications hinder radical changes. In short, the Unix operating system architects have their work cut out.

Acknowledgments

The authors thank the members of the Unix Heritage Society[15] and in particular Warren Toomey and Kirk McKusick for preserving and making available many important early Unix artifacts. They also thank the TUHS mailing list participants for their input and encouragement regarding this research. The authors are especially grateful to the anonymous reviewers and to Kirk McKusick, George Neville-Neil, Warren Toomey, and Alexios Zavras, for their detailed and insightful comments regarding earlier versions of this document. The research described has been carried out as part of the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732223.

References

[1] M. M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle," *J. Syst. Softw.*, vol. 1, pp. 213–221, Sep. 1984.

[2] I. Herraiz, D. Rodriguez, G. Robles, and J. M. Gonzalez-Barahona, "The evolution of the laws of software evolution: A discussion based on a systematic literature review," *ACM Comput. Surv.*, vol. 46, no. 2, pp. 28:1–28:28, Dec. 2013.

[3] D. L. Parnas, "Software aging," in *Proc. 16th Int. Conf. Softw. Eng.*, 1994, pp. 279–287.

[4] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Trans. Softw. Eng.*, vol. 27, no. 1, pp. 1–12, Jan. 2001.

[5] P. Avgeriou, P. Kruchten, R. L. Nord, I. Ozkaya, and C. Seaman, "Reducing friction in software development," *IEEE Softw.*, vol. 33, no. 1, pp. 66–73, Jan. 2016.

[6] L. Hatton, D. Spinellis, and M. van Genuchten, "The long-term growth rate of evolving software: Empirical results and implications," *J. Softw.: Evolution Process*, vol. 29, no. 5, pp. e1847–n/a, 2017, e1847 smr.1847.

15. https://www.tuhs.org/

[7] S. Koch, "Software evolution in open source projects—A large-scale investigation," *J. Softw. Maintenance Evolution: Res. Practice*, vol. 19, no. 6, pp. 361–382, 2007.

[8] H. Breivold, M. Chauhan, and M. Babar, "A systematic review of studies of open source software evolution," in *Proc. 17th Asia Pacific Softw. Eng. Conf.*, 2010, pp. 356–365.

[9] C. A. Conley and L. Sproull, "Easier said than done: An empirical investigation of software design and quality in open source software development," in *Proc. 42nd Hawaii Int. Conf. Syst. Sci.*, 2009, pp. 1–10.

[10] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-source software products," *IEEE Trans. Softw. Eng.*, vol. 30, no. 4, pp. 246–256, Apr. 2004.

[11] A. Capiluppi, A. E. Faria, and J. F. Ramil, "Exploring the relationship between cumulative change and complexity in an open source system," in *Proc. 9th Eur. Conf. Softw. Maintenance Reengineering*, 2005, pp. 21–29.

[12] M. Aram, S. Koch, and G. Neumann, "Long-term analysis of the development of the open ACS community framework," in *Proc. Open Source Solutions Knowl. Manage. Technological Ecosystems*, 2017, pp. 111–145.

[13] M. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Proc. Int. Conf. Softw. Maintenance*, 2000, pp. 131–142.

[14] A. Israeli and D. G. Feitelson, "The Linux kernel as a case study in software evolution," *J. Syst. Softw.*, vol. 83, no. 3, pp. 485–501, Mar. 2010.

[15] D. Spinellis, P. Louridas, and M. Kechagia, "The evolution of C programming practices: A study of the Unix operating system 1973–2015," in *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 748–759.

[16] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Manage. Sci.*, vol. 57, no. 7, pp. 1015–1030, 2006.

[17] A. Israeli and D. G. Feitelson, "The Linux kernel as a case study in software evolution," *J. Syst. Softw.*, vol. 83, no. 3, pp. 485–501, 2010.

[18] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Jul. 1976.

[19] D. G. Feitelson, "Perpetual development: A model of the Linux kernel life cycle," *J. Syst. Softw.*, vol. 85, no. 4, pp. 859–875, 2012.

