Privately Executable Examples

Viraj Kumar
virajkumar1@acm.org
Indian Institute of Science
Bengaluru, Karnataka, India

ABSTRACT

Executable examples enable students to check their comprehension of programming problems at any time before, during, or after implementation. Students express their understanding of a problem by specifying input-output pairs, and they receive immediate feedback on their understanding when these pairs are executed against correct and buggy solutions specified by the instructor. These solutions are typically executed on a server, and we find evidence that some students in an introductory programming course are wary of revealing their fragile problem comprehension in exchange for feedback. We propose a student-side mechanism that enables students to receive the same feedback privately. We also conduct a study to investigate differences in student ability to create valid and thorough examples using the server-side or the student-side feedback mechanisms.

CCS CONCEPTS

• **Social and professional topics** → **Student assessment**; **CS1**.

KEYWORDS

CS1, automated assessment, privacy, executable examples

ACM Reference Format:
Viraj Kumar. 2022. Privately Executable Examples. In *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol 1 (ITiCSE 2022), July 8–13, 2022, Dublin, Ireland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3502718.3524800

1 INTRODUCTION

When students are confronted with a programming problem, they must first comprehend its natural-language problem description. Students who fail to correctly comprehend the problem may spend considerable time solving a different problem [12, 15, 22]. Natural-language descriptions may be *genuinely* or *apparently* ambiguous, particularly in linguistically diverse contexts where neither students nor instructors are perfectly fluent in the language of instruction [1]. Wrenn and Krishnamurthi [23] note that students cannot check their comprehension of the problem by writing test cases: "if a student develops both their tests and implementation with the same misunderstanding of a problem, running those tests against their implementation will not reveal their misunderstanding". Therefore, they propose a mechanism – *executable examples* – that allows students to express their problem comprehension as a suite of input-output examples. A student submits their suite to a server, which provides immediate feedback on their problem comprehension by executing the suite against instructor-specified solutions. Students can refine their submissions, and the server logs the student's username and suite contents for each submission [23].

1.1 Motivating Question

The collection of such data raises privacy concerns [13] and triggers the motivating question for this study: Is there a *legitimate* educational need to log data traces for students' example suites? We believe this question is worth examining carefully, because although a server-side solution can eliminate privacy concerns by simply not logging data, there is a clear incentive to gather student-specific data traces: to support *learning analytics*. This is defined as "the measurement, collection, analysis and reporting of data about learners and their contexts, for purposes of understanding and optimizing learning and the environments in which it occurs" [17].

We do not seek to resolve the legitimacy question. In fact, we believe that a universal answer does not exist (Section 6.4). For situations where the answer is "no", we propose a student-side command-line mechanism that provides students immediate feedback on their suites by executing them against instructor-specified solutions while *guaranteeing privacy* (i.e., students reveal only their final suites). We achieve this by *obfuscating* instructor-specified solutions so that it is impractical for CS1 students to reverse-engineer them. Our two research questions are: In comparison to the server-side mechanism . . .

- **RQ1**: . . . do students find it harder to evaluate example suites using the student-side mechanism?
- **RQ2**: . . . is there a difference in the quality of final suites created by students using the student-side mechanism?

Following Wrenn and Krishnamurthi [23], we use two measures of suite quality:

- *Validity*: The proportion of examples in the suite that are consistent with the instructor-specified solution.
- *Thoroughness*: The proportion of buggy implementations for which at least one example in the suite is inconsistent.

2 RELATED WORK

Pardo and Siemens note that our growing capability to log and analyze the steps that students perform during a task makes it "possible to deploy new assessment techniques that measure more accurately the right [student] achievements", which "also raises the issue of privacy" [13]. It is difficult to define privacy, since there are differences in expectations of privacy between individuals [18],

ITiCSE '22, July 08–13, 2022, Dublin, Ireland

ACM ISBN 978-1-4503-9201-3/22/07…$15.00
https://doi.org/10.1145/3502718.3524800

as well as gender differences and differences between cultures [3]. In India, where this study was conducted, early work found less awareness of privacy issues among Indian students than their American counterparts [9], and a subsequent study found that students' immediate concerns for privacy remain low [10]. Even when students express concerns about the privacy of their data, their trust in their institution or their perception of a power imbalance may lead them to surrender their data [2, 20]. It is therefore imperative for instructors to have *legitimate* reasons for collecting student data. Drachsler and Greller have proposed an eight-point checklist for "managers and decision makers planning the implementation of Learning Analytics solutions" [7]. The third point on this checklist examines questions of legitimacy, including: "What data sources [do] you already have – aren't they enough?"

