JOURNAL OF INFORMATION SYSTEMS ENGINEERING & MANAGEMENT

Journal of Information Systems Engineering & Management, 2018, 3(2), 17
ISSN: 2468-4376

lectito

Predicting Software Flaws with Low Complexity Models based on Static Analysis Data

Lucas Kanashiro [1], Athos Ribeiro [1], David Silva [2], Paulo Meirelles [1, 3*], Antonio Terceiro [4]

[1] *FLOSS Competence Center (CCSL), University of São Paulo (USP),* São Paulo, BRAZIL
[2] *UnB Faculty in Gama (FGA), University of Brasília (UnB),* Brasília, BRAZIL
[3] *Department of Health Informatics, Federal University of São Paulo (UNIFESP),* São Paulo, BRAZIL
[4] *QA Services Team, Linaro Limited,* Curitiba, BRAZIL

*__Corresponding Author:__ paulormm@ime.usp.br

Citation: Kanashiro, L., Ribeiro, A., Silva, D., Meirelles, P. and Terceiro, A. (2018). Predicting Software Flaws with Low Complexity Models based on Static Analysis Data. *Journal of Information Systems Engineering & Management*, 3(2), 17. https://doi.org/10.20897/jisem.201817

Published: April 14, 2018

ABSTRACT

Due to the constant evolution of technology, each day brings new programming languages, development paradigms, and ways of evaluating processes. This is no different with source code metrics, where there is always new metric classes. To use a software metric to support decisions, it is necessary to understand how to perform the metric collection, calculation, interpretation, and analysis. The tasks of collecting and calculating source code metrics are most often automated, but how should we monitor them during the software development cycle? Our research aims to assist the software engineer to monitor metrics of vulnerability threats present in the source code through a reference prediction model, considering that real world software have non-functional security requirements, which implies the need to know how to monitor these requirements during the software development cycle. As a first result, this paper presents an empirical study on the evolution of the Linux project. Based on static analysis data, we propose low complexity models to study flaws in the Linux source code. About 391 versions of the project were analyzed by mining the official Linux repository using an approach that can be reproduced to perform similar studies. Our results show that it is possible to predict the number of warnings triggered by a static analyzer for a given software project revision as long as the software is continuously monitored.

Keywords: source code static analysis, source code metrics, common weakness enumeration, prediction, linux

INTRODUCTION

Source code static analysis is a good means to provide inputs to support software quality assurance. For instance, these inputs may be software structural attributes such as object-oriented metrics as presented by Chidamber and Kemerer (2002). Attributes related to the possible behaviors of the software at execution time (Ernst, 2005), which includes analyses of behaviors that may lead to security issues, could also serve as input provided for quality assurance (Ferzund et al., 2009; Misra and Bhavsar, 2003; Nagappan et al., 2006).

From the source code static analysis point of view, a vulnerability is the result of one or more flaws in software requirements, design, implementation, or deployment (Black et al., 2007). Vulnerability correction can be very expensive in later stages of a software development cycle, hence the importance of finding and correcting flaws in early stages, before they expose actual vulnerabilities.

Although no standard to define software flaws is widely accepted, some organizations define and classify them. The most accepted form is the one defined by MITRE, where software common flaws are cataloged as CWEs[1] (Common Weakness Enumeration). This catalog establishes a unique language to describe known software flaws, like divisions by zero and buffer overflows.

There are several static analyzers that can detect flaws (Black, 2001). With the support of such tools, this work presents a method to define low complexity models for the amount of flaws found in a software through a case study of two different types of common weaknesses detected in Linux: **Use of Uninitialized Variable** and **NULL pointer Dereference**. Thus, we have obtained low complexity polynomial functions to describe the number of these flaws present in Linux over time.

To show our ideas and results in this context, the remainder of this paper is organized as follows: the next section describes several related works investigated in this research. The third section shows the research design, discussing our research question as well as presenting the data collection and analysis approach. The fourth section presents the candidate low complexity models to predict flaws in Linux. The fifth section discusses the results, selecting and applying the candidate low complexity models. The last section concludes the paper, highlighting its main contributions and pointing to possibilities for future work.

