# Crash reproduction difficulty, an initial assessment

Cherry, Boris; Devroey, Xavier; Derakhshanfar, Pouria; Vanderose, Benoît

*Published in:*
Proceedings of the 19th Belgium-Netherlands Software Evolution Workshop (BENEVOL '20)

*Publication date:*
2020

*Document Version*
Publisher's PDF, also known as Version of record

# Crash reproduction difficulty, an initial assessment

Boris Cherry*, Xavier Devroey†, Pouria Derakhshanfar† and Benoît Vanderose*
†PReCISE, NaDI, Faculty of Computer Science, University of Namur, Namur, Belgium.
Email: boris.cherry@unamur.be, benoit.vanderose@unamur.be
†Delft University of Technology, Delft, The Netherlands. Email: x.d.m.devroey@tudelft.nl, p.derakhshanfar@tudelft.nl

***Abstract*—This study presents the initial step towards a thorough analysis of the difficulty to reproduce a crash using *search-based crash reproduction*. Traditionally, code size and complexity are considered representative indicators of the difficulty for search-based approaches, like *search-based unit test generation*, to generate tests. However, unlike unit test generation, crash reproduction does not seek to cover a set of behaviors but instead to generate one or more tests exercising a specific behavior reproducing a given crash. In this context, there is no guarantee that the indicators used for unit testing are still valid for crash reproduction. In this study, we seek to identify such indicators by considering various code metrics, code smells, and change metrics. We report our effort to collect those metrics for JCRASHPACK, a state-of-the-art crash reproduction benchmark, and an initial assessment by considering metrics individually. Our results show that although JCRASHPACK is larger than benchmarks used in previous studies, additional crashes should be added to improve its diversity and representativeness, and that no individual metric can be used to characterize the difficulty to reproduce a crash.**

***Index Terms*—Search-based crash reproduction, software measurement, code quality, change metrics.**

## I. INTRODUCTION

Information about an application crash, like a *stack trace* for Java applications, are usually reported to the developers through an issue tracker. Based on the report's information, the developers debug the software by identifying the root cause of the crash and applying a fix to the code. To ease their investigation, developers can start their debugging process by *reproducing* and *exposing* the crash, and (latter) write a test case to ensure that the fix does not induce regression errors [1]. Recent developments lead to the (partial) automation of the crash reproduction and exposure process. When a new issue is created, an automated process fetches the stack trace and try to generate a *crash reproducing test case* able to reproduce and expose the crash [2].

Various approaches have been developed to automate the generation of a crash reproducing test case [3]–[7]. Among those, search-based crash reproduction yields the best results by reproducing more crashes and generating helpful test cases [4]. Recently, a study [8] revisited the state-of-the-art approach by building JCRASHPACK [9], a benchmark containing crashes from open-source projects. They identified, among other challenges, code complexity and the difficulty of generating input data as the main barrier for search-based crash reproduction. This study, however, does not provide any indication about the *difficulty to automatically reproduce a given crash*. This is problematic both for practitioners and researchers. From a practitioner perspective, search-based crash reproduction is a best-effort technique relying on an evolutionary algorithm to explore the space of possible test cases to find the one able to reproduce the crash. This evolutionary algorithm can be configured using many different parameters. Indications on the right parameters to use for a given crash are essential to enhance the adoption of search-based crash reproduction by practitioners. From a researcher perspective, and despite the diversity of projects considered in JCRASHPACK, there is little indication that the set of crashes currently used to evaluate search-based crash reproduction approaches is diverse and representative enough to draw general conclusions [10].

In unit test generation, the factors considered to select the projects and classes under test to run an evaluation are the size (*e.g.,* number of classes, number of lines of code, *etc.*) and the complexity of the code (*e.g.,* number of branches, or the cyclomatic complexity) [11], [12]. Those factors are considered as an approximation of the difficulty for a search process to generate unit tests covering a project or a set of classes. Unlike unit testing, crash reproduction does not seek to achieve the full coverage of a set of classes but generates one or more tests exercising a specific behavior leading to a given crash. In this context, there is no guarantee that the factors used for unit testing are still valid for crash reproduction.

