




Towards Methods to Automatically Identify the Most Common Errors in Linux by Analyzing Git Commit Messages

Nikita Starovoytov
Applied mathematics department
Polzunov Altai State Technical University
Barnaul, Russia
0009-0007-0242-0198 

Nikolay Golovnev
Applied mathematics department
Polzunov Altai State Technical University
Barnaul, Russia
0009-0008-0258-4560 

Sergey Staroletov
Applied mathematics department
Polzunov Altai State Technical University
Barnaul, Russia
0000-0001-5183-9736 

Abstract—A lot of information circulates in system software environments, so it is advisable to use it to improve the operation of such systems. The Linux kernel not only comes with completely open source, but also the history of this code is completely available thanks to the git repository. We are primarily interested in error correction messages, whose text analysis can help in isolating the classes of the most typical errors. This paper expands on the previous work of one of the authors and suggests the use of data analysis methods. We look at methods for working with repository messages and ways to automatically find the most common errors in it. We calculate distances between the messages and cluster them. The results are done for the Thunderbolt repository inside the Linux kernel.

Index Terms—Linux, git, bugs, fuzzy phrase matching

I. INTRODUCTION

Today, closed software systems can no longer compete in quality with open ones, mainly due to the involvement of more qualified users who can not only test the software in the form of a black box, but also understand the code and suggest changes. The git version control system [1] and services based on it are based on a fork and pull request approach [2], with the help of which users easily propose changes, and administrators responsible for the repository have the opportunity to accept these changes after reviewing diffs. The git system is designed for distributed work and encourages many local changes (commits) so that developers can easily move between revisions. Each commit is accompanied by a comment about what was done. The system was originally created by Linus Torvalds to coordinate the development of the Linux kernel and it is Linus' second super successful project.

An operating system belongs to a class of system software that provides abstractions for accessing hardware from client code and the ability for such code to work cooperatively. With the increase in the number of developers in the world, an ever smaller percentage of them are capable of developing system program code, and therefore the development of

system code is not popular. However, there is a large amount of data circulating in system software environments that can be analyzed by today's popular data analysis methods. This paper follows this approach and proposes to analyze commit messages in the development of the Linux kernel by automated methods. A large number of commit messages indicate the presence of a big data in natural language; accordingly, some common patterns can be automatically identified from these data. We are primarily interested in the most typical errors in system software from among those identified and corrected.

Linux OS, based on an open modular kernel concept by Linus Torvalds, is being improved by a large number of developers, both individual and representatives of leading companies in the industry. The kernel is constantly evolving, all changes in it are carried out by committing changes to the developers' gits and some of them finally become available in the mainstream kernel at Torvalds GitHub [3]. Such a commit is usually verified by higher developers in the hierarchy using the pull-request mechanism.

The purpose of our work is to automatically analyze commits in the Linux kernel repository to identify the most representative bugs. In this paper, we mainly discuss and try data analysis methods for Linux commit messages.

The rest of the paper has the following structure. In Section II, we discuss known works on Linux bugs revealing and classification. Section III is about the internals of methods we use. Section IV is devoted to the implementation and evaluation. In Conclusion, we sum up and give a link to our software.

II. RELATED WORK

In the pioneering work [4] and then in [5], static analyzers were used to automatically check for potential errors in the Linux kernel code based on a given configuration over different kernels, classes of errors were defined as predefined messages of a static analyzer, and graphs of the evolution of

errors over time and for different subsystems were presented. Specifically, drivers have been found to be 3-7 times more error prone than other components.

In [6], an analysis of typical errors is made in the drivers of the Linux operating system. Here the concept of a typical error is introduced. It is specific to a large number of drivers (for example, resource leaks, incorrect use of locks), while a non-typical error is domain-specific for a particular driver. The authors manually analyzed the changes during the transition from one kernel version to another and compiled tables of the distribution of errors by classes. It was also found that drivers make 85% of all errors in the kernel. The paper [7] continues this work, summarizes various statistics on changes in the kernel and concludes that about 40% of changes between stable versions of the kernel are fixes of typical errors. Since more versions were analyzed and the code evolved, the author had to supplement the previous created classes. Such manual analysis is more difficult, but the authors note that it is more careful.

In [8], 5079 patches related to file systems made over 8 years were manually analyzed. Classes of bugs, the so-called bug patterns, are identified and graphs of their evolution are given, as a result, a dataset of 1800 bugs is compiled.

