Fuzz Testing Projects in Massive Courses

Sumukh Sridhara
UC Berkeley
sumukh@berkeley.edu

Brian Hou
UC Berkeley
brian.hou@berkeley.edu

Jeffrey Lu
UC Berkeley
jeffreylu017@gmail.com

John DeNero
UC Berkeley
denero@berkeley.edu

ABSTRACT

Scaffolded projects with automated feedback are core instructional components of many massive courses. In subjects that include programming, feedback is typically provided by test cases constructed manually by the instructor. This paper explores the effectiveness of *fuzz testing*, a randomized technique for verifying the behavior of programs. In particular, we apply fuzz testing to identify when a student's solution differs in behavior from a reference implementation by randomly exploring the space of legal inputs to a program. Fuzz testing serves as a useful complement to manually constructed tests. Instructors can concentrate on designing targeted tests that focus attention on specific issues while using fuzz testing for comprehensive error checking. In the first project of a 1,400-student introductory computer science course, fuzz testing caught errors that were missed by a suite of targeted test cases for more than 48% of students. As a result, the students dedicated substantially more effort to mastering the nuances of the assignment.

ACM Classification Keywords

K.3.2. Computers and Education: Computer and Information Science Education; Computer science education

Author Keywords

automated assessment; behavioral analytics; online learning

INTRODUCTION

Assignments that provide automated feedback to students are a core instructional tool in many massive courses [8, 18]. In technical subjects that involve programming, such as computer science and data science, assignments often ask students to implement a program according to a specification. Feedback is generated automatically by an *autograder*, a program that executes test cases against a student's implementation [12, 6, 13]. The test suite is typically constructed manually by the course instructors, with some tests distributed to students and some tests held out for assessment. This paper describes how *fuzz testing* [15] can complement manually constructed test cases to improve the feedback provided to students.

Fuzz testing involves randomly generating inputs to a program and then verifying some property of the output or program behavior. For example, this technique is often applied to find inputs on which a program will crash. In the context of large-scale education, randomized testing can be used by an autograder to verify that the output generated by a student's submission to a programming assignment matches the output produced by a reference implementation created by the instructor [19]. This paper describes the conditions necessary for fuzz testing to be used for autograding, as well as an array of options for distributing fuzz tests to students without exposing the instructor's reference implementation.

Programming assignments and automated feedback are often the primary mechanisms by which massive courses encourage students to explore the technical details of a subject. In such settings, test cases provide two complementary services to students. *Targeted test suites* highlight particular known points of confusion, often using simple inputs for which the correct output can be generated easily by hand. *Comprehensive test suites* attempt to detect all deviations from the specification. Targeted tests are helpful in providing preliminary feedback to students, while comprehensive tests encourage students to continue working on the assignment until it is completed correctly. When a comprehensive test suite fails to detect an error, the student may stop attempting to improve her or his submission. As a result, the student may inadvertently miss the learning opportunity of identifying and correcting the error.

To explore the impact of fuzz testing, we studied the behavior of students in a large on-campus CS 1 course with 1,400 enrolled students, CS 61A at UC Berkeley[1]. We focused on their first substantial programming project, which students completed either alone or in pairs. The most challenging question in the project asked students to write a function that simulates playing a dice game. Students were provided with both a manually constructed suite of targeted test cases and a fuzz test. Our autograder recorded each partial solution submitted for testing, so we were able to study students' progress through the problem.

We found that adding a fuzz test dramatically increased the number of incorrect solutions identified. More than 48% of

[1] `cs61a.org`.

L@S 2016, April 25–26, 2016, Edinburgh, Scotland Uk

ACM 978-1-4503-3726-7/16/04 . . . $15.00
DOI: **http://dx.doi.org/10.1145/2876034.2876050**

students created and tested a partial solution that passed the targeted test suite, but still contained a logical error identified by the fuzz test. Many random inputs were required to detect these errors; 6% of the errors we found were not discovered until more than 500 random inputs were explored.

