Techniques for detecting and deterring cheating in home exams in programming

Guttorm Sindre
Department of Computer Science
Norwegian Univ. of Science and Technology (NTNU)
Trondheim, Norway
guttorm.sindre@ntnu.no

Børge Haugset
Department of Computer Science
Norwegian Univ. of Science and Technology (NTNU)
Trondheim, Norway
borgeha@ntnu.no

***Abstract*— The Covid-19 pandemic led to increased use of home exams, with a perceived increase of cheating. Assessment integrity is a key challenge for higher education. Apart from remote proctoring, what other mitigations may be possible against cheating in home exams, and specifically for programming courses with huge classes? The paper presents our approaches to mitigate cheating, for CS1 based on questions with subtly different variants, for CS2 based on plagiarism detection and timestamps – in sufficient detail that others could use a similar approach. These two approaches can be partially effective against collaboration, but less so against contract cheating where help is acquired from an outside third party. Hence, towards the end of the paper we also outline possible approaches to mitigate such cheating, without or in addition to remote proctoring.**

Keywords—computing, programming, home exam, cheating, academic integrity

I. INTRODUCTION

During the Covid-19 pandemic, most universities were prevented from having supervised written exams on campus. In many cases, the chosen substitute was home exams, students receiving questions and delivering answers remotely. The unsupervised nature of such exams led to a perceived increase of cheating risks [1, 2]. Remote proctoring may mitigate cheating for home exams [3, 4] – though not 100% [5]. However, for many universities – including ours – remote proctoring was not a short-term option and might also raise legal [6] and ethical concerns [7, 8]. Many universities thus opted for unsupervised home exams where mitigation of cheating risk would mainly have to be done by means of changes to question design [9].

Pandemic or not, on-campus or off-campus assessment – cheating and assessment integrity is a key challenge for higher education in general, and in some cases especially for engineering education, as tech savvy students might have a bigger than average repertoire of cheating techniques. The research question addressed by this paper is: *Apart from remote proctoring, what other mitigations may be possible against cheating in home exams, and specifically for programming courses with huge classes?*

The rest of this paper is structured as follows: Section II presents related work on cheating in home exams and its potential mitigation, with special focus on programming exams. Section III then describes our experiences with approaches to detect cheating in our CS1 and CS2 exams. Section IV makes an in-depth discussion concerning which types of cheating threats our approach can address, and which types it will not address – for the latter outlining other possible approaches. Finally, section V concludes the paper.

II. RELATED WORK

There have been many publications about cheating in computer science courses. Harris [10] discusses plagiarism from peers and other sources, arguing that prevention is preferable to detection. Sheard et al. [11] made a study of student perceptions related to cheating and plagiarism, later followed up by several other studies, such as [12] surveying strategies for maintaining academic integrity in first year computing courses. Hellas et al. [13] studied plagiarism and collusion in take-home exams, using a combination of plagiarism detection and process tracing with time-stamps. This is quite similar to the cheating detection approach used in our CS2 course, except their approach was for research, suspected students being invited to interviews so that their behavior could be better understood, whereas our approach was aimed at prosecution of cheating cases. Apoorv et al. [14] present their own tool for plagiarism detection in take-home coding exams, while we used the well-known tools JPlag and MOSS, which are among those analyzed in [15] and [16]. Jeffries et al. [17] present an approach combining plagiarism detection with MOSS and analysis of change traces for the delivered code. Another work looking at detection of cheating via time stamps is [18]. Albluwi [19] performed a systematic review on plagiarism in computing education. He found that there is much research on prevention and detection of plagiarism, less on the relationship between pressure felt by the students and inclination to plagiarize.

Karnalim et al. [20] discuss the same problem as we experienced for the CS1 course, namely that strongly directed assignments (e.g., where the solutions are small and rather straightforward programs) do not lend themselves easily to plagiarism checking because many students may plausibly have used the same solution approach. Their proposal is to use syntactic similarity detection in such cases, rather than semantic similarity detection, but this might not have been effective in our case, as several of the students who had copied code from others had been quite apt at making syntactic changes to copied code (hoping to evade plagiarism checking) while maintaining the semantics.

Considering the approach of having different variants of programming questions, as used for two tasks in our CS1 course, this has been suggested by other authors before. Fendler and Godbey proposed something similar for math questions [21]. Rusak and Yan [22] report on an approach to auto-generate unique exams for all students in a statistics course for computer scientists, based on changing numerical parameters. Fowler and Zilles [23] present an approach to making minor variations of programming tasks along several different dimensions: changing names of variables, changing

names of functions, swapping order of parameters, adding or removing a function prototype, changing constant values, reversing polarity (e.g., asking to find the last rather than first, smallest rather than greatest), and change of data type. Of these, we only used change of constant values.

A completely different approach to mitigating collusion is presented in [24], partly giving students the same questions, but not in the same order – and each question had to be solved in a limited time-slot. Hence, a student B hoping to receive help from another student A on task X, might discover that student A had not yet received task X by the time B needed to deliver it. As long as A was busy enough with own tasks in the same time-slot, help would then be hard to get. Such a mitigation approach was however not viable for our CS1 exam, as our university's e-exam tool did not support more granular time-slotting within the exam. Also, the approach in [24] was found to be most effective if question sequencing could be based on previous knowledge of student competency levels (so that more clever students would always receive a certain task *after* weaker students), and we did not have such information since the CS1 exam was the first ever graded test they took at the university.

III. Our Mitigation Approaches

In our university, computing students have their introductory programming course (henceforth abbreviated CS1) in their first term (Autumn), and then a more advanced programming course in the second term (Spring). