Plagiarism in Take-home Exams: Help-seeking, Collaboration, and Systematic Cheating

Arto Hellas
University of Helsinki
Department of Computer Science
Helsinki, Finland
arto.hellas@cs.helsinki.fi

Juho Leinonen
University of Helsinki
Department of Computer Science
Helsinki, Finland
juho.leinonen@helsinki.fi

Petri Ihantola
Tampere University of Technology
Department of Pervasive Computing
Tampere, Finland
petri.ihantola@tut.fi

ABSTRACT

Due to the increased enrollments in Computer Science education programs, institutions have sought ways to automate and streamline parts of course assessment in order to be able to invest more time in guiding students' work.

This article presents a study of plagiarism behavior in an introductory programming course, where a traditional pen-and-paper exam was replaced with multiple take-home exams. The students who took the take-home exam enabled a software plugin that recorded their programming process. During an analysis of the students' submissions, potential plagiarism cases were highlighted, and students were invited to interviews.

The interviews with the candidates for plagiarism highlighted three types of plagiarism behaviors: help-seeking, collaboration, and systematic cheating. Analysis of programming process traces indicates that parts of such behavior are detectable directly from programming process data.

KEYWORDS

plagiarism, programming process data, educational data mining

ACM Reference format:
Arto Hellas, Juho Leinonen, and Petri Ihantola. 2017. Plagiarism in Take-home Exams: Help-seeking, Collaboration, and Systematic Cheating. In *Proceedings of ITiCSE'17, July 03-05, 2017, Bologna, Italy.*, , 6 pages.
DOI: http://dx.doi.org/10.1145/3059009.3059065

1 INTRODUCTION

Universities and other institutions are facing increased enrollments into their Computing programs. It is no surprise that teaching and learning is moving online at increased speed: there has been a push towards improving the feedback cycle that is associated with coursework and examinations. First, feedback should be available faster, and second, the costs should reduce. This is partially due to the instructional quality, but also due to the price of traditional education. As a downside of this online learning movement, academic dishonesty such as plagiarism has become easier and even more common [11].

Merriam-Webster online dictionary defines **plagiarism** as *an act of copying the ideas or words of another person without giving credit to that person* [15]. This is a significant problem especially in computing education [5, 6], where students actually see copying and pasting source code somehow more acceptable than doing the same with natural language and essays [1].

Because of the wide impact and importance of the topic, source code plagiarism has been studied for nearly three decades [12]. Much of the research has revolved around tools for detecting plagiarism (e.g., JPlag [17] or MOSS [4]) or students' attitudes towards plagiarism [23]. Tools and methods for detecting plagiarism are typically based on comparing similarities between code segments [14, 19]. Although modern learning environments provide rich log information on how students solve programming assignments [10], this knowledge is rarely applied in plagiarism detection.

In this work, we study plagiarism in take-home exams of an introductory programming course. As a part of take-home exams, students were expected to use a custom tool for downloading the take-home exam questions and answer templates so that the exam questions could be answered on a schedule that best fits the student. In addition, the tool recorded and sent the typing level changes made to the downloaded answer templates.

Our research questions for this study are as follows:

(1) What types of plagiarism happen in take-home exams?
(2) How can these different plagiarism types be identified from programming process data?

We answer these questions by using a mixed method approach. First, we checked all the exam answers using JPlag[1] and manually reviewed the results to identify suspicious cases. Next, we interviewed all the related students from where different motivations and trends explaining students' behavior emerged. Finally, we compared programming traces to these verified plagiarism cases in order to find whether different behaviors could be identified directly from the traces.

The rest of the article is structured as follows. Next, in Section 2, we outline the related research on source code plagiarism. This is followed by a description of our data and context in Section 3. Section 4 outlines the study methodology and the results, which are then discussed in Section 5. Section 6 concludes the article and provides directions for future work.

[1]The tool was selected based on the comparison of multiple plagiarism detection tools supporting Java [7]

2 PLAGIARISM IN PROGRAMMING

Many studies have been conducted on plagiarism in programming [1, 5, 11, 22, 23]. Especially, the attitudes of students [1, 11, 22] and academics [5] on what constitutes plagiarism in programming, which is quite different from traditional plagiarism as reusing code is often encouraged on introductory courses [5]. This can lead into a situation where the fine line between plagiarism and normal course practice becomes unclear to students [11]. Cosma and Joy [5] propose a definition of plagiarism based on survey answers from computer science educators based in the U.K. Based on the survey results, they define plagiarism as reusing code segments without properly citing the original author of the code. They also include obtaining the source code in their definition. They suggest that the main reason behind plagiarism in programming assignments might be that students are confused as to what constitutes plagiarism.

