Towards Comprehensive Metrics for Programming Cheat Detection

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

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

Benjamin Denzler
Computer Science and Engineering
University of California, Riverside
Riverside, California, USA
bdenz001@ucr.edu

ABSTRACT

Automated assistance for detecting cheating on programs has long been investigated by CS educators, especially with the rise of "homework help" websites over the past decade, and recently with AI tools like ChatGPT. The main detection approach has long been flagging similar submission pairs. Modern cheating, like hiring contractors or using ChatGPT, may not yield such similarity. And, cases based on similarity alone may be weak. Thus, over the past several years, building on logs from an online program auto-grader (zyBooks), we developed additional "cheating concern metrics": points rate, style anomalies, style inconsistencies, IP address anomalies, code replacements, and initial copying. Most are defined not only for one programming assignment but also across a set of assignments. The metrics can help catch more kinds of cheating, provide more compelling evidence of cheating, reduce false cheating accusations based on similarity alone, and help instructors focus their limited cheat-detection time on the most egregious cases. We describe the techniques, and our experiences (via our own Python scripts and a commercial tool) for several terms, showing benefits of having more metrics than just similarity. Of 30 cheating cases over 3 terms and 300 students, most were based on metrics beyond similarity, all students admitted, none later contested, and time per student was only 1-2 hours (far less than previously). Our goal is to prevent cheating in the first place, by reducing opportunity via strong detection tools, as part of a multi-faceted approach to having students truly learn and stay out of trouble.

CCS CONCEPTS

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

KEYWORDS

CS1, cheating, plagiarism, AI, homework, programming assignments

SIGCSE 2024, March 20–23, 2024, Portland, OR, USA.

ACM ISBN 979-8-4007-0423-9/24/03.
https://doi.org/10.1145/3626252.3630951

ACM Reference format:

Frank Vahid, Ashley Pang, and Benjamin Denzler. 2024. Towards Comprehensive Metrics for Programming Cheat Detection. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024). March 20-23, 2024.* Portland, OR, USA. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626252.3630951

1 INTRODUCTION

Cheating is a serious problem in programming courses [1]. Most cheating detection focuses on similarity checking [2, 3, 4, 5], which has detected much cheating over the years. However, similarity checking is known to have false positives, such as for small programs, yet a "many small programs" approach is popular especially in CS1 [6]. It can also have false positives when not much logical variation exists in possible solutions, even for larger programs. Pang [7] suggests introducing variability-inducing requirements to increase solution variability, but of course the approach is limited and does make the programs a bit harder for students. Other researchers have proposed complementary cheat detection techniques. For example, as some platforms record code history, some propose examining history for normal incremental development versus copying [8, 9]. Tahaei [10] proposed examining code history for cases where a student's code history shows one program being entirely replaced by another program, which is a common sign of copying. Similarly, Alzahrani [11] developed a tool to detect "drastic change" in code history. Drastic changes could be students copying a solution immediately, hopping between found solutions, or giving up on their attempts and pasting an online solution.

We switched to the zyBooks program auto-grader several years ago, which appears to be one of the most widely used program auto-graders in the U.S. [12]. zyBooks provides an IDE that records data on every program a student runs, including date, time, IP address, the code itself, test case results, score, and more. We naturally began using this log data to assist in our cheat detection efforts beyond similarity detection alone, writing scripts to measure time spent, detect code replacements, detect unusual constructs often found in contractor-written solutions, etc. These scripts are being developed as part of an NSF-funded project and the plan is to make them available to instructors in the near future. We also began using a beta tool from zyBooks itself that has begun incorporating such metrics. In this paper, we report our experiences developing and using this more comprehensive set of

cheat-detection metrics to detect cheating in our CS1 class at a large public university. We believe such tools will eventually become commonplace in more CS courses, to detect more kinds of cheating, and hopefully as a strong deterrent that prevents such cheating, akin to how nobody speeds right past a police officer. Deterrents of course are just one part of a more comprehensive approach to keeping students focused on learning.