In this exploratory study, we seek to identify such factors by considering various metrics from static code analysis, like complexity, coupling, *etc.*; code smells, as they can hamper testability; and change metrics, as complex code changes have a higher chance of injecting faults in the source code.

Our results show that although JCRASHPACK is larger than benchmarks used in previous studies, additional crashes should be added to improve its diversity and representativeness, and that no individual metric can be used to characterize the difficulty to reproduce a crash.

## II. BACKGROUND

### A. Search-based crash reproduction

Writing a crash reproducing test case is a useful, yet time-consuming and costly for debugging [1]. Hence, many automated techniques have been introduced to ease this process

 This research was partially funded by the EU Horizon 2020 ICT-10-2016-RIA "STAMP" project (No.731529).

Listing 1. XWIKI-13377 crash stack trace [8]

```
0  java.lang.ClassCastException: [...]
1      at [...].BaseStringProperty.setValue([...]:45)
2      at [...].PropertyClass.fromValue([...]:615)
3      at [...].BaseClass.fromMap([...]:413)
4      [...]
```

[3]–[6]. A previous study shows that search-based crash reproduction, which applies search-based test generation techniques to automate crash reproduction, is the most effective approach [4]. It has also been confirmed that crash reproducing test cases generated by this approach aid developers in fixing bugs [4].

Search-based crash reproduction takes as input the application, in which the crash happened, and a stack trace (reported in a crash report) with one of its frames indicated as the *target frame*. Then, it initiates a search process to generate a test case, which reproduced the given stack trace from the deepest frame up to the target frame. For instance, by passing the stack trace in Listing 1 as the given tack trace and frame 2 as the target frame, search-based crash reproduction generates a test case which reproduces the first two frames of the given stack trace with the same type of exception (`ClassCastException`).

*1) Fitness function:* To reproduce a given crash, search-based crash reproduction relies on a fitness function called CRASH DISTANCE (described in Equation 1) to evaluate the generated test cases, thereby guiding an evolutionary algorithm towards generating a crash reproducing test case for a given stack trace.

$$f(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_e) + max(d_t) & \textit{line is not reached} \\ 2 \times d_e(t) + max(d_t) & \textit{line is reached} \\ d_t(t) & \textit{exception is thrown} \end{cases} \quad (1)$$

Where $d_s(t) \in [0, 1]$ measures the distance between the execution of a generated test $t$ from reaching the line of the target frame (*target line*) using the approach level and branch distance [13]; $d_e(t) \in \{0, 1\}$ is a binary value indicating if $t$ throws the same type of exception as the given stack trace ($d_e(t) = 0$) or not ($d_e(t) = 1$); $d_t(t) \in [0, 1]$ compares the similarity of the frames in the given thrown stack trace by test $t$ against the frames in the given stack trace; and $max(.)$ indicates the maximum possible value for each heuristic.

Since considering $d_e(t)$ and $d_t(t)$ is only relevant if test $t$ covers the target line ($d_s(t) = 0$), CRASH DISTANCE (first line of Equation 1) sets the maximum value for these two heuristics before achieving the target line coverage. Therefore, $f(t) \in ]3, 6]$ before reaching the target line. Likewise, as shown by the second line of Equation 1, measuring the stack trace similarity ($d_t(t)$) is not relevant before fulfilling the exception coverage ($d_e(t)$), and thereby CRASH DISTANCE sets the maximum possible value for $d_t(t)$. Hence, $f(t) \in ]1, 3]$ before $t$ throws the same type of exception as the given stack trace. Finally, when $d_s(t)$ and $d_e(t)$ are zero, $f(t) \in [0, 1]$ according to the value of $d_t(t)$. Since the process is a minimization process, the three heuristics are equal to zero for a crash reproducing test case.