Joy et al. [11] studied students' understanding of plagiarism in a programming context. Similar to Cosma and Joy [5], they believe that students' confusion about plagiarism can lead to accidental plagiarism. They found that students are especially confused about reusing code from previous assignments, how accurately references should be cited, to what extent can the students collaborate on assignments, and that existing code should also be referenced when it is converted from one programming language to another.

Students may also perceive some forms of cheating as less severe than others [22]. For example, students view collaborating on assignments to not be as serious an offense as cheating in the exam. Students can be reluctant to report the cheating of fellow students if they observe it, which suggests that automatic ways of identifying cheating would be beneficial [22].

From the technical point of view, source code clone detection methods can be divided between text based (e.g. hashing of code lines), token based (i.e. similar to tokenization in compilers), tree based (e.g. normalized AST comparison), metrics based (i.e. combining multiple simple characteristics), graph based (e.g. depency graphs), and mixed approaches [19]. Although all approaches aim at coping with simple refactoring such as editing names or changing the order interchangeable code blocks, many tools are still insensitive to such edits – especially if multiple refactorings are combined to hide the origin of the work [7]. To alleviate these challenges, it has been recently proposed to incorporate edit history more closely into plagiarism detection process [8, 21]. These previous experiments on utilizing the history, however, deal with more coarse-grained edit data than what is available to us.

3 CONTEXT AND DATA

This study took place in an introductory programming course organized during fall 2016 at University of Helsinki, a research university in Europe. The course is organized in Java, and the contents of the course are similar to many other introductory programming courses offered at universities: variables, input/output, selection, objects, lists, maps, inheritance and so on.

Students who enroll to the BSc program as CS majors take the course during their first semester. Students with other majors take the course when it fits their schedule, given that they wish to take it. It is only mandatory to the CS majors. From the course population, roughly one third are CS majors, and two thirds potentially consider CS as a minor subject.

The grading of the course was based on a set of individual programming assignments and pair-programming assignments that correspond to 55% of the overall course mark, and three take-home programming exams that correspond to 45% of the overall course mark. The students had to gain at least half of the available points from the final take-home programming exam that covered the whole course in order to pass. The students in the course also had the option of participating in a traditional exam at the University facilities instead of the take-home exams.

All the programming assignments were administered using Test My Code [25]. Test My Code (TMC) provides a plugin for programming environments that makes it convenient to download, assess and return assignments. Over 95% of the participants used TMC with NetBeans, but some opted for Eclipse and IntelliJ IDEA.

Each take-home exam had three to five separate questions, and the students had four hours to complete the exam. Any kind of collaboration or help-seeking was strictly forbidden in the take-home exams, but the students were allowed to use the course materials and their own solutions to the course assignments while working on the take-home exam. Each take-home exam handout reminded the students of the rules. The university takes strict policy on plagiarism: plagiarism leads to a failed course mark and recurring offense leads to temporary expulsion from the university.

When a take-home exam was administered, the students could pick a suitable time from a set of days for the exam. In order to participate in the take-home exam, the students were expected to enable the logging of programming process in TMC and to download the assignment templates used in the exam. Downloading the templates, which also included the handouts, started the exam. The logs that TMC gathers includes every subsequent source code state as well as typical programming environment events such as running, debugging and testing the program.

In the course, 233 students participated in at least one take-home exam, and 204 participated in the final take-home exam from which the students had to gather at least half of the points to pass the course. For the analysis, data from all the students was used.

4 METHODOLOGY AND RESULTS

4.1 Identifying candidates

A preliminary list of candidates who potentially plagiarized in the take-home exams were identified using four separate methods:

(1) The JPlag-system [17] and a custom edit-distance metric was used to assess the similarity of students' solutions to the take-home exam questions. Students with very similar solutions to other students were suspect.
(2) Students' course mark from the programming assignments were contrasted with the take-home exam scores. Outliers, i.e. students with poor assignment scores and high take-home exam scores, were suspect.
(3) Pair-wise comparison of take-home exam start and end times as well as the submission times of the take-home exam questions. Students with similar start times, end times, and submission times were suspect.