RELATED WORKS

Evans and Larochelle (2002) states that although security vulnerabilities are well understood, it is not a common practice to include techniques to detect or avoid them in the development processes and suggests that instead of solely relying on programmers, tools to prevent and detect software vulnerabilities should be built and integrated in software development cycles. It is also known that static analysis helps programmers detect software vulnerabilities before they reach production (Evans and Larochelle, 2002) in a cheaper way than solely conducting manual inspections (Johnson et al., 2013), thus the interest in static analyzers.

Zheng et al. (2006) analyzed the effectiveness of security oriented static analyzers using tests and the number of failures reported by customers as parameters. They concluded that static analyzers are effective tools to find flaws in source code. However, a solution on how to monitor such flaws is not presented. Our research aims to find solutions for this issue.

Pan et al. (2006) proposes a method to predict if a file or function in a given project is buggy or bug-free based on the correlation between code quality metrics, like size, complexity and coupling, and software bugs. The authors used static analyzers to extract data from 76 revisions of Latex2rtf and 887 revisions of Apache HTTP so they could use the 10-fold cross-validation method with a machine learning algorithm to perform their predictions. The prediction precision was over 80% for files and over 70% for functions. The authors declared that the costs to run the static analysis to generate the method inputs were too high for large projects. In this paper we are testing low cost functions to predict some flaws, but running the analyses is still needed.

In another work, Penta et al. (2008) performed an analysis of the evolution of vulnerabilities detected by static analyzers in three different Free Software applications: Samba, Squid, and Horde. The authors analyzed how the number of vulnerabilities varies over time. Since they analyzed different development versions (not only releases), they could analyze aspects of the development process, like bug fixing efforts right before a release. The focus of this work was to understand how long vulnerabilities tend to remain in the system by modeling the vulnerabilities decay time with a statistical distribution and comparing the decay time of different classes of vulnerabilities. Our research is based on the hypothesis that to effectively take action on such bug fixing effort predictions, monitoring the number of flaws or the flaws decay time should be done continuously, during the software development cycle. This work is a first step towards such automation. First, we present a model to predict the number of flaws present in a specific version of the Linux kernel. Then, we show that such model must be continuously updated to be effective, i.e., the greater the difference between the last version used to train the model and the version used for the predictions, the less effective are the predictions made.

RESEARCH DESIGN

To guide the development of a reproducible method to define low complexity functions that enable the longitudinal study of software flaws, we addressed the following research question:

Q1 *– Is it possible to define low complexity models to predict flaws in the Linux source code?*

[1] cwe.mitre.org

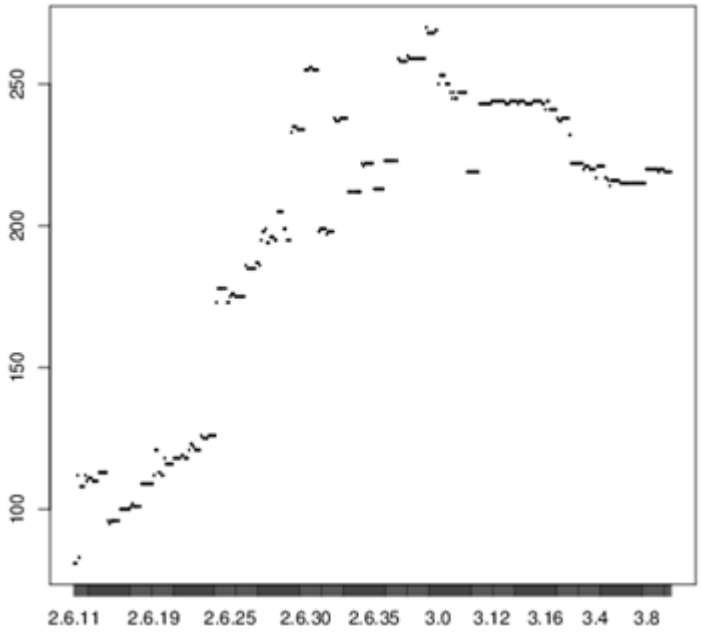

Figure 1. CWE 476 dereferences evolution

With low complexity functions, we are able to automate the continuous monitoring of flaws in a simple way, without the need to build complex models, which would be expensive to maintain. If we can easily infer the number of flaws predicted in a software as well, improvements in the software development cycle based on it become possible.