2 CHEATING CONCERN METRICS

2.1 Similarity

Similarity highlighting is the most common form of cheat detection. Some similarity checking tools are standalone like MOSS [2] and Jplag [3]. Most program autograders come with built-in similarity checking [13, 14, 15, 16]. With most similarity checkers from those autograders or in research tools [2, 3, 4, 5], for a given programming assignment (what we will call a "lab"), instructors are provided a sorted list of highly-similar program pairs (or a group) as seen in Figure 1.

Figure 1: List of highly-similar program pairs sorted by similarity concern. Each row is a student (names intentionally omitted).

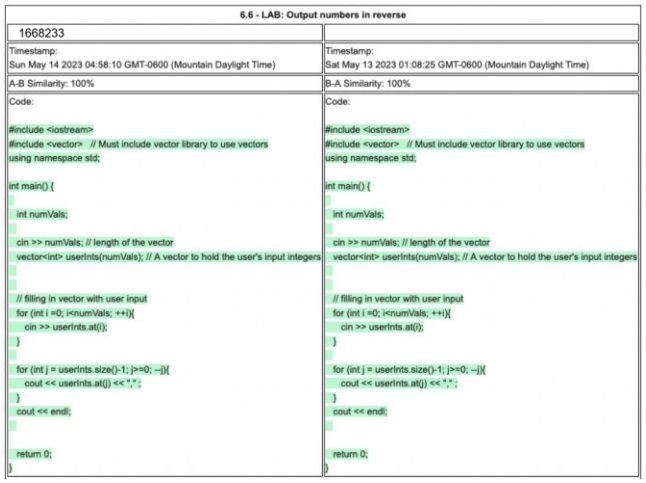


Figure 2: Most similarity checkers will highlight the code that is deemed similar across a pair of programs.

For a given pair, the similar code is highlighted, as in Figure 2. Similarity checkers typically ignore insignificant variations, like variable names, whitespace, comments, and even some statement ordering. With our tool, we have found that having 90% or higher similarity is suggestive of cheating, at least for labs with expected variation in solutions.

Most similarity checkers focus on one lab. Running a checker on every lab, and investigating similar pairs, can be cumbersome, especially if students are given multiple labs per week. Thus, in practice, many instructors run similarity checking on a few labs, such as one selected lab per week, or one from every few weeks.

Instructors are often interested in focusing their limited cheat-checking time on the most egregious cheaters. As such, we built a higher-level of similarity concern on top of a MOSS-like similarity checker. Given a set of labs, the technique runs the similarity checker on each lab, and for each student it counts the number of labs for which the student has a 90% or higher similarity with at least one other student. Because not all labs have much solution variability, we exclude low-variability labs where more than half the students have 90% or more similarity with at least one other student. We find low-variability labs with the assumption that the majority of the class isn't cheating (which could in some cases not be true of course). The remaining labs are "high variability" labs where 90% similarity is more likely to suggest cheating. We define the following metrics for each student:

- Similarity average: The average of the highest similarity score per lab for this student across all high-variability labs.
- Similar labs count: The number of high-variability labs where the student has 90% or higher similarity with at least one other student. Low-variability labs are ignored.

For every metric, we desire to have a "normalized" score that suggests to instructors the likelihood of the metric value indicating cheating. A score of 1 should almost certainly mean cheating (but never for sure -- instructors must always investigate), 0 means no evidence of cheating, and 0.5 is a border above which instructors might wish to investigate. We thus provide the following normalized similarity metric:

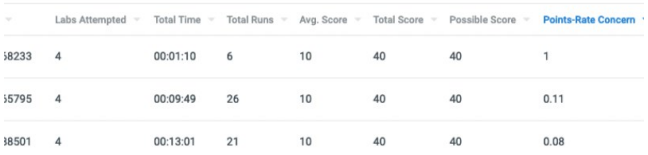
- Similarity concern: We map similarity average to the range 0 to 1 using standard deviations, with 0.5 being 1 standard deviation above the class average, and 1 being 2 standard deviations above. (Those standard deviation values could be adjusted).