(4) Pair-wise comparison of IP address spaces and IP addresses from which the submissions were made. Students with very similar IP addresses (outside known campus networks with roaming) were suspect.

The preliminary list was manually studied in order to remove false positives. False positives came mainly from JPlag and the custom edit-distance metric as some of the exam questions were highly structured.

After constructing a list of candidates approximately 15% of the participants were contacted for further interviews.

4.2 Contacting and interviewing the candidates

The students were contacted through email that informed them that they had been highlighted by a system that is used to detect plagiarism. The students were asked whether the information provided by the system makes any sense to them, and if they could recall of instances where they may have acted against the university policy. Finally, interview times were scheduled.

The interviews were scheduled so that students who were likely to have worked together were interviewed in successive time slots. The interview process was planned to consist of multiple steps, first showing the student a similarity matrix with one of the numbers being highlighted and telling them that their submission was similar to another student (or other students), then proceeding by showing the actual source codes, and finally asking for a potential explanation. Depending on the flow of the discussion, similar but not identical problems to those in the take-home exam were prepared for analysis.

As the students were contacted, a first type of plagiarism behavior – **help-seeking** – emerged. Multiple students responded to the initial email that another student in the course that they knew, typically a friend, had insisted for help, and that they could eventually no longer say no. These students were typically doing the exam in the same shared study space nearly at the same time, or, the student who was seeking help deliberately sought the other student out. In these cases, the students who were asked for help were typically performing well in the course, and the student who was asking for help knew of their performance. Students who had asked for help admitted their behavior.

The second plagiarism behavior – **collaboration** – emerged both through the data and through student interviews. Likely collaboration partners were identified through submission and programming process similarity, take-home exam start and end times, as well as possible previous collaboration in e.g. pair programming assignments. After contacting the students, some asked if they could come to talk either alone or together already before the assigned slot. Most of the students who were suspected for collaboration admitted doing so during the discussion, a typical approach was working on the exam in the same study space and discussing the problem with one another. Some initially denied any collaboration, but confessed afterwards.

The third plagiarism behavior – **systematic cheating** – emerged through individuals who did not respond to the contact request at all, and through students who did not have a clear colleague from whom they would have copied the solution. These typically emerged from abnormal correlations between the programming assignment scores and the take-home exam scores, as well as the analysis of the sources from which the exam questions were downloaded and from which the submissions were made to the assessment system. Here, typically, the students had acquired the exam questions before taking the exam, and had practiced the tasks beforehand. Such behavior was rather rare, and students who were suspected of such behavior admitted to their behavior.

Finally, a number of the students were acquitted during the interview. Such students were often high-performing, and could either easily explain the design decisions in the program, or explain how they would solve another problem, or both.

4.3 Analysis of Log Traces

To understand how these different plagiarism behaviors could be identified from the data, two methods were applied. First, we used an edit distance algorithm [24] to study how each student's distance to their final solution changed during the programming process. Then, we used a global alignment algorithm [16] to align each student's programming process with the programming process of other students.

The alignment algorithm creates a matrix of size $m * n$ for each student pair, where m is the event count for one student, and n is the event count for another student, and calculates the edit distance between all possible states. It then proceeds to identify the best alignment by constructing a path in the $m * n$ matrix that minimizes the path cost (edit distance) over the subsequent events from start to end. When the best alignment has been identified, the path cost is normalized by the number of events in the programming process to account for solutions of different length. Finally, the best alignment (here, average edit distance) for each student-pair is reported.

Note that as the calculation of the alignment is time-consuming when conducted on fine-grained data that contains each keystroke, only a subset of the events was used for constructing the alignments.

4.3.1 Help-seeking behavior and Systematic cheating. Two abnormal behaviors emerged during the analysis of the changes to the students' edit distances to their final solutions. There were students who had copy-pasted content to reach a solution, see Fig. 1, and there were programming processes that were very linear, see Fig. 2.

The first case (Fig. 1) is an example of a student who first attempted to construct a solution to the question alone, but could not do it. The student then proceeded to ask for help from an another student, who eventually provided code that the help-seeker could use. The second case (Fig. 2) is an example of a student who had struggled with the course beforehand, and had acquired a solution to the question. The student does not copy-paste the code, but it is obvious that the student is copying it from another source.

