Codehacks: A Dataset of Adversarial Tests for Competitive Programming Problems Obtained from Codeforces

Max Hort
Simula Research Laboratory
Oslo, Norway
Email: maxh@simula.no

Leon Moonen
Simula Research Laboratory
BI Norwegian Business School
Oslo, Norway
Email: leon.moonen@computer.org

2025 IEEE Conference on Software Testing, Verification and Validation (ICST) | 979-8-3315-0814-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICST62969.2025.10988963

***Abstract*—Software is used in critical applications in our day-to-day life and it is important to ensure its correctness. One popular approach to assess correctness is to evaluate software on tests. If a test fails, it indicates a fault in the software under test; if all tests pass correctly, one may assume that the software is correct. However, the reliability of these results depends on the test suite considered, and there is a risk of false negatives (i.e. software that passes all available tests but contains bugs because some cases are not tested). Therefore, it is important to consider error-inducing test cases when evaluating software.**

To support data-driven creation of such a test-suite, which is especially of interest for testing software synthesized from large language models, we curate a dataset (Codehacks) of programming problems together with corresponding error-inducing test cases (i.e., "hacks"). This dataset is collected from the wild, in particular, from the Codeforces online judge platform. The dataset comprises 288,617 hacks for 5,578 programming problems, each with a natural language description, as well as the source code for 2,196 submitted solutions to these problems that can be broken with their corresponding hacks.

***Keywords*—competitive programming, language model, dataset**

I. Introduction

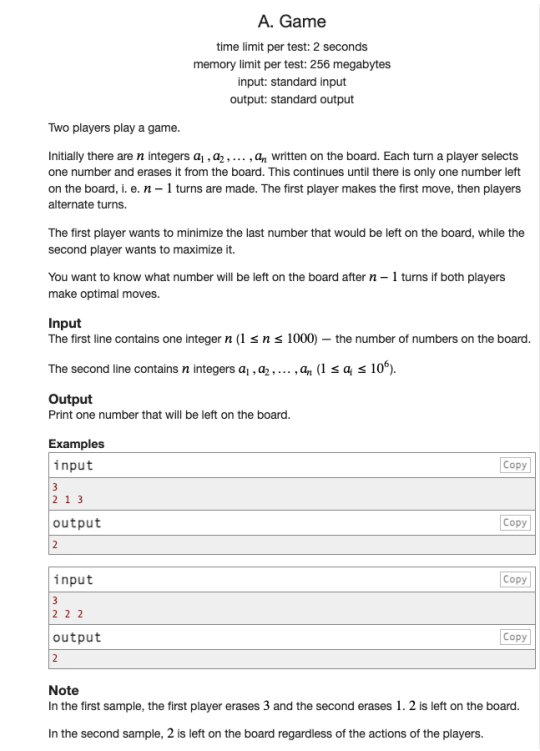
Large Language Models (LLMs) are increasingly being used to support us in our daily lives and have achieved high competitive performance on a variety of software engineering tasks (e.g., bug fixing, defect detection, program synthesis, program translation) [1]. However, LLMs have been shown to 'hallucinate' or 'confabulate', generating responses that may appear plausible but are incorrect. Although this has been mostly discussed in the context of natural language chats, it can also cause problems for software engineering tasks, such as code synthesis [2]. Therefore, it is of great importance to verify and ensure the correctness of the synthesized code [3]. One approach frequently used in practice is execution-guided evaluation [4], which applies a pre-defined test suite (e.g., unit tests, input-output pairs) to the generated code. If one of the tests fails, the snippet is treated as incorrect. If all tests pass, it is treated as a correct program synthesized by an LLM.

However, tests and, in particular, failure-inducing tests, are expensive and time-consuming to create [4], [5], [6]. As a result, there is a risk of false negatives where the code passes all available tests but still contains bugs because some cases were overlooked. To find such false negatives and ensure correctness of the code, additional tests are needed.

