A Review On Comparative Study Of Shell's In Linux

Shubhanga CS
departmnet of computer science
cshubhanga@gmail.com

Soorya Prakash Acharya
department of computer science
sooryapacharya@gmail.com

Smita Ravi Naik
department of computer science
smitan949@gmail.com

Sonali Shetty
department of computer science
sonalishetty235@gmail.com

Dr. G Srinivasan
department of computer science
srinivasgopalan@aiet.org.in

***Abstract*—There are several command-line interpreters, or "shells," that the Linux operating system supports. The main interface that allows the user to interact with the Linux kernel and execute scripts, programs, and instructions is provided by these shells, each of which has its own special features and powers. In this review article, the most popular Linux shells—Bourne-Again SHell (Bash), Z Shell (Zsh), Korn Shell (Ksh), Fish (Friendly Interactive SHell), and C Shell (Csh)—are thoroughly compared. In order to give consumers useful information to help them choose the best shell for their particular requirements, it tries to emphasize the unique qualities, performance characteristics, scripting capabilities, and user experience of each shell.**

Keywords— Linux Shells, Shell Comparison, Command Line Interpreters, Bash, Zsh, Fish, Tcsh, Scripting Languages, Shell Performance, User Interface Customization, Scripting Capabilities, CrossPlatform Compatibility, Shell Extensions, Command Line Productivity, Shell Efficiency, Open Source Shells, Interactive Shell Features, Shell Scripting, Terminal Emulators, Linux Operating System, Command Execution Speed, Shell Usability, Configuration Flexibility.

I. Introduction

Within the world of Linux operating systems, the shell functions as an essential interface that links users and the kernel. By converting user inputs into executable commands that communicate with the operating system's core, a shell serves as a command interpreter. It provides a strong and adaptable command-line environment and is essential in enabling communication between users and the system.A shell can be used for more than just running commands; it can be used to traverse files, launch applications, manage processes, and alter files. Because they offer a flexible and scriptable environment for both interactive use and automation, shells are an essential part of the Linux user experience.

Comprehending the function of shells is crucial for both administrators and users of Linux, since it serves as the entry point to fully utilizing the operating system's command-line interface. This overview lays the groundwork for a more thorough examination of Linux shells, including their history, functionality, and essential role in system administration and everyday computing duties.

II. DIFFERENT KIND OF SHELL'S

BASH, an acronym for "Bourne Again SHell," is a well-established open-source scripting language and command-line shell. Since its inception in 1989 as a component of Brian Fox's GNU Project, Bash has evolved into the standard shell for numerous Unix-like operating systems, such as Linux and macOS. It was developed from the Bourne Shell (sh) and combines elements from other Unix shells, making it a feature-rich tool that power users, developers, and system administrators use extensively.

Bash's main function is to read and carry out user commands via a command-line interface. Bash is a robust programming language with more capabilities than just a command interpreter. By writing scripts with variables, functions, control structures (if statements, loops), and other programming constructs, users take advantage of its scripting features to automate activities.

Environment variables for configuration, I/O redirection for controlling input and output, pipelines for linking commands, and task control for managing several processes at once are some of Bash's key features. These characteristics help Bash be more flexible and effective at handling a range of command-line activities. In Bash, environment variables are essential because they let users tailor their working environment and adjust the behavior of the shell. Users may remember and re-execute earlier instructions with the command history function, which improves productivity and navigation. Working with groups of files is made easier with the help of file globbing and wildcards, which simplify file-related tasks.

Bash's ability to be extended is demonstrated by its scripting functionality. The ability to write custom functions and scripts allows users to design system utilities, automate repetitive operations, and customize the shell for particular use cases. While integrating features from other shells, Bash maintains standards and backward compatibility, guaranteeing consistency with the original Bourne Shell (sh). This commitment encourages script portability across many Unix-like platforms and supports a smooth user experience. Users of Bash benefit from a thriving community, comprehensive documentation, tutorials, and online resources. This infrastructure of support helps users become proficient with Bash's capabilities and functions, which is one of the reasons it remains so popular in Unix and Linux contexts.