[20] P. Behnamghader, D. M. Le, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "A large-scale study of architectural evolution in open-source software systems," *Empirical Softw. Eng.*, vol. 22, no. 3, pp. 1146–1193, Jun. 2017.

[21] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger, "Analysing software repositories to understand software evolution," in *Software Evolution*. Berlin, Germany: Springer, 2008, pp. 37–67.

[22] R. Wettel and M. Lanza, "Visual exploration of large-scale system evolution," in *Proc. 15th Working Conf. Reverse Eng.*, Oct. 2008, pp. 219–228.

[23] E. Bouwers, J. P. Correia, A. v. Deursen, and J. Visser, "Quantifying the analyzability of software architectures," in *Proc. 9th Working IEEE/IFIP Conf. Softw. Archit.*, Jun. 2011, pp. 83–92.

[24] S. C. Johnson and B. W. Kernighan, "The programming language B," Bell Laboratories, Murray Hill, NJ, USA, Computer Science Tech. Rep. 8, Jan. 1977. [Online]. Available: http://web.archive.org/web/20180831015050/https://www.bell-labs.com/usr/dmr/www/bintro.html

[25] S. C. Johnson, "Lint, a C program checker," Bell Laboratories, Murray Hill, NJ, USA, Comput. Sci. Tech. Rep. 65, Dec. 1977. [Online]. Available: http://web.archive.org/web/20160412071448/http://files.cnblogs.com:80/files/bangerlee/10.1.1.56.1841.pdf

[26] B. W. Kernighan and L. L. Cherry, "A system for typesetting mathematics," Bell Laboratories, Murray Hill, NJ, USA, Comput. Sci. Tech. Rep. 17, May 1974. [Online]. Available: https://web.archive.org/web/20151029232442/http://tex.loria.fr/divers/unix-eqn1.ps.gz

[27] J. F. Maranzano and S. R. Bourne, "A tutorial introduction to ADB," Bell Laboratories, Murray Hill, NJ, USA, Comput. Sci. Tech. Rep. 62, May 1977. [Online]. Available: https://web.archive.org/web/20040324013641/https://wolfram.schneider.org/bsd/7thEdManVol2/adb/adb.pdf

[28] S. C. Johnson, "Yacc—yet another compiler-compiler," Bell Laboratories, Murray Hill, NJ, Comput. Sci. Tech. Rep. 32, Jul. 1975. [Online]. Available: https://web.archive.org/web/20170810013946/https://www.isi.edu/pedro/Teaching/CSCI565-Fall15/Materials/Yacc.pdf

[29] M. E. Lesk, "Lex—a lexical analyzer generator," Bell Laboratories, Murray Hill, NJ, Comput. Sci. Tech. Rep. 39, Oct. 1975, [Online]. Available: https://web.archive.org/web/20040324060316/http://wolfram.schneider.org:80/bsd/7thEdManVol2/lex/lex.pdf

[30] B. W. Kernighan, "UNIX for beginners," Bell Laboratories, Murray Hill, NJ, USA, Comput. Sci. Tech. Rep. 75, Feb. 1979. [Online]. Available: https://web.archive.org/web/20170711222622/http://wolfram.schneider.org/bsd/7thEdManVol2/beginners/beginners.pdf

[31] R. Morris and K. Thompson, "Password security: A case history," Bell Laboratories, Murray Hill, NJ, USA, Comput. Sci. Tech. Rep. 71, Apr. 1978. [Online]. Available: https://web.archive.org/web/20180317102420/http://wolfram.schneider.org/bsd/7thEdManVol2/password/password.pdf

[32] S. I. Feldman, "Make—A program for maintaining computer programs," Bell Laboratories, Murray Hill, NJ, USA, Comput. Sci. Tech. Rep. 57, Apr. 1977. [Online]. Available: https://web.archive.org/web/20040805040247/http://wolfram.schneider.org:80/bsd/7thEdManVol2/make/make.pdf

[33] B. W. Kernighan, "A typesetter-independent TROFF," Bell Laboratories, Murray Hill, NJ, USA, Comput. Sci. Tech. Rep. 97, 1982.