Empirical work [9] is devoted to a broad study of bugs in open-source software, including the Linux kernel. As for Linux, bugs are simply assigned to one of the subsystems (core, driver, network, FS, arch, other), while several open-source components are analyzed using message text from BugZilla with its vectorization and further automatic classification.

The work [10] is devoted to the study of 5741 Linux kernel bug reports, which were analyzed according to the description, comments and attached files from the Linux kernel bug tracker [11]. Bugs are classified into fast-reproducible (Bohrbug), difficult-to-reproduce (Mandelbug) or context-dependent, and are also defined categories from which the bug context depends, that is, errors with memory, not freed resources, etc. At the same time, the authors built a network based on the Linux call graph, with the help of which they track the impact of the functions affected in bug reports by counting various metrics.

Researchers in [12] present the results of compiling 42,060 kernels with all warnings enabled. As a result of the analysis of 400,000 warnings, they classified by type and distribution by kernel subsystems and identified drivers as the most vulnerable portion of the kernel.

The work [13] presents the PatchNet network, created as a result of automatic analysis of patches for the kernel, in order to predict whether a given patch will be accepted in the mainline kernel or not. For evaluation, the texts of the commit messages and the vector representation of the changes from the diff of the commit are used, which are then used to build a convolution neural network.

The research [14] is separately devoted to determining whether a patch to the kernel is a bug fix or not. The authors note that simple analysis based on commit messages does not always lead to results and propose a model that uses two classification algorithms: Learning from Positive and Unlabeled Examples and Support Vector Machine. It also uses features extracted from the commit diff.

In the study [15], the authors provide infrastructure,

classify and analyze Linux kernel-specific errors associated with errors in configuration files, as a rule, these are errors with dependencies. 95,854 Linux kernel builds were produced on random configurations, and of these, about 6% ended with errors, and which are discussed in the work. It is noted that the number of errors has decreased with previous findings, apparently due to testing processes with randomized configurations.

Summarizing the above on Linux bug analysis, it can be seen that (1) bugs in drivers are the most common; (2) different methods are used for classification, this is static analysis, build logs and patch analysis; (3) a lot of huge manual work has been done but the results may now be considered no longer relevant (the code is constantly changing). However, automatic classification by analyzing commits in git repositories has not been applied yet.

III. PRELIMINARIES

As for working with git repositories at the program level, what we are interested in, some is discussed in [16]. Probably, the most famous C-library for this is *libgit2*. For JVM programs, the *JGit* library [17] is popular. One can use *EGit* [18] to work with remote repositories, including pull requests, but this is not necessary for the current project because we are working with the mainline kernel with changes already accepted.

To compare commit messages, it is necessary to work out *fuzzy string matching* algorithms. Note that fuzzy string comparison is popular in bioinformatics and can also be modeled on non-deterministic automata for low-dimensional problems, although this is purely theoretically interesting [19]. Of practical interest are efficiently calculated string similarity measures, such as the Levenshtein distance.

Formally, the Levenshtein distance $L(s_1, s_2)$ [20] between strings s_1 and s_2 can be calculated according to the following formulas:

$$\begin{aligned}
 L(s_1, s_2) := & \forall i \in (0..|s_1|) : d_{i,0} := i + 1; \\
 & \forall j \in (0..|s_2|) : d_{0,j} := j + 1; \\
 & \forall i \in (1..|s_1|) : \\
 & (\forall j \in (1..|s_2|) : cost := (s_1[i-1] = s_2[j-1])?0 : 1 \\
 d_{i,j} := & \min(\min(d_{i-1,j} + 1, d_{i,j-1}), d_{i-1,j-1} + cost); \\
 & d_{|s_1|,|s_2|}) \quad (1)
 \end{aligned}$$

With it, to find the closest string to the existing ones, in the simplest implementation, one needs to calculate the distances between them using formula (1) and choose the minimum one. This method does not require preliminary preparation of strings and is susceptible to slight changes in them.