ZSH, which stands for "Z Shell," is a strong scripting language and command-line shell that may be used instead of Bash. It was created in 1990 by Paul Falstad as a more feature-rich and enhanced Bourne Shell (sh) with more capabilities. Zsh is a very flexible and user-friendly shell that combines functionality from other shells, such as Bash, KornShell, and others.

Zsh is a command-line interface that provides an interactive and effective user experience by interpreting and executing commands supplied by the user. Although Zsh and Bash are similar, Zsh has certain characteristics that make it different. Variables, arrays, and associative arrays are just a few of the programming features that Zsh's scripting language offers its users. Moreover, it has sophisticated scripting features including named directories, strong globbing patterns, and the capacity to create and utilize unique widgets for improved interactivity. One of Zsh's primary features is a comprehensive auto-completion system that lets users finish commands, routes, and parameters by just hitting the Tab key. With the help of this function, command-line efficiency is greatly increased and manual typing is decreased.

Prompt theming is a notable feature of Zsh that allows users to completely modify the appearance of the command prompt. To provide a customized and educational command-line experience, themes can contain details about the system, version control status, and current directory. Zsh has a syntax highlighting function that highlights various parts of a command in different colors, making command-line readability improved. Users will find it simpler to recognize and comprehend complicated commands with the help of this visual assistance. Popular frameworks like Oh-My-Zsh serve as examples of Zsh's plugin system, which makes it simple to integrate new features and themes. By adding plugins that provide features like enhanced syntax highlighting, more auto-completions, and more, users may increase Zsh's functionality. Zsh also has an intelligent correction mechanism that offers pre-execution suggestions for commands that are written incorrectly.

FISH, is a cutting-edge command-line interface that improves user experience on Unix-like computers. Its advanced syntax highlighting, which dynamically colors commands, arguments, and variables to help users rapidly comprehend and evaluate command structures, is one of its most notable features. This feature enhances the visual experience, particularly for those who use the command line frequently.

Fish shell is unique in that it has an auto-suggestions feature. Fish makes real-time completion suggestions to users as they input commands, taking into account their command history and available alternatives. This proactive approach is especially helpful for those who are new to the command line, since it not only expedites the command-entry process but also aids users in learning new commands and choices.

Fish's emphasis on consistency and simplicity in syntax makes it easier to use for users of different ability levels. By reducing the need for complex programming constructs, the shell enhances readability and user-friendliness. Fish emphasizes simplicity without sacrificing its scripting skills. Because of its strong scripting language support, users may write effective and potent scripts for customization and automation.

Fish shell's adaptability is further enhanced by built-in commands and functions. By offering an extensive collection of tools for frequently performed command-line operations, the shell minimizes the need for third-party programs. The provision of important functionality to users in an inclusive environment fosters productivity and efficiency.

Fish's programming language is created with readability for humans in mind. For customers who wish to take use of the power of scripting but may not be experienced developers, this functionality is especially helpful. Because of the language's simple and straightforward syntax, users may write and maintain scripts without encountering needless complexity.

Fish shell is a command-line interface that is both feature-rich and easy to use. With its emphasis on simplicity, clever auto-suggestions, dynamic syntax highlighting, and strong scripting features, Fish is a tempting choice for users looking for a fun and productive command-line interface. Fish shell offers an ideal blend of power and accessibility for both rookie and expert command-line users.

III. SHELLS INTERACTION WITH THE SYSTEM

A. Parsing and Interpretation

A shell's parsing and interpretation process starts when a user types a command into it in order to determine the user's purpose. This procedure, which divides the input command into its component components and gets it ready for execution, is essential to the shell's operation and takes many phases.

Tokenization is the initial stage of the parsing process, in which the input command is divided into distinct tokens by the shell according to whitespace characters (such tabs and spaces). Like command names, parameters, and redirection operators, each token stands for a unique component of the command. Tokens include, for instance, ls, -l, and /home/user in the command ls -l /home/user.

The command name, which is the first token in the command, is recognized by the shell once the input command has been tokenized. The executable program or built-in command that the shell needs to run is indicated by the command name. The command name is ls in the sample command ls -l /home/user.