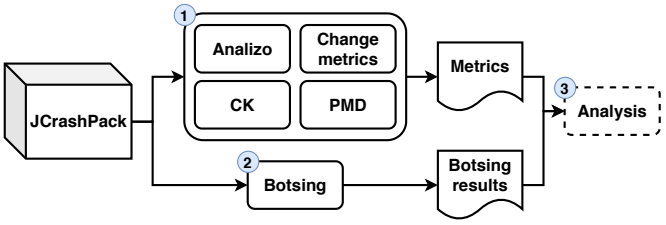


Fig. 1. Study set-up overview

*2) Benchmark-based evaluation:* A previous study [8] proposed JCRASHPACK [9]: a benchmark containing 200 stack traces from multiple industrial open-source Java projects. They performed an extensive evaluation of search-based crash reproduction and reported that this approach could not reproduce more than 50% of the crashes. After a manual analysis, they identified 13 types of challenges, among which code complexity and input data generation are the main obstacles.

Although characterizing the challenges can guide researchers towards designing new strategies to tackle the identified challenges, it does not help practitioners and researchers to indicate how hard it is to reproduce a crash. In this study, we try to answer this question by performing a more in-depth analysis using various static code analysis metrics.

*3) Search-based crash reproduction framework for Java:* BOTSING [7] is a well-tested and open-source search-based crash reproduction framework for Java crashes. This framework contains all of the approaches and techniques, introduced in related studies [14]–[16].

### B. Software measurement and code quality

Software quality assessment is a crucial activity in software development. Besides software inspection, software measurement is one of the main methods used to assess the quality of software. Multiple software metrics have been designed to establish meaningful relationships between measurable properties of software artefacts (*e.g.,* lines of code, cyclomatic complexity, etc.) and high-level software quality characteristics (e.g., testatbility, evolutivity, maintainability, *etc.*) [17]–[21].

Unfortunately, these relationships have yet to be accurately characterised (*i.e.,* with suitable thresholds to interpret quality based on measurement values) [22] and several limitations to metrics have been pointed out [23], [24].

However, software measurement provides a promising approach to easily extract relevant features of a code-base that can be quantitatively investigated and correlated.

## III. STUDY SET-UP

The main goal of our study is to answer the following research question: *How do measurable properties of software artefacts influence the difficulty to reproduce a crash using search-based crash reproduction?* To answer that question, we follow the steps presented in Figure 1. We use the crashes and projects from JCRASHPACK [9], and analyse them (① in Figure 1) using static code analysis (relying on Analizo [25], CK [26], and PDM [27]) and code change metrics. To

reproduce the crashes (② in Figure 1), we rely on the results of the evaluation of BOTSING from Derakhshanfar *et al.* [14]. Based on the collected metrics and the crash reproduction results, we perform various analysis (③ in Figure 1) described hereafter to answer our research question.

To run the different analysis (① in Figure 1) on JCRASHPACK, we used a dedicated server (Ubuntu 16.04, 64-bits, 125 GB memory, processor E5-2650 with 40 cores). For each version of the largest project (XWiki), CK ran on average for 12 hours, and Analizo ran between 16 and 21 hours. The other projects have a negligible execution time. As those tool work on a single thread, we ran them in parallel to save time.

### A. Static code analysis

We measure code metrics from three general categories : *static code metrics*, *code smells*, and *cyclomatic complexity*. Code metrics have been used to assess code quality *w.r.t.* various aspects like, for instance, maintainability [21] and testability [20]. In this study, we hypothesize that, as for search-based unit test generation [11], [12], code metrics provide indications about the size of the search space to explore in order to generate a crash reproducing test. Such metrics would help to specify more precisely the impact of *complex code* and *complex input data*, identified by Derakhshanfar *et al.* [8] as challenging for search-based crash reproduction.

