Impact of Several Low-Effort Cheating-Reduction Methods in a CS1 Class

Frank Vahid
Computer Science and Engineering
University of California, Riverside
Riverside, California, USA
vahid@cs.ucr.edu
Also with zyBooks

Kelly Downey
Computer Science and Engineering
University of California, Riverside
Riverside, California, USA
kldowney@ucr.edu

Ashley Pang
Computer Science and Engineering
University of California, Riverside
Riverside, California, USA
apang024@ucr.edu

Chelsea Gordon
zyBooks
Campbell, California, USA
chelsea.gordon@zybooks.com

ABSTRACT

Cheating in introductory programming classes (CS1) is a well-known problem. Various methods have been suggested to reduce cheating, but many are time-consuming, resource intensive, or don't scale to large classes. We introduced a class intervention having 6 low-effort commonly-suggested methods to reduce cheating: (1) Discussing academic integrity for 20-30 minutes, several weeks into the term, (2) Requiring an integrity quiz with explicit do's and don'ts, (3) Allowing students to retract program submissions, (4) Reminding students mid-term about integrity and consequences of getting caught, (5) Showing instructor tools in class (including a similarity checker, statistics on time spent, and access to a student's full coding history), (6) Normalizing help and pointing students to help resources. Via manual evaluation of similarity checker results on 7 held-constant labs with one instructor teaching 100-student sections, for two pre-intervention and two intervention sections, suspected-cheating reduced 62% (30.5% down to 11.5%). Because manual evaluation could be biased and is time consuming, we developed two automated coding-behavior metrics per lab -- time spent programming, and % of students with highly-similar code -- that may suggest how much cheating is happening. Time spent increased by 56% (7 min to 10.9 min), and % of students with highly-similar code dropped 48% (38.5% to 20%). We later repeated the intervention with a second instructor and different labs and achieved similar (in fact, even stronger) results, with time rising 84% (13 min to 24 minutes) and % dropping 66% (55.5% to 19%). All findings were statistically significant with $p < 0.0001$.

CCS CONCEPTS

• Social and professional topics - Professional topics - Computing education - Computing education programs - Computer science education - CS1

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada.

ACM ISBN 978-1-4503-9431-4/23/03. https://doi.org/10.1145/3545945.3569731

KEYWORDS

CS1, teaching, plagiarism, cheating, academic integrity, programming

ACM Reference format:

Frank Vahid, Kelly Downey, Ashley Pang and Chelsea Gordon. 2023. Impact of Several Low-Effort Cheating-Reduction Methods in a CS1 Class. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15--18, 2023, Toronto, ON, Canada. ACM, New York, NY, USA,* 7 pages. https://doi.org/10.1145/3545945.3569731

1 INTRODUCTION

Cheating on programming assignments in introductory programming classes (aka CS1) is a well-known problem [1, 2]. The temptations provided by websites dedicated to sharing class content, low-cost help/tutoring sites, and anonymous communication with classmates via real-time apps like Discord or GroupMe exacerbate the problem [3]. We sought cheating reducing methods that wouldn't require extensive resources, hours, or class redesigns, i.e., "low effort" methods.

We define cheating as a student submitting code that is not their own, typically by copying from a classmate or website (GitHub, Chegg, CourseHero, Quizlet, Wyzant, etc.), or by having someone write their code (friend, contractor, etc.), in a way that violates the class' policies.

Albluwi [1] proposed a framework from the field of fraud deterrence, namely the Fraud Triangle [4], to evaluate cheating reduction methods. Under the framework, three items are needed for cheating to occur:

- **Pressure (Press.):** Students need a good grade, and cannot (or are unwilling to) achieve the grade through non-cheating means.
- **Opportunity (Opp.):** Students have access to illicit working code, and think they won't be caught.
- **Rationalization (Rat.)**: Most students don't see themselves as cheaters, so justify their behavior. Ex: "I'm not learning from

this task", "Everyone else is copying", "Professionals copy all the time", "The expectations are unreasonable", etc.