Fuzz tests are easy to construct and distribute using the strategies we describe in this paper. They proved to be an effective complement to manually created tests even for a basic assignment: the first project in a CS 1 course. Our study of how students behave in a large-scale class provides compelling evidence that fuzz testing should be included regularly in autograded programming assignments.

AUTOGRADING

For many successful massive open online courses (MOOCs), programming projects are distributed with autograders that provide comprehensive feedback about the correctness of a student's implementation. These projects are not primarily used for assessment, but rather to maximize the instructional value of completing the assignment [4]. All students are encouraged to revise their project submissions until they are correct, and in the process they are exposed to important concepts from the course. Many popular programming-based MOOCs include comprehensive autograders, such as *Algorithms*[2], *Coding the Matrix*[3], *Foundations of Computer Graphics*[4], *Machine Learning*[5], and many others.

Several design issues arise in the construction of autograders that are relevant to the use of fuzz testing as described in this paper.

Test Distribution

Autograder tests can be distributed with a project or applied by an instructional server when an assignment is submitted. The advantages of distributing tests with a project are that they can be run offline, the computation required to execute the tests is distributed to students' machines, and students who do not want to share their progress with instructors due to privacy concerns still have access to test cases. However, one challenge of distributing tests to students is that they cannot include a solution to the project without potentially exposing that solution to the student.

Assessment

Not all test suites distributed with programming projects are designed to be comprehensive. Indeed, a common practice in programming-based courses is to withhold certain test cases for assessment. Students are sometimes encouraged to generate their own test cases to verify their implementations before submitting their work. These alternatives to comprehensive testing are certainly a proper fit for certain courses, but in this paper we focus on tests distributed to students that are intended to provide comprehensive evaluations of correct program behavior.

[2] `www.coursera.org/course/algs4partI`
[3] `www.coursera.org/course/matrix`
[4] `courses.edx.org/courses/BerkeleyX/CS_184.1x`
[5] `www.coursera.org/learn/machine-learning`

Targeted Tests

When designing the test suite of an autograder, it can be valuable to consider more than just comprehensive test coverage; instructors may also seek to provide guidance that properly matches each stage of a student's development process. We call any test case that isolates a particular issue a *targeted test*. Targeted tests complement comprehensive tests by focusing students' attention on particular details within a large program. These tests are often designed to make students think about certain edge cases that they may overlook in a first pass through an implementation.

Fully quantifying the value of targeted tests is beyond the scope of this paper, but in constructing our evaluation of fuzz testing we also discovered some evidence that targeted tests are a valuable complement to comprehensive tests. In the Spring 2015 offering of CS 61A, students were provided only with comprehensive tests for the question we describe later in the paper. In the Fall 2015 offering, they were provided with both targeted tests and comprehensive tests. When provided no targeted tests (Spring 2015), the median time between starting this question and passing at least one test case was 55 minutes. When provided targeted tests (Fall 2015), the median time to passing at least one test case was greatly reduced to 14 minutes. Targeted tests provide timely positive feedback that students are making progress in the correct direction.

Summary

Although the space of possible autograder designs is large, we focus on the particular setting in which a test suite is distributed to students with the project. The test suite includes both targeted tests to focus attention on key issues and comprehensive tests that attempt to detect all deviations from the assignment specification. The purpose of fuzz testing is to improve comprehensive tests while simplifying their development.

FUZZ TESTING

Fuzz testing—testing the behavior of a program on many random inputs—was first described as a technique to discover errors in mature UNIX tools [15]. Generating random inputs was sufficient at that time to crash 25-33% of core utility programs from various distributions of UNIX. This original paper aptly remarked that fuzz testing is best viewed as a complementary technique to conventional testing in which specific inputs are chosen manually by a programmer. A variety of work has improved and extended the original idea of fuzz testing, e.g. [7, 14]; randomized fuzz testing is now standard practice in software engineering [3]. The relative value of random testing is typically high compared to its implementation cost [11, 16]. Creating fuzz tests is often far less effort-intensive than designing tests manually.