An **unexploited resource** in this regard is the online judge platform *Codeforces*. Online judges are platforms that allow users to participate in programming competitions and solve programming tasks, often in the programming language of their choice. User submissions are evaluated on predefined test suites, and much care is taken to ensure that the test suites are comprehensive [7]. However, it has recently been shown that these test suites do not always cover all cases, allowing false negative submissions to slip through [7], [8].

Unlike other online judge platforms, Codeforces provides competitors the opportunity to identify such false negative submissions during a competition as a means to increase their score. They can do this by "hacking" the submissions of other competitors that already passed the predefined test suite. The submission of user A is hacked by user B, if B can find an input for which the submission fails (e.g., it generates a different output than a pre-specified solution). Figure 1 illustrates an example hack obtained from a Codeforces contest. It should be noted that an unsuccessful hacking attempt by user B results in a penalty to their score. The successful hacks from Codeforces provide a valuable learning resource to support the data-driven creation of test inputs to find false negative submissions and evaluate the quality of synthesized code.

This paper introduces *Codehacks,* a **novel dataset** curating failure-inducing test cases based on "hacks" that are automatically collected from the Codeforces coding platform. As online judges already evaluate submissions with a multitude of tests, these represent edge cases that are costly to create manually and are valuable resources for future test generation and validation approaches for synthesized code, in particular by LLMs. At the initial release, the dataset comprises of 288,617 hacks for 5,578 programming problems, each with a natural language description, as well as the source code for 2,196 submitted solutions to these problems that can be broken with their corresponding hacks. The necessary resources (dataset and scripts) to update, replicate and build on our work are provided at: **https://doi.org/10.6084/m9.figshare.24773754**



A. Game

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Two players play a game.

Initially there are n integers $a_1, a_2, \ldots, a_n$ written on the board. Each turn a player selects one number and erases it from the board. This continues until there is only one number left on the board, i. e. $n - 1$ turns are made. The first player makes the first move, then players alternate turns.

The first player wants to minimize the last number that would be left on the board, while the second player wants to maximize it.

You want to know what number will be left on the board after $n - 1$ turns if both players make optimal moves.

Input
The first line contains one integer n ($1 \le n \le 1000$) — the number of numbers on the board.

The second line contains n integers $a_1, a_2, \ldots, a_n$ ($1 \le a_i \le 10^6$).

Output
Print one number that will be left on the board.

Examples

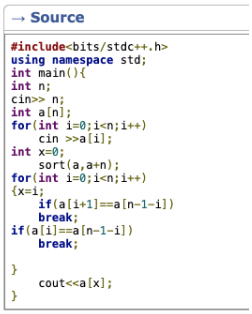
input Copy
3
2 1 3
output Copy
2

input Copy
3
2 2 2
output Copy
2

Note
In the first sample, the first player erases 3 and the second erases 1. 2 is left on the board.

In the second sample, 2 is left on the board regardless of the actions of the players.

(a) Problem description.

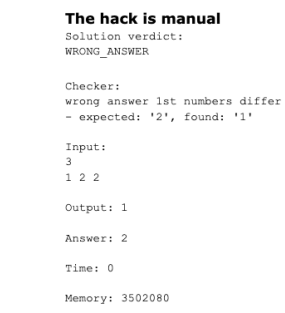


→ Source

```
#include<bits/stdc++.h>
using namespace std;
int main(){
int n;
cin>> n;
int a[n];
for(int i=0;i<n;i++)
    cin >>a[i];
int x=0;
    sort(a,a+n);
for(int i=0;i<n;i++)
{x=i;
    if(a[i+1]==a[n-1-i])
    break;
if(a[i]==a[n-1-i])
    break;

}
    cout<<a[x];
}
```

(b) Accepted solution.