We could, for example, make better decisions on when to increase the efforts to fix bugs or refactor code.

Data Collection

Although it is common to refer to the whole operating system as Linux, the name refers to the kernel. Its development has been active for over 20 years and it is written in the C programming language. The Linux source code showed itself adequate for this study due to its size and the fact that it is widely used around the world, which led us to believe there is a higher likelihood of finding flaws in it. A total of 391 versions of the kernel were analyzed. These version were downloaded from the project's official Git repositories[2]. By the time of our first analyses, the versions between 2.6.11 and 3.9 were available. After that, we have tested the candidate models in three recent kernel versions: 4.0, 4.5, and 4.9.

The static analyzer selected was Cppcheck. According to the description in its homepage[3], it aims to be a **sound** (Black, 2001) static analyzer, since its main goal is to find real bugs in the source code, not generating false positives. The main characteristic of sound static analyzers is the low rate of false positives. The Cppcheck output is composed of a location (source code line) and a message, describing a problem (warning or error). We ran Cppcheck 1.67 in each version of Linux and the corresponding output files for each of the analyzed versions are publicly available at github.com/lucaskanashiro/linux-analysis.

Data Analysis

The risk of Linux versions with more modules to stand out compared to other versions was identified upon analyzing the absolute number of flaws found. Thus, the number of flaws was normalized based on the number of modules in each version, resulting in the **flaw rate** per module that is our dependent variable. To analyze the influence of the project growth, we used the **total number of modules** as our independent variable. In this context, a kernel module is one C programming language source code file (header files are not counted for this matter).

Since there are several known flaws, we have conducted a previous study with 10 popular Free Software projects[4] from different domains to find the most recurrent flaws and select them to perform the analyses for this work. Since this paper proposes to find one model per flaw analyzed, we selected the two most frequent flaws found in those projects: **NULL pointer dereference** and use of **uninitialized variable**. Both are cataloged by MITRE and as Common Weakness Enumerations, the former as CWE476 and the latter as CWE457.

[2] git.kernel.org

[3] cppcheck.sourceforge.net

[4] Bash, Blender, FFmpeg, Firefox, Gstreamer, Inetutils, OpenSSH, OpenSSL, Python2.7, and Ruby-2.1

