# On the Naturalness of Bytecode Instructions

Yoonho Choi
yhchoi@handong.ac.kr
Handong Global University
Pohang, South Korea

Jaechang Nam*
jcnam@handong.edu
Handong Global University
Pohang, South Korea

## ABSTRACT

Bytecode is used in software analysis and other approaches due to its advantages such as high availability and simple specification. Therefore, to leverage these advantages in training language models with bytecode, it is important to clearly recognize the characteristics of the naturalness of bytecode. However, the naturalness of bytecode has not been actively explored.

In this paper, we experimentally show the naturalness of bytecode instructions and investigate their characteristics by empirically assessing 10 Java open-source projects. Consequently, we demonstrate that the bytecode instructions are more natural than source code representations and less natural than abstract syntax tree representations at a method-level. Furthermore, we found that there is no correlation between the naturalness of bytecode instructions and source code representations at a method-level. Our study supports that researchers need to deal with the characteristics of the naturalness of bytecode instructions in a different view from source code. We expect that these findings will be helpful for future work to study automated software engineering tasks such as automated debugging and vulnerability detection that use bytecode models.

## CCS CONCEPTS

• **Software and its engineering → Software notations and tools**; • **Computing methodologies → Natural language processing**; • **General and reference → Surveys and overviews**.

## KEYWORDS

Bytecode naturalness, n-gram model, code representation

**ACM Reference Format:**
Yoonho Choi and Jaechang Nam. 2022. On the Naturalness of Bytecode Instructions. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559559

## 1 INTRODUCTION

Bytecode is commonly used in software analysis and other approaches due to its advantages. For example, bug detection tools such as SpotBugs [6] and Google Error Prone [1] adopt JVM bytecode analysis because bytecode syntax is hardly changed and well-structured rather than high-level languages. Moreover, when the source code is not available or the information to be gathered in the analysis is only visible at the level of bytecode, adopting bytecode is an unavoidable choice [3, 10]. Therefore, to leverage these advantages in training language models with bytecode, it is important to clearly recognize the characteristics of the naturalness of bytecode.

However, the naturalness of bytecode has not been actively explored even if language modeling approaches based on the bytecode had been suggested. Kim et al. [9] proposed a method crash prediction technique by using bytecode instructions. Bryksin et al. [2] proposed a Kotlin compiler anomaly detection approach by using a language model trained with bytecode instructions. Their approaches assumed the naturalness of bytecode instructions. However, to our knowledge, the naturalness of bytecode instructions has not been experimented before. In contrast, the naturalness of source code has been in the spotlight with various source code representations by setting different abstract levels [5, 7, 8].

In this paper, we experimentally show the naturalness of bytecode instructions and investigate their characteristics. To explore the naturalness of bytecode, we first examine the naturalness of bytecode instructions in terms of cross-entropy. In addition, we study the relationship between the naturalness of bytecode instructions, source code, and abstract syntax tree (AST) representations to identify similarities and differences between them. If the naturalness of bytecode instructions is strongly correlated to source code or AST representations, we can take advantage of the findings from the previous studies about their characteristics. However, if not, we have to deal with the characteristics of the naturalness of bytecode instructions in a different view from source code.

Consequently, we demonstrate that the bytecode instructions are more natural than source code representations and less natural than AST representations at a method-level. Furthermore, we found that there is no correlation between the naturalness of bytecode instructions, source code, and AST representations at a method-level. Based on the findings, it is needed to explore the naturalness of bytecode instructions' characteristics, including relations between defects and the naturalness of bytecode and the complementary aspects between the naturalness of bytecode and source code. After understanding the characteristics, we can suggest more efficient and diverse approaches considering the use of bytecode, source code, and AST representations when we train various machine learning and deep learning models for software engineering. We expect that these findings will be helpful for future work to study various automated software engineering tasks such automated debugging and vulnerability detection that use bytecode models [2, 9, 11].