The hack is manual

```
Solution verdict:
WRONG_ANSWER

Checker:
wrong answer 1st numbers differ
- expected: '2', found: '1'

Input:
3
1 2 2

Output: 1

Answer: 2

Time: 0

Memory: 3502080
```

(c) Successful hacking attempt.

Fig. 1. Example of a hacked Codeforces submission with corresponding problem description and hacking attempt.

II. Related Work

A. Online Judge Datasets

Online judges provide programming problems to a wide audience and allow them to submit their own solutions which are evaluated on test suites to verify their correctness. The public sharing of this data allowed for the creation of multiple datasets that provide ample resources to support software engineering tasks. Over the last few years, datasets have been created that utilize data from online judge platforms such as Codewars, AtCoder, Kattis, Codeforces, Google Code Jam, CodeChef, and Hackerearth [9], [10], [11].

One of the largest and most frequently used data sets is CodeNet by Puri et al. [12]. CodeNet consists of over 14 million code samples in 55 programming languages, with sample input and output pairs (tests) for 98.5% of the code. These data are crawled from the Aizu and AtCoder online judge platforms. Subsets of CodeNet have been used to create program repair datasets [13], as well as incorporated in CodeContests, the dataset created for training the AlphaCode language model [11]. The complete CodeContests dataset contains problem descriptions, submissions, and test cases that were collected from the Aizu, AtCoder, CodeChef, Codeforces, and HackerEarth platforms.[1] The collected submissions are written in the three most frequently used programming languages: Python, Java, and C++.

To evaluate whether the tests for AtCoder are effective in detecting false negatives, Liu et al. [7] extracted 541,552 accepted solutions from 939 coding problems from the CodeContests dataset. After randomly generating additional tests, they found false negative submissions for 43.1% of the problems. A subset of these data containing 3,043 false negative submissions is shared as "TrickyBugs" to stimulate future research [14].

In addition, there are datasets that are completely based on Codeforces problems. An early dataset created in 2017 by Tan et al. [15] is called CodeFlaws. CodeFlaws is designed to support program repair tasks and consists of 7,436 programs from Codeforces (i.e., a user submission to a specific problem). Each submission consists of a rejected submission, classified in one of 39 defect classes, and an accepted one. Code4Bench introduced a richer dataset with 3,421,357 Codeforces programs written in 28 programming languages [16].

However, none of the existing works made use of the "hacking" functionality that Codeforces provides to curate their datasets. In particular, unlike the randomly generated tests by Liu et al. [7], these hacks are submitted by human contestants, which is helpful when evaluating code synthesis tools, trained on human-written code.

B. Program Synthesis

Program synthesis aims at the generation of a program with a specified target language, based on natural language descriptions or specifications of input-output pairs [17]. Advances in LLMs for code synthesis introduced well-performing models, such as AlphaCode which was able to rank in the top 54.3% in Codeforces coding competitions [11]. AlphaCode is a transformer model pre-trained on source code from GitHub repositories and fine-tuned on the CodeContests dataset. Another LLM for code generation is Codex [18], a GPT model fine-tuned on GitHub repositories for writing Python code.

Jain et al. [19] proposed an interactive system for code synthesis called Jigsaw. Jigsaw allows users to describe the program they want to generate with natural language and test cases (input and output pairs). Generation is then carried out by integrated LLMs, such as GPT-3 and Codex.

While Li et al. [11] evaluate AlphaCode on Codeforces contests, they mention that hacking was not performed during their evaluation of AlphaCode (i.e., the solutions submitted by AlphaCode to the contests cannot be hacked unlike regular submissions). The use of hacks as an additional resource for training was not considered.

[1] https://github.com/deepmind/code_contests

C. Test Generation

Tests are required to ascertain the correctness of software. Given the effort, cost, and time required to generate suitable tests for source code, test generation techniques have been developed to support software engineers with this task. Here, we outline recent approaches for test generation that utilize LLMs and could benefit from our *Codehacks* dataset.

