Software Testing is the Ideal Software Engineering Application for Present-Generation Large Language Models

ABSTRACT

Without denying the utility of current-generation Large Language Models (LLMs) for assisting developers in writing code, we propose that at present the most promising application of LLMs to software engineering can be found in the problem of software testing. The consequences of generating faulty or insecure code are limited or nearly eliminated when the code in question is test code, rather than production code. Given the frequency with which current LLMs generate incorrect code or answers to questions about code, this reduction of the costs of failure (and thus the costs of verification of LLM output) makes using LLMs in testing more attractive than in actual development. Furthermore, many software testing tasks that are not code-generation tasks benefit from "good-enough" answers, without requiring extreme accuracy (or precision) and allow an LLM to replace human effort that is often so cost-ineffective that in practice it is not attempted. This may make powerful testing techniques, such as mutation testing, that are currently limited in efficacy by costs, more practical; in a positive feedback loop, better test evaluation tools further reduce the costs of "bad" LLM-generated test code, by making it easier to detect.

CCS CONCEPTS

• **Software and its engineering** → **Dynamic analysis**; **Software testing and debugging**.

KEYWORDS

large language models, code generation, automated testing, test evaluation

ACM Reference Format:

. 2024. Software Testing is the Ideal Software Engineering Application for Present-Generation Large Language Models. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Much of the excitement (and fear) associated with the use of Large Language Models (LLMs) in software engineering has been associated with the idea of LLMs as assistants to, or replacements for, developers writing production code. This paper argues that, given the present state of LLMs, they are better suited to tasks related to *testing* code than to the core software engineering task of producing the code that is to be used in the real world. Current generation LLMs are powerful, but are also prone to (unpredictable) error and in need of close human supervision. Concerns about the correctness of LLM-generated code are greatly reduced in the context of using LLMs to generate tests; testing tasks imply that production code already exists. The existence of this code gives LLMs a rich context that likely allows them to perform better, and (even more importantly) provides a simple check on the "reasonableness" of the generated tests. If LLM-produced tests fail for existing, correct code, or do not improve objective, automated metrics such as code coverage or mutation score [19], they are, without costly human supervision or effort, easily seen to be inadequate. Uses of LLMs in testing tasks go beyond the simple idea of generating unit tests, but further applications may be best understood in the context of the core differences between generating production code and generating test code, which we discuss first.

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.

Conference'17, July 2017, Washington, DC, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

2 LLM-BASED CODE GENERATION: PRODUCTION CODE VS. TEST CODE

The rationale behind our argument that software testing is the most promising application of current-day LLMs, given their capabiities, can be most easily understood by examining the differences between using LLMs to generate *production* code (application code that is executed when the code is deployed: from the standpoint of testing, the "code under test" (CUT)) and using LLMs to generate *test code*, code not executed by users or in deployment, but used to find flaws in production code. We proceed by examining the possible outcomes of both tasks.

2.1 Case 1: Success

If an LLM generates correct, efficient production or test code, there is in one sense no difference: the result is as desired. However, there is still an advantage in the case of test code: it may not be neccessary to have a human inspect it and determine its validity. Production code, in most real-world contexts, must be inspected by another human being before used (even code generated by professional human programmers is usually subjected to substantial inspection in serious development efforts). LLM-generated test cases *can* be reviewed, of course, just as some human test code is subjected to inspection. However, test code can also be used without inspection, as if it were the output of a fuzzer or automated testing tool, which is only examined if the test produced fails.

2.2 Case 2: Obvious Failure

An LLM may also generate code that is obviously bad, and either fails to compile (e.g., type system violations) or fails under almost all runtime conditions. Such code is, often, correctable to code of case 1 or case 3, with little effort. In the case of test code, runtime failures take the form of failing tests that should not fail, e.g. where the LLM has produced an over-strict or simply incorrect test oracle. In these cases, the code can often be corrected by properly weakening the oracle or by modifying the expected output to match the

	How Hard is the Specification to Satisfy?	How Hard is the Specification to Check?
Production Code	Hard (Program Correctness)	Hard (Full Program Verification) (requires proof or at least very high quality test suite)
Test Code	Easy (Utility)	Easy (Improved Test Metrics) (e.g., coverage or mutants killed increase)

Table 1: Fundamental difference between production and test code

real (correct) output of code. Even obvious (in terms of visibility) failures in production code may sometimes require substantially more debugging effort, to ensure fixing the simple bug does not introduce a more subtle problem, but it is reasonable to consider this case one where neither kind of code generation has an obvious advantage. This leaves us with the third, and most dangerous, outcome of code generation.