To measure the different class-level and method-level *code metrics*, we used *CK* [26], a static analyser computing object-oriented programming metrics, completed by *Analizo* [25], a multi-language source code analyzer. For *code smells*, we used PMD [27], a static rule checker looking for programming flaws and bad designs, with the default set of rules for Java [28], including the `CyclomaticComplexity` rule to retrieve the *cyclomatic complexity* of the different methods.

Additionally to the different measures provided by the different tools (the complete list is available in [29]), we define and computed for each method the *simple_param_part* and *primitive_part*, denoting (resp.) the percentage of parameters with a primitive or Java-defined type, and the percentage of parameters with a primitive type. Those two measures provide an indication on the complexity of the possible input parameters for a given method.

### B. Code change metrics

Process metrics, like code change, have been used to identify potential defects in a source code [30]. As defects can manifest as crashes during the execution of a program, the presence of source code that has been frequently or recently updated could ease search-based crash reproduction [31]. On the contrary, stable source code might be less prone to crashes. For this study, we built a Python script, relying on the `PyGithub` [32] and `pandas` [33], [34] libraries, to compute the different metrics proposed by Rahman and Devanbu [30]: the number of commits on a file (*COMM*), the number of distinct developers having made a commit on a file (*DDEV*), the number of lines added to a file (*ADD*), the number of lines deleted from a file (*DEL*), the number of lines authored by the highest contributor of a file (*OWN*), the number of distinct developers who authored less than 5% of a file (*MINOR*).

### C. Crash reproduction results

We used the results of the evaluation of BOTSING from Derakhshanfar *et al.* [14] and the corresponding dataset [35], which applies BOTSING on 122 crashes. In this evaluation, each execution is repeated 30 times to address the randomness in the search-based algorithm. We only considered the results for *vanilla* crash reproduction without any seeding strategy and the CRASH DISTANCE fitness function. All other parameters were left to their default value.

From the evaluation results, we extracted for each frame of each crash the final value of the fitness function, denoted *fitness score* ($fs$) hereafter. As explained in Section II, a final fitness of 0 denotes an execution where BOTSING could reproduce the given crash from the given frame. Additionally to the fitness score, we compute for each frame the *reproduction rate* as the ratio between the number of times BOTSING could reproduce the frame ($fs = 0$) and the total number of executions for that frame (30 in the dataset). Following Soltani *et al.*'s recommendation, we consider the frame as *reproduced* if the reproduction rate is (strictly) above 50% [4], and *not reproduced* otherwise.

### D. Data analysis

We start by exploring the data collected in the previous steps (① and ② in Figure 1) using different plotting approaches. Those plots provide an overview of the data distribution across the different applications of JCRASHPACK. More specifically, we plot in boxplots the different class-level and method-level measures collected for the methods and classes pointed in the different frames of the crashes.

Next, we identify correlation between the different metrics and the fitness score($fs$) achieved by the final evaluation of the BOTSING fitness function described in Equation 1. For that, we apply Spearman's rank correlation ($\rho$) and Kendall's rank coefficient (with a significance threshold of 0.05) using `pandas` [33], [34] between each metric and $fs$.

The final fitness score denotes *how far* the search-based crash reproduction process could go. Since the CRASH DISTANCE considers the kind of exception thrown and the similarity of the stack trace only if the target line is reached (*i.e.,* the line where the exception is thrown, indicated in the target frame), we split the analysis for executions where $fs > 3$ (*i.e.,* the target line was not reached before exhaustion of the search budget) and $fs \leq 3$ (*i.e.,* the target line was reached before exhaustion of the search budget). The rational behind this decision lies in the different distances and heuristics used by CRASH DISTANCE: the approach level and branch distance (used by $d_s$ in Equation 1) to reach the target line ($fs > 3$), and the exception distance and stack trace similarity (used by $d_e$ and $d_t$ in Equation 1) to reproduce the exception thrown by the crashing executions. Additionally, we use Cohen's interpretation for effect size [36]: *small* for $0.10 < |\rho| \leq 0.30$, *medium* for $0.30 < |\rho| \leq 0.50$, and *large* for $|\rho| > 0.50$.