Tufano et al. [20] proposed ATHENATEST which treats unit test generation as a sequence-to-sequence learning task. Schaefer et al. [5] introduced TESTPILOT, a unit test generator based on Codex [18]. Codex is utilized without further retraining, solely by prompts including the respective function and examples.

Liu et al. [8] proposed a benchmarking framework (EvalPlus) to assess the correctness of LLM-synthesized code. EvalPlus employs both LLMs and mutation-based strategies to generate additional tests. In particular, ChatGPT is used to generate seed tests that are modified by mutation strategies. The prompt for ChatGPT includes the program solution, example test inputs, as well as an encouragement to generate interesting test inputs.

In many studies, the LLMs are used *off-the-shelf*, and the investigation is aimed at finding suitable prompts for the generation of tests [4]. Our work is orthogonal to such studies, as we aim to curate a high-quality dataset that can then be used to fine-tune an LLM for the task of generating failure-inducing tests given a natural language description of a programming problem.

III. Codeforces Dataset - Collection and Curation

Codeforces is a platform designed for practicing and participating in programming contests [16], [21]. Codeforces hosts contests consisting of multiple problems. Users can join a contest live, during which they can work on the problems for a designated time duration, but they can also solve problems offline, after the contests are over. Competitors have a free choice of programming language, while test inputs are independent of the programming language used [16]. During the duration of the contest, users can hack the submission of other contestants to gain additional points and improve their ranking. Hereby, an incorrect hacking attempt causes a point deduction. In addition, there are educational contests that allow for hacking up to 12 hours after contest completion for learning purposes.[2]

In the following, we outline the steps that were performed to collect hacks and submissions from competitions held on Codeforces and create the Codehacks dataset (Section III-A). Section III-B provides details on the collected hacking attempts, as well as a discussion of our choices on which hacks to include in the dataset. Section III-C describes the programming problems for which hacks were collected, and Section III-D discusses possible limitations of the collection process.

[2] https://codeforces.com/blog/entry/107753

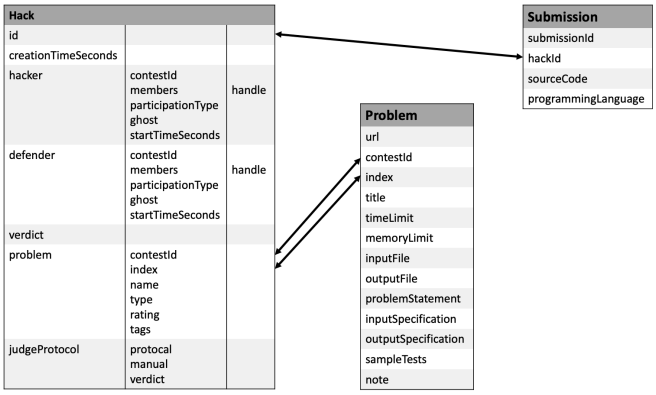


Fig. 2. Structure of the collected dataset.

A. Data Collection

To collect relevant data from Codeforces, we use their API[3] and a publicly available crawler.[4] With help of the Codeforces API, we obtain information about hacks. In particular, we first used the API to find the IDs for every contest held on Codeforces and afterwards obtain a list of hacking attempts for each contest. In total, from the $1,928$ contests currently on Codeforces, $1,647$ contests have hacking attempts from users, resulting in 393,382 successful hacks.[5]

While the API call provides useful information, such as the input used for a hacking attempt, the problem description is not provided. Therefore, we follow Tan et al. [15], to use and modify a publicly available crawler[4] to extract problem descriptions for each of the 5,578 problems with successful hacks. A description of the information obtained for hacks and problems is shown in Figure 2.

Continuing the crawling process to gather further details on submissions, such as their source code, turned out infeasible as they are part of the "robots.txt". Therefore, we used Code4Bench [16], a dataset from 2018 with $3,421,357$ submissions, to match the submission for our hacks with the already collected dataset. In total, the hacks from our dataset matched with 2,196 submissions from Code4Bench. We add the respective source code and programming language used to our dataset, as outlined in Figure 2.

