Efficient Test Generation for Dynamic Behaviors Leveraging Token-Level Input Commonalities

Yuxin Qiu
University of California at Riverside
Riverside, USA
yuxin.qiu@email.ucr.edu

Qian Zhang
University of California at Riverside
Riverside, USA
qzhang@cs.ucr.edu

Abstract

Software in the wild exhibits various computational efforts, such as dynamic behaviors arising from runtime invocations and nondeterministic behaviors introduced by probabilistic algorithms. These efforts complicate software testing because they require meaningful inputs for invocation, introduce additional branching decisions at runtime, and produce divergent outcomes for identical inputs.

To address these challenges, we propose an effective testing framework that leverages commonality analysis across inputs, execution traces, and outcomes. This paper presents our preliminary experiment, which only leverages token-level commonalities to synthesize strings as inputs for testing dynamic invocations in Java applications. Specifically, we analyze naming patterns of loadable targets for runtime invocations and generate inputs following such patterns. Because existing string mutation operators cannot easily synthesize meaningful inputs for runtime invocations, we construct a token tree to prioritize tokens that are likely to be used for invocation based on common vocabularies such as prefixes or suffixes. To address the challenge that branching decisions and execution paths are not deterministic, we enhance traditional branch coverage guided fuzzing by monitoring dynamic behaviors based on class hierarchies and call graphs.

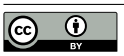
In the future, we plan to extend our test generation framework for dynamic behaviors to include nondeterministic behaviors. Nondeterminism often stems from probabilities and randomization, causing programs to branch into multiple outcomes despite identical inputs. We want to systematically explore as many of these outcomes as possible. Leveraging the insight from our preliminary research that there exist common patterns such as shared prefixes, we plan to define constraints to enable early termination when identical execution trace prefixes are detected, allowing us to improve testing efficiency while covering diverse execution paths.

CCS Concepts

• **Software and its engineering** → **Search-based software engineering**; **Software testing and debugging**;

Keywords

Software testing, token synthesis, commonality analysis



FSE Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728509

ACM Reference Format:
Yuxin Qiu and Qian Zhang. 2025. Efficient Test Generation for Dynamic Behaviors Leveraging Token-Level Input Commonalities. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728509

1 Introduction

The prevalence of artificial intelligence has introduced increasing computational efforts in emerging software systems, including dynamic behaviors due to runtime invocations [20, 26, 31], nondeterministic behaviors resulting from randomization [11], and nonterminism [18]. For example, users must pass in a meaningful string argument to specify which tokenizer (*e.g.,* `bert-base-uncased`) to use for classification in the Deep Java Library (DJL) [1]. However, the fundamental question of *"How can we efficiently generate and execute tests to effectively exercise dynamic, nondeterministic behaviors in an application?"* has been largely overlooked in the literature.

Test Generation Challenges. Fuzz testing has emerged as a highly effective input generation technique for large software systems [3, 23, 24]. Typically, fuzzers iteratively generate new test inputs by mutating seed inputs and save them if they exercise new branches. However, naïvely applying existing fuzzers to dynamic, nondeterministic behaviors is *inefficient* for three reasons.

- *Invalid Strings.* Traditional fuzzers struggle to synthesize valid string inputs for dynamically loadable targets such as classes and methods. For example, running a fuzzer JQF [24] on DJL programs for 24 hours failed to generate any valid method names, leading to repeated `NoSuchMethodExceptions` without any deeper execution path exploration.
- *Ineffective Branch Coverage.* Fuzzing techniques often build on an inherent assumption that branch coverage is a meaningful signal for guiding input generation. However, this signal may fail to differentiate dynamically loaded code. For example, the branch coverage for accessing `name` and `age` fields in `UserDetails` class via reflection [2] is identical, despite accessing different attributes.
- *Dynamic Branches.* Fuzzers often assume a static set of executable branches. However, testing dynamic behaviors requires expanding this scope to include branches that are introduced and executed after dynamic loading. Nondeterminism introduces additional complexity, as the same input may traverse different execution paths across test runs.