Many "high-effort" cheating-reduction methods exist. Examples (and the Fraud Triangle side addressed) include:

- (Press.) Extensive help, days/nights/weekends, via office hours, free tutors / learning assistants, rapid response to discussion posts or emails, etc. [5]
- (Press.) Pair programming [6, 7]
- (Opp.) New programming assignments each term [1, 8]
- (Opp.) Auto-randomized assignments so each student's task is unique (typically auto-graded too)
- (Opp.) "Authentic assessments" wherein each student does a custom assignment [1, 9]
- (Opp.) More proctored assessments [9]
- (Opp.) 1-on-1 grading with the student or frequent progress checks [10, 11]
- (Opp.) Heavy plagiarism detection use [12]

Because instructors often don't have the time or resources to apply those methods extensively throughout a course, we sought to know: *Is there a set of "low-effort" methods that can be applied with no substantial class redesign, and little time from an instructor (a few hours per term), that reduce cheating?* We describe several low-effort methods and summarize our experiences illustrating their impact.

2 LOW-EFFORT CHEATING-REDUCTION METHODS

We selected numerous low-effort cheating-reduction methods from those proposed by various instructors and CS education researchers. We applied six as a class intervention, in Figure 1.

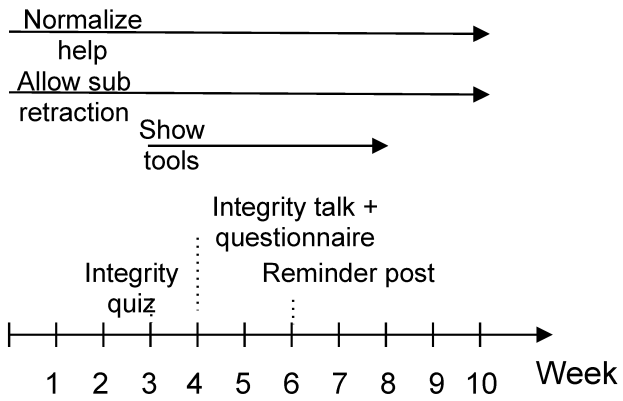

Figure 1: The intervention, consisting of six low-effort methods, in a 10-week term.

1. Having an academic integrity talk: This method addresses rationalization by helping students see the class' importance, the professor's reasonableness, and that writing code is different than understanding/modifying code. It addresses opportunity by clarifying what is a violation, noting violations will be sought, explaining similar code is not normal and tools easily detect copying, noting that students caught cheating get stiff academic consequences, etc.

2. Giving an integrity quiz: Some instructors require 100% on an integrity quiz, whose questions explicitly cover what is allowed and not allowed. Such a quiz ensures collaboration policies are understood; often students don't know what is cheating. This method addresses rationalization by making it harder for students to justify cheating (especially via "I thought it was OK"). The method may address opportunity by making cheating detection efforts and consequences clear. In previous work, 1 and 2 were shown to decrease cheating in an online masters CS course [14].

3. Allowing students to retract submissions: Malan [15] notes much cheating occurs in panicked desperation before a deadline, so allows students to retract submissions (via a "regret clause"). Doing so helps with pressure. Students may have a better perspective after a deadline.

4. Reminding students about cheating: A reminder may help with rationalization (reminding students of reasoning) and opportunity (reminding of getting caught). A tradeoff exists with creating a state of fear in students, so such a reminder might just be a simple discussion forum post or 5 minute talk in lecture.

5. Showing tools: Although instructors may warn students that a similarity checker like MOSS [16][17] will be run, students often don't believe it, or don't realize the tool's power. Showing tools in class can help address opportunity, helping students realize they are likely to get caught.

6. Normalizing help: Students in intro programming courses may not realize that needing help is normal [18]. Instructors can remind students that help is normal. Syllabi can point to help resources like office hours, tutoring services, a lab where students can ask questions, an online discussion forum, etc. This method addresses pressure since students getting help may not get desperate. It can address rationalization by making it harder to dismiss the professor as unreasonable.