The preliminary our papers [21], [22] demonstrate the use of the Levenshtein distance, but now we would like to apply another method known from its use in search engines (“bag of words” to convert a phrase into a vector + cosine similarity between vectors to further determine the minimal distance). To calculate the distance between commits using the cosine similarity approach, it is required to represent the commit message string as a vector (from the features as the words of the message). Here we denote $w_{i,j}$ as the sign of the presence of the word j in the string i , while n specifies the number

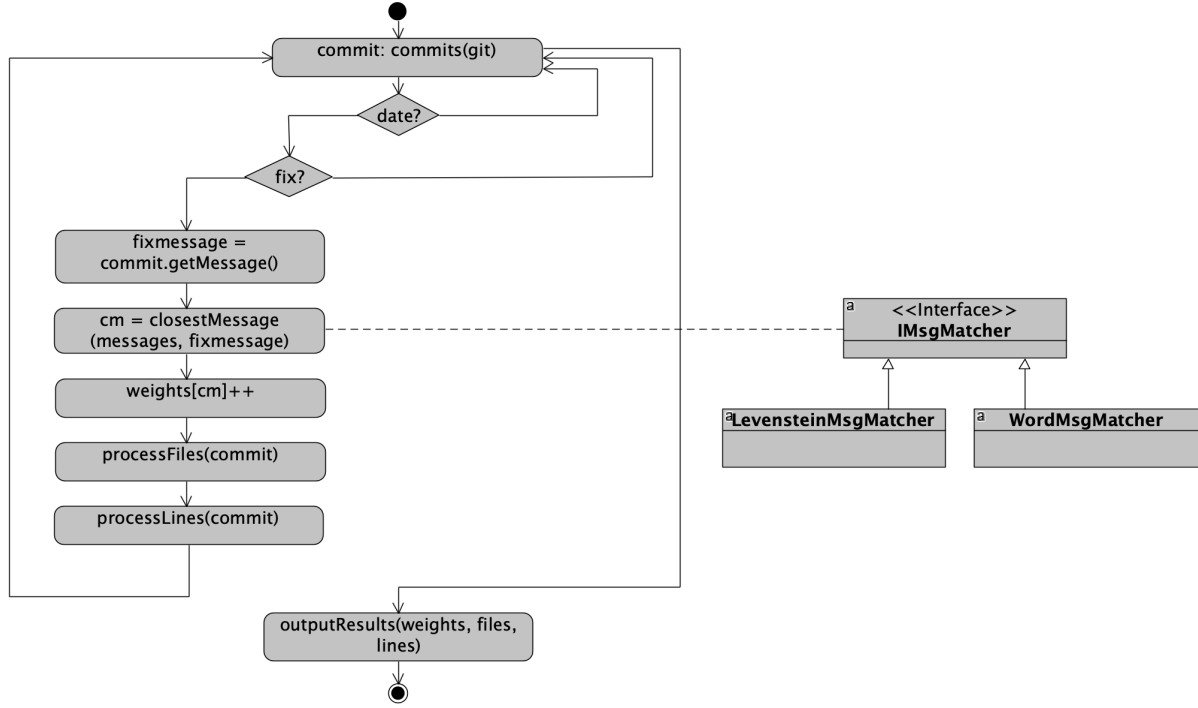


Fig. 1. Solution diagram for handling a repository with bug fix messages

of words in the dictionary of unique words. Then we can calculate the cosine similarity between the vectors:

$$D(s_1, s_2) := D((w_{1,1}, w_{1,2}, \dots, w_{1,n}), (w_{2,1}, w_{2,2}, \dots, w_{2,n})) = \frac{\sum_{i=1}^n w_{1,i} \cdot w_{2,i}}{\sqrt{\sum_{i=1}^n w_{1,i}^2 \cdot \sum_{i=1}^n w_{2,i}^2}} \quad (2)$$

In this case, permutations of words in a string will not change anything.

If we use dictionaries that give the stem word form for each word (without cases, endings, etc.), we can get rid of the problems of counting the same words in different components of vectors. The process is called lemmatization [23] or lemma normalization. In the simplest case, the Catvar dictionary can be used [24]. Here, for each word from the commit message (column 1), its normalized form can be obtained (column 2), for example, here is a dictionary fragment for the words “fix”:

fix	fix	\$N\$
fix	fix	\$V+0\$
fixed	fix	\$V+ed\$
fixed	fix	\$V+en\$
fixes	fix	\$N+s\$
fixes	fix	\$N+s\$
fixes	fix	\$V+s\$
fixing	fix	\$V+ing\$

In more advanced cases, the StanfordCoreNLP API [25], [26] as a Java library can be used. Since not only the stem, but also the part of speech is known for each word, when converting

phrases into vectors, it is advisable to filter them, cutting off articles and frequently used words.