According to the Codeforces data sharing guidelines, we provide the URL to each problem description crawled.[6] An explanation of the required steps and relevant source code is provided in our online Appendix.[7]

B. Hack Verdicts

Each hacking attempt is evaluated by the Codeforces platform to determine whether it was successful or not (i.e., whether the hacker is able to show erroneous behavior of an accepted submission). For example, to make a submission fail

[3] https://codeforces.com/apiHelp
[4] https://github.com/Nymphet/codeforces-crawler/tree/master
[5] We accessed the API on the 18th of November 2024.
[6] https://codeforces.com/blog/entry/967
[7] https://doi.org/10.6084/m9.figshare.24773754

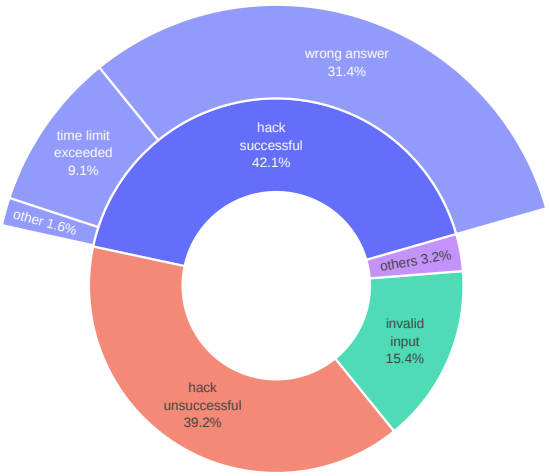


Fig. 3. Verdicts of the submitted hacks.

due to a "time limit exceeded" error, one often needs to create as many inputs lines as allowed by the problem specification. To make a submission fail due to a wrong answer, one needs to find an input that is not covered by the online judges test set. Figure 3 illustrates the distribution of the verdicts among the 1,002,339 hacking attempts, of which 393,382 (42.1%) are successful. The majority of successful attempts force the submission to generate a wrong answer (31.4% of total) and the next most successful type group makes the submission exceed predefined time limits (9.1% of total). The remaining types of errors that are induced include runtime errors and exceeding memory limits. 39.2% of the submitted hacks are not able to point out errors in the submissions (hack unsuccessful), while 15.4% do not conform to the input specifications of the problems and are therefore not executed (invalid input). We also observed that 3.2% of the hacking attempts received other tags (i.e., generator incompilable, generator crashed, ignored, other, testing). We treat these, as well as invalid inputs, as unsuccessful hacking attempts and exclude them from the dataset.

We note that the Codeforces API abbreviates long input sequences (e.g., an input can consist of more than 10,000 lines) by substituting parts of the input with "..." [16]. This is the case for 104,765 of 393,382 successful hacking attempts. These are omitted from the collection as their input cannot be reproduced, leaving 288,617 hacks for the dataset collection.

C. Problem Types

Each Codeforces problem is categorized based on different tags that indicate the type of problem (e.g., dynamic programming). Moreover, problems have a certain difficulty level ranging from 800 to 3500, to indicate how difficult it is for a user to solve this problem.[8] In Figure 4, we illustrate frequency of problem types and their respective difficulty level for the collected hacks. The tags are ordered by frequency in the datasets, showing that the most frequently used tags to describe problems with successful

[8] Note that 2,415 of the problems did not have a specified difficulty level.