Commonality Analysis Based Testing. Our overall insight is that we can address these challenges for testing dynamic, nondeterministic behaviors by analyzing *commonalities* in test inputs, execution traces, and program outcomes. As shown in Figure 1, we *observe*

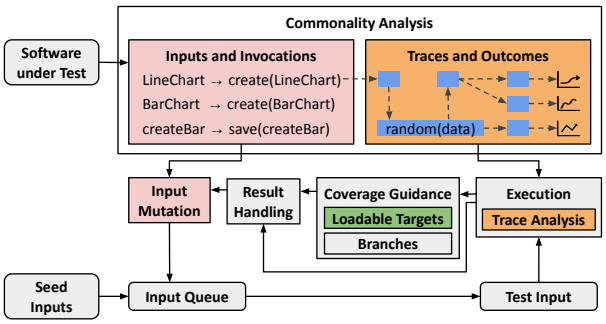


Figure 1: Overview of Testing with Commonality Analysis.

that (1) runtime invocations use strings as arguments to construct loadable target names and (2) tokens sharing common vocabularies (*e.g.,* the suffix `Chart` in `LineChart` and `BarChart`) are likely to represent different loadable targets at the same site of runtime invocations (*e.g.,* the method `create`). We thus *hypothesize* that we can prioritize tokens with similar vocabularies to improve the reachability of runtime invocations. Furthermore, while probabilistic and randomized algorithms may cause identical inputs to yield different outcomes (*e.g.,* generating varied line charts in Figure 1), we *observe* that similar preceding execution traces often lead to consistent results. We thus *hypothesize* that we can leverage trace analysis for early termination to efficiently explore diverse execution paths.

SynToken. In this paper, we present our initial exploration on token-level input commonalities to generate tests for dynamic invocations in Java applications [7, 19, 31, 34]. Specifically, we propose SynToken, a two-fold test generation tool as elaborated below.

1. Commonality Based Token Substitution and Prioritization. Based on the insight that loadable targets (*e.g.,* classes, methods, fields, and libraries) for runtime invocations in Java often adhere to naming conventions, such as camel cases [29], where strings are repeated with common prefixes or suffixes, SynToken increases the likelihood of generating strings of dynamically loadable targets. SynToken first generates a token dictionary from source code to include all possible loadable targets present in the software and its included libraries. Naïvely enumerating all tokens does not consider the fact that not all tokens contribute equally to the reachability of runtime invocations and would result in a large number of futile attempts. To expedite this process, SynToken constructs *a token tree, a tree-based data structure representing prefixes and suffixes*, to prioritize tokens with the same structure based on the naming patterns.

2. Loadable Target Monitoring. Unlike conventional fuzzers that primarily rely on branch coverage [3] or performance metrics [21, 33] for guidance, SynToken's loadable target monitoring adds three additional categories: (1) all loadable targets within the software, (2) all sibling targets related to previously executed tests, and (3) all sub-targets derived from previously executed tests. Any inputs leading to either new branch coverage or successful dynamic loading are then retained for subsequent mutations.

Preliminary Findings. We conducted an evaluation with 71 subject programs from XCorpus [13], a widely used corpus of real-world Java projects. We compared SynToken with two alternatives: (1) SynToken without tree-based input prioritization and (2) traditional input generation with the AFL engine [3, 24]. Overall,

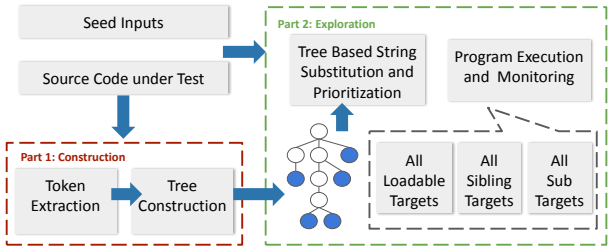


Figure 2: Synthesizing Tokens for Testing Dynamic Behavior.

SynToken was able to cover all targets that can be dynamically loaded and used and reveal all 280 manually inserted bugs residing in the loadable targets. Per the open science policy, we provide the artifacts to SynToken at https://doi.org/10.5281/zenodo.15278627. We summarize our key findings as follows.

- String substitutions in SynToken enable to generate meaningful inputs to reveal bugs, such as `NullPointerException` and `IllegalArgumentException`, that reside in loadable targets.
- Analyzing input commonalities is highly effective for exercising dynamically loaded code. Input prioritization based on analyzing common prefixes and suffixes provides up to 10× speedup compared to random string substitution in the token dictionary.

Future Plan. Moving forward, we plan to investigate how to extend this approach to other dynamic and nondeterministic behaviors. For example, we will identify commonalities across traces and outcomes to enhance the coverage of diverse execution behaviors.