[34] AT&T, Ed., UNIX System Readings and Applications, vol. I. Englewood Cliffs, NJ, USA: Prentice Hall, *1978 (Bell Syst. Tech. J.*, vol. 57, no. 6, Jul./Aug. 1978).

[35] AT&T, Ed., UNIX System Readings and Applications, vol. II. Englewood Cliffs, NJ, USA: Prentice Hall, *1987 (AT&T Bell Laboratories Tech. J.*, vol. 63, no. 8, Oct. 1984).

[36] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Commun. ACM*, vol. 17, no. 7, pp. 365–375, Jul. 1974.

[37] K. L. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984.

[38] B. W. Kernighan, "PIC—A language for typesetting graphics," *Softw.: Practice Exp.*, vol. 12, pp. 1–21, 1982.

[39] J. L. Bentley, L. W. Jelinski, and B. W. Kernighan, "CHEM—A program for phototypesetting chemical structure diagrams," *Comput. Chemistry*, vol. 11, no. 4, pp. 281–297, 1987.

[40] R. Pike and K. Thompson, "Hello world," in *Proc. USENIX Tech. Conf. Proc.*, Winter 1993, pp. 43–50.

[41] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from Bell Labs," *Comput. Syst.*, vol. 8, no. 2, pp. 221–254, 1995.

[42] O. Babaoglu and W. Joy, "Converting a swap-based system to do paging in an architecture lacking page-referenced bits," in *Proc. 8th ACM Symp. Operating Syst. Principles*, 1981, pp. 78–86.

[43] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *ACM Trans. Comput. Syst.*, vol. 2, no. 3, pp. 181–197, Aug. 1984.

[44] R. Sandberg, "The design and implementation of the Sun network file system," in *Proc. USENIX Assoc. Conf. Proc.*, Jun. 1985, pp. 119–130.

[45] W. F. Jolitz and L. G. Jolitz, "Porting UNIX to the 386: A practical approach. Designing a software specification," *Dr. Dobb's J.*, vol. 16, no. 1, Jan. 1991.

[46] W. R. Stevens and J.-S. Pendry, "Portals in 4.4BSD," in *Proc. USENIX 1995 Tech. Conf. Proc.*, Jan. 1995, pp. 1–1.

[47] M. K. McKusick and G. R. Ganger, "Soft updates: A technique for eliminating most synchronous writes in the fast filesystem," in *Proc. USENIX Annu. Tech. Conf. Freenix Track*, Jun. 1999, pp. 1–18.

[48] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The zettabyte file system," in *Proc. 2nd Usenix Conf. File Storage Technol.*, Apr. 2003.

[49] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2004, pp. 15–28.

[50] D. Spinellis, "Another level of indirection," in *Beautiful Code: Leading Programmers Explain How They Think*, A. Oram and G. Wilson, Eds. Sebastopol, CA, USA: O'Reilly and Associates, 2007, ch. 17, pp. 279–291.

[51] D. M. Ritchie, "A retrospective," *Bell Syst. Tech. J.*, vol. 56, no. 6, pp. 1947–1969, Jul./Aug. 1978.

[52] K. Thompson, "UNIX time-sharing system: UNIX implementation," *Bell Syst. Tech. J.*, vol. 56, no. 6, pp. 1905–1929, Jul/.Aug. 1978.

[53] L. Rosler, "The evolution of C—Past and future," *Bell Syst. Tech. J.*, vol. 63, no. 8, pp. 1685–1699, Oct. 1984.

[54] S. C. Johnson and D. M. Ritchie, "Portability of C programs and the UNIX system," *Bell Syst. Tech. J.*, vol. 57, no. 6, pp. 2021–2048, Jul./Aug. 1978.

[55] D. M. Ritchie, "The evolution of the UNIX time-sharing system," *AT&T Bell Laboratories Tech. J.*, vol. 63, no. 8, pp. 1577–1593, Oct. 1984.