Application

Fuzz tests are commonly used to assess the security of a program given some randomized inputs [5, 10]. During execution, the program state is monitored to ensure that security constraints are maintained. In common usage, fuzz tests watch for a specific behavior (such as a buffer overflow or

program crash), often ignoring the actual output of the program. The correct output for a random input is often unknown, because no reference implementation exists for most programs being tested in this way. By contrast, when applying fuzz testing to programming projects, a reference implementation (the instructor solution) often exists. With a reference implementation, we can apply fuzz testing to student submissions to test for correctness.

Creating and Distributing Fuzz Tests

Any program can be fuzz tested for correctness as long as the behavior of the program for a space of inputs is uniquely defined by the assignment specification. The behavior of the reference implementation must be deterministic, and the specification of the implementation must precisely resolve any possible sources of non-determinism (such as breaking ties in a maximization).

To fuzz test student code, the autograder must programmatically feed random input into the student's program. Since the input domain for the program is known, an instructor can easily generate valid inputs. Existing work on fuzz testing tools provides a variety of strategies for the generation for complex inputs [9, 2, 20].

These inputs alone help verify that the program does not crash on valid inputs, but do not verify correctness of the program. Instead, some mechanism must provide access to the correct response for a random test according to the reference implementation, but without distributing the reference to students.

Our solution is to generate one set of random fuzz tests using a fixed random seed. The same set of tests is distributed to all students, ensuring fair and consistent autograder responses across multiple autograder runs, while maintaining the simplicity of randomly generated tests. We compute the correct response for each test according to the reference implementation to obtain the expected fuzz test output. The following options describe methods in which this expected output can be distributed.

1. **Raw Output.** This approach provides students with the reference implementation's raw output for each fuzz test input. The reference output is compared to the output generated by the student's program. Since each run of the autograder tests the same inputs, it is possible for students to hard-code their implementation to pass the fuzz test, without fully completing the assignment.

2. **Obfuscated Output.** Obfuscating the random inputs and outputs is one method to discourage students from hard-coding their answer to match the correct fuzz test outputs. Students are only informed whether their implementation passes or fails a sequence of fuzz tests, with no further information.

   To implement this behavior, students are given the encrypted output of the instructor solution, rather than the raw output. The encrypted output can be computed using a one-way hash function H that combines all fuzz test outputs for the instructor solution, $fuzz(solution)$, and produces a hash $h_{correct} = H(fuzz(solution))$. A student's implementation is correct if their hash $H(fuzz(student))$ matches $h_{correct}$.

3. **Traces.** If the internal state of the student's code can be inspected by the autograder, it is possible to provide even more detailed feedback to the student. Specifically, the autograder can not only state that a specific fuzz test failed, but rather exactly when the internal state differed from the reference state. This level of detail requires that tests include a serialized list of all program states in the trace. This additional information allows fuzz tests to pinpoint the places in which a student's implementation deviates from the reference, as with a targeted test, without requiring careful test generation from the instructor.

4. **Hidden.** It is also possible to keep the fuzz tests hidden from student view, instead choosing to apply fuzz tests as a way to assess student progress and identify frequent errors. Instructors can use these data to alter their assignments accordingly. If desired, the instructors can publish the most common errors as well as test cases that highlight these errors. Knowing the most common incorrect outputs allows instructors to determine common bugs and more effectively address student questions either in-person or online. This information becomes more important in massive classes as the observed set of unique incorrect implementations grows large.