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ASE '22, October 10–14, 2022, Rochester, MI, USA*
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3559559

**Table 1: The ten Projects used in the empirical study**

| Name | Version | # Byt. Mthd. | # Src. Mthd. | Matched |
|---|---|---|---|---|
| arthas | 3.5.5 | 637 | 4,383 | 202 |
| dbeaver | 21.3.5 | 41,901 | 36,668 | 27,403 |
| dubbo | 3.0.67 | 9,793 | 20,790 | 6,804 |
| EventBus | 3.3.1 | 1,027 | 527 | 367 |
| guava | 31.0.1 | 2,722 | 11,661 | 1,406 |
| jadx | 1.3.3 | 8,178 | 8,696 | 5,350 |
| retrofit | 2.9.0 | 309 | 238 | 142 |
| RxJava | 3.1.3 | 25,625 | 32,492 | 19,651 |
| spring-boot | 2.6.4 | 24,619 | 38,645 | 15,317 |
| zxing | 3.4.1 | 1,923 | 2,563 | 847 |

## 2 BACKGROUND

One of the representative statistical language models is an n-gram model. Given a word sequence that consists of $m$ words, $w_1 w_2 ... w_m$, the n-gram model computes the probability of each word as follows: $P(w_{1:m}) = \prod_{i=1}^{m} P(w_i|w_{1:i-1})$ , where $P(w_{1:m})$ is the probability of words. The n-gram model approximates the product of the probability based on a Markov property of order $n-1$, which makes it faster. However, the n-gram models suffer from an issue caused by data sparsity [7]. To mitigate the data sparsity issue, smoothing techniques have been introduced and widely used. The Modified Kneser-Ney smoothing technique shows the best performance to improve the n-gram model in the source code domain [7]. In addition, Karampatsis et al. [8] represented that big code does not imply big vocabulary and byte-pair encoding is more effective than other approaches to deal with the issue. For the source code modeling by using n-gram models, Jimenez et al. [7] found that different tokenizers lead to different levels of naturalness, and a proper choice of $n$-order is 4 or 5 for 8 tokenizers.

## 3 RESEARCH QUESTIONS

To explore the naturalness of bytecode instructions, we address the following research questions.

- *RQ*1: Do bytecode instructions have naturalness?
- *RQ*2: Is the naturalness of bytecode correlated with the naturalness of source code?

How natural are bytecode instructions? Is the naturalness observed at a similar degree to the naturalness of source code? These questions have not been investigated before. We investigate the degree of naturalness of bytecode instructions at a method-level and then compare the naturalness of bytecode with the naturalness of source code. If we can observe the similarity of naturalness between bytecode and source code, we can use bytecode naturalness as the alternative to source code naturalness when we cannot access the source code. If not, we need to further investigate the unique meanings of bytecode of software to properly adopt it.

## 4 METHODOLOGY

### 4.1 Data Collection

For the empirical study, we collected source code, bytecode and AST representations from 10 Java projects as listed in Table 1. To collect the projects, we searched for GitHub projects written in 'Java'. Then, we select the top-10 projects based on the number of 'stars' that Java language takes the major portion of the implementation, and clone them to obtain the source code data. For the bytecode data, we build every project by using open JDK 11 and disassemble the generated class files into assembly to read bytecode instructions by using ASMTools 7.0. To collect ASTs, we used JavaParser 3.24.2.

### 4.2 Data Preprocessing

*4.2.1 Extracting Methods and Static Linking.* To assess the naturalness of bytecode instruction, source code, and AST representations at a method-level, we extract the methods of each disassembled bytecode file and source code file. We adopt a method-level for our investigation because the most fine-grained units of a sequence of instructions could be found in a method. In the case of compiling lambda expressions, the compiler generates dummy methods within the class file for the lambda expressions. For inner classes including anonymous classes, new class files are created for the inner classes. To keep the consistency of method bodies between the source code and bytecode, we statically merge the separated lambda expression into the method that contains the lambda expressions. In addition, we also merge methods in the separated inner class files into the original method that includes the inner classes' methods.

Then, we assign key values for every method based on each method signature, the name of each class, and the name of a parent directory. By using the name of the class and the directory, we distinguish methods that have the same method signature.