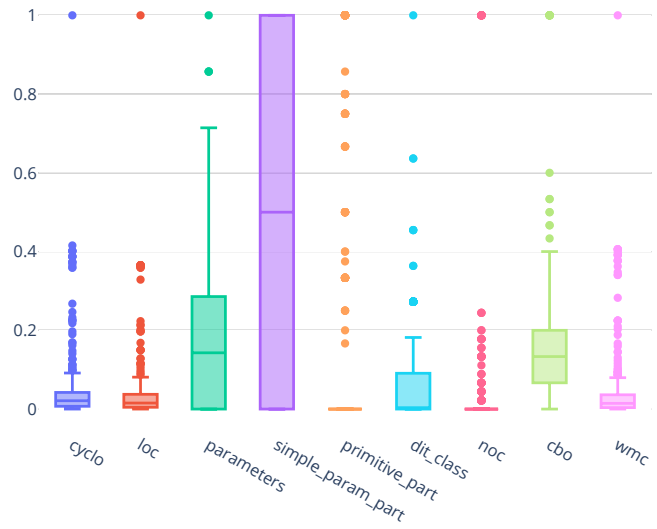


Fig. 2. Excerpt of the distribution of the normalized static code metrics values: cyclomatic complexity (*cyclo*), number of lines of code (*loc*), number of parameters (*parameters*), percentage of parameters with a primitive or Java-defined type (*simple_param_part*), percentage of parameters with a primitive type (*primitive_part*), depth of inheritance tree (*dit_class*), number of children (*noc*), coupling between objects (*cbo*), weighted methods complexity per class (*wmc*).

TABLE I
PEARSON'S CORRELATIONS ($r$) VALUES WITH MEDIUM EFFECT SIZES FOR CLASS *(c)* AND METHOD *(m)* LEVEL METRICS.

| **Name** | *fs* | $\rho$ | **Description** |
|---|---|---|---|
| maxNestedBlocks (c) | $> 3$ | 0.34 | Max. number of nested blocks. |
| variables (m) | $> 3$ | 0.42 | Num. of variable declarations. |
| tryCachQty (m) | $> 3$ | 0.31 | Num. of *try/catch*. |
| wmc (c) | $> 3$ | 0.34 | Class complexity. |
| loc (m) | $> 3$ | 0.44 | Num. of line of codes. |
| cbo (m) | $> 3$ | 0.33 | Coupling between objects. |
| maxNestedBlocks (m) | $> 3$ | 0.39 | Max. number of nested blocks. |
| rfc (m) | $> 3$ | 0.45 | Response for a class. |
| cyclo (m) | $> 3$ | 0.37 | Cyclomatic complexity. |
| mmloc (c) | $> 3$ | 0.45 | Max. method LOC. |
| simple_param_part (m) | $\leq 3$ | -0.42 | Perc. of simple parameters. |
| primitive_part (m) | $\leq 3$ | -0.40 | Perc. of primitive parameters. |
| cyclo_add (m) | $\leq 3$ | 0.54 | Lower frames add. complexity. |

## IV. INITIAL RESULTS

### A. Data visualisation

Figure 2 shows the distribution of the different measures for the frames of the crashes in JCRASHPACK. All the measures are normalized to be represented on the plot. Only the percentage of parameters with a primitive or Java-defined type (*simple_param_part*) has a good distribution. For all the other measures, the median is below 0.2 with outliers up to 1.0. We observe the same distributions of values for the change metrics (omitted due to space constraints), where only the number of distinct developers (*DDEV*) is well distributed. This suggests that additional crashes should be added to JCRASHPACK to improve its diversity and representativeness [10].