3 CS1 AND THE INTERVENTION

Our CS1 is at a large public state university denoted as an "R1" (research active), offered every 10-week quarter to 300-500 students (half computing majors, half in other science/engineering majors that require CS1), via ~100-student sections, with two instructor-led 80-min lecture sessions and one teaching-assistant-led 110-minute lab session per week, in C++. The course uses a zyBook [19], with weekly: before-lecture interactive readings having ~100 learning questions (Participation Activities or PAs), ~20 code reading or writing homework problems (Challenge Activities or CAs), and 5-8 weekly programming assignments (Lab Activities or LAs -- we mostly use "zyBooks maintained labs" for LAs). All are auto-graded with immediate feedback, partial credit, and unlimited resubmissions. The course is "flipped" with active tasks during lecture. The course allows collaboration within constraints, and in lab sessions students get started on the weekly LAs. All LA coding is done in zyBook's coding window (no external tools allowed). The course grade is typically 10% PAs, 10% CAs, 20% LAs, 5% class participation, and the remaining 50-60% from a midterm exam and final exam, taken in-person, half multiple-choice and half code-writing. The class achieves consistent results across quarters and

instructors, with strong grades (more As/Bs than Cs/Ds/Fs) and a low DFW rate (D, F, or withdrawal) usually below 20%. End-of-term evaluations are positive, usually in the top 20% of all courses on campus.

One instructor who regularly teaches two sections per year applied the six low-effort cheating reduction methods in Spring 2021 and Fall 2021, under an approved protocol by our university's IRB (institutional review board).

1. The talk was in Week 4 for 30 minutes, attendance required, followed by a mandatory questionnaire requiring 100% correctness. All students participated.
2. The integrity quiz was in Week 3, with 15 multiple choice questions, 100% correctness required. Example question: "I am allowed to show my non-working code to a classmate to get help debugging" (true) or "I am allowed to show my working code to a struggling classmate" (false). Upon answering, the quiz system showed further explanations.
3. Students were informed via the syllabus and announcements of a form to retract program submissions up to one week after submitting, with no admission of anything, yielding a 0 but no penalty or referral.
4. A 1-paragraph reminder announcement was posted in the class learning management system in Week 6.
5. In weeks 3-8, the instructor showed the similarity checker and student program analysis tools during lecture about 5 times total, in natural ways like "Let's see if anyone had similar ways of solving quiz 3" or "Who volunteers to let us see their code history for lab 7?".
6. Help was normalized via syllabus text, pointers to resources, and frequent reminders by the instructor that getting help is normal. The instructor also live-coded in lecture and made mistakes, and praised mistakes made by students as learning opportunities.

The total instructor/TA time spent on the intervention was about 5 hours the first term, and about 3 hours the second term due to using already-prepared items.

We compared the two intervention sections with the instructor's prior two offerings, Fall 2019 and Spring 2020, -- the "pre-intervention" sections. The sections' differed in those 6 methods as shown in Table 1.

Method	Pre-intervention	Intervention
1. Integ. talk	~5 min in Wk 1	~30 min in Wk 4.
2. Integ. quiz	None (2-3 questions on Wk1 syllabus quiz)	15-question Wk3 quiz
3. Allow retract.	None	Announced & allowed
4. Remind	None (beyond perhaps brief comment)	Wk 6 posted announcement
5. Show tools	Not deliberate, shown 1-2 times	~5 deliberate showings Wks 3-8
6. Help	1-2 sentences in syllabus	Syllabus paragraph + pointers + freq. reminders

Table 1: **Pre-intervention and intervention class sections with respect to low-effort cheating reduction methods.**

In the intervention section, the instructor was deliberate in *not* doing much beyond those six methods, i.e., not making additional discussion forum posts, or talking extensively about integrity in later lectures, to aid in determining the impact of those specific low-effort methods.

We found 12 labs that were identical (among ~70 labs) across the pre-intervention and intervention terms. Among those, we selected the 7 that had good solution variability so similarity checking could detect copying. Table 2 summarizes the lab content, instructor solution's lines of code (LOC), and week. The labs spanned weeks 4-8. LOC is the number of lines of code in the instructor's solution including blank lines.