2 Preliminary Research on Analyzing Input Commonalities

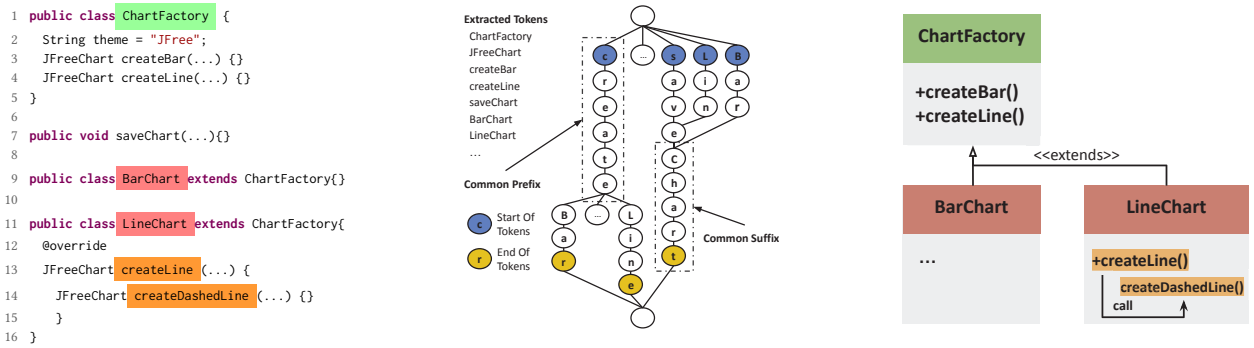
Figure 2 shows an overview of synthesizing tokens based on input commonalities for an application with dynamic invocations: (1) token tree construction and token substitution (Section 2.1) and (2) loadable target exploration and monitoring (Section 2.2).

2.1 Token Substitution and Prioritization

Exploring code after runtime invocations requires an understanding of possible loadable targets (*i.e.,* valid string arguments). Loadable targets in Java often adhere to naming conventions, such as camel cases [4, 14, 29]. Class names like `BarChart` and `LineChart` share the suffix `Chart`, while method names like `createBar` and `createLine` repeat the prefix `create`. Prior research has leveraged such naming patterns [17] for API recommendation tasks [16] or code completion tasks [30]. By leveraging this insight, we can effectively navigate the token space, prioritize replacement tokens, and enhance the generation of valid loadable targets.

Step 1. Token Extraction. Using JavaParser [15] to facilitate token analysis, we traverse Abstract Syntax Trees (ASTs) of the program under test (*e.g.,* Figure 3a) and target specific nodes such as class definitions and method definitions, because they primarily represent possible loadable targets. We generate multiple token dictionaries for each category (*e.g.,* classes, methods) and then combine them.

Step 2. Tree Construction. Prior work that replaces an input with a randomly selected token ignores the fact that loadable target



(a) Source Code of the Program under Test. (b) Extracted Tokens and Tree Construction. (c) Loadable Targets.

Figure 3: An Example of Token Tree Construction and Loadable Targets for JFreeChart [5] in XCorpus [13].

names usually have recurring prefixes or suffixes. We thus conclude that not all tokens contribute equally to the exploration of dynamic behaviors. In fact, based on our experiments, among 3,511 tokens extracted from the real-world project JFreeChart [5], only 607 tokens are relevant to dynamic class loading, which indicates that the random replacement from dictionaries would result in 83% futile attempts. Therefore, we construct a token tree to relate tokens that share common vocabularies. For instance, in Figure 3b, `createBar` and `createLine` share identical nodes `c`, `r`, `e`, `a`, `t`, and `e` as their common prefix, and `BarChart` and `LineChart` share the same nodes `C`, `h`, `a`, `r`, and `t` as their common suffix.

Step 3. Prioritized Input Selection. Based on the class and method relations presented in the token tree, we traverse the token space and prioritize tokens with the longest common prefix or suffix as the next generated input (*i.e.,* another possible loadable target). We use a tunable parameter to decide whether to search in the prefix tree or suffix tree. Traversals in the prefix tree will proceed top-down from the top root node, while traversals in the suffix tree will proceed bottom-up from the bottom leaf node.