[56] W. Toomey, "The restoration of early UNIX artifacts," in *Proc. USENIX Annu. Tech. Conf.*, 2009, pp. 20–26.

[57] W. Toomey, "First edition unix: Its creation and restoration," *IEEE Ann. History Comput.*, vol. 32, no. 3, pp. 74–82, Jul.-Sep. 2010.

[58] D. Spinellis, "A repository of Unix History and evolution," *Empirical Softw. Eng.*, vol. 22, no. 3, pp. 1372–1404, 2017.

[59] W. Toomey, "Unix: Building a development environment from scratch," in *Reflections on Operating Systems—Historical and Philosophical Aspects*, L. Demol and G. Primiero, Eds. New York, NY, USA: Springer, 2017.

[60] J. Lions, *Lions' Commentary on Unix 6th Edition with Source Code*. Peer-to-Peer Communications, San Jose, CA, USA, 1996.

[61] M. J. Bach, *The Design of the UNIX Operating System*. Englewood Cliffs, NJ, USA: Prentice Hall, 1986.

[62] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*. Boston, MA, USA: Addison-Wesley, 1988.

[63] M. K. McKusick, K. Bostic, and M. J. Karels, *The Design and Implementation of the 4.4BSD Unix Operating System*. Reading, MA, USA: Addison-Wesley, 1996.

[64] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Reading, MA, USA: Addison-Wesley, 2004.

[65] M. K. McKusick, G. Neville-Neil, and R. N. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed. Reading, MA, USA: Addison-Wesley Professional, 2014.

[66] E. I. Organick, *The Multics System: An Examination of its Structure*. Cambridge, MA, USA: MIT Press, 1972.

[67] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*, 1st ed. Hoboken, NJ, USA: Wiley Publishing, 2012.

[68] V. Basili, C. Caldiera, and D. H. Rombach, "Goal question metric paradigm," in *Encyclopedia of Software Engineering*. New York, NY, USA: Wiley, 1994, vol. 2, pp. 528–532.

[69] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Hoboken, NJ, USA: Wiley Publishing, 2009.

[70] J. Tyree and A. Akerman, "Architecture decisions: Demystifying architecture," *IEEE Softw.*, vol. 22, no. 2, pp. 19–27, Mar. 2005.

[71] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Hoboken, NJ, USA: Wiley, 1996.

[72] N. Harrison, P. Avgeriou, and U. Zdun, "Using patterns to capture architectural decisions," *IEEE Softw.*, vol. 24, no. 4, pp. 38–45, Jul./Aug. 2007.

[73] J. Singer, S. E. Sim, and T. C. Lethbridge, "Software engineering data collection for field studies," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. London, U.K.: Springer, 2008, pp. 9–34.

[74] D. Spinellis, "Documented Unix facilities over 48 years," in *Proc. 15th Conf. Mining Softw. Repositories*, May 2018, pp. 58–61.

[75] M. McIlroy, "Interview with Michael S. Mahoney," Aug. 1989, current Dec. 2018. Archived doi: 10.5281/zenodo.2525529. [Online]. Available: https://www.princeton.edu/ hos/mike/transcripts/mcilroy.htm

[76] K. Thompson, "Interview with Michael S. Mahoney," Jun. 1989, current Dec. 2018. Archived doi: 10.5281/zenodo.2525529. [Online]. Available: https://www.princeton.edu/~hos/mike/transcripts/thompson.htm

[77] P. H. Salus, *A Quarter Century of UNIX*. Boston, MA, USA: Addison-Wesley, 1994.

[78] M. K. McKusick, "Twenty years of Berkeley Unix: From AT&T-owned to freely redistributable," in *Open Sources: Voices from the Open Source Revolution*, C. DiBona, S. Ockman, and M. Stone, Eds. Newton, MA, USA: O'Reilly, 1999, pp. 31–46.

[79] P. Seibel, *Coders at Work: Reflections on the Craft of Programming*. New York, NY, USA: Apress, 2009, ch. 12: Ken Thompson, pp. 449–483.