The shell parses the remaining tokens to extract the command's parameters and arguments after recognizing the command name. Arguments can be file names, directory paths, or other arguments that provide the command further information. Options, also referred to as switches or flags, alter how a command behaves. The -l option in the ls -l /home/user command tells the ls command to provide comprehensive details about files and directories.

The shell recognizes redirection operators, which manage the input and output streams for commands, in addition to command names and parameters. The operators ?, \, and | are frequently used for redirection purposes. They divert output to a file, input from a file, and pipe output from one command to another, respectively. The output of the ls program, for instance, is sent to a file called files.txt when the command ls ? files.txt is used.

The shell has successfully disassembled the input command into its component pieces and determined the command name, parameters, and redirection operators once the parsing step is finished. This procedure guarantees that the order is accurately comprehended and ready to be carried out. After that, the shell carries out the command, interacting with the operating system and carrying out the user-specified actions by using the information that has been processed.

B. *System Calls*

Shells act as a go-between for the operating system kernel and users, making it easier to control system resources and carry out tasks. System calls are one of the main ways that shells communicate with the kernel. The kernel provides these system calls, which enable processes—including shell processes—to ask the operating system for services.

Fork() and exec() are two of the many system calls that shells use, and they are essential for starting processes and executing commands. The child process, created by the fork() system function, is a perfect replica of the parent process, down to the address space and execution context. This method facilitates concurrency and multitasking by allowing shells to launch additional processes to carry out instructions. The exec() system function is used to replace the child process's memory image with a new program image—typically, the executable that corresponds to the command to be executed—after a new process is formed using fork(). By loading the desired program into the newly formed process, this technique enables shells to execute commands. In essence, the command execution replaces the shell process.

Shells are used for file manipulation and input/output activities, in addition to process management. They make use of system methods like open(), read(), write(), and close(). To open or create new files, use the open() system function, which returns a file descriptor representing the opened file. Shells can then utilize this file descriptor to use the read() and write() system functions, respectively, to read from or write to the file. The close() system function is used to terminate the file descriptor and release the related resources when file operations are finished.

Shells can successfully modify files and communicate with other system resources thanks to these system functions. Shells use the kernel's system call interface to connect with the underlying operating system in order to carry out various tasks such as managing processes, executing commands, and working on files. The command-line interface is built on the tight relationship between shells and the kernel, which enables users to effectively control and manage their systems through shell interactions.

C. *Environment Variables*

Environment variables are dynamic values that influence the behaviour of commands and scripts run within the shell environment. Shells are essential for handling these variables. Important data including user preferences, system setups, and executable file paths are stored in these variables. To properly modify and improve their working environments, users must comprehend how shells manage and change environment variables.

The current user's username (USER), home directory (HOME), and shell-specific variables like the path (PATH) variable, which specifies the directories the shell searches for executable files, are just a few examples of the vast array of data that environment variables in shells might include. Environment variables can also hold information about terminal settings, system locales, and language preferences, which can affect how commands and programs run in the shell environment.

Shells enable users to dynamically customize and configure the shell environment by giving them access to a variety of ways for interacting with environment variables. Using built-in shell commands, users may examine and modify the current set of environment variables. Examples of these commands include env and printenv. To guarantee that variables are available in later commands and scripts, users can designate variables for export to child processes using the export command, for example.

Moreover, shells allow environment variables to be added, changed, and removed in order to customize the shell environment to meet particular needs. Assignment statements (VAR=value) can be used to change variables that already exist. Alternatively, users can unset variables completely (unset VAR) or prepend or append values to variables. With the help of these features, users may adjust their work surroundings, personalize the behavior of commands, and optimize their workflow to suit their unique needs and preferences.

Additionally, environment variables provide communication between various shell environment processes and applications. Users may propagate data across many processes and scripts by exporting variables, which makes it easier for different system components to integrate and communicate with one another.