Lab	**LOC**	**Wk**
Lab 1: Interstate highway numbers: Output features like primary/auxiliary, N/S/E/W, etc.	36	4
Lab 2: Seasons: Takes a date as input and outputs the date's season.	85	4
Lab 3: Max and min: 3 ints input, output largest and smallest.	41	5
Lab 4: Leap-year: Given year, write function returning whether leap year.	40	5
Lab 5: Even/Odd Values in Vector: Reads ints, outputs if all even, odd, or neither.	58	7
Lab 6: Word Frequencies - functions: Reads a list of words, outputs the words/freq.	42	8
Lab 7: Contact List: Read list of names and phone numbers. Lookup num by name.	36	8

Table 2: **Summary of selected labs.**

4 SUSPECTED CHEATING BEFORE AND AFTER

We trained a teaching assistant (TA) with 1 year experience to detect cheating via the zyBooks built-in similarity checker (simchecker). For all 28 labs (7 labs * 4 terms), the TA was instructed to focus on pairs reported by the simchecker to have above 9.0 similarity or higher (max is 10.0), with 9.0 chosen from our past cheating investigation experiences. The TA was told to examine each pair manually and determine whether the pair's code was very likely a case of copying, either from each other or a common online source. Telltale signs included identical statement selection and ordering, variable declaration approach (early/late, initialized or not), variable names, spacing, brace usage, comments, and anomalies from the class & book style (untaught constructs, highly-optimized code, strange spellings, etc.). The TA was instructed to flag any highly-suspected cases, though not 100% sure cheating occurred. For each lab, the TA reported the % of students that did the lab who were suspected of copying. The TA spent about 20 hours on the cheating analysis.

Figure 2 shows results. Substantial reductions are evident for nearly all labs, averaging 32% and 29% (avg 30.5%) pre-intervention, and 13% and 10% (avg 11.5%) after.

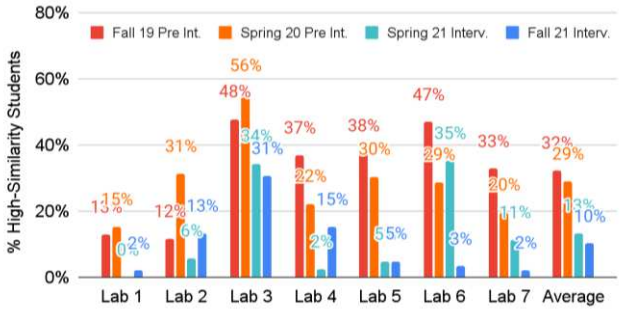

Figure 2: % of high-similarity students before and after the intervention. Substantial reductions are seen.

5 AUTOMATED METRICS

We sought automated metrics, to: (1) avoid human bias, (2) detect some cheating not yielding similar code pairs, and (3) automate future analyses. We defined two such metrics:

- Median time: zyBooks reports median time per student per lab. We required all coding be done in the system (no external IDEs were allowed for labs we examined). The median is less influenced by outliers than the mean. Copying code from online, a contractor, or classmate, may result in less time. Note: [13] found students who copy mostly do so from the start; only 10-20% copied after trying the lab.
- % of high-similarity students: For a given LA, we run the simchecker and auto-count the students appearing at least once with a 0.9 or higher, and divide by the total students who submitted that lab. [14] used a similar metric, to avoid instructor bias.

We sought to further verify the time metric. In Fall'21, we sanctioned 10 students (of ~100) for cheating. Figure 3 plots average time for all students on Week 6 labs (one of the more challenging weeks), sorted by time, showing sanctioned students as orange triangles. The sanctioned students generally appear on the left, supporting the use of time as a general cheating indicator, and is consistent with our interviews with sanctioned students who often state they didn't have time to work on their programs. (Note: One of the students near 50 on the x axis was doing the work but sanctioned for sharing solutions with another student).