For example, in Figure 3b, when the current input is `createBar` and SynToken searches in the prefix tree, SynToken traverses the token tree along the common parent nodes and identifies that `createLine` is the token having the longest common prefix with `createBar`. Then `createLine` will be selected as the next test input. Importantly, the subtrees including tokens like `saveChart` are deprioritized, enabling an expedited search process.

Mutation. We do not sacrifice the capability of generating arbitrary inputs and retain random bit and byte mutations to include the capability of classical fuzz testing techniques.

2.2 Runtime Monitoring

Relying solely on branch coverage signal is insufficient for testing dynamic behaviors. To tackle this challenge, we incorporate runtime monitoring for loadable classes and methods accessible within the software. Based on the insight that different loadable targets may implement the same interface or method, we analyze inheritance relations for classes and methods (*e.g.,* Figure 3c). For classes, we extract the *class hierarchy* by monitoring the AST attribute `childrenNodes` of `ClassOrInterfaceDeclaration` nodes. For methods, we determine the *call graph* by investigating the AST attribute `childrenNodes` of `MethodDeclaration` nodes, because the node of a callee method is present in `childrenNodes` of the node of its caller method. For each loaded target, we monitor if its sibling targets and sub-targets are also loaded.

3 Preliminary Evaluation

We seek to answer the following research questions.

RQ1 How effective and efficient is SynToken in generating inputs to explore dynamic behaviors?

RQ2 How effective is SynToken in revealing bugs residing in the loadable targets?

Subjects. We evaluated SynToken on XCorpus [13], a well-known benchmark set of real-world Java projects [27, 32]. 71 out of its 76 projects demonstrate dynamic behaviors and thus are used as our evaluation subjects.

Baselines. We compared SynToken with two baselines.

- JQF-AFL-fuzz. It is traditional coverage-guided fuzzing with the AFL engine, which is built on top of JQF for Java programs. It uses branch coverage as guidance and random byte replacements as input generation. The generated inputs are arbitrary strings.
- SynToken-Dictionary. This option disables token tree guided input prioritization from SynToken. It replaces the current input with a random string selected from the token dictionary.

3.1 Exercising Loadable Targets

To evaluate the efficacy and efficiency of SynToken in exploring dynamic behaviors, we ran it for three hours and measured the percentage of exercised targets out of all loadable targets. We also compared SynToken against JQF-AFL-fuzz to assess the benefit of using existing tokens and against SynToken-Dictionary to assess the speedup from tree-based input prioritization.

Due to space limit, we will discuss five sample subjects in detail, which reflect the minimum, maximum, median, lower quartile, and upper quartile lines of code (LOC) among the 71 subjects we study. Their accumulative loadable target coverage is shown in Figure 4.

SynToken exercised all loadable targets within two minutes for every subject. In contrast, JQF-AFL-fuzz *failed* to generate any meaningful inputs, resulting in repeated `ClassNotFoundExceptions` and

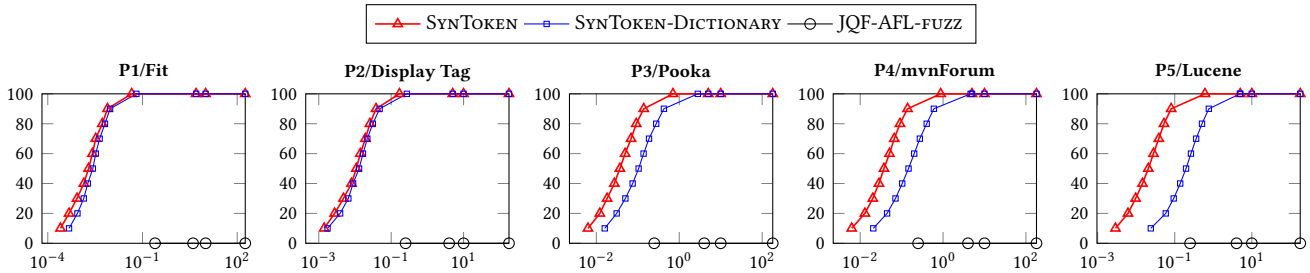


Figure 4: Accumulated Loadable Target Coverage of Sample Subject Programs. The y-axis indicates the percentage of coverage, and the x-axis represents wall-clock time in minutes on a logarithmic scale.

`NoSuchMethodExceptions`, as shown in Figure 4 where JQF-AFL-fuzz's results overlap with the x-axis.