5. **Final Assessment.** Fuzz tests are useful for assessing the correctness of submitted assignments, even if the fuzz tests are never distributed to students. Using fuzz tests for assessment saves the effort of constructing new tests for each course offering. In some courses, test cases that are used to assign a student's grade are never released to students, even after scores are assigned, in part due to the effort required to generate new test cases for future course offerings. However, when students are not provided with the inputs that caused their program to fail, they lose the opportunity to understand why they were penalized. Communicating fuzz tests to students does not compromise the set of hidden tests available for future course offerings.

IMPLEMENTATION

We explored the use of fuzz testing in an autograder by adding a fuzz test to an existing manually created test suite of the first project in a CS 1 course. We distributed the fuzz test using the *Obfuscated Output* method described in the previous section.

The Game of Hog

The first project in our CS 1 class is implementing a simulator for the game of Hog [17]. In Hog, two players alternate turns trying to be the first to end a turn with at least 100 total points. On each turn, the current player chooses some number of dice to roll, up to 10. That player's score for the turn is the sum of the dice outcomes. There are some additional rules to make the game more exciting to play and challenging to implement.

We used a fuzz test to verify student implementations of the game's main simulation loop, the fifth of ten questions in the project. This main loop updates the players' scores after each

Students Attempting Project	1,392
Students Attempting Target Question	1,355
Students Completing Project	1,331
Code Snapshots	486,482
Average Snapshots Per Student	349
Incorrect Attempts at Target Question	48,079
Snapshots with Fuzz Test Errors	22,375

Figure 1. Collected Dataset

roll of the dice. CS 1 students find this question challenging to implement because they must use iteration to simulate each turn of the game, modify multiple variables, and apply functions that were implemented in previous parts of the project. Our students commonly struggle with off-by-one errors where they simulate an additional turn after the game should have ended, infinite loops where their games never end as a result of forgetting to modify variables, or incorrect score updates.

The number of possible inputs to this function is extremely large because each player can roll a number of dice between 0 and 10 for all 10^4 game states, creating at least 10^{10} unique inputs to define the players' strategies.[6] Attempting to verify the correctness of a student implementation by enumerating all legal inputs is infeasible. However, fuzz tests are an easy way to augment hand-designed test cases to provide better coverage of the input space. In our implementation, the random input dictated both players' strategies and all dice outcomes.

Autograder Interface

Project source files are distributed with a suite of test cases and an open-source autograder called OK[7]. Students use OK to run correctness tests on their implementation after unlocking conceptual questions [1]. Every question has several instructor-designed test cases to provide students with targeted feedback on the correctness of their solution. Students are strongly encouraged to correct their solutions to pass all test cases before progressing to the next question.

Students can run the provided tests on a particular question as often they would like. Each time students run the tests, a snapshot of their code, as well as metadata about which question they are working on and the number of tests that their solution passes, is sent to the OK server. These snapshots allow us to continuously monitor student progress, which we have used to develop a large dataset of in-progress work, as seen in Figure 1.

After running OK on a particular question, students receive feedback on the correctness of their program. When students fail a test case, OK displays the test that triggered the failure, the expected value, and the student's returned value. This interface provides students with enough information to start debugging their error. Once their solution passes all tests for a question, students know it is safe to proceed to the next question.

[6] If a random sequence of dice outcomes is included in the input as well, the space of possible inputs grows much larger still.

[7] `okpy.org`

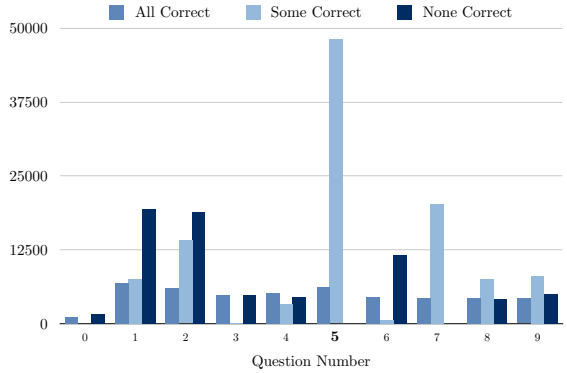

Figure 2. The distribution of attempts on each question of Hog. In the Fall 2015 offering of the course, we provided both targeted tests and fuzz tests for Question 5.