D. *Filesystem Manipulation*

Using a variety of utilities and commands, shells are effective tools for navigating, manipulating, and managing files and directories inside the filesystem. These commands improve productivity and workflow efficiency by providing users with a smooth and effective approach to carry out typical filesystem operations straight from the command line.

The mkdir command's generation of directories is one of the essential filesystem activities that shells provide. By passing the required directory names as parameters to the mkdir command, users can create new directories in the filesystem hierarchy. This feature makes it possible for users to set up their filesystem in a way that best suits their needs, which makes managing and organizing data more effective.

Shells offer commands to create, remove, copy, move, and alter files and folders in a similar manner. Users can copy files and directories from one location to another using the cp command, but they can also destroy files and directories with the rm command. Users may rename files and transfer them between directories with ease thanks to the mv command, which makes filesystem movement easier. Users can also alter the timestamp on existing files or create new, empty files using the touch command.

System calls that are made by the shell to communicate with the filesystem constitute the foundation of these filesystem activities. For instance, the unlink() system call removes a file or directory from the filesystem, but the mkdir() system call creates a new directory with the given name. In a similar vein, files and directories can be renamed by using the rename() system function, which modifies the filenames in the filesystem hierarchy.

Users may effectively and easily carry out a variety of filesystem activities by utilizing these filesystem utilities and commands. Shells offer a single interface for manipulating files, abstracting system calls and filesystem operations into easily understood commands. Regardless of technical proficiency, users may efficiently manage filesystem resources thanks to this abstraction layer, which also improves usability.

E. Scripting and Automation

One of the main features of shells is that they may allow scripting and automation. By writing and running shell scripts, users can automate tedious chores and carry out complicated processes. Shell scripts are basically text files that are run by the shell as a single program. These text files contain sequences of shell commands and control structures, such loops and conditionals. With the help of this functionality, users may automate a variety of operations, including data processing, process management, and system administration, so greatly increasing the power and flexibility of the shell.

By utilizing the shell's syntax and command set, shell scripting makes it possible for scripts to easily carry out operations including program execution, file manipulation, and system monitoring. Additionally, shell scripts are effective tools for automating intricate operations because they may call other programs, initiate utilities, and utilize built-in shell functions like text manipulation, variable replacement, and input/output redirection.

When a shell script runs, the shell examines each command in turn, parses its syntax, interprets it, and then invokes the appropriate system functions to communicate with the operating system. Interpretive execution enables dynamic execution flow, where control structures based on conditional evaluations or iterative processes over data sets may be used to change the script's behavior.

Utilizing the huge ecosystem of Unix/Linux commands and tools is one of the main advantages of shell scripting, since it allows scripts to accomplish a wide range of activities. With the straightforward syntax of shell commands, a shell script can, for example, automate file backups, the monitoring of system logs for certain occurrences, or the bulk processing of picture files.

Shell scripting also makes it easier to automate system administration chores like setting up users' accounts, installing applications, and configuring the system. System administrators may guarantee consistency, lower the possibility of human mistake, and save a substantial amount of time and effort by scripting these procedures.

But there are duties associated with shell scripting's capabilities. It necessitates a deep comprehension of command tools, shell syntax, and system architecture. To avoid unexpected effects on the system, scripts must be properly designed to manage mistakes, provide security, and operate effectively.

F. Job Control

The shell's job control capabilities are essential for effective system administration and user interaction in the multitasking environments of Unix and Linux. The term "job control" describes the shell's capacity to control the execution of several processes, giving users the freedom to execute commands in the background, pause and resume running programs, and change the order in which they are executed. This feature makes it possible for the user to manage system resources and processes much more effectively, which leads to a workflow that is more dynamic and effective.

1) Background Execution and Process Suspension:

The capability of job control to execute tasks in the background is one of its main features. Users can launch a process and return to the shell prompt instantly by adding an ampersand (&) to a command. This enables the simultaneous execution of other tasks. Long-running programs that don't need frequent user input benefit greatly from this non-blocking style of operation.