### B. Correlation analysis

Table I provides the significant correlation values with a large or medium effect size. At the class level, all the metrics correlates with a medium effect size. At the method level, several metrics correlate with a medium effect size. Notably, the response for class (*rfc*), and the maximal method length (*mmloc*) have the highest correlation.

For executions where the target line was reached ($fs \leq 3$), the percentage of parameters with a primitive or Java-defined type (*simple_param_part*) and the percentage of parameters with a primitive type (*primitive_part*) correlate with a medium effect size. Notably the additive cyclomatic complexity (*cyclo_add*), computed by adding the cyclomatic complexity of the of the methods pointed by the different frames in a stack trace correlates (0.54) with a strong effect size.

The complete list of correlation coefficients between the fitness score and the different metrics is available in [29].

## V. DISCUSSION AND FUTURE WORK

*a) Data collection:* Collecting large amounts of data is not trivial. Additionally to the problems coming from the disparity of format for the analysis reports, some tools have costly operations like parsing all the files in a project or building a dependency graph. Executing those operations on large projects part of JCRASHPACK, like the different versions of XWiki (with around 102.5 MB of source code), poses several challenges. We ran our different analysis on a dedicated (powerful) server, allowing to reduce the execution time. In our future work, we will reduce the analysis cost by focusing on the metrics and corresponding elements of the projects relevant to characterize the difficulty to reproduce a crash.

*b) Code smells:* Our data collection also includes the identification of code smells for the different projects of JCRASHPACK. We did not discussed how the presence of smells can influence the difficulty to reproduce a crash. Our initial assessment indicates that the correlation between the number of rules violated in a target class and the final fitness score is significant but very low, both for execution reaching and not reaching the target line.

*c) Machine learning:* The dataset we collected will be used for more advanced analysis and data visualisation approaches used in the machine learning community. For instance, the application of a decision or regression tree learning algorithm would help understanding the factors (*e.g.,* code smells) impacting the difficulty to reproduce a crash.

*d) Configuration of the search-process:* Finally, for our evaluation, we considered only one configuration of BOTSING (the *vanilla* one from Derakhshanfar *et al.* [14]). However, the search-process can be tuned using many different parameters, which can be hard for newcomers and hamper the industrial deployment of search-based crash reproduction. One of our end-goal is to define a configuration recommender for BOTSING which can, based on a crash and the application source code repository, recommend an optimal configuration to apply to reproduce the crash.

## REFERENCES


[1] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

[2] D. Gagliardi, M. Audren De Kerdrel, L. Andreatta, N. Bertazzo, C. Formisano, D. Gagliardi, J. Gorroñogoitia Cruz, C. Landry, and R. Tejada, "STAMP Deliverable D4.4: Final public version of API and implementation of services and courseware," STAMP project, Tech. Rep., 2019. [Online]. Available: https://www.stamp-project.eu/view/main/deliverables

[3] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "A bug reproduction approach based on directed model checking and crash traces," *Journal of Software: Evolution and Process*, vol. 29, no. 3, p. e1789, mar 2017.

[4] M. Soltani, A. Panichella, and A. Van Deursen, "Search-Based Crash Reproduction and Its Impact on Debugging," *IEEE Transactions on Software Engineering*, 2018.

[5] N. Chen and S. Kim, "STAR: Stack trace based automatic crash reproduction via symbolic execution," *IEEE Trans. on Software Engineering*, vol. 41, no. 2, pp. 198–220, 2015.

[6] J. Xuan, X. Xie, and M. Monperrus, "Crash reproduction via test case mutation: Let existing test cases help," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM, 2015, pp. 910–913.

[7] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. Van Deursen, "Botsing, a Search-based Crash Reproduction Framework for Java," in *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia*. ACM/IEEE, aug 2020.

[8] M. Soltani, P. Derakhshanfar, X. Devroey, and A. van Deursen, "A benchmark-based evaluation of search-based crash reproduction," *Empirical Software Engineering*, vol. 25, no. 1, pp. 96–138, jan 2020.