Compared to SynToken-Dictionary, SynToken achieved an average speedup of 4.18× across subjects. The speedup increases as the subject size grows. For example, on P1 with 3,456 LOC and 374 loadable targets, SynToken is 1.33× faster, while on P5 with 412,995 LOC and 3,206 loadable targets, the speedup reaches 9.72×.

3.2 Bug Detection

We used mutation analysis to assess SynToken's bug detection capability in terms of the ability to discover injected bugs residing in dynamically loaded classes and methods. We systematically created faulty mutants by leveraging three mutation operators, *i.e.,* Assignment Statement Insertion (M1), Function Argument Replacement (M2), and Typecast Operation Insertion (M3), to inject commonly seen exceptions from the prior study [12] into original subjects. In total, we injected 280 bugs into our 71 subjects. We measured the number of bugs detected by SynToken in three hours. Remarkably, SynToken uncovered all the injected bugs by generating inputs corresponding to loadable target names and invoking targets to reveal the bugs within the loaded targets. On the contrary, JQF-AFL-fuzz failed to detect any inserted bugs, because it wasted its fuzzing time on generating invalid string names.

4 Related Work

Analyzing Dynamic Behaviors. Efforts have been made to extend unsound static analyses to accommodate dynamic behaviors [9, 10], where dynamic profiling is often employed as a complementary approach of unsound static analyses. However, how to efficiently generate tests for effectively exploring dynamic behaviors is overlooked, which motivates our research.

Fuzz Testing. Instead of relying on branch coverage [3, 25], several techniques adopt custom guidance mechanisms [21, 22, 33, 35]. However, none of them provides guidance for dynamic behaviors. Prior work also explored token characteristics in dictionary-based fuzzing. Salls et al. [28] construct a dictionary of all tokens from the source code for testing JavaScript interpreters. However, they do not consider input commonalities, such as prefixes and suffixes, to expedite input exploration. AFL [3] and libFuzzer [6] support user-defined dictionaries that contain language keywords, multi-byte magic values, *etc.* However, their mutations are not aligned with tokens, and they still mainly rely on byte-level mutations.

Coding Conventions. Developers commonly follow coding conventions such as camel casing when naming identifiers, and these widely used vocabularies have been leveraged in various program analysis tasks [8, 17, 30]. However, none of these techniques are designed or used for test generation.

5 Future Plans

Our eventual goal is to systematically study various intricate dynamic and nondeterministic behaviors in emerging software and to build comprehensive reachability analysis tools for them. SynToken's underlying idea of leveraging commonalities to enhance testing efficacy and efficiency for runtime invocation can be extended to other dynamic and nondeterministic features.

For example, nondeterminism, which often arises from probabilistic implementation and randomized processes, can cause programs to diverge into multiple outcomes despite identical inputs. We *observe* that there are commonalities across nondeterministic executions, where executions sharing common prefixes have a high probability of producing the same outcomes. We thus *hypothesize* that we can improve testing efficiency for nondeterminism by terminating executions early when common execution prefixes are detected, thereby shifting to other unexplored execution paths.

Specifically, we will (1) instrument the software to record execution traces; (2) define constraints that establish early termination conditions by comparing the current execution trace against previously recorded traces; and (3) enable early termination when these constraints indicate that the current execution path shares a common prefix with any previously explored path.

6 Conclusion

In this paper, we open a research problem of generating tests for dynamic and nondeterministic behaviors. The preliminary research analyzes input commonalities at the token level. This idea of commonality analysis can be extended to other dynamic and nondeterministic behaviors by incorporating a wider range of characteristics, including test inputs, execution traces, and program outcomes.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 2426161, Cisco gift funding, and UCR Senate Awards. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] 2025. https://github.com/deepjavalibrary/djl/blob/master/extensions/tokenizers/src/test/java/ai/djl/huggingface/tokenizers/TokenClassificationTranslatorTest.java#L95.

[2] 2025. https://github.com/jmix-framework/jmix/blob/3070342f8fbcf82e47a9bb148fd1fdde3b4a96b0/jmix-multitenancy/multitenancy/src/main/java/io/jmix/multitenancy/core/impl/TenantProviderImpl.java#L63.

[3] 2025. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[4] 2025. Code Conventions for the Java Programming Language. https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html.

[5] 2025. JFreeChart. https://www.jfree.org/jfreechart/.