[80] J. Schilling, "User maintained programs in the second edition," TUHS—The Unix Heritage Society mailing list, Dec. 2016. [Online]. Available: http://minnie.tuhs.org/pipermail/tuhs/2016-December/007561.html

[81] S. Johnson, "What sparked lint? [was: Unix stories]," The Unix Heritage Society mailing list, Jan. 2017, Accessed on: 21, Nov. 2017, Archived by WebCite at [Online]. Available: http://www.webcitation.org/6v8TXa7kK

[82] Bell Laboratories, *UNIX Programmer's Manual. Volume 1*, 7th ed. Murray Hill, NJ, USA: Bell Telephone Laboratories, 1979.

[83] Bell Laboratories, *UNIX Programmer's Manual. Volume 2—Supplementary Documents*, 7th ed. Murray Hill, NJ, USA: Bell Telephone Laboratories, 1979.

[84] S. R. Bourne, "An introduction to the UNIX shell," in *UNIX Programmer's Manual. Volume 2—Supplementary Documents*, 7th ed. Murray Hill, NJ, USA: Bell Telephone Laboratories, 1979.

[85] D. M. Ritchie, "The C programming language—reference manual," in *UNIX Programmer's Manual. Volume 2—Supplementary Documents*, 7th ed. Murray Hill, NJ, USA: Bell Telephone Laboratories, 1979.

[86] S. C. Johnson, "Lint, a C program checker," in *UNIX Programmer's Manual. Volume 2—Supplementary Documents*, 7th ed. Murray Hill, NJ, USA: Bell Telephone Laboratories, 1979.

[87] M. E. Lesk, "TBL—A program to format tables," in *UNIX Programmer's Manual. Volume 2—Supplementary Documents*, 7th ed. Murray Hill, NJ, USA: Bell Telephone Laboratories, 1979.

[88] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou, "The use of summation to aggregate software metrics hinders the performance of defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 5, pp. 476–491, May 2017.

[89] C. B. Seaman, "Qualitative methods," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. London, U.K.: Springer, 2008, pp. 35–62.

[90] K. Thompson, "Users' reference to B," Internal Bell Labs Technical Memorandum. [Online]. Available: https://archive.org/details/users-ref-to-b, Jan. 1972, MM-72-1271-1, filing case 39199-11.

[91] L. Nyman and M. Laakso, "Notes on the history of fork and join," *IEEE Ann. History Comput.*, vol. 38, no. 3, pp. 84–87, Jul. 2016.

[92] *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX) Base Specifications, Issue 7*, IEEE Standard 1003.1–2017, 2017s.

[93] D. Libes and S. Ressler, *Life with UNIX*. Englewood Cliffs, NJ, USA: Prentice Hall, 1989.

[94] S. C. Johnson, "A tour through the portable C compiler," in *UNIX Programmer's Manual. Volume 2—Supplementary Documents*, 7th ed. Murray Hill, NJ, USA: Bell Telephone Laboratories, 1979.

[95] S. C. Johnson and M. E. Lesk, "Language development tools," *Bell Syst. Tech. J.*, vol. 56, no. 6, pp. 2155–2176, Jul./Aug. 1978.

[96] J. L. Bentley, "Programming pearls: Little languages," *Commun. ACM*, vol. 29, no. 8, pp. 711–721, Aug. 1986.

[97] P. Hudak, "Domain-specific languages," in *Handbook of Programming Languages, vol. III: Little Languages and Tools*, P. H. Salus, Ed. Indianapolis, IN, USA: Macmillan Technical Publishing, 1998.

[98] M. Fowler, *Domain-Specific Languages*. Boston, MA, USA: Addison-Wesley, 2010.

[99] S. R. Bourne, "The UNIX shell," *Bell Syst. Tech. J.*, vol. 56, no. 6, pp. 1971–1990, Jul.-Aug. 1978.

[100] A. V. Aho, B. W. Kernighan, and P. J. Weinberger, "Awk—A pattern scanning and processing language," *Softw.: Practice Exp.*, vol. 9, no. 4, pp. 267–280, 1979.