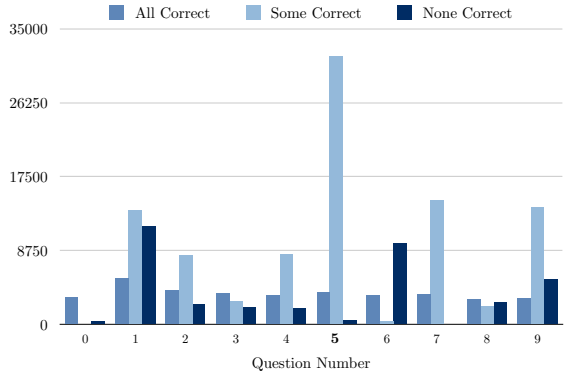

Figure 3. The distribution of attempts on each question of Hog. In the Spring 2015 offering of the course, we provided only a few manually designed fuzz tests for Question 5. This offering had approximately 30% fewer students than Fall 2015, so the vertical axis has been scaled accordingly.

Figure 2 (Fall 2015) and Figure 3 (Spring 2015) show how often students ran the autograder to check their solutions for each question of Hog in two separate offerings of the course. For each question, we classify each attempt to pass the autograder into three scenarios: no test cases passed ("None Correct"), some test cases passed ("Some Correct"), and all test cases passed ("All Correct"). The question that we have targeted for fuzz testing is Question 5. It is the most challenging question on the project and therefore requires a large number of autograder runs before passing all tests. In our project, a question refers to a subtask of the project that has tests for correctness.

Student Experience

We investigated student experience of the fuzz test by observing the number of questions asked about the fuzz test on the class forum. The online Q&A forum, called Piazza, allows students to ask questions of their peers and instructors. The median response time for public posts about the project was 27 minutes for peer responses and 37 minutes for an instructor response.

Our choice to obfuscate the output of the fuzz test significantly increased the amount of time required for the instruc-

tors to assist a student with a question about the fuzz test, both in office hours and on Piazza. Due to the obfuscated output, students' only recourse for help if they were failing the fuzz test was to share their code in office hours or make a private post with their code attached for instructors to view. There were 86 such private posts (which is about 1 post for every 8 students who failed the fuzz test). The median time for a response from an instructor to these posts was 2.3 hours due to the difficulty of identifying the student's error in the large space of potential errors identified by the fuzz test.

In a midterm survey with 1,245 responses, 575 students (46%) reported spending at least one hour trying to debug errors caught by the fuzz test; 19% reported spending more than four hours. 346 students (28%) responded that they did not fail the fuzz test or did not remember the fuzz test.

EVALUATION

We first quantified how fuzz testing complemented our manually constructed test cases by measuring the number of students who passed all of our manually constructed test cases but failed the fuzz tests.

Student Fuzz Test Design

We designed a fuzz test that simulates 100 games of Hog. Before distributing the project, we randomly generated 100 fuzz test inputs and evaluated the staff solution on each of these inputs. We computed a hash from the 100 staff solution outputs, which we then distributed to students along with the targeted tests.

Usage Pattern

For the question that we analyzed, students were provided with a sequence of 9 targeted tests, as well as the fuzz test described above, which was executed only after the students passed the previous 9 tests.

This analysis excludes students who opted out of uploading progress as well as those who worked on the project without running the provided tests.

Fuzz Test Evaluation

To analyze the effects of additional fuzz tests on detecting student errors, we developed a larger fuzz test than the one distributed to students originally. We randomly generated 1,000 test inputs. We graded all snapshots in which students modified the originally provided code for the target question, resulting in a total of 173,468 snapshots of student code.