2.3 Case 3: Subtle Failure

The most troublesome case for LLM-generated code, as with human-generated code, is that of subtle failure, faults which only exhibit under rare input conditions. For production code the consequences can be arbitrary; the faulty code may not be detected until it is deployed, with the usual potentially disastrous consequences and costs. The large research field of software testing essentially exists in order to limit such occurrences.

A particularly important subtle failure mode of LLM-generated production code is code that is "correct" but *insecure*. Previous studies have shown that, like code taken from sources such as stackoverflow [9], LLM-code may be prone to taking the "easy path" of functionality without security [25, 28, 29]. Given the difficulty of testing for insecurity and the cost or impossibility of "building security in after the fact," this can be a particularly disastrous failure mode, and one that is almost always subtle. There is, essentially, no equivalent for this failure mode in test code, which may fail to check for security, but in general cannot itself introduce a vulnerability.

In fact, the consequences of rare failure in test code are generally bounded. A test is fundamentally either too weak or too strong. That is, it either does not fail when it should, or fails when it should not fail. The latter case, if subtle and only exposed after some change to the code under test, is subtle in the sense that it is not immediately detected, but can only be subtle in the sense that it is hard to understand as a bug in the test when it *exposes a complexity in the specification of the software.* In our view, test code that exposes such a complexity, especially a subtle one, is not a "failure" at all, but is instead a *contribution to a software development effort.* No such positive statement can be made about a subtle bug in a program!

The remaining case is test code that subtly fails to catch some erroneous aspects of behavior of the program under test. However, this is, essentially, the typical case for all individual tests, manually or automatically constructed. Production code is generally *uniquely* responsible for certain behaviors of a program; test code is generally only partially responsible for detecting flaws in behavior, in well-designed test efforts. It is seldom the case that anyone assumes a particular test is meant to catch all possible faults in a program, or even in the code it executes. Therefore, test code that *detects at least some bad behaviors* is always useful, and this minimal criteria can be easily established by using simple, fully automated techniques such as mutation testing.

It is well known that LLMs often vary in their performance of a task, sometimes more and sometimes less successful; in testing, it may often be practical to simply use a large number of LLM generated tests for the same code, assuming that redundnacy will be low cost (after all, in fuzzing thousands of ineffective tests are executed for each "useful" test). An equivalent approach for production code seems unlikely to usually be possible.

One way to consider the fundamental difference is shown in Table 1. Generating production code requires satisfying a complex, hard-to-construct (even for humans), usually implicit but also problem-specific, specification: that the code has what is essentially a conceptually single desired behavior. Determining that a proposed solution fails to satisfy this difficult-to-satisfy specification is also extremely hard: it is the general program verification problem, if guarantees are desired, and the general program testing problem, if the problem is rephrased as finding a counterexample showing the LLM output is incorrect.

In contrast, generating a useful test means only generating code that increases test utility: that is, code that can catch *some* bugs. No one expects that a single test (or even test suite) will detect all possible bugs. Further, determining if a proposed incremental addition to tests is useful is often easy: if the test increases either code coverage or killed mutants, then the test is clearly of value. Fuzzers frequently automatically determine "interestingness" of thousands of potential tests per second.

Asking an LLM to solve an easier problem where failure or success is easy to evaluate is obviously a more likely-to-succeed strategy in cases where the LLM is not infallible.

3 LLM-SPECIFIC VIRTUES OF TEST CODE

In addition to the fundamental differences in the tasks discussed above, some unusual features of test code make it particularly suitable for LLM generation.

Researchers have defined a wide range of metrics, including BLEU [24], ROUGE-L [18], and even more code-specific metrics that perform AST matching and dataflow analysis such as CodeBLEU [27], CrystalBLEU [6] and CodeBertScore [37], to determine how well LLM-generated code matches "something a developer might write." Fundamentally, these lexical metrics are just an approximation of what generated code should look like, structurally and syntactically; they only accidentally capture any semantic value, due to their generality. As noted above, for test code similarly generic *semantic* checks are available, including most obviously code coverage and ability to kill mutants. These are suitable for

fully automated reinforcement learning based approaches, without requiring the human intervention often invoked to check production code via Reinforcement Learning with Human Feedback (RLHF) [22].