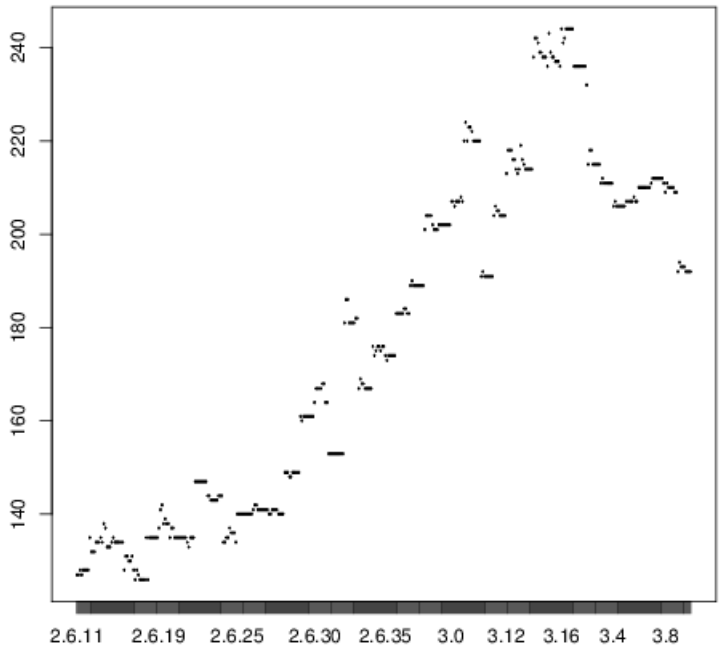

Figure 2. CWE 457 dereferences evolution

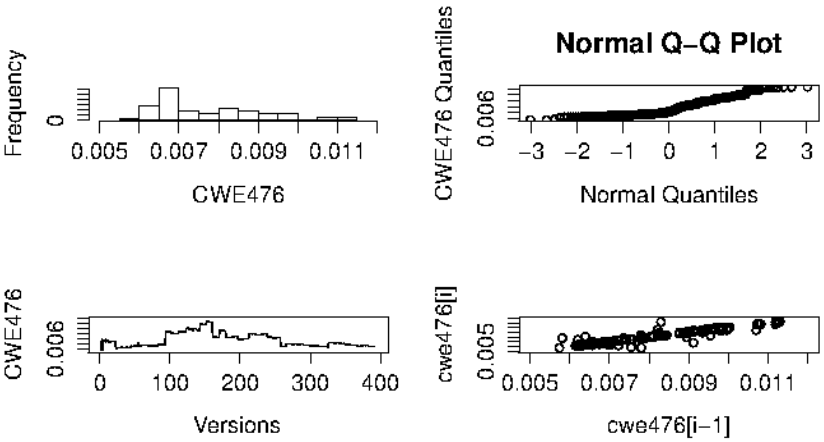

Figure 3. 4-Plot for CWE 476 – Number of NULL pointer

As we can see in **Figures 1** and **2**, in the course of Linux evolution, the total number of flaws dereferences does not always increase. It increases until a certain point and then there is a tendency towards stabilization, which is, most likely, the result of the Linux development process evolution. To understand the behavior of the data we performed some analyses, such as correlation matrix between the flaws and the number of modules, using the Pearson correlation coefficient (Rossman, 1996).

Also, to confirm that the data can not be represented by a linear function, we used the 4-plot technique developed by NIST[5]. This technique facilitates the understanding of the data distribution and its randomness. For this study, identifying these characteristics was necessary to obtain information for the definition of a model which satisfies the studied data set, or that could, at least, eliminate unwanted options. For Linux, the data does not follow a normal distribution, but a long-tailed distribution, so that it is not constant and varies widely, as we can see in **Figures 3** and **4**. Both analyses showed that the flaw rates were not strongly correlated and they did not fit a normal distribution, which corroborates with the fact that exponential and weibull distributions can model the behavior of defect occurrence in a wide range of software (Penta, 2002; Jones, 1991; Li et al., 2004; Wood, 1996). With this in mind we discarded the possibility of using a linear model.

Since our proposal is based on low complexity models and linear functions do not fit the data set, we investigated polynomial models. Initially, an identification of outliers (Hawkins, 1980) work was done for the definition of the polynomial models through a technique presented by Tukey (1977), using a box plot. The outliers found were removed to improve the candidate model accuracy. After that, parametric and non-parametric models