For each snapshot, we evaluated the student code on all 1,000 fuzz test inputs, producing a vector of 1,000 student outputs. We assumed that if two snapshots produced the same output vector, they contained the same logical errors. The snapshots that we examined manually all validated this assumption.

Figure 4 shows the frequency of the most common logical errors. Each bar represents one response vector, ordered from most to least common. The figure shows that the 10 most common logical errors account for 41% of snapshots with errors. The 50 most common errors account for 71% of snapshots. Figure 5 describes the logical errors that resulted in

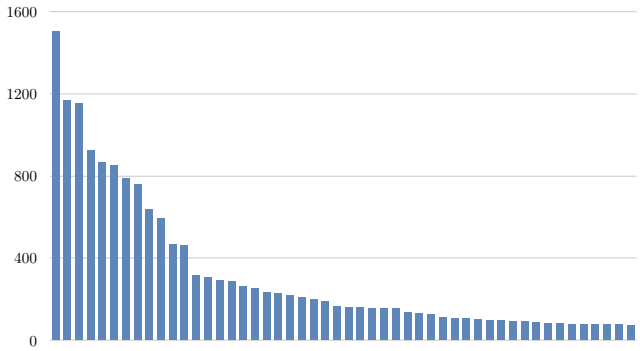

Figure 4. The 50 most common errors revealed by the fuzz test followed by a long tail of 1,185 other errors (not shown). Each bar represents the frequency of one output vector. We assumed that if two snapshots produced the same output vector, they contained the same logical errors.

Rank	Snapshots	Logical Error
1	1,502	Did not modify returned variables
2	1,171	Incorrectly determined tens digit for numbers larger than 100
3	1,156	Incorrectly determined tens digit for single-digit numbers
4	925	Did not switch player after turns
5	867	Classified 2 as a nonprime number
6	854	Stopped after two turns
7	789	Did not apply a rule in some cases
8	759	Did not implement a rule
9	637	Stopped after one turn
10	594	Gave all points to one player

Figure 5. The 10 most common errors revealed by the fuzz test.

the ten most common errors. However, a long tail of student errors account for a small but significant number of snapshots. In total, the fuzz test discovered 1,235 distinct errors. This large number of unique errors is consistent with previous work in applying fuzz tests to evaluate open-ended programming questions: A system that used a fixed error model to identify the differences between student and reference implementations could only find a difference conforming to the model in 65% of cases, indicating a large space of possible logical errors [19].

In a massive course, fuzz tests can identify groups of students that have the same logical error, even when that error is quite rare. Finding these groups can be helpful to the students by connecting them with each other and helpful to instructors who can attempt to assist all of them at once.

Infrastructure

To analyze data from fuzz tests at this scale, we created an autograding infrastructure that dynamically adjusts capacity based on demand. During our use, it was capable of testing more than 5,000 submissions per minute (each running a fuzz test with 1,000 inputs). This throughput can be used to generate quick analysis of student progress for instructors.

1. **Results Database.** We use a cluster of three virtual machines running *Redis* (a key-value datastore) to store the results.
2. **Job Queue.** Each of the 173,468 student code snapshots is inserted into a job queue that is also backed by the Redis cluster.
3. **Grading Containers.** All of the grading happens inside of a *Docker* container that is sandboxed from the host machine and has all of the dependencies to grade the project pre-installed.
4. **Grading Servers.** Virtual Machines run 10 workers that each grab snapshots from the job queue, launch grading containers, and communicate the results to the results database. Grading is a CPU intensive operation which requires relatively powerful virtual machines. We reduce costs by 70% through the usage of Preemptible Instances that may be terminated at any time. This an acceptable tradeoff because any lost work is easily recomputable.
5. **Autoscaling.** Google Compute Engine was configured to automatically adjust the number of grading servers based on average CPU load of the existing servers. This results in the system rapidly creating many grading servers, each of which results in 10 new workers. Once grading was completed, the machines were automatically terminated to reduce costs.