The first case is typical to help-seeking behavior, and the second case was seen in both behaviors (those students who had sought out for help, and those students who had acquired the question and worked out a solution beforehand). Figure 3 shows a pattern which was more typical to those students who worked on the assignment in a normal fashion – we will look into this in further detail next.

4.3.2 Collaboration. The alignment of students' programming process data resulted in pair-wise alignment scores for each student.

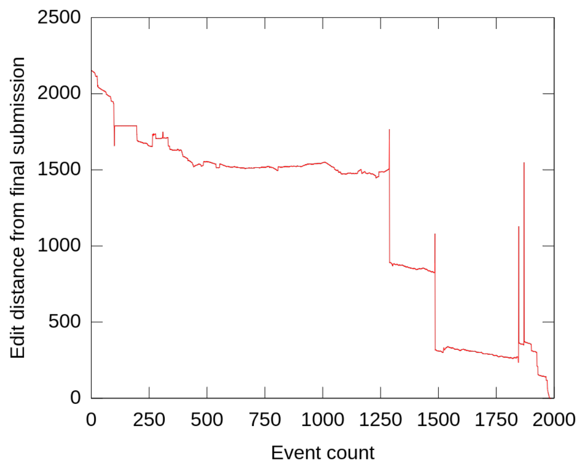

Figure 1: A student who first worked on a problem alone (events 0-1250). After being stuck for a while, the student asked for a solution from a peer. Parts of the received solution was pasted into the student's project to reach a final solution.

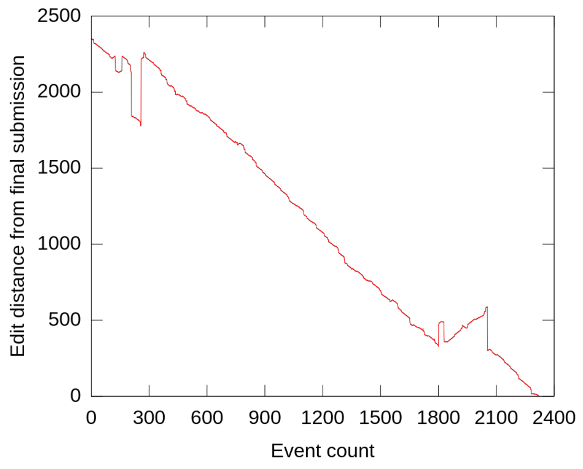

Figure 2: A student had previously asked and received a solution for the problem from a peer. The student mimicked the solution, but did not copy-paste it. At the end, the student modified the variable and method names, and somewhat reorganized the project.

Upon analysis, it became evident that the alignment of those students who have collaborated to reach a solution was significantly better than the alignment of random students. Such students could be detected using an outlier test [18].

A visual analysis of the alignments supported the finding. Figure 4 illustrates the average edit distance of two sample students to all other students in the class. In the Figure, the whiskers extend from each end of the box for a range equal to 1.5 times the interquartile range, and the box spans the range of values from the first quartile to the third quartile of the data. Outliers (detected by Gnuplot) are marked as dots.

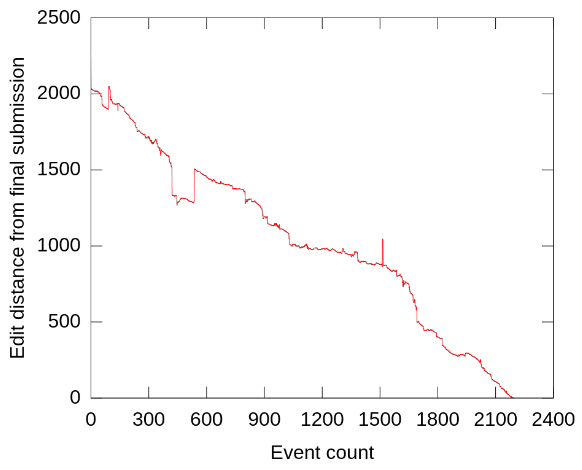

Figure 3: A student who had worked on the problem and eventually reached a solution. The behavior did not indicate copy-pasting or mimicking.

The student on the left hand side, i.e. "Candidate", had collaborated with another student; the collaboration partner was the one with the smallest average edit distance. The student on the right hand side, i.e. "Random", is a randomly picked student who was not suspected of plagiarism, no outliers were detected for the student. Rosner's Extreme Studentized Deviate test [18] confirmed the existence of two outliers for the candidate when the strictness (probability) was set as $p = 0.05$. When the strictness was increased ($p = 0.00001$), one outlier was identified. After removing the outlier, Kolmogorov-Smirnov test indicated that the data was normally distributed ($p > 0.15$).