To further verify the % high-similarity metric, Figure 4 shows % of highly-similar students on four Week 6 labs. On specific labs where students were caught cheating, similarity scores are nearly 100%, versus 38% for the rest of the class. In fact, those students caught cheating had higher similarity scores even on labs they weren't specifically caught cheating on, averaging 73%, as shown. (For 6.21, one student had 8.9 similarity, just below our 9.0 cutoff, causing the 80% value in the plot).

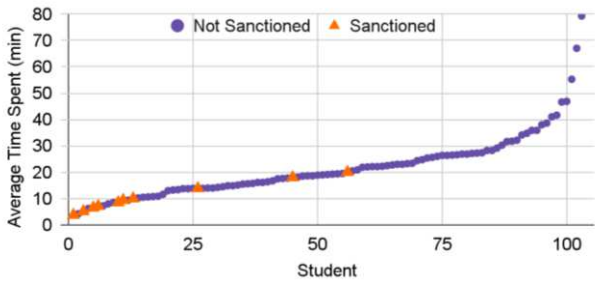

Figure 3: Average time for each student on each Week 6 lab. Students sanctioned for cheating (on any labs, not just in Week 6) tend to appear to the lower left.

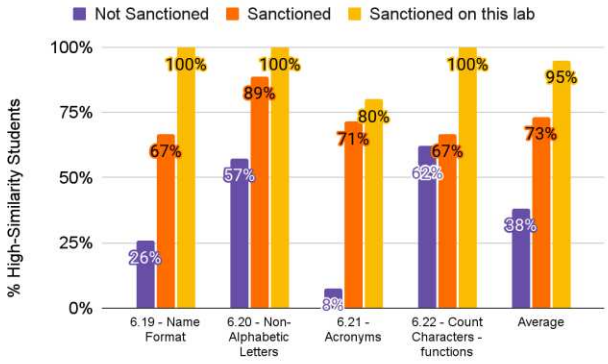

Figure 4: % high-similarity students for 4 Week 6 labs.

Neither metric is a smoking gun -- some low-time students are fast coders, and some similarity is due to coincidence, allowed collaboration, or low-variability solutions. But, the data suggests the metrics are useful as general indicators.

Our hypothesis was this: *The intervention sections would see an increase in median time, and decrease in % of high-similarity students.* Both differences might suggest students were working more independently on their programs, and resorting less to cheating.

6 RESULTS USING AUTOMATED METRICS

Figure 5 shows time for pre-intervention and intervention terms. Figure 6 shows % high-similarity students. Median time increased, as hypothesized. The % dropped, also as hypothesized. Median time for pre-intervention terms was 6.7 and 7.3, vs. 10.8 and 11 for intervention terms, averaging 7 for pre-intervention and 10.9 for intervention (56% increase). % of high-similarity students was 40% and 37% for pre-intervention terms, vs. 20% and 20% for intervention terms, averaging 38.5% vs. 20% (48% decrease).

7 RESULTS BY A SECOND INSTRUCTOR

Given the positive results obtained in Spring & Fall 2021, we enlisted a second instructor to attempt the intervention in Spring 2022 as well. That instructor was the "main" instructor of our CS1, usually teaching 2-3 100-student sections each term, and leading other instructors who teach additional sections (3-5 sections are taught each term).

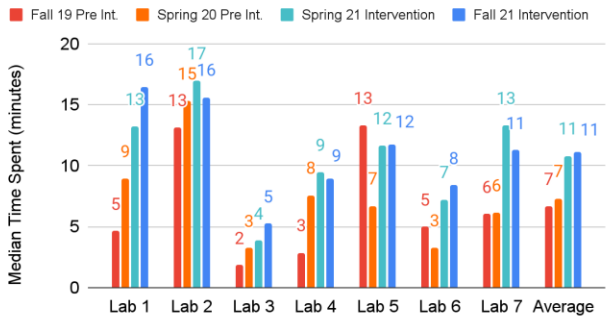

Figure 5: Median time, before and after intervention. Time rose as hypothesized.