The job took about three hours to complete and utilized approximately fifty distinct virtual machines. The total cost of the grading servers was $7.53, showing that fuzz testing is logistically feasible for grading in massive classes, despite the distributed infrastructure required for fast analysis.

Results

After analyzing the in-progress work of students, we discovered that 656 students (48.4% of students who attempted the question) were able to write code that passed the 9 manually constructed tests, but still contained a logical error identified by the fuzz test. Without this feedback from the fuzz test, half of the course would have progressed with incorrect implementations.

Figure 6 shows the distribution of attempts made by students in this group before finally passing the provided fuzz test. The median time for a student to resolve an error revealed in a fuzz test was 3.4 hours. This time stands in stark contrast to the median time of 18 minutes to resolve errors revealed in targeted tests. We attribute this difference to both the comprehensiveness of fuzz tests at finding errors (seen in Figure 4) as well as the obfuscation of the fuzz test output provided to students.

In Figure 7, we analyze how additional fuzz test inputs help detect additional bugs. First, all snapshots are tested with 1,000 inputs. Snapshots that produce a different output vector than the reference solution are labeled as incorrect. Then, we determine how many incorrect snapshots match the reference solution for the first k fuzz tests inputs. When only one fuzz test input is considered ($k = 1$) against 22,375 snapshots with an incorrect implementation, only 16,010 (71.6%)

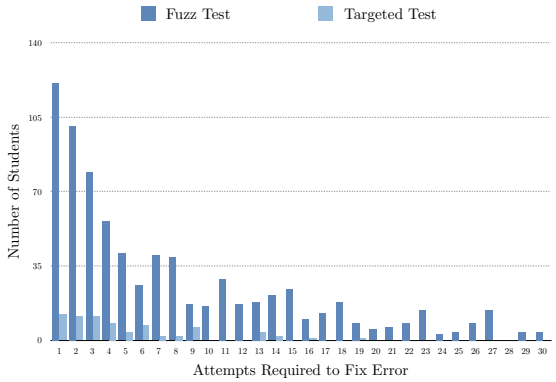

Figure 6. Distribution of attempts required by students to resolve errors for each test type

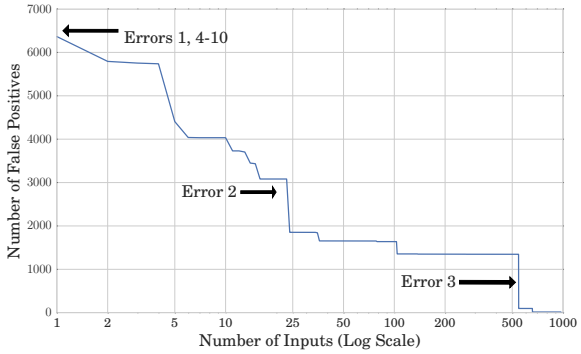

Figure 7. As the number of fuzz test inputs increases, more snapshots are revealed to have errors. The sharp drops are labeled with the error from Figure 5 that was caught by that test input.

are identified as containing a logical error. The fuzz test that we provided to students only contained 100 inputs. Figure 7 shows that this was insufficient to catch all student errors. The first 100 fuzz tests only caught 92.7% of snapshots with errors. Therefore, some students were given full credit without a correct solution.

In addition, Figure 7 shows that some fuzz test inputs are more useful for detecting errors than others. For example, inputs 105 to 545 were unable to detect many more errors than the first 104 inputs alone. However, input 546 tested a rare scenario where students were asked to find the tens digit for a three-digit number, rather than a two-digit number. Because the fuzz test inputs are randomly sampled from the space of inputs, these uncommon scenarios may not be tested until a large number of inputs are randomly generated.