Students in our CS1 course write code using Prutor [6], one of many online programming platforms that take frequent snapshots of each student's code, capturing not just the *outcome* but the *process* of code development. Over the past decade [4, 14], a rich body of research has provided a compelling answer to the legitimacy question for gathering this type of process-level data instead of just the outcomes (see [21] for a recent survey). While frameworks for assisting novice programmers such as the Design Recipe from *How to Design Programs* [8] and the PCDIT framework [11] encourage students to develop examples (or *cases* in the latter framework), we are unaware of any argument that addresses the legitimacy question for gathering process-level data on example suite development. Wrenn and Krishnamurthi [23] demonstrate that the immediate feedback provided by their tool can help students improve their understanding of the problem at hand. We propose and evaluate a mechanism for the same purpose that does not reveal any process-level data.

3 STUDENT-SIDE EXAMPLE EVALUATION

To evaluate student examples on the student's own machine, we obfuscate the instructor's solutions (written in C). For this study, we used the Tigress C obfuscator [19] which can transform the code in a variety of sophisticated ways. We find that one of the basic Tigress transformations – Virtualize [1] – produces code that is sufficiently obfuscated to make it impractical for CS1 students to extract the instructor solutions in human-readable form. The resulting C code can be compiled and executed by the student from the command line like any other C program.

There are two limitations to this approach. First, the resulting C code is often so large that students cannot execute it using free online programming environments. [2] Several students rely on such platforms when they cannot access on-campus computing facilities, such as during the prevailing lockdown. Second, similarly adequate obfuscators may not be available for other programming languages.

4 STUDY DESIGN

This in-person study was conducted in Fall 2021, with students in an accelerated 6-week CS1 course (using the C programming language) at a premier research institution in India. There are two lecture hours and one tutorial hour per week. Further, at the time of enrollment, students are randomly assigned to one of four batches. Each batch attends a weekly 3-hour in-person lab session on a particular day (Monday to Thursday), during which students can seek the assistance of the course instructor as well as two Teaching Assistants.

We conducted the study during Week 5, and 97 students out 108 gave consent for their anonymized submissions to be used in this research study. Students solved two programming problems (described below) and completed three graded tasks. Task 1 asked students to create an example suite for one of the given problems using the server-side mechanism (Prutor). Students were free to evaluate their suites any number of times using a familiar sequence of two mouse-clicks. Although the server logged their username and attempts, this process-level data was *not* examined in this study. For Task 2, students created an example suite for the other problem using the student-side mechanism (the command line). Once again, students were free to evaluate their suites as often as they wished, this time using a two command-line steps they had exposure to in prior weeks: compiling and running the code. In the two weeks prior to this study, four similar problems were discussed in lectures and tutorials. For each problem, an initial suite of valid examples was presented, and the thoroughness of the suite was improved during the discussion. For Tasks 1 and 2, students were informed that their *final* example suites would be graded based on validity and thoroughness.

Students were split into two groups A (including 47 consent-giving students) and B (including 50 consent-giving students), and the problems for Tasks 1 and 2 were assigned as shown in Table 1.

Table 1: Task → Problem assignment for Groups A and B

Group	Days	Task 1	Task 2
A (47)	Mon, Wed	Problem 1 (server)	Problem 2 (student)
B (50)	Tue, Thur	Problem 2 (server)	Problem 1 (student)

4.1 Programming Problems

Each of the programming problems asked students to write a function that processes a given array of integers. We anticipated that students would find it substantially easier to create examples for Problem 1. To explain why, we reproduce the problem specifications, including examples given to students.

4.1.1 Problem 1: Sorted. An array of length `n` is sorted in *ascending* order if: `a[0]` ≤ `a[1]` ≤ ... ≤ `a[n-1]`. Similarly, the array is sorted in *descending* order if: `a[0]` ≥ `a[1]` ≥ ... ≥ `a[n-1]`. Consider a function with the following prototype:

```
int sorted(unsigned int n, const int a[n]);
```

This function should return: 2 if the array is sorted in *both* ascending and descending order, 1 if the array is sorted *only* in ascending order, −1 if the array is sorted *only* in descending order, and 0 otherwise.