[6] 2025. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[7] Anwar Alqaimi, Patanamon Thongtanunam, and Christoph Treude. 2019. Automatically Generating Documentation for Lambda Expressions in Java. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 310–320. https://doi.org/10.1109/MSR.2019.00057

[8] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. 2002. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering* 28, 10 (2002), 970–983. https://doi.org/10.1109/TSE.2002.1041053

[9] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 241–250. https://doi.org/10.1145/1985793.1985827

[10] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-Depth Investigation into Debloating Modern Java Applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/3368089.3409738

[11] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2016. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 327–342. https://doi.org/10.1145/2837614.2837639

[12] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. 2015. Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 134–145. https://doi.org/10.1109/MSR.2015.20

[13] JB Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017. XCorpus – An executable Corpus of Java Programs. *Journal of Object Technology* 16, 4 (Aug. 2017), 1:1–24. https://doi.org/10.5381/jot.2017.16.4.a1

[14] Eric Enslen, Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2009. Mining source code to automatically split identifiers for software analysis. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. 71–80. https://doi.org/10.1109/MSR.2009.5069482

[15] Roya Hosseini and Peter Brusilovsky. 2013. Javaparser: A fine-grain concept indexing tool for java problems. In *CEUR Workshop Proceedings*, Vol. 1009. University of Pittsburgh, 60–63.

[16] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API Method Recommendation without Worrying about the Task-API Knowledge Gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 293–304. https://doi.org/10.1145/3238147.3238191

[17] R. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1073–1085. https://doi.ieeecomputersociety.org/

[18] Hrishikesh Karmarkar, Raveendra Kumar Medicherla, Ravindra Metta, and Prasanth Yeduru. 2022. FuzzNT : Checking for Program Non-termination. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 409–413. https://doi.org/10.1109/ICSME55016.2022.00049

[19] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 507–518. https://doi.org/10.1109/ICSE.2017.53

[20] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 507–518. https://doi.org/10.1109/ICSE.2017.53

[21] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 254–265.

[22] Shaohua Li and Zhendong Su. 2023. Accelerating Fuzzing through Prefix-Guided Execution. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 75 (apr 2023), 27 pages. https://doi.org/10.1145/3586027

[23] Valentin Manes, HyungSeok Han, Choongwoo Han, sang cha, Manuel Egele, Edward Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* PP (10 2019), 1–1. https://doi.org/10.1109/TSE.2019.2946563

[24] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

[25] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[26] Felipe Pontes, Rohit Gheyi, Sabrina Souto, Alessandro Garcia, and Márcio Ribeiro. 2019. Java Reflection API: Revealing the Dark Side of the Mirror. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 636–646. https://doi.org/10.1145/3338906.3338946

[27] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 251–261. https://doi.org/10.1145/3293882.3330555

[28] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. Token-Level Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2795–2809. https://www.usenix.org/conference/usenixsecurity21/presentation/salls

[29] Bonita Sharif and Jonathan I. Maletic. 2010. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *2010 IEEE 18th International Conference on Program Comprehension*. 196–205. https://doi.org/10.1109/ICPC.2010.41

[30] Jieke Shi, Zhou Yang, Junda He, Bowen Xu, and David Lo. 2022. Can Identifier Splitting Improve Open-Vocabulary Language Model of Code?. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1134–1138. https://doi.org/10.1109/SANER53432.2022.00130

[31] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features-A Benchmark and Tool Evaluation. In *Asian Symposium on Programming Languages and Systems*. Springer, 69–88. https://doi.org/10.1007/978-3-030-02768-1_4

[32] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the Recall of Static Call Graph Construction in Practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1049–1060. https://doi.org/10.1145/3377811.3380441

[33] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MEMLOCK: Memory Usage Guided Fuzzing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 765–777. https://doi.org/10.1145/3377811.3380396

[34] Erik Wognsen, Henrik Karlsen, Mads Olesen, and René Hansen. 2014. Formalisation and analysis of Dalvik bytecode. *Science of Computer Programming* 92 (10 2014), 25–55. https://doi.org/10.1016/j.scico.2013.11.037

[35] Qian Zhang, Jiyuan Wang, and Miryung Kim. 2021. HeteroFuzz: Fuzz Testing to Detect Platform Dependent Divergence for Heterogeneous Applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 242–254. https://doi.org/10.1145/3468264.3468610