Software code has been fundamentally shown to be *natural*: in other words there are common, repetitive patterns that can be learned by neural models [13]. This naturalness is a big reason why techniques such as code generation work so well, with LLMs learning common patterns of code from the vast corpus of open source projects.

Source code for tests is much more structurally regular than even normal source code. For example, unit tests are usually both loop and condition free, and can be broken down into setup/call/check patterns that are common across many tests. This has been observed to the point that formal models including only assignments, calls, and assertions have been successfully used as paradigms for unit test generation [14]. Further, the widespread use of a small number of unit testing frameworks (e.g., JUnit, GoogleTest, and Python's unittest module) makes test code even more similar across programs.

Modern LLMs have been pretrained on millions of source code tokens, thus these models have a strong understanding of code semantics and behavior. This can be seem with CodeBERT [7] embeddings outperforming prior work such as Code2Vec [1] and TF-IDF [32], with the CodeBERT vectors capturing some notion of semantic similarity rather than pure token match (as in techniques such as TF-IDF). These training bases contain a large number of tests, with many signals including naming and structure distinguishing the test code within these corpuses, making models "aware" of the tightly constrained nature of test code. Thus one can generate entire test suites with a powerful model such as ChatGPT [35, 30]. Even smaller models, trained with this relationship [26, 17] show significant promise, generating tests that compile, execute successfully, and increase coverage.

QUERY:

```
Can you write a DeepState harness to test a binary search
routine called using the function binsearch:
int binsearch(int a[], size_t N, int target)
where binsearch returns -1 if the target is not found,
otherwise the location of target in a?
```

RESPONSE:

```
#include <deepstate/DeepState.hpp>
extern "C" {
  int binsearch(int a[], size_t N, int target);
}
using namespace deepstate;
TEST(BinarySearch, BasicFunctionality) {
  // Define the size of the array
  size_t array_size = 10;
  // Generate a random array
  int a[array_size];
  for (size_t i = 0; i < array_size; ++i)
    a[i] = DeepState_Int(); // Generate a random integer

  // Sort the array (binary search requires a sorted array)
  std::sort(a, a + array_size);
  // Generate a random target value
  int target = DeepState_Int();
  // Perform binary search
  int result = binsearch(a, array_size, target);
  // Check the result
  if (result == -1) {
    ASSERT_FALSE(std::binary_search(a, a + array_size, target));
  } else {
    ASSERT_EQ(a[result], target);
  }
}
```

Figure 1: ChatGPT 3.5 generated response to query to generate a DeepState harness to test C binary search function.

4 BEYOND GENERATING UNIT TESTS

The discussion above largely focuses on using LLMs to generate "test code"—typically, unit tests. However, LLMs are also well-suited for more esoteric testing tasks, including some that do not involve generating code at all.

4.1 Other Test Generation Tasks

The context of existing production code gives LLMs more to work with, and the context of existing tests for production code may give LLMs even more to work with: LLMs can be used to augment existing tests with (better) oracles [5]. This is a particulary promising application, since the tendency of oracle quality to lag behind code coverage is an enduring and under-addressed problem in software testing [16].

Another critical problem in software testing is that while developers are at least familliar with writing unit tests, few have experience in writing property-based tests or fuzz-drivers [10], a different task that is often perceived as very difficult. Generalizing oracle generation from specific to parameterized unit tests is a potential long-term solution to this problem, and there is already work underway to generate fuzz drivers using LLMs [36], including by Google's OSS-Fuzz team, one of the most important industrial fuzzing efforts [20]. In fact, even a free general-purpose tool such as ChatGPT 3.5 can generate fuzz harnesses for specific fuzzing tools, e.g., Trail of Bits' DeepState "unit fuzzing" tool [11], as shown in Figure 1.

Finally, LLMs can directly be used as fuzzers, for compilers and other language tools [34] or even deep-learning libraries themselves [4].

4.2 Non-Code-Generation Testing Tasks

Mutation testing has long been one of the most promising ways to evaluate testing efforts; however, it is also expensive, both in terms of computational and human effort. Recently, LLMs have been used to significantly reduce the computational costs of mutation testing, by predicting (without running them) which mutants will kill which tests [15]. An obvious next step is to apply LLMs to the human-effort-intensive problems of identifying equivalent mutants [23] and prioritizing which mutants to examine [12].