[1] https://tigress.wtf/virtualize.html

[2] For instance, the instructor solution for Problem 1 (Section 4.1.1) has 21 lines of code and 457 characters (excluding comments), whereas the obfuscated solution has 277 lines and 19,449 characters.

Examples: If `a[3] = {1, 2, x}` then the function should return: 1 if $x \geq 2$ and 0 if $x < 2$.

Starter code: Students were given starter code for expressing the examples above in C, and were asked to add their own examples.

```
int a1[] = {1, 2, 2};
int a2[] = {1, 2, 1};
TEST test[] = {
  {.label = "ex1", .n = 3, .a = a1, .output = 1},
  {.label = "ex2", .n = 3, .a = a2, .output = 0}
};
```

4.1.2 Problem 2: Permutation. Suppose the elements of an array `a[]` of length `n` are a permutation of $0, 1, \ldots, n-1$ for some positive integer `n`. If so, the inverse permutation is an array `a_inv[]` of length `n` such that: `a[a_inv[i]] = a_inv[a[i]] = i` for all $0 \leq i < n$. Consider a function with the following prototype:

```
int permutation(unsigned int n, const int a[n], int a_inv[n]);
```

This function should return: 1 if the array `a[]` is a permutation of $0, 1, \ldots, n-1$ for some positive integer `n` (if so, the function should also initialize `a_inv[]` as the inverse of permutation of `a[]`), and 0 otherwise.

Example: If `a[3] = {x, 2, 0}` then the function should return 0 if `x` is not equal to 1. Otherwise, it should return 1 and `a_inv[3]` must be the array `{2, 0, 1}`.

Starter code: Students were given starter code for expressing the examples above in C, and were asked to add their own examples.

```
int a1[] = {1, 2, 0};
int a1_inv[] = {2, 0, 1};
int a2[] = {2, 2, 0};
TEST test[] = {
  {.label = "ex1", .n = 3, .a = a1, .a_inv = a1_inv,
    .output = 1},
  {.label = "ex2", .n = 3, .a = a2, .output = 0}
};
```

4.1.3 Comparison of Problems. We anticipated that students would find it easier to create examples for Problem 1 for two reasons. First, we expected students to be more familiar with the idea of a sorted array than the idea of a permutation and its inverse. Second, it is easy to copy-paste a given example for Problem 1 and modify the array to be sorted or unsorted. For Problem 2, in contrast, the syntax depends on whether the input array is a permutation (in which case the inverse must also be specified) or not (the inverse need not be specified). Calculating the inverse is also burdensome, and we observed several students calculating inverses by hand.

For both Tasks 1 and 2, students executed their examples against a correct solution only. After completing Tasks 1 and 2, all students completed a common Task 3: implementing solutions for these two problems using Prutor. To help students test their code, we provided test cases that students could execute via Prutor. The two examples provided in the starter code for each Problem were included as *visible* test cases (i.e., students could see each of the inputs, the corresponding expected output, and their own function's corresponding outputs). For each Problem, we also provided several *hidden* test cases. Prutor does not allow students to see the inputs or expected outputs for these tests, but it reports the number of hidden test cases that the student's code passes. Students were informed that their implementations would be graded based on a more thorough test suite than the visible and hidden tests provided.

As students debugged their code using these instructor-specified test-cases, they were encouraged to amend, extend, or otherwise modify their example suites for Tasks 1 and 2 before finally submitting their work. However, these modifications were not mandatory.

4.2 Student Survey

To understand how students perceived the two Tasks and the two evaluation mechanisms (relevant for **RQ1**), we asked students to complete an *optional* survey after completing Tasks 1 and 2. Our study design ensures that examples for both Problems were created using the server-side mechanism by about half the students, and using the student-side mechanism by the rest (Table 1). We can thus examine whether the mechanism had any influence on students' perceived difficulty in creating examples.

Out of the 47 students in Group *A* who gave consent, 24 filled this optional survey. Similarly, 28 out of the 50 students in Group *B* who gave consent filled out this survey.