For the purposes of searching for strings with “strong components” or relevant words/tokens (i.e., to reduce the weights of frequently occurring words in a string), the tf-idf approach [27] can be applied. It does frequency counting, and with this, the vector components (features) instead of word appearance (1 or 0) will contain tf-idf weights. If we denote n_w as the number of occurrences of the word w into a commit message $m \in M$, and n_w as the total number of words in the document, and $|M|$ as the total number of messages, then:

$$\text{tf-idf}(w, m, M) := \text{tf}(w, m) \times \text{idf}(w, M) = \frac{n_w}{\sum_k n_k} \times \log \frac{|M|}{|\{m_i \in M \mid w \in d_i\}|} \quad (3)$$

If we are able to vectorize commit messages, then it makes sense to try to cluster them automatically. Clustering or cluster analysis involves the vectorization of given objects, calculating the distances between them according to a certain metric and dividing objects into clusters or groups of nearby objects. Vectorization involves the selection of key entities of objects and their presentation as a set of vectors of the same dimension. The clustering algorithm is a function $X \rightarrow Y$ that assigns a cluster identifier $y \in Y$ to any object $x \in X$. Some popular clustering algorithms are K-means, DBSCAN, and hierarchical clustering. The K-means algorithm iteratively minimizes the total square deviation of cluster points from the centers of these clusters [28]. The density-based spatial clustering of applications with noise (DBSCAN) algorithm groups points in a high-density area into one cluster, while marking lonely points as noise [29]. With hierarchical clustering, a tree (dendrogram) is built,

from leaves to root. Initially, each object is contained in its own cluster. Next, an iterative process of merging the two nearest clusters takes place until all clusters are combined into one, or the required number of clusters is found [30].

IV. ON THE IMPLEMENTATION OF OUR APPROACH

A. General approach

To work with git repositories, we use the JGit library. With it, we can iterate over commits and insert conditions for them. Therefore, it is suggested to use all available information for analysis: commit messages can be used to evaluate the most common error messages, and information about changed files will help to find, for example, drivers with a large number of bug fixes. The solution scheme is shown in Fig. 1. Preliminary experiments were described in [21], [22], in the present paper, we, first of all, generalize the methods for finding the nearest messages and perform clustering.

In this solution, we get all the commits of a given repository, filter them according to the dates of interest, and extract from them only those that explicitly indicate that this is a bug fix. In the current implementation, a list of key words in the message is just checked, but we keep in mind the discussion in [14]. We counted the number of commits and fixes over the past 5 years in Fig. 2, it is interesting that it almost does not change, apparently this is due to the workload of people who check them.

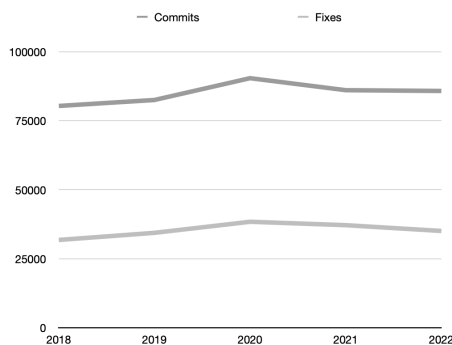


Fig. 2. The number of commits and fixes in torvalds/linux for the last five years

Next, we obtain the commit message and look for the one closest to it (*closestMessage*), delegating the responsibility for getting it to one of the implementations of the *IMsgMatcher* interface (see again Fig. 1). For the found message, we increase the weight. We also save information about files (*processFiles*) and lines with changes (*processLines*). After the loop over all commits is completed, we display a generalized result for the found messages by searching for a given number of messages with maximum weights, as well as information about the most modified files.

B. Method selection. Experimenting with the Thunderbolt Driver Repository

In order to compare the discussed methods, we ran a series of experiments using a relatively small Thunderbolt repository [31] with 650 fix commits. We used the discussed methods for $L()$ and $D()$ calculation; the last one is done with using preliminary lemmatization.



Fig. 3. A thunderbolt 2 network adapter

Thunderbolt (see Fig. 3) is a relatively new interface developed by Intel and Apple. However, this is nothing more than an output of the PCIe bus lanes to the outside. In the modern version, the standard is merged with USB4 [32], respectively, patches in the kernel address this topic.