[5] itl.nist.gov/div898/handbook/eda/section3/eda3332.htm

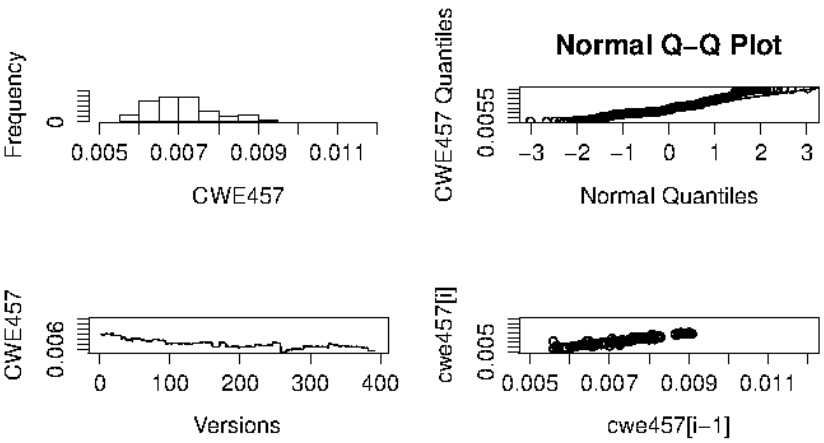

Figure 4. 4-Plot for CWE 457 – Use of uninitialized variable

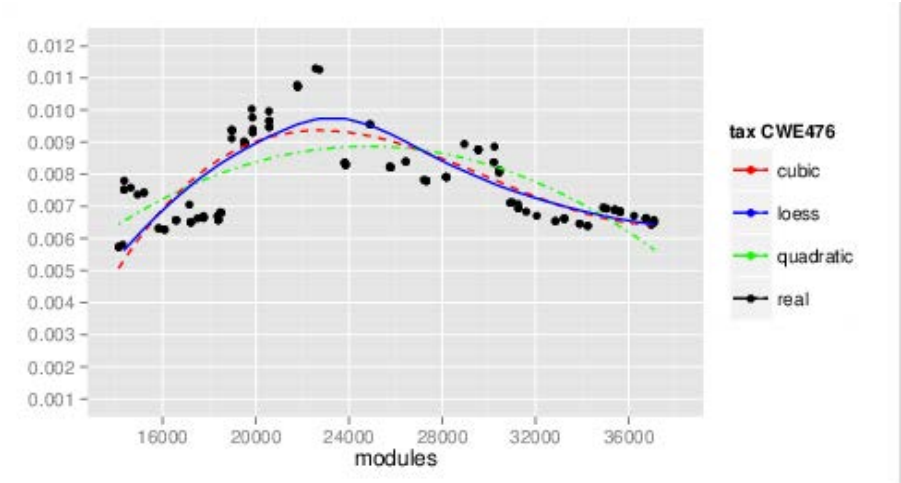

Figure 5. CWE 476 data models prediction curve

were built. The non-parametric model is a black box method from which it is not possible to obtain a mathematical function. In this study, it is used as a base for the definition of a parametric model. As a result, we have a function that receives the total number of modules in the project as input and returns the flaw rate as output.

We also have used the Locally Weighted Regression (LOESS) non-parametric method to provide a smooth surface using a non-parametric regression (Cleveland and Devlin, 1988). In short, the LOESS method applies several minor regressions to the data set as well as it can guide the definition of a possible parametric model. Therefore, we have applied polynomial regressions to obtain the parametric models definition for this study.Posteriorly, to validate the obtained models, we used the K-Fold cross-validation technique, which consists in dividing the data set in *K* groups, where one of these groups is used to test the model and the other groups are used to train the model (Picard and Cook, 1984). With this kind of cross-validation, we can test the model with values from different versions of the kernel, obtaining the model prediction error. We use the mean squared error, which guarantees the model reliability, given that this error has a low value. After the validation, the model errors were compared and we could chose the best model, getting to low complexity (polynomial) functions capable of predicting the Linux source code flaw rates proposed in this paper.

Finally, we have tested the candidate models collecting new data from recent Linux versions (such as 4.0, 4.5, and 4.9) and comparing them to the predicted values from our low complexity polynomial functions

LOW COMPLEXITY MODELS DEFINITION

To achieve a lower error, we could use a polynomial function with a higher degree. However, to build a suitable model, we need to avoid overfitting, the extreme adjustment of the model to our data set, which would disturb the extrapolation for future inputs. Our approach based on a non-parametric model as reference helps avoiding a possible overfitting of the built model, providing a more flexible model.

For the models definition, we identified and removed the outliers in the data set, defined the non-parametric model with the LOESS method and, finally, defined two parametric models through polynomial regression: one