The survey asked the following three questions:

(1) For which Task was it easier to create examples? (Multiple-choice, options shown in Figure 1)
(2) For which Task was it easier to evaluate examples? (Multiple-choice, options shown in Figure 2)
(3) For examples created using the command line, your instructor can see only your final submission. For examples created using Prutor, your instructor can additionally see all your past submissions. How do you feel about this? (Open ended)

5 RESULTS

In this Section, we perform several hypothesis tests to compare two proportions. In each case, our null hypothesis is that the two proportions are equal, and we calculate the χ_1^2 statistic and the corresponding p-value using the test by Campbell [5] and Richardson [16]. We first examine the results from the optional student survey.

5.1 Survey Results

5.1.1 Question 1. Figure 1 shows the percentage of responses in each group for the five given options related to *creating* examples. None of the between-group differences are statistically significant, suggesting that the mechanism for *evaluating* examples did not strongly influence students' perception of the relative difficulty in *creating* examples. Most students found it easier to create examples for Problem 1 (Sorted), in line with our expectations (Section 4.1.3).

5.1.2 Question 2. Figure 2 shows the percentage of responses in each group for the five given options related to *evaluating* examples. The highest percentage of respondents in both groups (more than 40%) find that the two evaluation mechanisms are "about the same". However, among those who thought it was slightly or much easier to evaluate examples for Problem 1 (Sorting), there were

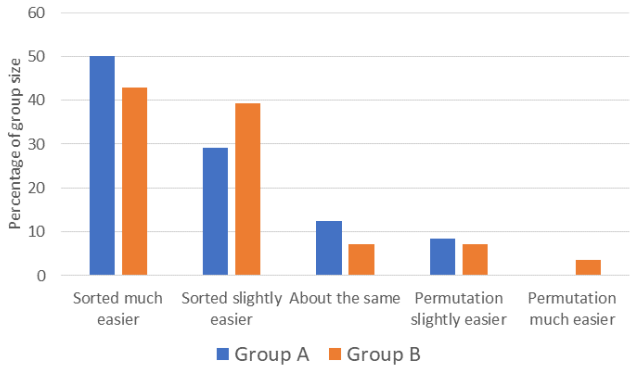

Figure 1: Responses to the question "For which Task was it easier to create examples?"

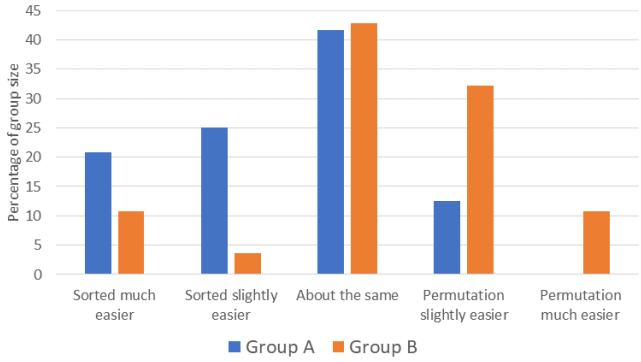

Figure 2: Responses to the question "For which Task was it easier to evaluate examples?"

a significantly higher percentage of respondents from Group A (45.8%, server-side mechanism) than from Group B (14.3%, student-side mechanism); hypothesis test for difference of proportions: $\chi^2_1 = 6.146$, $p = 0.0132$. Similarly, among those who that it was slightly or much easier to evaluate examples for Problem 2 (Permutation), there were a significantly higher percentage of respondents from Group B (42.9%, server-side mechanism) than from Group A (12.5%, student-side mechanism); hypothesis test for difference of proportions: $\chi^2_1 = 5.691$, $p = 0.0171$.

5.1.3 Question 3. The majority of students (19 out of 24 in Group A, 18 out of 28 in Group B) gave a neutral answer to the open-ended survey question. Typical responses include "I'm ok either way", "doesn't matter", and "Fine". Only three students (2 from Group A, 1 from Group B) expressed a level of comfort. One of these opinions explicitly refers to the recording of code (not examples): "Its completely OK because one uses past attempts to figure out where the error was in one's code, and the professor sees one's process of code making". Another such opinion came with a caveat: "I feel that it is really helpful if the Professor could see my past attempts and it is an excellent idea, where the instructor could keep track of a student's progress. But this function is futile unless the instructor has time enough to check each individual student's progress and he does cares [sic] enough."