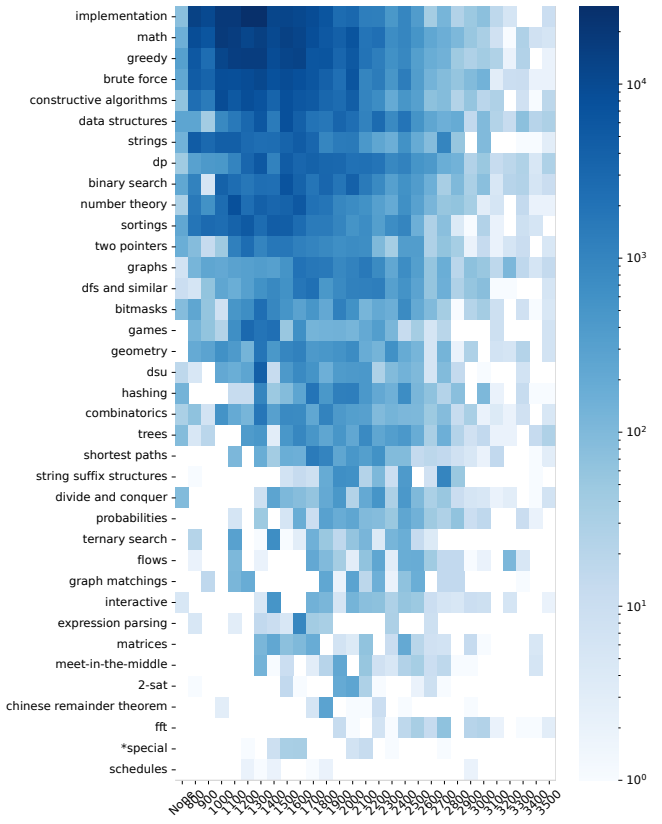


Fig. 4. Distribution of hacks per problem tag (y-axis) and difficulty (x-axis).

hacks are: implementation, math, greedy. Moreover, we can observe that there is a larger number of hacks for problems with lower difficulty levels, which could be caused by the overall distribution of problems on Codeforces (e.g., there are more problems with lower difficulty), or the proficiency level of contestants (e.g., contestants that solve problems with low difficulty levels have a lower rating and therefore create solutions that are "hackable").

D. Limitations

The quality of the successful hacks depends on the pre-defined test suite employed by Codeforces for each of the problems. There are no regulations on how many tests a problem should include, and therefore it is not possible to judge the coverage or completeness of the test suites. This could lead to some of the hacks collected being "easy" due to small initial test suites, while other problems with more extensive test suites may not be covered because no successful hacks were submitted for them.

IV. Conclusions and Future Usage

This paper introduces Codehacks, a dataset of user-submitted hacks that reveal errors in submissions to programming problems that the standard test suites on Codeforces are not able to cover. In total, this data set comprises 288,617 successful hacking attempts and 2,196 source code submissions for 5,578

programming problems. For each of the problems, we include the natural language description.

Curating a collection of failure-inducing test cases for programming problems is of interest, as such test cases can be difficult and costly to find. We hope that the provided dataset can function as a valuable resource for software testing and test generation.

One promising application area is to improve code synthesis. It is known that high-quality tests can help improve code synthesis [22], and that it is important to consider edge cases when testing programs, as these are the most likely to point out flaws in the program logic. Our dataset can be used to fine-tune an LLM for generating such edge cases (i.e., generate likely error-inducing tests), based on natural language descriptions of the problem for which code is synthesized.

Another promising field for applying our dataset is test generation in an adversarial setting. In this case, an LLM such as AlphaCode or Codex is used to synthesize code from natural language descriptions, while another model is trained to generate challenging test inputs from natural language descriptions. These test inputs are then used to determine the quality of the code generated by the first model. Similarly to the iterative approach of Liventsev et al. [23], who generated code and utilized feedback from failed tests for iterative debugging and repair, this adversarial framework can be extended to iteratively generate novel challenging tests, forcing the code synthesizer to generate more robust code.

ACKNOWLEDGMENT

We would like to thank **MikeMirzayanov** for creating the Polygon and Codeforces platforms.