*4.2.2 Tokenization.* In language modeling, tokenization is one of the most important steps because the language model computes the probability of sequences composed of tokens. Jimenez et al. [7] investigated the impact of using different tokenizers and abstraction levels for source code and found the probability of sequences is affected by the visit strategies such as depth-first search (DFS) or breadth-first search (BFS) [7].

Based on this finding, we adopt the four different AST and two different source code representations used in the previous study [7]: 1) BFS based ASTs, 2) pruned BFS (PBFS) based ASTs, 3) DFS based ASTs, 4) pruned DFS (PDFS) based ASTs, 5) Java grammar (JG) based source code representation, and 6) UTF non-alphanumeric delimiter based source code representation. Pruned versions of the AST representations remove nodes that serve a structural purpose, e.g., every variable name is preceded by a node of type *NameExpr* [7]. We exclude the other two representations containing *comments* used in the previous study because *comments* are removed after compiling. Bytecode is tokenized as a sequence of instructions following the previous studies [2].

Tokenizers applied to AST representations and JG tokenizer are implemented based on the JavaParser, and we use Terrior 5.6 tokenizer to implement the UTF tokenizer.

### 4.3 N-Gram Model Configurations

We build n-gram models for every project separately by using KenLM [4] as KenLM provides the Modified Knerser-Ney smoothing technique, a well-adopted smoothing technique in the source code corpus. For a model configuration, we select the n-order from 2 to 6 because the cross-entropy value of source code and AST representations converges at the n-order of 4 or 5 [7].

**Table 2: Average numbers of unique tokens in each project (*Uniq.*), and median (*Med.*), mean (*Avg.*), and standard deviation (*Std.*) of cross-entropy values of methods in the dataset computed by bigram, trigram, 4-gram, 5-gram, and 6-gram models**

| | | bigram | | | trigram | | | 4-gram | | | 5-gram | | | 6-gram | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Uniq. | Med. | Avg. | Std. | Med. | Avg. | Std. | Med. | Avg. | Std. | Med. | Avg. | Std. | Med. | Avg. | Std. |
| Inst. | 152.9 | 9.15 | 10.54 | 9.75 | 6.19 | 7.82 | 9.21 | 4.86 | 6.75 | 8.45 | 4.23 | 6.23 | 8.15 | 4.04 | 5.99 | 8.04 |
| BFS | 64.9 | 6.34 | 7.58 | 5.36 | 4.82 | 5.61 | 4.14 | 4.02 | 4.64 | 4.14 | 3.5 | 4.19 | 3.91 | 3.1 | 3.96 | 3.96 |
| PBFS | 62.5 | 5.86 | 7.03 | 5.11 | 4.33 | 5.1 | 4.01 | 3.74 | 4.44 | 3.73 | 3.35 | 4.08 | 3.57 | 3.06 | 3.91 | 3.60 |
| DFS | 64.9 | 3.83 | 4.39 | 2.52 | 3.33 | 3.84 | 2.50 | 2.68 | 3.09 | 1.90 | 2.53 | 2.88 | 1.84 | 2.35 | 2.73 | 1.86 |
| PDFS | 62.5 | 4.45 | 5.06 | 4.22 | 3.33 | 3.83 | 3.06 | 3.07 | 3.51 | 2.38 | 2.82 | 3.23 | 2.20 | 2.67 | 3.11 | 2.25 |
| JG | 17168.5 | 30.36 | 55.15 | 122.44 | 14.9 | 37.73 | 89.12 | 13.08 | 35.64 | 86.88 | 12.01 | 34.3 | 85.87 | 11.75 | 33.89 | 85.61 |
| UTF | 9879.3 | 99.61 | 532.2 | 1825.6 | 60.98 | 444.27 | 1550.0 | 58.03 | 423.88 | 1522.8 | 57.7 | 420.99 | 1520.6 | 57.31 | 420.37 | 1520.4 |