The remaining 12 opinions expressed a degree of discomfort. Some responses include "I am slightly uncomfortable that Professor can see my past attempts", "Should ask us first", and "its sorta creepy". Two of these responses expressed concern because prior attempts might be graded, including: "For some of the previous test cases, I input buggy test cases which did not give a correct answer, so I would be glad if my proffesor [sic] only focuses on the final test cases." (Several students used the terms "test cases" and "examples" interchangeably.) We note that 9 out of these 12 opinions were from students in Group B, who used the server-side mechanism to create examples for the more challenging Problem 2 (see Figure 1).

Table 2: Between-group differences in valid suites by Problem

Problem 1			Problem 2		
Group	Size	Valid	Group	Size	Valid
A	45	42	A	47	38
B	49	37	B	50	48
$\chi^2_1 = 5.497$, $p = 0.019$			$\chi^2_1 = 5.473$, $p = 0.0193$		

5.2 Quality of Examples

Next, we examine the quality of example suites created by students using the two previously stated metrics: validity and thoroughness.

5.2.1 Validity. In our study, example suites were invalid for one of two reasons: either the code containing the examples failed to compile, or it caused a run-time error during evaluation. It is clear from Table 2 that for Problem 1, a significantly higher proportion of students in Group A ($\frac{42}{45} = 93.3\%$) were able to create valid suites, whereas the corresponding proportion for Group B is just $\frac{37}{49} = 75.6\%$. This difference in proportions is statistically significant ($p = 0.019$). For Problem 2, a significantly higher proportion (96%) of students in Group B were able to create valid suites than students in Group A (80.9%).

5.2.2 Thoroughness. For each Problem in our study, students' examples were evaluated *only* against the instructor-specified reference solution for that Problem (for both the server-side and student-side mechanisms). As students worked on Task 3 (implementing solutions for Problems 1 and 2), some of their buggy attempts were identified by instructor-specified test cases (visible and hidden). Students were encouraged (but not required) to execute these buggy solutions against their own examples and augment these examples if they failed to detect the bugs. Thus, the study provided students with an opportunity to improve the thoroughness of their suites.

Of the 94 submissions received for Task 3, three failed to compile. Out of the remaining 91 submissions, we identified 33 buggy implementations for Problem 1 (Sorted), and 22 buggy implementations for Problem 2 (Permutations) using a more thorough set of tests than the visible and hidden tests provided to students. We manually inspected all solutions that passed these tests to convince ourselves that they were correct implementations.

For this study, we use these student-created buggy implementations to assess the thoroughness of student-created example suites.

As a benchmark, the thoroughness of the visible test cases was 15.2% for Problem 1 and 31.8% for Problem 2. In contrast, the thoroughness of the combined set of visible and hidden test cases was 51.5% for Problem 1 and 63.6% for Problem 2.

For Problem 1, the average thoroughness of students' suites was nearly identical for Group A (51.80%) and Group B (51.67%). In fact, for 32 out of the 33 buggy implementations for Problem 1, the percentage of student suites that can detect a bug is nearly identical for both groups. Figure 3 compares the performance for both groups on these 33 implementations, ordered along the x-axis from "obviously buggy" (all student suites can detect a bug) to "subtly buggy" (very few student suites can detect a bug). The only exception is Implementation 13 in this ordering, which is shown in Figure 4. The proportion of student suites that can detect bugs in this implementation differs significantly from Group A (57.14%) to Group B (81.08%); $\chi_1^2 = 5.145$, $p = 0.0233$.