5 DISCUSSION

5.1 Behavior traits

Multiple behavior traits were detected during the analysis. Help-seeking students were typically struggling in the course and sought to pass by begging for answers from their peers. These students did not always simply copy and paste the solutions that they received, but for example, mimicked the solutions by typing them in keypress by keypress; it was also typical to modify the variable names and method names in order to mask obvious copy-pasting.

Collaborators had planned out meeting together with other students to work on the take-home exams. They worked together to reach a solution. The programming process often contains missteps and rethinking the solution, and copy-paste behavior or mimicking code received from others is almost absent. Collaboration can be detected from the process logs if the student is collaborating with a peer who is also taking the exam. Their solutions and problem solving processes were typically very similar, and they had started the exam nearly at the same time.

Systematic cheating was related to e.g. harvesting solutions using a separate account. However, the students typically stated that it

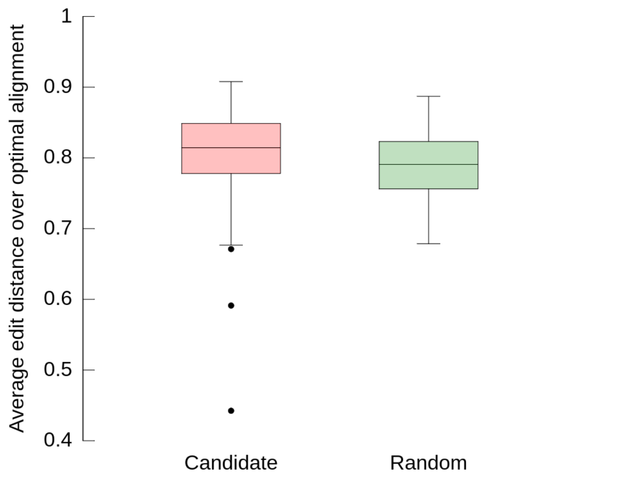

Figure 4: Two sample students and their alignment scores with every other student who worked on the assignment plotted using a box plot. The student labeled "Candidate" had collaborated with another student (the lowest point), whilst the student labeled "Random" had not.

was a "bad idea that they had in the middle of the night". Such behavior where students create multiple accounts in order to gain access to course content that is used for assessment purposes has recently been also identified in MOOCs [20].

When considering these behaviors, distinguishing between them is not easy, and it is possible that the same student shows multiple behaviors. For example, if a student had asked for a solution for a problem from a peer long before he or she started to work on the exam, it is a good question whether it should be considered systematic cheating. Similarly, collaboration was always planned and in that sense systematic.

5.2 Process data and course assignments

There has been an increase in the use of students' programming process data in research [10]. Such data shows plenty of promise in e.g. learning how students learn. However, as we have observed here, it is possible that the data that we gather is not an accurate representation of the students' knowledge.

When analyzing the students' programming process data, we observed that systems such as JPlag [17] are rather poor in detecting plagiarism from problems that are highly structured. If the assignment outline contains cues on class names, method names, variable names, etc., the solutions that students create are very similar. On the other hand, more variance is included in open-ended assignments. Both have also benefits and downsides in how easy they are to assess, and what types of tests one can write for them [9].

The use of process data as a part of the assessment and detection of plagiarism has multiple benefits. We could, for example, detect copy-paste events as well as programming behavior that shows that a student is mimicking a previously acquired solution.

5.3 Take-home exams

Our observations come from a set of take-home exams that have been used to assess students' programming competence. Take-home exams in the Computer Science education context have been previously studied by Leinonen et al. [13], who had access to data that was recorded from both the exam and normal course assigments. Their work has been more focused on identifying who is typing the code, whilst our work focuses on whether the student is doing the work on his or her own without help from others.

The take-home exam setup is an alternative version of the lab exam by Bennedsen and Caspersen, who note that such an exam *provides a valid and accurate evaluation of the student's programming capabilities, evaluates the process as well as the product, and encourages the students to practice programming throughout the course* [3]. Whilst in their context the students come to a specific time slot and lab for the exam, the approach is limited in terms of the number of students who can attend the exam. As a benefit, plagiarism is harder. In our context, on the other hand, there is no upper bound to the number of participants, and less resources and TAs are needed, but there are also more opportunities for plagiarism.