[9] P. Derakhshanfar and X. Devroey, "Jcrashpack: A java crash reproduction benchmark (version 1.0.1) [data set]," 2020. [Online]. Available: https://zenodo.org/record/3766689

[10] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press, 2013, p. 466.

[11] G. Fraser and A. Arcuri, "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 1–42, dec 2014.

[12] A. Panichella and U. R. Molina, "Java unit testing tool competition - Fifth round," in *Proceedings - 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing, SBST 2017*, 2017, pp. 32–38.

[13] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, jun 2004.

[14] P. Derakhshanfar, X. Devroey, G. Perrouin, A. Zaidman, and A. van Deursen, "Search-based crash reproduction using behavioural model seeding," *Software Testing, Verification and Reliability*, vol. 30, no. 3, p. e1733, 2020.

[15] P. Derakhshanfar, X. Devroey, A. Zaidman, A. van Deursen, and A. Panichella, "Good things come in threes: Improving search-based crash reproduction with helper objectives," in *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia*. ACM/IEEE, aug 2020.

[16] P. Derakhshanfar, X. Devroey, and A. Zaidman, "It is not only about control dependent nodes: Basic block coverage for search-based crash reproduction," in *International Symposium on Search Based Software Engineering*. Springer, 2020, pp. 42–57.

[17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[18] F. B. Abreu and R. Carapuça, "Object-oriented software engineering: Measuring and controlling the development process," in *Proceedings of the 4th international conference on software quality*, vol. 186, 1994.

[19] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the Relation of Test Smells to Software Code Quality," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, sep 2018, pp. 1–12.

[20] M. Bruntink and A. van Deursen, "Predicting class testability using object-oriented metrics," in *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*, 2004, pp. 136–145.

[21] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111 – 122, 1993, object-Oriented Software.

[22] J. Pantiuchina, M. Lanza, and G. Bavota, "Improving Code: The (Mis) Perception of Quality Metrics," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, no. Section V. IEEE, sep 2018, pp. 80–91.

[23] N. E. Fenton and M. Neil, "Software metrics: successes, failures and new directions," *Journal of Systems and Software*, vol. 47, no. 2-3, pp. 149–157, 1999.

[24] K. El-Emam, "Object-oriented metrics: A review of theory and practice," in *Advances in software engineering*. Springer, 2002, pp. 23–50.

[25] A. Terceiro, J. Costa, J. Miranda, P. Meirelles, L. R. Rios, L. Almeida, C. Chavez, and F. Kon, "Analizo: an extensible multi-language source code analysis and visualization toolkit," in *Brazilian conference on software: theory and practice (Tools Session)*, 2010.

[26] M. Aniche, *Java code metrics calculator (CK)*, 2015, available at https://github.com/mauricioaniche/ck.

[27] *Java rulechecker (PMD), version 6.23*, 2020, available at https://pmd.github.io.

[28] *PMD default rule set*, available at https://pmd.github.io/pmd-6.23.0/pmd_rules_java.html.

[29] B. Cherry, "JCrashPack 2.0: Search-based crash reproduction hardness analysis," Master's thesis, Univesrity of Namur, Jun. 2020. [Online]. Available: https://researchportal.unamur.be/en/studentTheses/jcrashpack20

[30] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, may 2013, pp. 432–441.

[31] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: a model to identify how bugs are introduced in software components," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1294–1340, mar 2020.

[32] *PyGithub : a library to use the GitHub API v3*, available at https://pygithub.readthedocs.io.

[33] The pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[34] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61.

[35] P. Derakhshanfar, X. Devroey, G. Perrouin, G. Perrouin, A. Zaidman, and A. van Deursen, "Replication package of "Search-based Crash Reproduction using Behavioral Model Seeding"," Oct. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3673916

[36] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken, *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge, 2013.