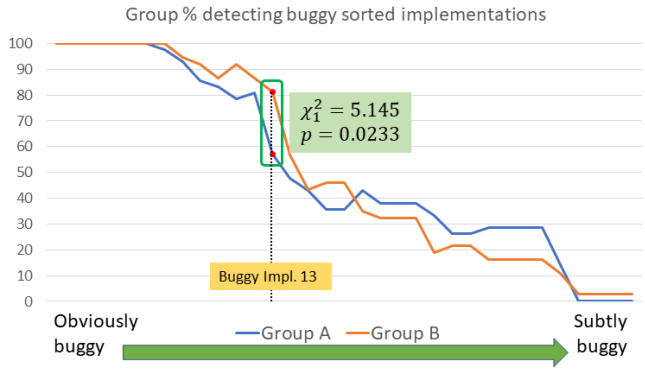

Figure 3: Comparison of groups on each of 33 buggy implementations for Problem 1 (Sorted), ordered from "obviously buggy" to "subtly buggy"

```
1  int sorted(unsigned int n, const int a[n]) {
2    if (n <= 1) { return 2; }
3    for (int i = 0; i <= n-2; i++) {
4      if (a[i] == a[i+1]) {
5        return 2;
6      } else if (a[i+1] >= a[i] && a[i+2] >= a[i+1]) {
7        return 1;
8      } else if (a[i] >= a[i+1]) {
9        return -1;
10     } else {
11       return 0;
12     }
13   }
14 }
```

Figure 4: Buggy Implementation 13 for Problem 1

For Problem 2, the average thoroughness of students' suites was again nearly identical for Group A (59.81%) and Group B (59.19%). In fact, for all 22 buggy implementations for Problem 2, the percentage of student suites that can detect a bug is nearly identical for both groups (Figure 5): none of these differences is statistically significant.

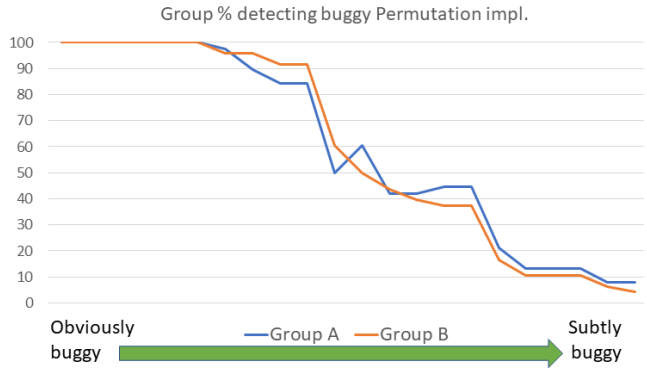

Figure 5: Comparison of groups on each of 22 buggy implementations of Problem 2 (Permutation), ordered from "obviously buggy" to "subtly buggy"

6 DISCUSSION

6.1 Limitations

We acknowledge four main limitations of our study due to which our findings may not generalize well. First, there is a significant gender bias among our sample of 97 students: only 14 self-identify as female. In India, such a gender ratio is typical in CS1 courses at top-tier institutions, but it is atypical at the majority of institutions offering CS1. Second, our student-side mechanism relies on effectively obfuscating instructor solutions, which we have demonstrated only for C programs. Third, our study is based on only two problems, both involving array traversal. Fourth, all but 4 responses for the optional survey were recorded during the 3-hour lab session. Thus, students who struggled with the problems may be underrepresented among survey respondents.

6.2 RQ1

In comparison to the server-side mechanism, do students find it harder to evaluate example suites using the student-side mechanism? Based on our findings in Section 5.1.2 and Section 5.2.1, we believe that the answer to **RQ1** is "yes" i.e., a substantial number of students appear to find it harder to evaluate example suites with the student-side mechanism.

We identify two possible and related explanations for these findings: discomfort with the command-line interface in general and, more specifically, discomfort with fixing syntax errors using the compiler from the command line. For the majority of students, this course provides their first exposure to the command line. In the four weeks preceding this study, students were exposed to the command-line interface during Weeks 1, 2, and 3, and to Prutor's GUI during Weeks 3 and 4. Despite this exposure, we observed some students struggling with the command line during this study. We suspect that most of the students who found both evaluation mechanisms similar (Section 5.1.2) were comfortable with the command-line interface, whereas most of the other students found the GUI-based server-side mechanism far easier to handle. To eliminate this potential source of difference, we are working on a GUI for the student-side mechanism.

We interpret the results in Section 5.2.1 as follows. For the server-side mechanism, students wrote their examples using Prutor. This provides a modern IDE with standard features such as line-numbering, auto-indentation and highlighting to indicate syntax errors. In contrast, many students using the student-side mechanism edited their code using a rudimentary text editor (`gedit`) which provides only some of these features. This may have contributed to the substantially higher number of syntactically invalid suites created by these students. To eliminate this potential source of difference in future studies, we will familiarize students with a modern IDE accessible from the command line.