CONCLUSION

Fuzz testing student submissions by comparing their output on random inputs to the output of a reference implementation is both easy to implement and effective for comprehensive testing. When combined with targeted tests that are manually constructed to identify common issues, test suites can enhance the instructional value of programming projects by guiding students to correct implementations. Our empirical

evaluation exposed some surprising results: fuzz testing identified errors that were missed by hand-crafted tests in almost half of student implementations. A small number of random inputs was insufficient for identifying errors; many submissions matched the reference for hundreds of random inputs before diverging. The number of unique errors that arose in the one programming problem we studied was enormous: 1,235 unique errors among 1,355 students. With such a variety of possible issues that may arise in programming problems in a large course, fuzz testing should be considered an essential tool for providing automated feedback at scale.

Acknowledgements

This work was supported by a Google Research Cloud Credits Award.

REFERENCES

1. Basu, S., Wu, A., Hou, B., and DeNero, J. Problems before solutions: Automated problem clarification at scale. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, ACM (2015), 205–213.
2. Becker, S., Abdelnur, H., Obes, J. L., State, R., and Festor, O. Improving fuzz testing using game theory. In *Network and System Security (NSS), 2010 4th International Conference on*, IEEE (2010), 263–268.
3. Bounimova, E., Godefroid, P., and Molnar, D. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, IEEE Press (Piscataway, NJ, USA, 2013), 122–131.
4. DeNero, J., and Klein, D. Teaching introductory artificial intelligence with pac-man. In *Proceedings of the Symposium on Educational Advances in Artificial Intelligence* (2010).
5. Dormann, W., and Plakosh, D. Vulnerability detection in activex controls through automated fuzz testing. *Unpublished working paper* (2008).
6. Edwards, S. H., and Perez-Quinones, M. A. Web-cat: automatically grading programming assignments. In *ACM SIGCSE Bulletin*, vol. 40, ACM (2008), 328–328.
7. Forrester, J. E., and Miller, B. P. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, USENIX Association (Berkeley, CA, USA, 2000), 6–6.
8. Glassman, E. L., Singh, R., and Miller, R. C. Feature engineering for clustering student solutions. In *Proceedings of the First ACM Conference on Learning @ Scale Conference*, L@S '14, ACM (New York, NY, USA, 2014), 171–172.
9. Godefroid, P., Kiezun, A., and Levin, M. Y. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, ACM (New York, NY, USA, 2008), 206–215.
10. Godefroid, P., Levin, M. Y., Molnar, D. A., et al. Automated whitebox fuzz testing. In *NDSS*, vol. 8 (2008), 151–166.
11. Hamlet, D., and Taylor, R. Partition testing does not inspire confidence (program testing). *IEEE Trans. Softw. Eng. 16*, 12 (Dec. 1990), 1402–1411.
12. Hollingsworth, J. Automatic graders for programming classes. *Commun. ACM 3*, 10 (Oct. 1960), 528–529.
13. Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, ACM (New York, NY, USA, 2010), 86–93.
14. Miller, B. P., Cooksey, G., and Moore, F. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, ACM (New York, NY, USA, 2006), 46–54.
15. Miller, B. P., Fredriksen, L., and So, B. An empirical study of the reliability of unix utilities. *Commun. ACM 33*, 12 (Dec. 1990), 32–44.
16. Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, IEEE Computer Society (Washington, DC, USA, 2007), 75–84.
17. Parlante, N., Zelenski, J., Dodds, Z., Vonnegut, W., Malan, D. J., Murtagh, T. P., Neller, T. W., Sherriff, M., and Zingaro, D. Nifty assignments. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, ACM (New York, NY, USA, 2010), 478–479.
18. Piech, C., Sahami, M., Huang, J., and Guibas, L. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, L@S '15, ACM (New York, NY, USA, 2015), 195–204.
19. Singh, R., Gulwani, S., and Solar-Lezama, A. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, ACM (New York, NY, USA, 2013), 15–26.
20. Tsankov, P., Dashti, M. T., and Basin, D. Secfuzz: Fuzz-testing security protocols. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST '12, IEEE Press (Piscataway, NJ, USA, 2012), 1–7.