This is just one approach to mapping, which has worked well for us. Other more rigorously-determined approaches are possible and a subject for future investigation.

We also track which other students appeared as highly-similar to the current student for each counted lab. If another student appears in half or more of the counted labs, their name appears in a "potential collaborator" list with an orange flag. If the other student appears in 2/3, we red flag them as a "very likely collaborator".

2.2 Points Rate Concern

If an instructor requires all development be in a tool that records develop time and run information, then those values can help detect cheating: Students with unusually low time or runs, yet high scores, might be copying from external sources (of course, they might just be proficient programmers). Thus, we provide some development metrics to instructors, as shown in Figure 3.



	Labs Attempted	Total Time	Total Runs	Avg. Score	Total Score	Possible Score	Points-Rate Concern
58233	4	00:01:10	6	10	40	40	1
55795	4	00:09:49	26	10	40	40	0.11
58501	4	00:13:01	21	10	40	40	0.08

Figure 3: Student roster showing time and points metrics. Each row is a student (names intentionally omitted).

Relevant metrics include the time each student spent, and the number of times the student ran their code. Students with low time (or runs) but high scores are suspect. As such, we define the following normalized metric:

- Points-rate concern: We calculate the ratio of TotalScore / TotalTime for each student. Those scores are mapped to 0 to 1 such that 1 standard deviation above the class average yields 0.5, and 2 standard deviations yields 1.

Again, the standard deviation values could be adjusted; we have found the above work well. We note that the standard deviation approach assumes a class with sufficient students doing the labs, such as perhaps 25 or more students. For smaller classes, another mapping approach might be needed.

2.3 Style Anomalies

Most instructors have their class follow a particular "coding style", usually (but not always) following their textbook's style. Being required to follow a coding style is common in industry too. Example style features include:

- Variable naming (camelCase, under_score, etc.)
- Brace style (opening brace on the same line or next line)
- Variable declaration rules (at top or spread out, initialized or not, multiple per line or not, etc.)
- Constructs taught (user-defined functions or not, conditional operators or not, etc.)
- Indent amount (2 space, 3 space, 4 space, etc.)
- Use of while (true) loops and break (common with AI)

Typically dozens of such style features exist.

When students copy code from online sources, have an external person program for them (friend, contractor), or let AI program for them, often those programs' style departs from the class style. We call those "style anomalies".

Figure 4 shows code with several style anomalies from our class' style: late declarations, short variable names, initialized declarations, calling of an untaught "min" function, no blank lines, and more. Figure 5 shows some code that is left-aligned, usually resulting from a copy-paste of code from another source.

Our technique defines about 20 style features (a unique set of ~20 for each of Python, Java, C++, and C), with defaults set to a common style but configurable by an instructor. Then, we define the following metric:

- Style anomaly count: A count of the number of style anomalies found in a student's programs.

Our tool originally used regular expressions to detect style anomalies, though we recently developed a version that uses a more symbolic approach.

```
#include <iostream>
#include <vector>
using namespace std;
int GetVectorMin(const vector<int> &myVec) {
  if (!myVec.size()) {
    //empty
    return -1;
  }
  int mn = myVec[0];
  for (int val: myVec) {
    mn = min (mn, val);
  }
  return mn;
}
int main() {
   int n;
   cin >> n;
   vector <int> myVec(n);
   for (int i=0; i<n; i++) {
    cin >> myVec[i];
  }
   cout << GetVectorMin(myVec) << endl;
   return 0;
}
```

Figure 4: Style anomalies are styles that depart from the class requirements or norm, like the late declaration of mn, or the use of array brackets in myVec[i].

Figure 5: Left-aligned code is a style anomaly, which usually occurs when a student copy-pastes code from another source. This figure shows our tool's auto-detection and highlighting of the anomaly.

For most style features, by default our tool counts every instance of a style variation, such as counting every time an untaught construct like a user-defined function appears. For some features, the anomaly by default is counted just once no matter how often it appears, such as varying from normal indenting amount. The instructor can change that setting for any feature. Furthermore, all features by default are weighted by 1; the instructor can increase the weight of any feature. For example, we forbid use of array brackets in our C++ class (instead requiring use of v.at() notation), so we weigh the style feature "array brackets" with a higher value such as 5 instead of 1.