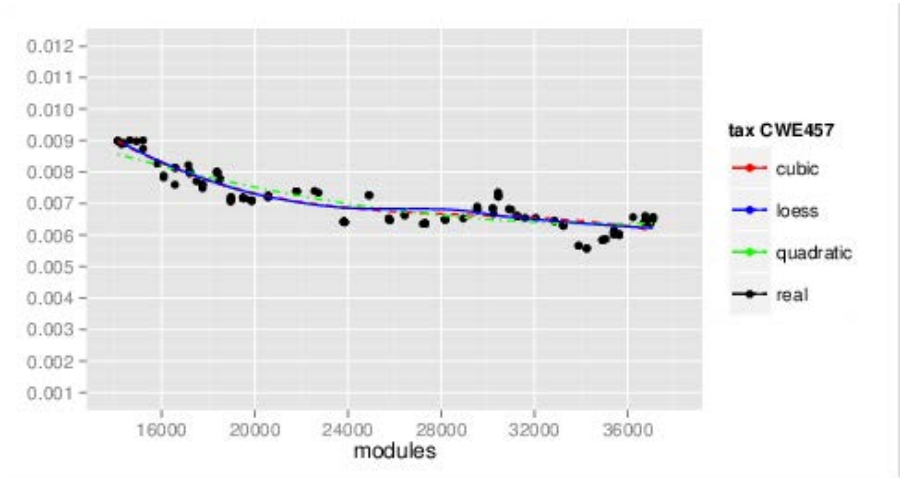

Figure 6. CWE 457 data models prediction curve

Table 1. Error Obtained Through cross-validation

Model	CWE	Mean squared error
Quadratic	476	0.000000958
Cubic	476	0.000000862
Quadratic	457	0.000000137
Cubic	457	0.000000112

quadratic polynomial and one cubic polynomial. In **Figures 5** and **6**, we present the "real" flaw rate values and flaws evolution curves ("cubic", "quadratic" and "LOESS") in the built models.

Figure 5 shows that the built cubic model is the one with the best approximation for the reference model curve (LOESS model). It is especially good for data extrapolation on data sets higher or lower than the data interval used in this study. On the edges, the cubic model has a better approximation from the reference model, while the quadratic model tends to diverge from it. Still, although all the models presented in **Figure 6** have a good approximation for the reference model curve, the cubic model fits the reference model better than the quadratic one, both in the minimum and maximum limits.

Finally, to compare the models, we performed a cross-validation with the K-fold method to analyze the performance of both models in a data set that was not used for training those models. In short, we used a ten-fold cross-validation ($K = 10$) for this study, which means dividing the sample in ten groups for training and testing (Kohavi, 1995). **Table 1** presents the mean squared error (MSE), associated to each of the models obtained through the cross-validation. For instance, the MSE measures the quality of a prediction model, so its values closer to zero are better, and they always are non-negatives. Therefore, the low error ($MSE < 0.000001$), obtained from the presented functions, guarantees the models reliability.

For both CWEs, the cubic model stood out. The error associated with the quadratic model for CWE476 was approximately 11.14% higher than the cubic model in "real" analysis situations, while the error associated with the quadratic model for CWE457 was approximately 22.32% higher than the cubic model[6]. Therefore, in both cases, the cubic models are better for predicting the Linux flaw rate evolution, since they show a better approximation from the reference models when it comes to the extrapolation of the analyzed data boundaries.

RESULTS

For each CWE studied, two models were defined and one was selected. Thus, it is possible to use low complexity functions, with low error, to define a model to predict flaws in Linux, answering our research question. This is feasible when we can obtain a meaningful data set, as it was the case for Linux, where we used 391 different versions of the project. In short, comparing and selecting the models for each CWE, we have Equations 1 and 2 representing the selected cubic models:

$$\begin{aligned} tax\_CWE476(modules) = {} & (1.911224 * 10^{-15}) * modules^3 \\ & - (1.72028 * 10^{-19}) * modules^2 \\ & + (4.857479 * 10^{-6}) * modules \\ & - (0.03460173) \end{aligned} \quad (1)$$

[6] These numbers can be verified by comparing the values in Table I, dividing the quadratic error values by the cubic ones

Table 2. Some flaw rates and predictions

Version	tax_CWE476	tax CWE457	Modules
linux-v2.6.11	0.005735325	0.008992424	14123
linux-v2.6.39	0.008927095	-	30245
linux-v3.16-rc3	-	0.006577706	37095
linux-v3.9-rc8	0.006460368	0.005663884	33899
linux-v4.0.0	0.006435337	0.006036167	37793
linux-v4.5.0	0.006829370	0.005503281	40209
linux-v4.9.0	0.007679160	0.004861321	42234

Table 3. Comparison between model and real value CWE 476

Version	model_tax_CWE476	real_ tax_CWE476
linux-v4.0	0.006435337	0.006085783
linux-v4.5	0.006829370	0.005968813
linux-v4.9	0.007679160	0.005729981

Table 4. Comparison between model and real value CWE 457

Version	model_tax_CWE457	real_ tax_CWE457
linux-v4.0	0.006036167	0.006562062
linux-v4.5	0.005503281	0.006615434
linux-v4.9	0.004861321	0.006179855

$$\begin{aligned} tax\_CWE457(modules) = {} & (-6.466983 * 10^{-16}) * modules^3 \\ & + (5.603787 * 10^{-11}) * modules^2 \\ & - (1.639652 * 10^{-16}) * modules \\ & + (0.02287291) \end{aligned} \quad (2)$$

It can be verified that the coefficients of the equations 1 and 2 are low values, since the flaw rates are also low. It is uncommon to find buggy functions in Linux (Ferreira et al., 2016). So, when this already low value is divided by the number of modules, which tends to increase over time, we get very low values for the flaw rates.