## 4.4 Evaluating Naturalness

*4.4.1 Evaluation Metric.* To evaluate the naturalness of bytecode, source code, and AST representations, we adopt cross-entropy as an evaluation metric. The degree of cross-entropy implies how the model is 'surprised' by a test sequence based on their trained sequences. Lower cross-entropy values mean the test sequence is more natural and predictable. If a given sentence, $s$, composed of $m$ words and the n-gram model is trained with n-order, $n$, the cross-entropy values are computed as follows:

$$CrossEntropy(s) = -\frac{1}{m}\sum_{i=1}^{m} P(w_i|w_{i-n+1:i-1}).$$

*4.4.2 Evaluation Strategies.* For each project, we divide the data into two groups, which are unmatched methods and matched methods. The matched methods are ones that have the same key in bytecode methods and source code methods. Then, we split the matched methods into 10 folds to evaluate the naturalness of the methods with ten iterations. For each iteration of evaluation, 10% of matched methods are used as a test set and the others (unmatched methods and 90% of remaining matched methods) are used to train the model. For instance, if a project has 100 bytecode methods, 200 source code methods, and 50 matched methods, a bytecode model will be trained by 95 instances except for the 5 matched methods for each iteration. At the same time, a source code model will be trained by 195 instances except for the 5 matched methods. This is for acquiring training instances at most and assessing the naturalness within a whole software, not just for the matched methods.

# 5 RESULTS

## 5.1 Dataset

Table 1 shows the statistics of the processed project data. Each column represents the project name, its version, the number of methods in bytecode (# Byt. Mthd), the number of methods in source code (# Src. Mthd.), and the number of matched methods (Matched). We can observe the differences among the number bytecode, source code, and matched methods. One of the reasons is due to the excluded source code in the build process of projects according to the configurations. The characteristics of bytecode discussed in Section 4.2.1 is another reason. In addition to the characteristics, there are methods such as 'init<>' generated by a compiler. These methods cannot be matched with ones in the source code.

## 5.2 RQ1: Degree of the Naturalness

*5.2.1 Protocol.* To address RQ1, we measure the cross-entropy values based on the methodology discussed in Section 4. Then, we compare the median and mean of the cross-entropy values generated by the language models adopting the bytecode tokenizer, the four AST tokenizers, and the two source code tokenizers.

*5.2.2 Result.* Table 2 shows the average number of unique tokens in each project, and median, mean, and standard deviation of cross-entropy values of methods in the dataset computed by bigram, trigram, 4-gram, 5-gram, and 6-gram models in each column. The first column of the table represents the tokenizers discussed in Section 4.2.2 and *Inst.* (third row) is the tokenizer for the bytecode instructions. The overall results indicate that every representation shows the naturalness of software, including bytecode instructions. The median of bytecode instructions decreases from 9.15 to 4.04 and the mean decreases from 10.54 to 5.99 as the n-order increases. In addition, the bytecode instructions are more natural than two source code representations (i.e. *JG* and *UTF*) and less natural than the rest of AST representations (*BFS*, *PBFS*, *DFS*, *PDFS*).

**Finding 1:** Bytecode instructions have naturalness. In addition, the bytecode instructions show less naturalness than AST, but more naturalness than source code.

## 5.3 RQ2: Correlations of the Naturalness

*5.3.1 Protocol.* First, we rank the matched methods of each project from the most natural one to the least natural one by using the cross-entropy values. Second, we compute rank correlations among the cross-entropy values generated by the language models adopting the bytecode tokenizer, the four AST tokenizers, and two source code tokenizers. We used the two representative measures, Spearman's *rho* and Kendall's *tau* as in the previous study [7].

The absolute value of *rho* and *tau*, which are correlation coefficients of the Spearman test and Kendall test respectively, indicates how strongly the pair are correlated to each other ranging from 0 to 1. In this study, if a strong correlation is observed, it implies that two different models evaluate one method to a similar degree in terms of naturalness. After we compute the correlation coefficient, the median value is evaluated over ten projects' correlation coefficients whose p-value is less than 0.05.