2.4 Style Inconsistencies

We found a key indicator of cheating is variation in coding style across the same student's set of programs. For example, as shown in Figure 6, a student might in one program have standalone closing braces before a subsequent else, but in another program might put closing braces on the same line as the else. As another example, one program might use 3-space indents, another 4-space indents, and then another back to 3-space. Normal students don't change style like that; such inconsistency is usually indicative of copying from various online sources whose styles vary, or from an AI tool which may not always use the same style.

We define about 30 style features per language (Python, Java, C++, C). For a given student, we analyze their programs and determine the "majority" style for each feature, e.g., brace on standalone line, 3-space indents, for loop index declared in loop, etc. We refer to the set of majority style features as the base style. Then, we define the following metric for each student:

- Style inconsistency count: A count of the number of style inconsistencies in a student's programs.

We also define "style inconsistency concern" as a 0-1 metric, like done for earlier metrics using standard deviations.

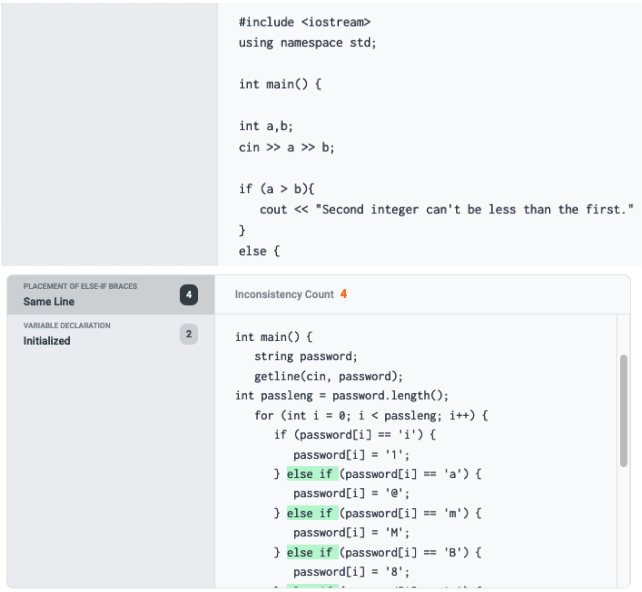


Figure 6: Style inconsistencies are unnatural variations in style across programs for the same student, such as different brace styles between the above two programs from the same student. This figure shows our tool's auto-detection and highlighting of departures from the student's majority style.

2.5 Code Replacement / Initial Copying

If instructors require all development be done in a tool, then the history of the code can reveal possible cheating. We sometimes see students whose very first code instance is a nearly-complete solution, perhaps 100 or more lines of code. In contrast, most students would start perhaps with 20 lines and then run that code, then add 10-20 lines and run that code, etc. On a related note, sometimes students will be working on a solution, cannot get it functioning correctly, and then suddenly in their code history an entirely different solution appears. Some students do this multiple times; we call those "solution hoppers". We thus define two more metrics:

- Initial code sizes: The average number of lines of code in the student's very first run in their code history for each program.
- Code replacement count: A count of the number of times two subsequent code runs are drastically different.

We use a standard text diff to detect code replacements, though more sophisticated and accurate approaches are surely possible.

We define an "Initial code size concern" metric as normalized to 0 - 1 using the standard deviation approach. Likewise we define a "code replacement concern" metric similarly.

We are working on detecting actual copy-pastes and using those to further help detect initial copying or code replacements.

2.6 IP Address

Our system logs IP addresses for each submission. We can analyze the locations for oddities. For example, sometimes students will give an external programmer their login credentials. This may lead to a student's program coming from overseas. Or, we have seen nearly-simultaneous submissions from two different IP addresses for one student, suggesting a student may have somehow divided their work with a classmate or friend.