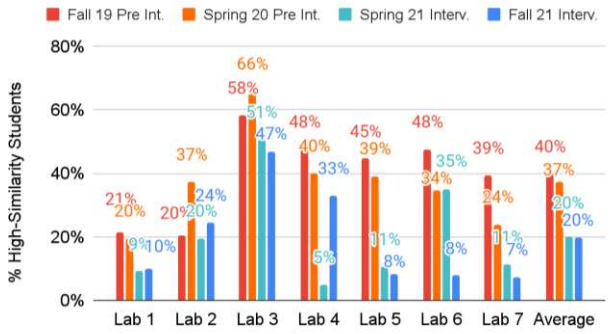

Figure 6: % of high-similarity students, before and after intervention. % dropped, as hypothesized.

That instructor applied the intervention in Spring 2022 for two ~100-student sections. We compared that instructor's "pre-intervention" Fall 2021 and Winter 2022 terms with their Spring 2022 intervention term. The instructor assigned about 50 LAs across those terms; we compared LAs that were identical in all three terms, and that had variability in their solutions. Labs 1, 3, 4, and 7 were identical to those used earlier in this paper, but the instructor didn't use the other three labs; we replaced them with other labs the instructor did use. Instructor-solution size and approximate week, written as (size, week), were : (36, 4), (24, 6), (36, 6), (45, 6), (35,7), (38,7), and (39, 9). Figures 7 and 8 show results. As hypothesized, time rose, from 13 min (avg) to 24 minutes, and % dropped, from 55.5% (avg) to 19%.

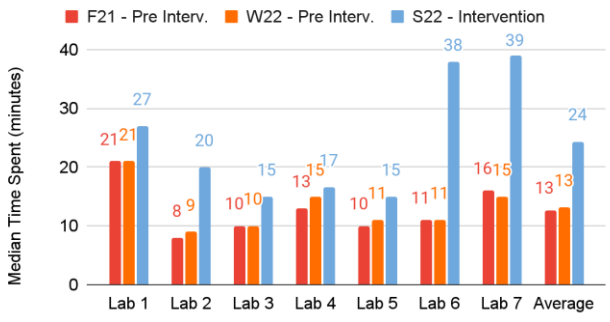

Figure 7: Median time, second instructor. Time rose as hypothesized. Note: Some labs differ from earlier.

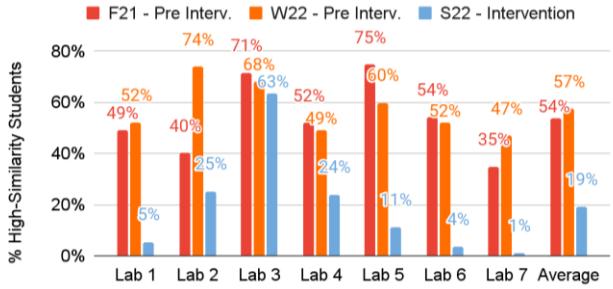

Figure 8: % of high-similarity students, second instructor. The % dropped as hypothesized.

8 STATISTICAL SIGNIFICANCE

A linear mixed effects model was fitted to the data for each instructor separately, so that the effect of our intervention could be estimated while controlling for specific labs. The dependent variable was the log transformation of minutes spent, as the residuals of raw minutes spent violated the assumption of normality. We included lab activity as a nested random effect within academic term, to control for the varied difficulty of labs. For the first instructor, a t-test using Satterthwaite's method revealed a significant effect of condition ($t(1,2834) = 6.513$, $p < .0001$). Cohen's d was calculated in accordance with [20] to obtain the partial effect size ($d=.24$). For the second instructor, a t-test using Satterthwaite's method revealed a significant effect of condition ($t(1,5978) = 6.059$, $p < .0001$). Cohen's d was calculated to obtain the partial effect size ($d=.72$).