This work is supported by the Research Council of Norway through the secureIT project (IKTPLUSS #288787), and by the European Union through the Horizon Europe Marie Skłodowska-Curie Actions (#101151798).

REFERENCES

[1] C. Niu, C. Li, B. Luo, and V. Ng, "Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*. Vienna, Austria: International Joint Conferences on Artificial Intelligence Organization, Jul. 2022. doi: 10.24963/ijcai.2022/775 pp. 5546–5555.

[2] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large Language Models for Software Engineering: Survey and Open Problems," Oct. 2023.

[3] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On Reliability of Patch Correctness Assessment," in *International Conference on Software Engineering (ICSE)*, May 2019. doi: 10.1109/icse.2019.00064 pp. 524–535.

[4] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "CodeT: Code Generation with Generated Tests," Nov. 2022.

[5] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "Adaptive Test Generation Using a Large Language Model," Feb. 2023.

[6] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, S.-C. Cheung, and J. Kramer, "Finding Failure-Inducing Test Cases with ChatGPT," Jun. 2023.

[7] K. Liu, Y. Han, J. M. Zhang, Z. Chen, F. Sarro, M. Harman, G. Huang, and Y. Ma, "Who Judges the Judge: An Empirical Study on Online Judge Tests," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Seattle WA USA: ACM, Jul. 2023. doi: 10.1145/3597926.3598060 pp. 334–346.

[8] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," in *Advances in Neural Information Processing Systems*, vol. 36, Dec. 2023, pp. 21 558–21 572.

[9] D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt, "Measuring Mathematical Problem Solving With the MATH Dataset," in *Thirty-Fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, Aug. 2021.

[10] F. Ullah, H. Naeem, S. Jabbar, S. Khalid, M. A. Latif, F. Al-turjman, and L. Mostarda, "Cyber Security Threats Detection in Internet of Things Using Deep Learning Approach," *IEEE Access*, vol. 7, pp. 124 379–124 389, 2019. doi: 10.1109/access.2019.2937347

[11] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022. doi: 10.1126/science.abq1158

[12] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks," Aug. 2021.

[13] J. A. Prenner and R. Robbes, "RunBugRun – An Executable Dataset for Automated Program Repair," Apr. 2023.

[14] K. Liu, Y. Han, Y. Liu, Z. Chen, J. M. Zhang, F. Sarro, G. Huang, and Y. Ma, "TrickyBugs: A Dataset of Corner-case Bugs in Plausible Programs," in *Proceedings of the 21st International Conference on Mining Software Repositories*. Lisbon Portugal: ACM, Apr. 2024. doi: 10.1145/3643991.3644870 pp. 113–117.

[15] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: A programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017. doi: 10.1109/icse-c.2017.76 pp. 180–182.

[16] A. Majd, M. Vahidi-Asl, A. Khalilian, A. Baraani-Dastjerdi, and B. Zamani, "Code4Bench: A multidimensional benchmark of Codeforces data for different program analysis techniques," *Journal of Computer Languages*, vol. 53, pp. 38–52, Aug. 2019. doi: 10.1016/j.cola.2019.03.006

[17] X. Chen, C. Liu, and D. Song, "Execution-Guided Neural Program Synthesis," in *International Conference on Learning Representations*, Apr. 2019.

[18] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," Jul. 2021.

[19] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, Jul. 2022. doi: 10.1145/3510003.3510203 pp. 1219–1231.

[20] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit Test Case Generation with Transformers and Focal Context," May 2021.

[21] M. M. Rahman, B. C. Das, A. A. Biswas, and M. M. Anwar, "Predicting Participants' Performance in Programming Contests using Deep Learning Techniques," Feb. 2023.

[22] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program Synthesis with Large Language Models," Aug. 2021.

[23] V. Liventsev, A. Grishina, A. Härmä, and L. Moonen, "Fully Autonomous Programming with Large Language Models," in *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2023. doi: 10.1145/3583131.3590481