We have not yet automated the calculation of an IP address concern metric. We currently look at IP addresses manually for already-concerning students. But factors in such a calculation might include the distance of the IP addresses from the university, and the variation among IP addresses for a student, especially if close in time.

2.7 Overall Concern

Instructors can select any set of labs, after which our tool calculates most of the above metrics. Then, instructors can sort by any metric, and focus on the highest-concern students.

However, because instructors have limited time, we are interested in defining an "overall concern" metric, which effectively combines all the metrics to yield a single 0-1 concern metric value. Instructors could thus sort by that metric and focus on the highest concern students.

First, we sort the students by the max across all of that student's metric values. Next, we secondary sort by the second largest metric value, and so on. Note that this equates to concatenating a student's metric values into one number, then sorting the digits in descending order, and finally sorting students based on that number. For example, if Student A had a 0.7 similarity concern score and a 0.9 style anomaly score and if we ignore other concern metrics, Student A's overall concern would be 0.97. If Student B

has 0.8 similarity and 0.0 style anomaly scores, their overall concern would be 0.80.

Actually, because individual concern metrics can range from 0 to 1, we shift each metric down by 0.1, with a floor of 0.0, so that each individual concern metric ranges from 0.0 to 0.9, before we concatenate them into an overall concern metric.

More powerful approaches are surely possible for determining overall concern. For example, some approaches might take a more "intelligent" path through the individual concern metrics. A student might have high similarity, but if they also have a low points-rate concern (meaning they are spending a lot of time), and they have no style anomalies or inconsistencies, we might be less concerned. Or, if a student has high style anomalies, and their IP addresses all come from overseas, we might express very high overall concern.

3 EXPERIENCES USING MULTIPLE METRICS FOR CHEATING DETECTION

We summarize our experiences using the above metrics for cheating detection across 3 terms from 2021 - 2023. For student privacy reasons and under the auspices of our IRB (institutional review board), we group and summarize cases, and in particular we do not list specific students nor do we list specific terms, to reduce chances of sanctioned students detecting themselves in our writeup.

Across the three terms, we sanctioned 30 students for serious cheating. Nearly all received Fs, though a few received course grade reductions (1-2 letter grade deductions). Each term had about 100 students, so the sanctioning rate was about 30 / 300 = 10%. All sanctioned students eventually admitted to cheating.

The tools have been evolving over the past 2 years. For all three terms, we had automated points rate, style anomaly, and similarity concern metrics. We had access to IP addresses. We also had easy access to entire coding histories, so we could see if a student was working normally or had initial copies or code replacements in their history. This access included seeing for a selected student the latest-highest scored program submitted for every selected lab all on a single page, so we could quickly scroll and see all the labs submitted by that selected student. In the latest term, an overall concern metric was made available, and an automated style inconsistency tool was developed.

We found ourselves making heavy use of three metrics -- points rate, style anomaly, and similarity -- to find students of concern. Our weekly routine evolved towards selecting all labs from the past 2-3 weeks (totaling 10-15 labs), running tools, and then:

- First, we'd sort by the points rate concern metric to find students getting high scores but spending little time relative to classmates. Typically about 5 students in our class of 100 would have points rate concern values above 0.7. For each, we would scroll through the "all labs on a single page" view to visually detect style anomalies or style inconsistencies. If none, we'd quickly look at code history for a few labs, and if development looked normal, we'd assume the student was just a fast programmer. If detected, we'd look a bit more at code history, look at the similarity reports, and decide whether to contact the student about possible cheating.
- Second, we'd sort by the style anomaly concern metric to find students with code departing from class style. That list was typically about 5-10 students having concern values over 0.7. For each, we'd scroll through the "all labs on a single page" view for inconsistency in style.
- Third, we'd look at the similarity data showing orange and red flagged students, to look for cases of repeated copying among classmates.