5 CONCLUSION

We consider it fortunate that LLMs are likely highly suited to testing tasks, because, frankly, software developers generally do not enjoy testing [33], and therefore often do not put sufficient effort into testing [2] or reviewing test code [31]. Removing the burden of unpleasant and onerous (and often poorly done) tasks from humans is, we claim, the ideal use of automation. Better testing is, additionally, even more critical in a world where some portion of the code under test is to be generated by LLMs, which may produce new and surprising kinds of bugs that humans have trouble imagining, and thus addressing with appropriate tests.

Finally, as a provocative point, we suggest that in the long term, LLMs may offer a solution to one of the more long-standing and controversial problems of software engineering [21]: DeMillo, Lipton, and Perlis famously argued in the late 1970s that program verification could never be "proof" in the sense that mathematical proof is, because program proofs are simply *too boring* to be subjected to the social aspects of mathematical proof [3]. LLMs, however, may be applicable to the Leibnizian dream of program proof, in that they may be capable of many of the critique and refinement functions of human analysis of proofs, but without the limitation of being bored by tedious detail and extreme length. LLMs are already being used to help generate program proofs [8]; in the long run they may help make such proofs more meaningful and valuable, as well.

REFERENCES

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *CoRR*, abs/1803.09473, 2018.

[2] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 559–562, 2015.

[3] Richard A De Millo, Richard J Lipton, and Alan J Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, 1979.

[4] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 423–435, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. TOGA: A neural method for test oracle generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2130–2141. ACM, 2022.

[6] Aryaz Eghbali and Michael Pradel. Crystalbleu: Precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[8] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1229–1241. ACM, 2023.

[9] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136, 2017.

[10] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. Some problems with properties. In *Proc. Workshop on the Human Aspects of Types and Reasoning Assistants (HATRA)*, 2022.

[11] Peter Goodman and Alex Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018.

[12] Alex Groce, Iftekhar Ahmed, Josselin Feist, Gustavo Grieco, Jiri Gesi, Mehran Meidani, and Qihong Chen. Evaluating and improving static analysis tools via differential mutation analysis. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 207–218, 2021.

[13] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847, 2012.

[14] Josie Holmes, Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevi n Kellar, and James O'Brien. TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, 20(1):57–78, 2018.

[15] Kush Jain, Uri Alon, Alex Groce, and Claire Le Goues. Contextual predictive mutation testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 250–261, New York, NY, USA, 2023. Association for Computing Machinery.

[16] Kush Jain, Goutamkumar Tulajappa Kalburgi, Claire Le Goues, and Alex Groce. Mind the gap: The difference between coverage and mutation score can guide testing efforts. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 102–113, 2023.

[17] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

[18] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[19] Richard J. Lipton, Richard A DeMillo, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[20] Dongge Liu, Jonathan Metzman, Oliver Chang, and Google Open Source Security Team. AI-powered fuzzing: Breaking the bug hunting barrier. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html.

[21] Donald MacKenzie. *Mechanizing proof: computing, risk, and trust*. MIT Press, 2004.

[22] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[23] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple fast and effective equivalent mutant detection technique. In *International Conference on Software Engineering*, 2015.

[24] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[25] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

[26] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. Cat-lm: Training language models on aligned code and tests, 2023.

[27] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020.

[28] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. Lost at c: A user study on the security implications of large language model code assistants. *arXiv preprint arXiv:2208.09727*, 2023.

[29] Mohammed Latif Siddiq and Joanna Santos. Generate and pray: Using sallms to evaluate the security of llm generated code. *arXiv preprint arXiv:2311.00889*, 2023.

[30] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. An empirical study of using large language models for unit test generation, 2023.

[31] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. When testing meets code review: Why and how developers review tests. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 677–687, New York, NY, USA, 2018. Association for Computing Machinery.

[32] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[33] Philipp Straubinger and Gordon Fraser. A survey on what developers think about testing. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 80–90, 2023.

[34] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal fuzzing via large language models, 2023.

[35] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation, 2023.

[36] Cen Zhang, Mingqiang Bai, Yaowen Zheng, Yeting Li, Xiaofei Xie, Yuekang Li, Wei Ma, Limin Sun, and Yang Liu. Understanding large language model based fuzz driver generation, 2023.

[37] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code, 2023.