6.3 RQ2

In comparison to the server-side mechanism, is there a difference in the quality of final suites created by students using the student-side mechanism? The findings above suggest that the answer to **RQ2** is "yes" when the metric for the quality of suites is validity. On the other hand, when the metric is thoroughness, our findings in Section 5.2.2 strongly suggest that the answer to **RQ2** is "no": for every buggy implementation except one, there is no statistically significant difference in the proportion of example suites created using the server-side or the student-side mechanism that is capable of detecting a bug.

We tried to seek an explanation for the differing performance on the single exception (Buggy Implementation 13 for Problem 1, Figure 4). The implementation is correct when the array length $n \leq 1$ (due to the test on Line 2). When the array length $n = 2$, there may be an out-of-bounds array access on Line 6. For a C program, the run-time behavior may depend on the value of other (memory adjacent) variables. Hence, we checked whether any of the observed difference between Group *A* and Group *B* suites on this implementation is due to examples with $n = 2$. We found no such differences. It just turns out that a greater proportion of students from Group *B* had created examples with array length $n \geq 3$ (using the student-side mechanism) that detected the error. One such input was a hidden test case provided for Task 3: `a[3] = {-1, -2, -1}`.

6.4 The Legitimacy Question

In our view, gathering process-level data for student-created examples is legitimate only if instructors can use this data to help students either directly (as noted by one student in Section 5.1.3), or indirectly by "understanding and optimizing learning and the environments in which it occurs" [17]. During this study, students received help from the instructor and TAs only when they requested help. All these requests fell into two categories: help with understanding a technical nuance of the problem specification, and help with understanding the natural-language description itself. We illustrate each type of request with an example, and discuss the potential for process-level data to assist instructors in fielding such help requests.

For Problem 1, the most commonly asked question was: "What should the `sorted()` function return when the array has length $n \leq 1$?" Students who asked this question were unfamiliar with the concept of *vacuous truth* and were therefore unable to understand why such an array is sorted in *both* ascending and descending order according to the specification of Problem 1 (Section 4.1.1). Each of these students had created an example for such an input by specifying the expected output as 0, and both mechanisms provided feedback on this error. Some of these students deduced that the correct output is 2 through a process of trial-and-error, but they wanted to understand why this was the correct output. An analysis of process-level data could help the instructor recognize that a specific technical point is a common source of confusion, and therefore suggest a way to help (e.g., by providing a suitable learning resource).

For Problem 2, several students were unfamiliar with the term "inverse" and struggled to parse the given definition. Some of these students interpreted this as "reverse", even though the stated definition of inverse (Section 4.1.2) was clearly not the definition of reverse, and despite the example provided at the end of the problem specification in which the inverse of `{1, 2, 0}` is *not* the reverse `{0, 2, 1}`. Our initial response to their query was always similar to "Read the definition of inverse carefully". However, some students required further assistance, including an explanation of the problem in a language they were more comfortable with. We do not see the utility of process-level data in assisting such students, except possibly in identifying them. However, such identification could raise ethical concerns.

We therefore believe that there is no universal answer to the legitimacy question we have posed.

7 CONCLUSIONS

We have presented a mechanism that allows students to *privately* self-assess their comprehension of programming problems by creating input-output examples, without revealing anything to the instructor except possibly their final suite of examples. Our student-side mechanism presently requires familiarity with the command-line interface, and therefore it may be slightly harder for students to create syntactically valid examples and execute them than with the GUI for our server-side mechanism. (We plan to eliminate this difference in future studies.) On the other hand, we find no significant difference in the thoroughness of example suites created by students using the two mechanisms. There may be scenarios where it is legitimate to log process-level data as students create examples. In all other cases, our student-side mechanism provides the benefits of immediate feedback while guaranteeing students' privacy.

REFERENCES

[1] GS Adithi, Akshay Adiga, K Pavithra, Prajwal P Vasisht, and Viraj Kumar. 2015. Secure, Offline Feedback to Convey Instructor Intent. In *2015 IEEE Seventh International Conference on Technology for Education (T4E)*. IEEE, 105–108.