*5.3.2 Result.* Figure 1 represents the rank correlation between all tokenizer pairs generated by bigram to 6-gram models. The upper triangle gives the median of the *rho* and the lower triangle gives the median of the *tau*.

First, we can observe there is a negligible correlation between bytecode instructions and all other representations (less than or equal to 0.19) in terms of the Spearman correlation coefficients in every n-order configuration. The similar trend is also observed in

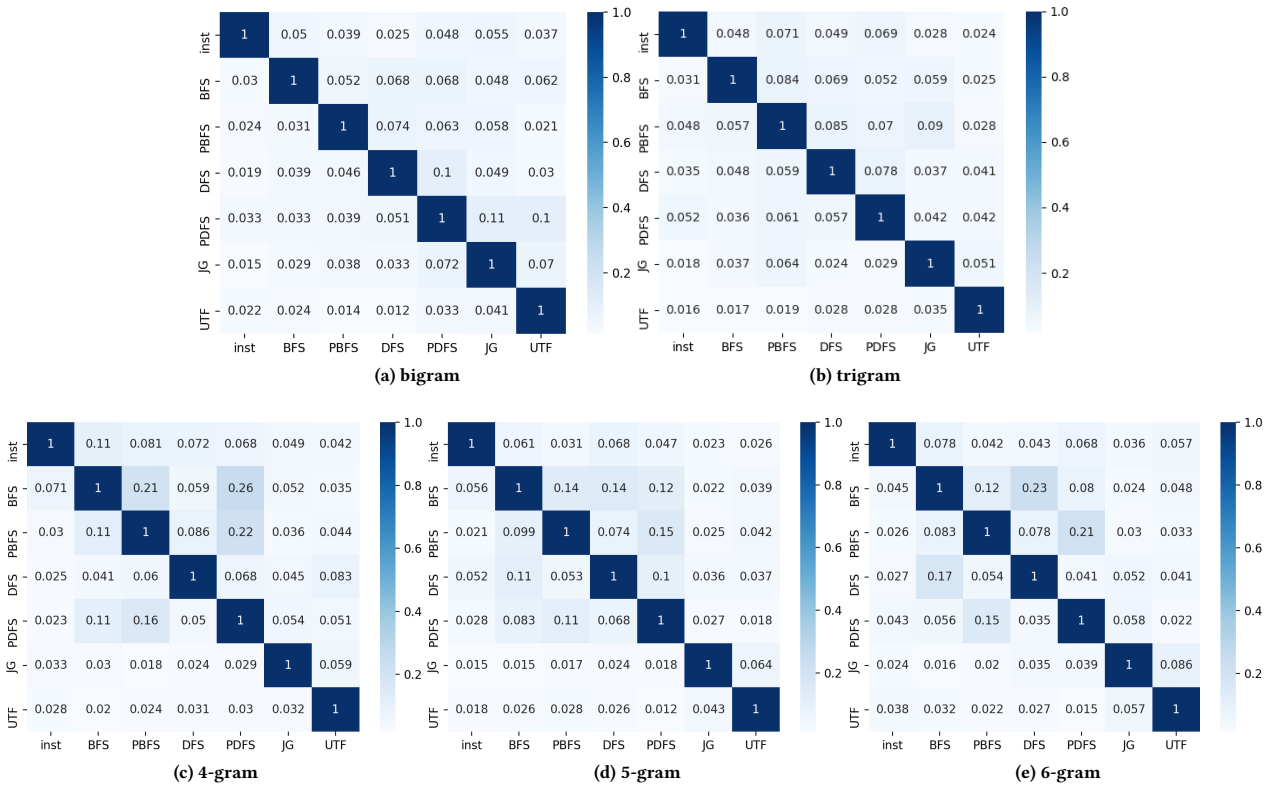


**Figure 1: Rank correlation matrices between every tokenizer pair from bigram to 6-gram models. The upper triangle gives the median value of the Spearman rank correlation coefficients and the lower diagonal represents the median value of the Kendall rank correlation coefficients over all projects.**

the Kendall correlation coefficients. This observation indicates that the rank of the natural methods assessed by bytecode instructions is very different from any other representation.