In an ideal setup, however, there would be no need for an exam. Currently, the line that separates the take-home exams and the course assignments from each other is the way that we expect the students work on them. Course assignments can be worked on with peers, but take-home exams should be completed individually. In addition, each take-home exam must be completed within four hours from the start of the exam, while the students have one week to complete each course assignment set.

One way to reduce the need for the take-home exam could – for example – be making it mandatory for the students to allow recording their programming process in the course. As a consequence, one could use algorithms such as the ones proposed by Ahadi et al. [2] to identify students who handle the course content well, and focus on only those students who struggle. This would have downsides though; some could plagiarize, and it is a good question whether the students would see such an approach as fair or ethical.

5.4 Limitations of study

Our study has multiple limitations, which we acknowledge next. First, as the exams were taken at home and other premises that the students chose to use, we do not know if we were able to reach all the students who did plagiarize. It is possible that, for example, a student had help from someone who is not on the course. It is also possible that we only caught those students who did not mask their plagiarism well enough.

Second, our setup has likely increased the tendency to plagiarize. It is likely that some of the students who participated in the take-home exam would not have attended a normal pen-and-paper exam at all. That is, the take-home exam opportunity can increase the tendency to plagiarize. Such behavior has been previously observed, for example, in business studies [26].

Third, a part of the students who we invited to the interviews were acquitted, and the final set of students who did plagiarize in the course is smaller than the interviewed population – we cannot claim that the behaviors that were identified during the interviews are a representative sample. Further studies, also from other contexts, are needed.

Fourth, we do not know whether the students plagiarized during the normal programming assignments as recording the programming process from the normal assignments was not mandatory. It is likely that information from such behavior would bring additional insight to those students who chose to plagiarize during the exams.

6 CONCLUSIONS AND FUTURE WORK

In this work we studied how students plagiarize in take-home exams. During the interview process with students who were suspected of plagiarism, three behavior types stood out: (1) help-seeking, (2) collaboration, and (3) systematic cheating. Students who were seeking for help were typically struggling on the course or in the exam, and insisted on help from their peers. Students who were collaborating worked on the exam together, and the students who were systematically cheating had e.g. created fake accounts to gain access to the exam questions in the submission system.

Furthermore, we analyzed the logs that were recorded as the students were working on the take-home exams, and found patterns that have the potential for identifying students who have plagiarized. A linear solution process in an assignment that is known to be challenging for students indicates plagiarism as does pasting parts of the solution. Collaboration where students in the same course are working together to solve the exam can potentially be detected through alignment of the programming process. At the same time, our approach is not a panacea, as we cannot – for example – identify students who have collaborated with external parties.

These results provide further means to detect plagiarism for instructors and researchers, and can be used as a discussion starter if an institution is considering the use of take-home exams. Furthermore, the recent ITiCSE working group on Educational Data Mining and Learning Analytics in Programming [10] indicated that more and more institutions are starting to gather such data. Researchers who are interested in, for example, building models of the student as a learner [27] or automatically identifying students who are struggling [2, 10] are likely to benefit from introducing plagiarism-related metrics to their models.

As a part of our future work, we are looking into incorporating data gathering into the whole introductory programming course in order to see to what extent the students plagiarize within the course assignments, and whether that behavior is linked with the plagiarism behavior in the take-home exams. We are also looking into additional means of reducing plagiarism on the courses, and considering the opportunity of removing exams completely.

REFERENCES

[1] Cheryl L Aasheim, Paige S Rutner, Lixin Li, and Susan R Williams. 2012. Plagiarism and programming: A survey of student attitudes. *Journal of Information Systems Education* 23, 3 (2012), 297.

[2] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. 2015. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the eleventh annual International Conference on International Computing Education Research.* ACM, 121–130.

[3] Jens Bennedsen and Michael E. Caspersen. 2007. Assessing Process and Product. *Innovation in Teaching and Learning in Information and Computer Sciences* 6, 4 (2007), 183–202. DOI: http://dx.doi.org/10.11120/ital.2007.06040183

[4] Kevin W Bowyer and Lawrence O Hall. 1999. Experience using" MOSS" to detect cheating on programming assignments. In *Frontiers in Education Conference, 1999. FIE'99. 29th Annual*, Vol. 3. IEEE, 13B3–18.