Additionally, job control makes it easier to pause running processes by utilizing signals. For instance, using Ctrl+Z will usually cause a foreground task to be suspended. By initiating a SIGSTOP signal, this step puts the process in a paused state without ending it. The shell may be used to handle suspended processes. It maintains a job table that lists all current and suspended tasks and enables users to list, restart, or end them as

2) Resuming Execution

It is possible to restart suspended processes in the background or in the foreground. The fg command allows it to take control of the terminal and communicate with the user directly by resuming a paused operation in the foreground. In contrast, a stopped task can be resumed in the background by using the bg command. This allows the job to continue running without obstructing the shell or needing immediate user interaction.

3) Process Priority Management

Moreover, shells let users control process priority by utilizing the "niceness" idea. Higher priority processes use more CPU time than lower priority processes. Commands such as nice and renice, which in the end invoke the setpriority() system function, allow users to modify a process's niceness and hence its scheduling priority.

4) Signal Management

These job control characteristics are based on how the shell uses signal processing and system calls. With the use of the kill() system function, a process can receive signals from the shell or other processes, allowing for actions like termination (SIGTERM), forced termination (SIGKILL), and suspension (SIGSTOP). Furthermore, a more reliable method for handling signals is offered by the sigaction() system call, which enables programs to designate unique handlers for particular signals. This makes it possible for processes to respond to signals from the operating system or other processes with graceful shutdowns, restarts, or other unique behaviors.

G. *Input and Output Redirection*

The usual input, output, and error streams of commands can be redirected by users thanks to shell support for input and output redirection. File descriptors and system methods like dup2(), which duplicate file descriptors to reroute input and output to files or other processes, are used to accomplish this capability.

H. *Security Considerations*

Shell scripts are vulnerable to injection attacks, privilege escalation, and inappropriate processing of user input, just like any other executable code. Validating input data, limiting the usage of root capabilities, and using tools like Shell Check for static analysis are all recommended practices for shell script security.

I. *Process Management*

Shells handle all aspects of process management, including its creation, observation, and termination. Fork() and wait() are two examples of system functions they use to start child processes and coordinate their execution. Shells also keep track of information about processes that are currently in use, such as their process IDs (PIDs) and execution status.

TABLE 1

COMPARATIVE STUDY ON SHELLS IN LINUX

Feature	Bash	Zsh	Fish	Tcsh
Autocompletion	Basic	Advanced	Advanced	Advanced
Scripting Capabilities	Excellent	Excellent	Good	Good
Interactive Use	Good	Excellent	Excellent	Good
Customization	Moderate	High	High	Moderate
Plugin System	Limited	Extensive	Moderate	Limited
Learning Curve	Moderate	Steep	Easy	Moderate
Performance	Good	Good	Excellent	Good

IV. CONCLUSION

In conclusion, the wide variety of shells that are accessible in the Linux environment greatly enhances the operating system's adaptability and versatility. Reviewing a variety of shells, including Fish, Zsh, and Bash, reveals a feature-rich environment that accommodates a wide range of user needs and preferences. Because of its widespread use and powerful scripting features, Bash is still a go-to option for a lot of users and system administrators. With its additional customisation choices and potent tab-completion, Zsh is a compelling choice for those looking for a more engaging experience. Especially for those who are unfamiliar with the command line, Fish Shell stands out as a contemporary and friendly option thanks to its emphasis on user-friendliness, syntax highlighting, and clever auto-suggestions.

Because each shell is different and offers advantages over the others, users may customize their command-line experience to fit their tastes and workflow. With these varied solutions available, the Linux ecosystem can accommodate users with different use cases and skill levels, creating a dynamic and inclusive environment. Exploration and development of shells adds a great deal to the overall user experience as the Linux community grows, allowing for more effective and efficient interaction with the command line's strong capabilities. The choice of a shell ultimately comes down to personal taste, workflow demands, and individual user requirements, which reflects the flexible and open character of the Linux operating system.

REFERENCES

[1] Shivangi Shandilya,Surekha Sangwan,Ritu Yadav (2014). "Shell Scripting and Shell Programming in Unix", (IJIRT), vol. 1, no. 11, pp 2349-600.