[101] L. E. McMahon, "SED—A non-interactive text editor," in *UNIX Programmer's Manual. Volume 2—Supplementary Documents*, 7th ed. Murray Hill, NJ, USA: Bell Telephone Laboratories, 1979.

[102] B. W. Kernighan and D. M. Ritchie, "The M4 macro processor," in *UNIX Programmer's Manual. Volume 2—Supplementary Documents*, 7th ed. Murray Hill, NJ, USA: Bell Telephone Laboratories, 1979.

[103] S. I. Feldman, "Make—A program for maintaining computer programs," *Softw.: Practice Exp.*, vol. 9, no. 4, pp. 255–265, 1979.

[104] W. N. Joy, S. L. Graham, C. B. Haley, M. K. McKusick, and P. B. Kessler, "Berkeley Pascal user's manual," in *UNIX Programmer's Manual—Volume 2c—Supplementary Documents: 4.2 Berkeley Software Distribution*. Berkeley, CA, USA: Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Aug. 1983.

[105] W. N. Joy and M. Horton, "Ex reference manual," in *UNIX Programmer's Manual—Volume 2c—Supplementary Documents: 4.2 Berkeley Software Distribution*. Berkeley, CA, USA: Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Aug. 1983.

[106] W. Joy, "An introduction to display editing with vi," in *UNIX Programmer's Manual—Volume 2c—Supplementary Documents: 4.2 Berkeley Software Distribution*. Berkeley, CA, USA: Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Aug. 1983.
[107] W. Joy, "An introduction to the C shell," in *UNIX Programmer's Manual—Volume 2c—Supplementary Documents: 4.2 Berkeley Software Distribution*. Berkeley, CA, USA: Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Aug. 1983.
[108] J. S. Quarterman and J. C. Hoskins, "Notable computer networks," *Commun. ACM*, vol. 29, no. 10, pp. 932–971, Oct. 1986.
[109] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Reading, MA, USA: Addison-Wesley, 1992.
[110] A. Hume, "Grep wars: The strategic search initiative," in *Proc. EUUG Spring 88 Conf.*, 1988, pp. 237–245.
[111] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, "4.2BSD system manual," in *UNIX Programmer's Manual—Volume 2c—Supplementary Documents: 4.2 Berkeley Software Distribution*. Berkeley, CA, USA: Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Aug. 1983.
[112] P. Karn, "The KA9Q internet (TCP/IP) package: A progress report," in *Proc. 6th ARRL Comput. Netw. Conf.*, 1987, pp. 91–94.
[113] M. Seltzer and M. Olson, "LIBTP: Portable, modular transactions for UNIX," in *Proc. Winter 1992 USENIX Conf.*, Jan. 1992, pp. 9–26.
[114] *FreeBSD Handbook*, Revision 47376 ed., The FreeBSD Documentation Project, Oct. 2015.
[115] T. S. Killian, "Processes as files," in *Proc. USENIX Summer 84 Conf.*, 1984, pp. 203–207.
[116] G. Lehey, *The Complete FreeBSD*, 4th ed. Newton, MA, USA: O'Reilly Media, 2006.
[117] P.-H. Kamp and R. N. M. Watson, "Jails: Confining the omnipotent root," in *Proc. 2nd Int. Syst. Admin. Netw. Conf.*, May 2000.
[118] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, May 2014.
[119] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," Mitre Corp., Bedford, MA, USA, Tech. Rep. MTR-2547, vol. 1, Nov. 1973.
[120] K. J. Biba, "Integrity considerations for secure computer systems," Mitre Corp., Bedford, MA, USA, Tech. Rep. MTR-3153, Rev. 1, Apr. 1977.
[121] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 101–112.
[122] G. E. Moore, "Cramming more components onto integrated circuits," *Electron.*, vol. 38, no. 8, pp. 114–117, Apr. 1965.
[123] D. Spinellis, "A tale of four kernels," in *Proc. 30th Int. Conf. Softw. Eng.*, May 2008, pp. 381–390.
[124] H. v. Vliet, *Software Engineering: Principles and Practice*, 3rd ed. Hoboken, NJ, USA: Wiley Publishing, 2008.
[125] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation Optimization*, Mar. 2004.