The running time of all solutions is approximately equal (about 250 seconds). We used the kernel up to the commit *3ecc37918c80*. The top 10 error messages obtained and their weights are given below:

1. Using the Levenshtein distance $L(s_1, s_2)$:

```
Few fixes and cleanups. / 16
thunderbolt: Fix typo in comment / 14
thunderbolt: Fix -Wrestrict warning / 13
test from previous fixes. / 12
Fixes the following W=1 kernel build
warning(s): / 11
"thunderbolt: Fix for v6.0 final / 10
Fixes: dacb12877d92 ("thunderbolt: Add
support for on-board retimers") / 9
Fix this by saving pointer to the parent
device before calling / 9
usb: mtu3: fix failed runtime suspend in
host only mode / 9
Fixes: 046beelf9ab8 ("thunderbolt: Add
MSI-X support") / 8
```

2. Using the distance between the vectors $D(s_1, s_2)$:

```
thunderbolt: fix -wrestrict warning / 212
thunderbolt minor updates and fixes / 152
thunderbolt: fix typo in comment / 148
fixes for thunderbolt device dma
protection / 132
thunderbolt: fix a leak in tb_retimer_add
() / 128
fixes: 54e418106c76 ("thunderbolt: add
debugfs interface") / 120
thunderbolt: fix typos in clx enabling /
118
thunderbolt: fix for v5.9-rc6 / 118
thunderbolt: fixes for v5.9-rc4 / 118
fixes and features: / 116
thunderbolt: fix to check for kmemdup
failure / 116
```

3. Using distance between vectors $D(s_1, s_2)$ + TF-IDF:

```
minor driver fixes and improvements over
the usb tree / 608
so add a quirk that fixes it. we also
need to expand the quirk table to /
608
lots of tiny dwc3 fixes and updates for
the ip block that is / 606
usb gadget fixes and additions all over
the place / 606
force enum tb_drom_entry_type to unsigned
to fix the following error: / 606
other tiny janitorial and cleanups fixes
for lots of different usb / 604
fix a few typos and wording mistakes in
the acpi device enumeration / 604
thunderbolt: fix spelling mistake "
simultaneously" -> "simultaneously" /
604
```

4. Using distance between vectors $D(s_1, s_2)$ + TF-IDF inverse:

```
thunderbolt: fix typo in comment / 20
thunderbolt: switch: fix kernel-doc
descriptions of non-static functions
/ 18
thunderbolt: nhi: fix kernel-doc
descriptions of non-static functions
/ 16
thunderbolt: path: fix kernel-doc
descriptions of non-static functions
/ 16
thunderbolt: eeprom: fix kernel-doc
descriptions of non-static functions
/ 16
thunderbolt: ctl: fix kernel-doc
descriptions of non-static functions
/ 16
few minor cleanups and fixes / 16
thunderbolt: fix to check the return
value of kmemdup / 16
few fixes and cleanups. / 14
thunderbolt: fix some kernel-doc comments
/ 14
```

As a result, it can be seen that the conclusions of the analysis depend on the method used. In the first case, we see that a large number of messages were not very informative, and there were important fixes related to pointers, sleep mode, and interrupts. The found messages generally confirm our knowledge of the subject area – in the top there are fixes about USB and PCIe (MSI-X is used to deliver interrupts). In the second case, we also see fixes related to typos, as well as the well-known DMA vulnerability (see for example an analysis of this known issue in [33]) and memory leaks. The conclusion of the third case is very different and here, apparently, the messages turned out to be almost all unique and specific to the given area. The last output is generally similar to the first two, but there are almost all repeated

phrases with minor changes that need to be manually cleaned up.