Table 2 presents the defined models with the flaw rates in some known points, as first collected version (2.6.11), last analyzed version (3.9), the version with the higher flaw rates (2.6.39 for CWE476 and 3.16 for CWE457), as well as, recent Linux versions for prediction points (4.0, 4.5, and 4.9).

In the one hand, the flaw rate for CWE476 (NULL pointer dereference) peaked in version 2.6.39 and then started to decrease, but as soon as the number of modules starts to increase, it tends to increase, reaching a new peak and finally starting to decrease again. In the other hand, the flaw rate for CWE457 (Uninitialized variables) tends to decrease, as shown in **Table 2**. We can observe that the flaws always oscillate with a few peaks, but the values tend to decrease. The values for CWE476 emphasize that fact, since the flaw rate on the second peak is lower than the previous one.

After aplying the models to kernel versions later than 3.9, we compared the real flaw rate values and the values estimated by our model. We obtained the real values by dividing the number of occurrences of CWE476 and CWE457 by the number of modules of the respective kernel versions. The real values are shown in **Tables 3** and **4**.

By analyzing the real values and the values proposed by the models for Linux versions greater than 3.9, we can see that the further the latest version used to train the model is from the version analyzed, the greater the error on its predictions. This leads to the need of updating the model for a certain interval of new Linux releases, making it possible to reduce the error, improving the precision of the statistical model predictions. An interesting means to achieve such updates is by performing continuous static analysis on software repositories (the kernel repository in this specific context) and automate the model updating tasks. In the next Section we discuss our final remarks and our plans for future work, including the development of a platform to perform continuous static analysis on software repositories and make the analyses available.

FINAL REMARKS

By identifying frequent flaws in source code, such as NULL pointer dereferences and uses of uninitialized variables (CWE476 and CWE457, respectively), a series of exploratory analyses were carried out and we defined low complexity models to predict the mentioned flaws. We observed, from Cppcheck reports, that source code flaws are not easily detected in Linux, which gave us a small number of flaws per module in the performed analysis.

To test the defined models, we also analyzed Linux versions that were more recent than the ones analyzed to define the selected functions, using the presented models to predict the flaw rates, and comparing the obtained values with the results of static analysis. The results showed that it is possible to predict the number of flaws

triggered by the static analyzer for a given version of the kernel, as long as the revisions used for training the model are not distant from the version we want to generate predictions for. It is important to say that this may not be true for other software, since it depends on how much is changed between versions. But the opposite also holds: the model may work better for software with smaller deltas between versions and this should be investigated in future work.

In this study, we did not consider the possibility of the used static analyzer to report false alarms (Landi, 1992) (false positives), increasing the flaw rate of this work with fake data (not real flaws), or false negatives, omitting flaws from our research, which would decrease our flaw rates. Even with this limitation, since the used sample is large, statistically, the proposed models are valid. They could also be applied with other sources for the flaws, like manually mining repositories for bug fixes or searching mailing lists for confirmed bug reports, cases where most flaws, if not all of them, are positives. We did not follow this approach because we intend to scale this research using an automated approach.

In our analysis, we could observe that the presented rates of flaws per module have peaks during the evolution of Linux's source code, that represent phases of source code's growth, where flaws were added, and refactoring phases, where flaws were fixed. Thus, a next step for this research would be applying the analysis of other factors that influence the variation of source code flaws, such as the number of developers who are working with quality assurance. This would make it possible to define multivariate models to aid software engineers on making decisions around the software development cycle.Moreover, from the defined low complexity models analyses, we want to identify some aspects of the development process as a next step, besides studying the flaws history in the course of different releases. For instance, the models could be used to analyze the development cycle in specific moments, such as new features development (increase in number of flaws) and moments of stabilization, refactoring, and bug fixing (decrease in number of flaws) through Linux development. The reason to build less complex models is the fact that they will require less effort to perform future analysis. In the future, we want to automate all these analyses and perform continuous monitoring of flaws in source code.