[126] D. Sjøberg, G. Bergersen, and T. Dybå, "Why theory matters," in *Perspectives on Data Science for Software Engineering*, T. Menzies, L. Williams, and T. Zimmermann, Eds. Boston, MA, USA: Morgan Kaufmann, 2016, pp. 29–33.
[127] D. I. K. Sjøberg, T. Dybå, B. C. D. Anda, and J. E. Hannay, "Building theories in software engineering," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. London, U.K.: Springer, 2008, pp. 312–336.
[128] F. P. Brooks, *The Mythical Man Month*. Reading, MA, USA: Addison-Wesley, 1975.
[129] R. Love, *Linux Kernel Development*, 3rd ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2010.
[130] D. Bovet, *Understanding the Linux kernel*, 3rd ed. Sebastopol, CA, USA: O'Reilly, 2006.
[131] *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Programming Interface (API) (C Language)* ISO Standard ISO/IEC 9945–1:1996, 1996 (IEEE/ANSI Std 1003.1, 1996 Edition).
[132] *Information Technology—Portable Operating System Interface (POSIX)—Part 2*: Shell and Utilities ISO Standard ISO/IEC 9945–2:1993, 1993 (IEEE/ANSI Std 1003.2-1992 & IEEE/ANSI 1003.2a-1992).
[133] *American National Standard for Information Systems—programming language—C*, ANSI Standard ANSI X3.159–1989, Dec. 1989, (also ISO/IEC 9899:1990).
[134] *Programming Languages—C* ISO Standard ISO/IEC 9899:1999, 1999.
[135] *Programming Languages—C* ISO Standard ISO/IEC 9899:2018, 2018.
[136] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (Dagstuhl seminar 16162)," *Dagstuhl Rep.*, vol. 6, no. 4, pp. 110–138, 2016. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6693
[137] M. Bagherzadeh, N. Kahani, C.-P. Bezemer, A. E. Hassan, J. Dingel, and J. R. Cordy, "Analyzing a decade of Linux system calls," *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1519–1551, Jun. 2018.
[138] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Bell Syst. Tech. J.*, vol. 57, no. 6, pp. 1905–1929, Jul./Aug. 1978.
[139] W. R. Stevens, *UNIX Network Programming: Networking APIs: Sockets and XTI*, vol. 1, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1998.
[140] A. S. Tanenbaum, *Operating Systems: Design and Implementation*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1997.
[141] A. Lewis, *AT&T Settles Antitrust Case; Shares Patents*. New York, NY, USA: New York Times, Jan. 25, 1956, pp. 1,16.
[142] N. Takahashi and T. Takamatsu, "UNIX license makes Linux the last missing piece of the puzzle," *Ann. Bus. Administ. Sci.*, vol. 12, pp. 123–137, 2013.

Diomidis Spinellis is a professor of software engineering with the Department of Management Science and Technology, Athens University of Economics and Business, Greece and director of the University's Business Analytics Laboratory. He is the author of two award-winning books, Code Reading and Code Quality: The Open Source Perspective. His most recent book is Effective Debugging: 66 Specific Ways to Debug Software and Systems. He has contributed code that ships with Apple's macOS and BSD Unix, and is the developer of CScout, UMLGraph, dgsh, and other open-source software packages, libraries, and tools. He served as an editor in chief for *IEEE Software* over the period 2015–2018. He is a senior member of the IEEE.

Paris Avgeriou is a professor of software engineering with the University of Groningen, The Netherlands where he has led the Software Engineering research group since September 2006. His research interests lie in the area of software architecture, with strong emphasis on architecture modeling, knowledge, evolution, patterns and technical debt. He is an editor in chief of the *Journal of Systems and Software*, as well as an associate editor for *IEEE Software*. He has co-organized several international conferences (e.g., ECSA and ICSA) and workshops (mainly at ICSE). He is a senior member of the IEEE.