C. Clustering results for Thunderbolt

We selected the cosine similarity method with tf-idf. We used the clusterization with some pre-defined cluster centers from which we selected 8 more or less relevant. The resulting centroids are presented below:

1. 5.47% [support, xdomain, lane, tunnel]
2. 4.50% [warning, gadget, usb, over]
3. 3.86% [return, check, value, kmemdup]
4. 3.54% [cleanup, minor, kunit, documentation]
5. 3.22% [spelling, mistake, usb, seq_puts]
6. 2.57% [typo, comment, enabling, clx]
7. 2.57% [doc, kernel, static, description]
8. 2.25% [leak, memory, sw, failure]

The percentage here is calculated from the number of messages containing information about corrections. There can be several such fixes in one commit. Let us try to discuss the nature of these fixes:

- 1st cluster: *xdomain*-related bug fixes and fixes for DMA vulnerabilities, aimed at ensuring the security and protection of the system from possible attacks, associated with the use of incorrect memory access.
- 2nd cluster: fixes for build-time warnings related to use the gcc compiler for the Linux kernel, aimed at eliminating warning messages that occur when compiling the driver code. These fixes do not change the functionality of the code, but rather are intended to provide a cleaner and safer build of the project.
- 3rd cluster: fixes for problems that occur when a null pointer is returned from the *kmemdup* function in case of memory allocation failure. Fixes include checking the return values of functions and handling cases when the memory allocation failed, in order to take appropriate measures to restore or gracefully terminate the execution.
- 4th, 7th clusters: corrections to documentation in the Linux kernel: fixing inaccuracies, errors, and out-of-date information, as well as supplementing incomplete or insufficient informative comments. The purpose of such corrections is to ensure that the documentation is accurate, actual, and complete. Fixes may also include improvements to readability, style, and conformance of comments to code.
- 5th, 6th clusters: classes of fixes related to minor changes and code cleanup in the Linux kernel, includes fixes for formatting errors and other minor issues. These changes do not affect the functionality of the code. Such fixes usually include indentation, alignment, removal of extra spaces, correct use of indentation and other similar changes, which help to preserve code formatting conventions.

- 8th cluster: fixes for memory leak bugs related to improper release of resources. This includes situations where memory release does not occur or occurs at the wrong time or place.

Analyzing the clusters found, it seems that the most representative errors are related to the quality of the code. This coincides on the whole with our previous studies on the whole kernel [21], [22]. Such errors, however, can shadow the so-called domain-specific errors (related precisely to the nature of the code in a particular repository).

At the same time, the relatively low number of fixing commits prevents us from determining a large number of them. There is also the problem of how to isolate the most interesting specific cases that are representative. For example, for the generalized message [*typo, comment, enabling, clx*], the most interesting (domain-specific) word is *clx* (possible related to CLx low-power link states), however, the words here are ordered in descending order of the word weight metric and *clx* is less interesting according to the aglorhythmic point of view. Nevertheless, if we search for all such words with inverse tf-idf in front, we get poor grouping results. We will try to solve this in future works.

V. CONCLUSION

Using the presented methods of data analysis, we are relatively quickly able to obtain both the most representative error messages of error fixes. At the moment, we have developed an infrastructure for bypassing commits and analyzing them with two methods of analyzing similar strings and clustering the results to obtain generalized error classes. The results we obtained can be used in teaching both system programming and data analysis.

This way we can quickly get into the problems of a particular Linux driver or component and take a retrospective of changes, since we can easily select the paths in the git for analysis and set the time from and to which we need to analyze the commits.

The current software is written in Kotlin and Python, presented in [34] and will be further improved.

In the future, we are going to analyze the main components of the Linux operating system kernel and discuss the identified bugs in the form of a separate article. It would also be interesting to obtain a retrospective of error classes by years.