[5] Georgina Cosma and Mike Joy. 2008. Towards a definition of source-code plagiarism. *IEEE Transactions on Education* 51, 2 (2008), 195–200.

[6] Fintan Culwin, Anna MacLeod, and Thomas Lancaster. 2001. Source code plagiarism in UK HE computing schools. *Issues, Attitudes and Tools, South Bank University Technical Report SBU-CISM-01-02* (2001).

[7] Jurriaan Hage, Peter Rademaker, and Nikè van Vugt. 2011. Plagiarism Detection for Java: A Tool Comparison. In *Computer Science Education Research Conference (CSERC '11).* Open Universiteit, Heerlen, Open Univ., Heerlen, The Netherlands, The Netherlands, 33–46.

[8] Frederik Hattingh, Albertus AK Buitendag, and Jacobus S Van Der Walt. 2013. Presenting an alternative source code plagiarism detection framework for improving the teaching and learning of programming. *Journal of Information Technology Education* 12 (2013), 45–58.

[9] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research.* ACM, 86–93.

[10] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, and others. 2015. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports.* ACM, 41–63.

[11] Mike Joy, Georgina Cosma, Jane Yin-Kim Yau, and Jane Sinclair. 2011. Source code plagiarism—a student perspective. *IEEE Transactions on Education* 54, 1 (2011), 125–132.

[12] Thomas Lancaster and Fintan Culwin. 2004. A comparison of source code plagiarism detection engines. *Computer Science Education* 14, 2 (2004), 101–112.

[13] Juho Leinonen, Krista Longi, Arto Klami, Alireza Ahadi, and Arto Vihavainen. 2016. Typing Patterns and Authentication in Practical Programming Exams. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16).* ACM, New York, NY, USA, 160–165. DOI: http://dx.doi.org/10.1145/2899415.2899472

[14] Vítor T Martins, Daniela Fonte, Pedro Rangel Henriques, and Daniela da Cruz. 2014. Plagiarism detection: A tool survey and comparison. In *OASIcs-OpenAccess Series in Informatics*, Vol. 38. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[15] Merriam-Webster Online. 2016. Merriam-Webster Online Dictionary. (2016). http://www.merriam-webster.com

[16] Saul B Needleman and Christian D Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology* 48, 3 (1970), 443–453.

[17] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding plagiarisms among a set of programs with JPlag. *J. UCS* 8, 11 (2002), 1016.

[18] Bernard Rosner. 1975. On the detection of many outliers. *Technometrics* 17, 2 (1975), 221–227.

[19] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495.

[20] Jose A. Ruiperez-Valiente, Giora Alexandron, Zhongzhou Chen, and David E. Pritchard. 2016. Using Multiple Accounts for Harvesting Solutions in MOOCs. In *Proceedings of the Third (2016) ACM Conference on Learning @ Scale (L@S '16).* ACM, New York, NY, USA, 63–70. DOI: http://dx.doi.org/10.1145/2876034.2876037

[21] Johannes Schneider, Avi Bernstein, Jan Vom Brocke, Kostadin Damevski, and David C Shepherd. 2016. Detecting Plagiarism based on the Creation Process. *arXiv preprint arXiv:1612.09183* (2016).

[22] Judy Sheard, Martin Dick, Selby Markham, Ian Macdonald, and Meaghan Walsh. 2002. Cheating and Plagiarism: Perceptions and Practices of First Year IT Students. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '02).* ACM, New York, NY, USA, 183–187. DOI: http://dx.doi.org/10.1145/544414.544468

[23] Simon and Judy Sheard. 2016. Academic Integrity and Computing Assessments. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW '16).* ACM, New York, NY, USA, Article 3, 8 pages. DOI: http://dx.doi.org/10.1145/2843043.2843060

[24] Esko Ukkonen. 1985. Algorithms for approximate string matching. *Information and control* 64, 1-3 (1985), 100–118.

[25] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. 2013. Scaffolding Students' Learning Using Test My Code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13).* ACM, New York, NY, USA, 117–122. DOI: http://dx.doi.org/10.1145/2462476.2462501

[26] Tim West, Sue Ravenscroft, and Charles Shrader. 2004. Cheating and moral judgment in the college classroom: A natural experiment. *Journal of Business Ethics* 54, 2 (2004), 173–183.

[27] Michael Yudelson, Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. 2014. Investigating automated student modeling in a Java MOOC. In *Educational Data Mining 2014.*