[2] M. Seltzer and M. Olson, “LIBTP: Portable, modular transactions for UNIX,” in Proc. Winter 1992 USENIX Conf., Jan. 1992, pp. 9–26.

[3] Evolution Of The Unix System Architecture:An Exploratory Case Study.

[4] 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.

[5] 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

[6] Kerrisk, M. (2010). The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press.

[7] Robbins, A., & Beebe, N. (2005). Classic Shell Scripting: Hidden Commands that Unlock the Power of Unix. O'Reilly Media, Inc.

[8] Love, R. (2010). Linux System Programming: Talking Directly to the Kernel and C Library. O'Reilly Media, Inc.

[9] 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.

[10] 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.

[11] Stevens, R. P., Rago, S. A., & Fenner, B. (2013). Advanced Programming in the UNIX Environment. Addison-Wesley Professional.

[12] Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts. John Wiley & Sons, Inc.

[13] A. Caroline Mary, "Shellshock Attack on Linux Systems – Bash", (IRJET 2015), vol. 2, no. 8,pp. 2395-0072.

[14] nixCraft.cyberciti.biz.

[15] Evolution of Shells.

[16] W. Toomey, “Unix: Building a development environment from

[17] scratch," in Reflections on Operating Systems—Historical and Philosophical Aspects, L. Demol and G. Primiero, Eds. New York, NY,

[18] Tycho Kirchner, Konstantin Riege & Steve Hofmann, "Bashing irreproducibility with shournal," (Springer 2024) 14:4872.

[19] Kartalopoulos, S. V., "Differentiating Data Security and Network Security," Communications, 2008. vol. 08, pp.1469 - 1473,19 - 23 May 2008.

[20] http://www.thegeekstuff.com/2010/07/executeshell-script.

[21] http://en.wikipedia.org/wiki/Shell_script.

[22] http://www.calpoly.edu/~rasplund/script.html.

[23] http://supportweb.cs.bham.ac.uk/docs/tutorials/d.ocsystem/build/tutorials/unixscripting/unixscripting.html.

[24] Robin Harder, "Using Scopus and OpenAlex APIs to retrieve bibliographic data for evidence synthesis. A procedure based on Bash and SQL", (Elsevier 2024).

[25] J.A.Bradshaw, K.J.Carden & D.Riordan, "Ecological applications using a novel expert system shel",(cabios) vol. 7, no. 1,pp. 1991.

[26] W. Joy, "An introduction to the C shell, in UNIX Programmer's", vol. 2,Berkeley, CA.

[27] Diomidis Spinellis, Senior Member,"Evolution of the Unix System Architecture:An Exploratory Case Study",(IEEE 2021), vol. 47, no. 6, june 2021.

[28] https://Linux.org." (2023).

[29] Bridges, C., Yeomans, B., Iacopino, C., Frame, T. E., Schofield, A., Kenyon, S. et al. Smartphone qualification and Linux-based tools for CubeSat computing payloads. In Proceedings of the 2013 IEEE Aerospace Conference, Mar. 2013. [Online] Available: http://dx.doi.org/10.1109/AERO.2013.6497349.

[30] Min-kyu Choi , Rosslin John Robles, Chang-hwa Hong2), Tai-hoon Kim1), "Wireless Network Security: Vulnerabilities, Threats and Countermeasures", Vol. 3, No. 3, July, 2008

[31] Kartalopoulos, S. V., "Differentiating Data Security and Network Security," Communications, 2008. ICC '08. IEEE International Conference on, pp.1469-1473, 19-23 May 2008.

[32] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," Linux J., vol. 2014, no. 239, May 2014.

[33] 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.

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

[35] R. Love, Linux Kernel Development, 3rd ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2010.

[36] D. Bovet, Understanding the Linux kernel, 3rd ed. Sebastopol, CA, USA: O'Reilly, 2006.

[37] Kumar, Surender, and Rupinder Kaur. "Plant disease detection using image processing-a review." *International Journal of Computer Applications* 124.16 (2015).

[38] 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.

[39] W. R. Stevens, UNIX Network Programming: Networking APIs: Sockets and XTI, vol. 1, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1998.

[40] 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.