REFERENCES

- [1] *git*. [Online]. Available: <https://git-scm.com>
- [2] GitHub, *Proposing changes to your work with pull requests*. [Online]. Available: <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests>
- [3] L. Torvalds, *Linux kernel*. [Online]. Available: <https://github.com/torvalds/linux/>
- [4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 73–88.
- [5] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in Linux: Ten years later," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011, pp. 305–318.
- [6] V. Mutilin, E. Novikov, and A. Khoroshilov, "Analysis of typical errors in Linux OS drivers (in Russian)," *Proceedings of the Institute for System Programming of the Russian Academy of Sciences*, vol. 22, pp. 349–374, 2012. [Online]. Available: <https://www.elibrary.ru/item.asp?id=20278337>
- [7] E. M. Novikov, "Evolution of the Linux OS kernel (in Russian)," *Proceedings of the Institute for System Programming of the Russian Academy of Sciences*, vol. 29, no. 2, pp. 77–96, 2017. [Online]. Available: <https://www.elibrary.ru/item.asp?id=29118078>
- [8] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of Linux file system evolution," *ACM Transactions on Storage (TOS)*, vol. 10, no. 1, pp. 1–32, 2014.
- [9] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical software engineering*, vol. 19, pp. 1665–1705, 2014.
- [10] G. Xiao, Z. Zheng, B. Yin, K. S. Trivedi, X. Du, and K.-Y. Cai, "An empirical study of fault triggers in the Linux operating system: An evolutionary perspective," *IEEE Transactions on Reliability*, vol. 68, no. 4, pp. 1356–1383, 2019.
- [11] *Kernel.org Bugzilla*. [Online]. Available: <https://bugzilla.kernel.org>
- [12] J. Melo, E. Flesborg, C. Brabrand, and A. Wasowski, "A quantitative analysis of variability warnings in Linux," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, 2016, pp. 3–8.
- [13] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo, "PatchNet: Hierarchical deep learning-based stable patch identification for the Linux kernel," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2471–2486, 2019.
- [14] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in *2012 34th international conference on software engineering (ICSE)*. IEEE, 2012, pp. 386–396.
- [15] M. Acher, H. Martin, J. A. Pereira, A. Blouin, D. E. Khelladi, and J.-M. Jézéquel, "Learning from thousands of build failures of Linux kernel configurations," Ph.D. dissertation, Inria; IRISA, 2019.
- [16] S. Chacon and B. Straub, *Pro git*. Springer Nature, 2014.
- [17] *JGit – Eclipse*. [Online]. Available: <https://eclipse.org/jgit/>
- [18] *EGit*. [Online]. Available: <https://www.eclipse.org/egit/>
- [19] S. Staroletov, "Model checking games and a genome sequence search," in *Journal of Physics: Conference Series*, vol. 1679, no. 3, 2020, p. 032020.
- [20] V. I. Levenshtein, "Binary codes with correction of dropouts, insertions and character substitutions (in Russian)," in *Reports of the Academy of Sciences*, vol. 163, no. 4. Russian Academy of Sciences, 1965, pp. 845–848.
- [21] S. M. Staroletov, "Researching the most common bugs in the Linux kernel by analysing commits in the git repository (in Russian)," *System Administrator*, vol. 4(197), pp. 73–77, 2019. <http://samag.ru/archive/article/3859>. [Online]. Available: <https://www.elibrary.ru/item.asp?id=37252881>
- [22] S. Staroletov, "A survey of most common errors in Linux kernel," *SYRCoSE Poster session*, 2017.
- [23] M. Hann, "Towards an algorithmic methodology of lemmatization," *Bulletin Association for Literary and Linguistic Computing*, vol. 3, no. 2, pp. 140–150, 1975.
- [24] *Categorical Variation Database (version 2.1)*. [Online]. Available: <https://github.com/nizarhabash1/catvar>
- [25] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.
- [26] *Class StanfordCoreNLP*. [Online]. Available: <https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/pipeline/StanfordCoreNLP.html>
- [27] G. Salton, E. A. Fox, and H. Wu, "Extended boolean information retrieval," *Communications of the ACM*, vol. 26, no. 11, pp. 1022–1036, 1983.
- [28] H. Steinhaus *et al.*, "Sur la division des corps matériels en parties," *Bull. Acad. Polon. Sci*, vol. 1, no. 804, p. 801, 1956.
- [29] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *kdd*, vol. 96, no. 34, 1996, pp. 226–231.
- [30] J. H. Ward Jr, "Hierarchical grouping to optimize an objective function," *Journal of the American statistical association*, vol. 58, no. 301, pp. 236–244, 1963.
- [31] *Drivers – Thunderbolt*. [Online]. Available: <https://github.com/torvalds/linux/tree/master/drivers/thunderbolt>
- [32] *USB Type-C System Overview*. [Online]. Available: <https://www.usb.org/sites/default/files/D1T1-2%20-%20USB%20Type-C%20System%20Overview.pdf>
- [33] R. Sevinsky, *Funderbolt Adventures in Thunderbolt DMA*, 2013. [Online]. Available: <https://media.blackhat.com/us-13/US-13-Sevinsky-Funderbolt-Adventures-in-Thunderbolt-DMA-Attacks-Slides.pdf>
- [34] *Linux Kernel Analysis*, 2023. [Online]. Available: <https://github.com/NicolayGolovnev/LinuxKernelAnalysis>