In fact, our next step involves developing a platform to perform static analysis on different software repositories with different static analyzers. This will provide the data needed to perform studies like the present research and take decisions to improve the software development cycle and the software quality itself. We also intend to open the collected data, which might be useful for other researchers and software developers.

ACKNOWLEDGEMENTS

National Council for the Improvement of Higher Education (CAPES) for supporting Lucas Kanashiro and Athos Ribeiro during the production of this paper. Also, Fabio Kon (University of São Paulo) for reviewing and providing important feedback on this paper.

REFERENCES

Black, P. E. (2001). Static analyzers in software engineering.

Black, P. E., Kass, M. and Koo, M. (2007). Source code security analysis tool functional specification version 1.0. *In Special Publication (NIST SP) 500-268, NIST*, pp. 5. https://doi.org/10.6028/NIST.SP.500-268

Chidamber, S. and Kemerer, C. (2002). *A metrics suite for object oriented design*. MIT, Cambridge, MA, USA.

Cleveland, W. S. and Devlin, S. J. (1988). Locally weighted regression: An approach to regression analysis by local fittting.

Ernst, M. D. (2005). *Static and dynamic analysis: synergy and duality*.

Evans, D. and Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), 42–51. https://doi.org/10.1109/52.976940

Ferreira, G., Malik, M. M., Kästner, C., Pfeffer, J. and Apel, S. (2016). Do #ifdefs influence the occurrence of vulnerabilities? An empirical study of the linux kernel. *CoRR,* vol. abs/1605.07032. [Online]. Available at: http://arxiv.org/abs/1605.07032. https://doi.org/10.1145/2934466.2934467

Ferzund, J., Ahsan, S. and Wotawa, F. (2009). Empirical evaluation of hunk metrics as bug predictors.

Hawkins, D. M. (1980). *Identification of outliers*. Londres, Chapman and Hall. https://doi.org/10.1007/978-94-015-3994-4

Johnson, B., Song, Y., Murphy-Hill, E. and Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681. https://doi.org/10.1109/ICSE.2013.6606613

Jones, W. D. (1991). Reliability models for very large software systems in industry. In *Software Reliability Engineering, International Symposium on. IEEE*, pp. 35–42. https://doi.org/10.1109/ISSRE.1991.145351

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection.

Landi, W. (1992). Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4), 323–337. https://doi.org/10.1145/161494.161501

Li, P. L., Shaw, M., Herbsleb, J., Ray, B. and Santhanam, P. (2004). Empirical evaluation of defect projection models for widely-deployed production software systems. *ACM SIGSOFT Software Engineering Notes*, 29(6), 263–272. https://doi.org/10.1145/1041685.1029930

Misra, S. and Bhavsar, V. (2003). Relationships between selected software measures and latent bug-density: Guidelines for improving quality.

Nagappan, N., Ball, T. and Zeller, A. (2006). Mining metrics to predict component failures.

Pan, K., Kim, S. and E. J. W. Jr. (2006). Bug Classification Using Program Slicing Metrics. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 31–42. https://doi.org/10.1109/SCAM.2006.6

Penta, M. D., Cerulo, L. and Aversano, L. (2008). The Evolution and Decay of Statically Detected Source Code Vulnerabilities. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 101–110. https://doi.org/10.1109/SCAM.2008.20

Picard, R. R. and Cook, R. D. (1984). Cross-validation of regression models. *Journal of the American Statistical Association*, 79(387), 575–583, 1984. https://doi.org/10.1080/01621459.1984.10478083

Rossman, A. J. (1996). *Workshop statistics: Discovery with data.*

Tukey, J. W. (1977). *Exploratory data analysis.*

Wood, A. (1996). Predicting software reliability. *Computer*, 29(11), 69–77. https://doi.org/10.1109/2.544240

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P. and Vouk, M. (2006). On the value of static analysis for fault detection in software.