For suspicious cases after the above investigation, we would always first reach out to students asking about their labs, without accusing them of cheating, and invite a conversation either via email or a Zoom call. In general, we found:

- Style anomalies were the "smoking gun" in a majority of cases, yielding very compelling evidence of cheating. Code would have advanced techniques, like pointers, infinite loops with breaks, user-defined functions (before we taught that), range-based for loops, etc. Upon conversing with the student, they would often have no understanding of those techniques. But in a few cases, they had prior experience and could easily explain their used techniques.
- Style inconsistency was the second most compelling evidence of cheating. Students usually could not justify why they would substantially change their style from one lab to another. We cannot recall any justified cases of such inconsistency; it was always due to copying from different sources or from AI being inconsistent.

Style anomalies or inconsistencies were the main factors in about 2/3 of all our sanctioned cases. The most common sources that students described during our discussions were Chegg, Quizlet, Coursehero, GitHub, Stack Overflow, and more, though often students just googled for solutions without remembering which site returned a hit. Somewhat amusingly, sometimes students would copy-paste a solution in the wrong language, and other times the code they pasted would include the surrounding items on the website like the request for a solution or thanking the solution provider. In the most recent term, copying from ChatGPT was also a commonly-stated source by students.

About a half dozen of the students admitted to having another person program for them. Some hired contractors online, and paid them for solutions -- typical rates were about $30-$50 per solution. Some had family or friends doing their programming for them. In some cases, the IP addresses provided compelling evidence, where students gave their login credentials to someone else, causing their submissions to be coming from overseas, or from two places at once.

We also found similarity was compelling evidence in about 1/3 of cases. Usually, similarity was due to one student providing

excessive help to a classmate. We would see one student's code history clearly showing hard work, and another student showing little work and usually scoring high on the points rate concern metric. We could also see from timestamps who was completing the work first. In discussions, one student would often say they felt bad for their friend and just wanted to help. Sometimes they would share code with the classmate but ask them not to copy it, but then the classmate would get desperate and copy.

Sometimes, similarity was due to independent copying from the same online source. Some students would have identical code but not know each other, due to both copying from Chegg. On some occasions, an online contractor would be paid by two students independently, and just give them both the same code. And, we found that ChatGPT sometimes generates the same code for different students too. In all these cases, there were usually also style anomalies or inconsistencies, so the similarity was just further evidence of cheating.

Points rate concern was helpful in detecting potential cases, though itself was not "evidence" of cheating.

We typically spent about 1-2 hours per week detecting, investigating, and reporting cheating. Some weeks we spent none, other weeks 2-4 hours. The tools did most of the hard work of detecting suspicious cases. For a week's suspicious cases, we'd spend about an hour total doing the manual investigations of the 5-10 highlighted students. The most time-consuming task was the discussions with students, especially when using Zoom meetings vs email, which we greatly prefer over emails because we can not only progress more quickly towards "clear cheating" or "the student is actually programming", but also because we find it is better for the student since we are respectful and reassuring and thus can help reduce their worry and stress (of which there can be a lot).

4 DISCUSSION

The comprehensive metrics approach has substantially changed our cheating detection efforts.

- Detection previously was based entirely on similarity for a single lab. We can now detect a much broader range of cheating.
- With detailed logs in recent years, detection was a cumbersome manual process. Today, the aforementioned tools make finding suspicious cases almost trivial, just running the tools (which takes < 1 minute) and then sorting by a concern metric, followed by a bit of clicking to examine student code.
- Every student in those three terms eventually admitted to cheating, as the evidence became overwhelming. The students would realize they could not explain their style anomalies or inconsistencies, justify having identical code to an unknown classmate, or explain why all their code is coming from another country. Most students would initially try to justify such things, but as we would show more evidence, they would say "Ok, ya, I copied from Chegg" or "Well, OK, I admit I used ChatGPT, I was just desperate", etc. Previously, with similarity checking alone, students would often not admit, instead sticking with "We just discussed the program together" or "I looked at his code but I didn't copy-paste it".
- Since using the more comprehensive metrics, not a single case has been contested or overruled after being referred to student conduct. The evidence is likely just too overwhelming. In the past, students would sometimes contest the cases, and on occasion the student conduct panels would say that the evidence wasn't sufficiently conclusive that cheating occurred. We have not experienced that in over two years now.