[2] Susanne Barth and Menno DT De Jong. 2017. The privacy paradox–Investigating discrepancies between expressed privacy concerns and actual online behavior–A systematic literature review. *Telematics and informatics* 34, 7 (2017), 1038–1058.

[3] Steven Bellman, Eric J Johnson, Stephen J Kobrin, and Gerald L Lohse. 2004. International differences in information privacy concerns: A global survey of consumers. *The Information Society* 20, 5 (2004), 313–324.

[4] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. 2014. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences* 23, 4 (2014), 561–599.

[5] Ian Campbell. 2007. Chi-squared and Fisher–Irwin tests of two-by-two tables with small sample recommendations. *Statistics in medicine* 26, 19 (2007), 3661–3675.

[6] Rajdeep Das, Umair Z Ahmed, Amey Karkare, and Sumit Gulwani. 2016. Prutor: A system for tutoring CS1 and collecting student programs for analysis. *arXiv preprint arXiv:1608.03828* (2016).

[7] Hendrik Drachsler and Wolfgang Greller. 2016. Privacy and Analytics: It's a DELICATE Issue a Checklist for Trusted Learning Analytics. In *Proceedings of the*

Sixth International Conference on Learning Analytics amp; Knowledge (Edinburgh, United Kingdom) *(LAK '16)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/2883851.2883893

[8] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to design programs: an introduction to programming and computing*. MIT Press.

[9] Ponnurangam Kumaraguru and Lorrie Cranor. 2005. Privacy in India: Attitudes and awareness. In *International workshop on privacy enhancing technologies*. Springer, 243–258.

[10] Ponnurangam Kumaraguru and Niharika Sachdeva. 2012. Privacy in India: Attitudes and awareness v 2.0. *Available at SSRN 2188749* (2012).

[11] Oka Kurniawan, Cyrille Jégourel, Norman Tiong Seng Lee, Matthieu De Mari, and Christopher M Poskitt. 2021. Steps Before Syntax: Helping Novice Programmers Solve Problems using the PCDIT Framework. *arXiv preprint arXiv:2109.08896* (2021).

[12] Dastyni Loksa and Amy J. Ko. 2016. The Role of Self-Regulation in Programming Problem Solving Process and Success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 83–91. https://doi.org/10.1145/2960310.2960334

[13] Abelardo Pardo and George Siemens. 2014. Ethical and privacy principles for learning analytics. *British Journal of Educational Technology* 45, 3 (2014), 438–450.

[14] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 153–160.

[15] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/3230977.3230981

[16] JT Richardson. 2011. The analysis of 2× 2 contingency tables–yet again. *Statistics in medicine* 30, 8 (2011), 890–author.

[17] George Siemens and Phil Long. 2011. Penetrating the fog: Analytics in learning and education. *EDUCAUSE review* 46, 5 (2011), 30.

[18] Daniel J Solove. 2008. Understanding privacy. (2008).

[19] Clark Taylor and Christian Collberg. 2016. A tool for teaching reverse engineering. USENIX Workshop on Advances in Security Education, ASE 2016, co-located with the 25th USENIX Security Symposium. All rights reserved.; 2016 USENIX Workshop on Advances in Security Education, ASE 2016, co-located with the 25th USENIX Security Symposium.

[20] Yi-Shan Tsai, Alexander Whitelock-Wainwright, and Dragan Gašević. 2020. *The Privacy Paradox and Its Implications for Learning Analytics*. Association for Computing Machinery, New York, NY, USA, 230–239. https://doi.org/10.1145/3375462.3375536

[21] Maureen M Villamor. 2020. A review on process-oriented approaches for analyzing novice solutions to programming problems. *Research and Practice in Technology Enhanced Learning* 15, 1 (2020), 1–23.

[22] Jacqueline Whalley and Nadia Kasto. 2014. A Qualitative Think-Aloud Study of Novice Programmers' Code Writing Strategies. In *Proceedings of the 2014 Conference on Innovation amp; Technology in Computer Science Education* (Uppsala, Sweden) *(ITiCSE '14)*. Association for Computing Machinery, New York, NY, USA, 279–284. https://doi.org/10.1145/2591708.2591762

[23] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. Association for Computing Machinery, New York, NY, USA, 131–139. https://doi.org/10.1145/3291279.3339416