**Finding 2:** The naturalness of bytecode instructions is not correlated to AST and source code representations in our empirical settings. It implies that the naturalness of bytecode instructions has different characteristics from ones of other representations.

Interestingly, there is no clear correlation between the source code and the AST representations. We can observe the different results from the previous study [7] that found strong correlations between source code representations at a file-level. We can find *BFS*, *PBFS*, and *PDFS* tokenizers as the most correlated in 4-gram (Fig. 1 (c)), 5-gram (Fig. 1 (d)), and 6-gram (Fig. 1 (e)), but still, they have a negligible or weak correlation. In addition, we could find that using different n-order affects the degree of correlations. In our observation, 4-gram (Fig. 1 (c)) show the largest Spearman coefficients (0.26 between *BFS* and *PDFS*), and 6-gram (Fig. 1 (e)) represent the largest Kendall coefficients (0.17 between *DFS* and *BFS*) rather than others, but we cannot find any trend.

**Finding 3:** At a method-level, there is no clear correlation even between the source code representations and AST representations. Considering the previous study [7], this implies that adopting different granularity leads to different results.

## 6 CONCLUSION AND FUTURE WORK

We experimentally show that the bytecode has naturalness. In addition, we observed that the naturalness of bytecode has different characteristics compared to the naturalness of source code and AST representations. The replication package is available here[1]. Based on the findings, we will extend our study as follows:

- Exploring deeper the characteristics of the naturalness of bytecode such as the naturalness of bytecode generated from different high-level languages.
- Proposing defect and vulnerability prediction approaches based on the naturalness of different software representations, including bytecode instructions and other source code and AST representations.
- Designing neural machine translation-based automated program repair at a bytecode-level.

## ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1F1A1063049).

[1]https://github.com/ISEL-HGU/NaturalnessOfBytecode

# REFERENCES


[1] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 14–23. https://doi.org/10.1109/SCAM.2012.28

[2] Timofey Bryksin, Victor Petukhov, Ilya Alexin, Stanislav Prikhodko, Alexey Shpilman, Vladimir Kovalenko, and Nikita Povarov. 2020. *Using Large-Scale Anomaly Detection on Code to Improve Kotlin Compiler.* Association for Computing Machinery, New York, NY, USA, 455–465. https://doi.org/10.1145/3379597.3387447

[3] Albert E., Gordillo P., Livshits B., Rubio A., and Sergey I. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode.. In *Automated Technology for Verification and Analysis. (ATVA 2018, Vol. 11138)*. Springer, Cham. https://doi.org/10.1007/978-3-030-01090-4_30

[4] Kenneth Heafield. 2011. KenLM: Faster and Smaller Language Model Queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, Edinburgh, Scotland, 187–197. https://www.aclweb.org/anthology/W11-2123

[5] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the Naturalness of Software. *Commun. ACM* 59, 5 (apr 2016), 122–131. https://doi.org/10.1145/2902362

[6] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Not.* 39, 12 (Dec. 2004), 92–106. https://doi.org/10.1145/1052883.1052895

[7] Matthieu Jimenez, Cordy Maxime, Yves Le Traon, and Mike Papadakis. 2018. On the Impact of Tokenizer and Parameters on N-Gram Based Code Analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 437–448. https://doi.org/10.1109/ICSME.2018.00053

[8] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1073–1085.

[9] Sunghun Kim, Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, and Shivkumar Shivaji. 2013. Predicting Method Crashes with Bytecode Operations. In *Proceedings of the 6th India Software Engineering Conference* (New Delhi, India) *(ISEC '13)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/2442754.2442756

[10] X. Leroy. 2003. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning* 30 (2003), 235–269. https://doi.org/10.1023/A:1025055424017

[11] Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2015. Analysis of Android Inter-App Security Vulnerabilities Using COVERT. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 725–728. https://doi.org/10.1109/ICSE.2015.233