While zyBooks provides a beta tool, we are developing our own scripts that simply take log files as input and generate similar metric values. Other platforms can also be used, but they may capture different metrics which may yield different results.

Ultimately, our goal is not to punish cheating but rather to prevent cheating. A strong deterrent helps, reducing "opportunity" per the fraud triangle [1]. As such, we show students some of our tools -- in a friendly way, like "Let's see how much time folks spent programming last week", or "Let's see if anyone came up with similar solutions on the quiz problem", pulling up the tools in anonymous mode. We also address the other two parts of the triangle, aiming to reduce "pressure" via heavily scaffolded learning and second-chance policies, and aiming to reduce "rationalization" via establishing rapport, having reasonable policies, discussing moral/ethical issues with students, and motivating why learning programming is useful.

5 CONCLUSIONS

The more comprehensive set of cheating concern metrics described in this paper have: (1) Assisted us in detecting cheating cases that would have gone unnoticed just considering similarity, (2) Helped provide stronger cases for cheating referrals, while also exonerating students who might have otherwise been referred based just on similarity, (3) Saved hours of instructor time per week by auto-highlighting the most egregious cases across a set of programs. We have especially found that the style anomaly and the style inconsistency metrics have become our preferred metrics, often providing "smoking gun" evidence, sometimes in conjunction with similarity, and other times when similarity doesn't even apply. As such, we encourage the CS community to integrate such techniques into their cheat detection approaches. More generally, we believe strong cheat detection can provide a deterrent to cheating, as part of a multi-faceted approach to reducing cheating that might include moral/ethical aspects, increased/improved proctored assessments, and more.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Nos. 2111323 and 2313793.

REFERENCES

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

[2] 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).

[3] Software plagiarism detector, https://jplag.ipd.kit.edu/, accessed 2022.

[4] Prechelt, L., Malpohl, G. and Philippsen, M. Finding plagiarisms among a set of programs with JPlag. Journal UCS, 8(11), 2002.

[5] Lisa Yan, Nick McKeown, Mehran Sahami, and Chris Piech. 2018. TMOSS: Using Intermediate Assignment Work to Understand Excessive Collaboration in Large Classes. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18). Association for Computing Machinery, New York, NY, USA (pp. 110–115).

[6] Allen, J.M., Vahid, F., Downey, K. and Edgcomb, A.D., 2018, June. Weekly programs in a CS1 class: Experiences with auto-graded many-small programs (MSP). In 2018 ASEE Annual Conference & Exposition.

[7] Pang, A., & Vahid, F. (2023, June). Variability-Inducing Requirements for Programs: Increasing Solution Variability for Similarity Checking. In Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (pp. 430-435).

[8] Ericson, B.J. and Miller, B.N., 2020, February. Runestone: A Platform for Free, Online, and Interactive Ebooks. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education (pp. 1012-1018).

[9] Hagan, D. and Markham, S., 2000, December. Teaching Java with the BlueJ environment. In Proceedings of Australasian Society for Computers in Learning in Tertiary Education Conference ASCILITE.

[10] Tahaei, N. and Noelle, D.C., 2018, August. Automated plagiarism detection for computer programming exercises based on patterns of resubmission. In Proceedings of the 2018 ACM Conference on International Computing Education Research (pp. 178-186).

[11] Alzahrani, N., & Vahid, F. (2022, August). Detecting Possible Cheating In Programming Courses Using Drastic Code Change. In 2022 ASEE Annual Conference & Exposition.

[12] Gordon, C.L., Lysecky, R. and Vahid, F., 2021, July. The rise of program autograding in introductory cs courses: A case study of zylabs. In 2021 ASEE Virtual Annual Conference Content Access.

[13] https://www.gradescope.com/, accessed Aug 2023.

[14] https://www.vocareum.com/, accessed Aug 2023.

[15] www.zybooks.com, access Aug 2023 (https://www.zybooks.com/catalog/zylabs-programming/).

[16] https://www.codio.com/, accessed Aug 2023.