For the percentage of high-similarity students metric, a generalized linear mixed effects model with a Poisson distribution was fitted to the data. The dependent variable was the normalized count of high-similarity students in each lab. We included lab activity as a nested random effect within academic term, to control for the varied difficulty of labs. We observed a significant effect of condition for the first instructor ($z(1,28) = -8.945$, $p < .0001$, Cohen's $d=1.2$). We observed a significant effect of condition for the second instructor ($z(1,20) = -14.37$, $p < .0001$, Cohen's $d=2.1$).

9 DISCUSSION

Another explanation of the data could be that students -- having been told instructors would be looking for cheating, could see a history of runs, and could run a similarity checker -- did fake runs to increase time, and modified code to beat the similarity checker. However, the TA's manual analysis, coupled with the instructor examining code, found most code exhibited expected student development processes. We did find a handful of students in the intervention classes who were deemed cheating, who seemed to *try* (unsuccessfully) to beat the similarity checker by modifying variable names and spacing (horizontal and vertical) -- perhaps not realizing that these changes do not impact the similarity checker.

Our interventions introduced six methods all at once. Ideally, the impact of each would be known, but such isolation did not seem

feasible, requiring running dozens of intervention sections to create sufficient ability to analyze the impact of each method (i.e., of each parameter, per experimental design techniques [21]). Most schools don't have enough sections or students for that. Also, we suspect doing just one method would have less impact; the collection may be more powerful on the student's perception that "This class doesn't allow cheating," with the sum being greater than the parts. Also, since all six methods only required a few hours total, we were not compelled to prune methods. But, learning the impact of each method may be future work.

We chose the six methods after studying cheating-reduction research, conversing with instructors about cheating reduction, attending conference sessions on cheating reduction (such as a birds-of-a-feather session at the SIGCSE conference [22] a few years earlier, and integrity panels involving instructors and students), participating on our campus' academic integrity committee and learning of techniques from committee members, and learning from our own teaching experiences over the past decade. We do not claim those six methods are the best. Rather, our goal was to determine if *some* set of low-effort methods could have much of an impact, and it seems that they can.

The interventions can be used in any class whether using zyBooks or not. The "Show tools" item requires some cheating-detection tools to be used (they are built into zyBooks).

Upon reflection, we wish we had included a seventh method, of accepting late LA submissions with a small penalty, or of allowing students to make up missed LAs. While being more effort than the other six methods, accepting lates / makeups is still relatively low effort, and its reduction of "pressure" around deadlines may help. We hope to add that method in future work.

A concern many instructors have is that focusing on cheating may hurt their end-of-term student evaluation scores. Thus, for interest (and not part of the main results of this paper, since evaluation scores can depend on numerous other factors), we report the evaluation scores of the first intervention instructor, shown as: instructor score / course score. Anything above 4.0 is generally good; our CS department average is usually 4.3-4.4:

- Pre-intervention
  - Spring 2019: 4.82 / 4.64
  - Fall 2021: 4.85 / 4.76
- Intervention
  - Spring 2020: 4.38 / 4.29
  - Fall 2021: 4.23 / 4.26

The intervention terms scores were pulled down by 3 or 4 "1" scores (not present pre-intervention) and some comments along the lines of: Maybe if the instructor focused more on teaching and less on cheating, we wouldn't need to cheat. The drop suggests instructors wishing to keep strong evaluations may need to take special care. Since collecting data for this paper, the instructor taught another term (Spring 2022) using the same interventions but taking extra care to not over-emphasize cheating. Student time and similarity data were consistent with the other intervention terms, but evaluation scores rose to 4.65 / 4.48. This gives hope that the cheating-reduction methods can be applied while maintaining good evaluation scores, but future focused work would be needed for more robust results on evaluation score impacts.

10 CONCLUSIONS

We examined the impact on student behavior when incorporating six low-effort cheating-reduction methods into a CS1 class. Via manual analysis and two automated metrics, we found those methods appear to have a positive impact on reducing cheating and increasing earnest behavior. As such, CS1 instructors (and instructors of other CS courses) may wish to consider incorporating some or all of the methods, requiring about 5 hours in a first term, and just a couple hours in subsequent terms. Not all cheating was eliminated, and thus future work remains to continue to try to reduce cheating, subject to available resources.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 2111323.

REFERENCES

[1] Albluwi I. Plagiarism in programming assessments: a systematic review. ACM Transactions on Computing Education (TOCE). 2019 Dec 9;20(1):1-28.

[2] Shaw, M., Jones, A., Knueven, P., McDermott, J., Miller, P. and Notkin, D., 1980. Cheating policy in a computer science department. ACM SIGCSE Bulletin, 12(2), pp.72-76.

[3] O'Malley M, Roberts TS. Plagiarism on the rise? Combating contract cheating in science courses. International Journal of Innovation in Science and Mathematics Education. 2012 Nov 15;20(4).

[4] Cendrowski H. and Martin J. 2015. The Fraud Triangle. John Wiley 8 Sons, Ltd, Chapter 5, 41--46.

[5] Doebling A, Kazerouni AM. Patterns of Academic Help-Seeking in Undergraduate Computing Students. In21st Koli Calling International Conference on Computing Education Research 2021 Nov 18 (pp. 1-10).

[6] Williams L, Upchurch RL. In support of student pair-programming. ACM SIGCSE Bulletin. 2001 Feb 1;33(1):327-31.

[7] Urness T. Assessment using peer evaluations, random pair assignment, and collaborative programing in CS1. Journal of Computing Sciences in Colleges. 2009 Oct 1;25(1):87-93.

[8] Simon. 2017. Designing programming assignments to reduce the likelihood of cheating. In Proceedings of the 19th Australasian Computing Education Conference (ACE'17). ACM, New York, NY, 42–47.

[9] Sheard J, Butler M, Falkner K, Morgan M, Weerasinghe A. Strategies for maintaining academic integrity in first-year computing courses. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education 2017 Jun 28 (pp. 244-249).

[10] Ernest Ferguson. 1987. Conference grading of computer programs. In Proceedings of the 18th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'87). ACM, New York, NY, 361–365.

[11] Sukhodolsky J. How to Eliminate Cheating from an Introductory Computer Programming Course. International Journal of Computer Science Education in Schools. 2017 Oct 31;1(4):25-34.

[12] Pawelczak D. Benefits and drawbacks of source code plagiarism detection in engineering education. In2018 IEEE Global Engineering Education Conference (EDUCON) 2018 Apr 17 (pp. 1048-1056). IEEE.

[13] N. Alzahrani and F. Vahid. Detecting Possible Cheating In Programming Courses Using Drastic Code Change. ASEE 2022.

[14] Mason, T., Gavrilovska, A. and Joyner, D.A., 2019, February. Collaboration versus cheating: Reducing code plagiarism in an online MS computer science program. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (pp. 1004-1010).

[15] Malan DJ, Yu B, Lloyd D. Teaching academic honesty in CS50. InProceedings of the 51st ACM Technical Symposium on Computer Science Education 2020 Feb 26 (pp. 282-288).

[16] Schleimer S, Wilkerson, DS, Aiken A. Winnowing: local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data (pp. 76-85).

[17] Novak M, Joy M, Kermek D. Source-code similarity detection and detection tools used in academia: a systematic review. ACM Transactions on Computing Education (TOCE). 2019 May 21;19(3):1-37.

[18] Lewis CM. Twelve tips for creating a culture that supports all students in computing. ACM Inroads. 2017 Oct 27;8(4):17-20.
[19] zyBooks, www.zybooks.com, 2022.
[20] Westfall J, Kenny DA, Judd CM. Statistical power and optimal design in experiments in which samples of participants respond to samples of stimuli. Journal of Experimental Psychology: General. 2014 Oct;143(5):2020.
[21] Ryan TP, Morgan JP. Modern experimental design. Journal of Statistical Theory and Practice. 2007 Dec 1;1(3-4):501-6.
[22] SIGCSE 2018 birds-of-a-feather group. GitHub, Tutors, Relatives, and Friends: Combating the Wide Web of Plagiarism: the Discussion Continues. https://sigcse2018.sigcse.org.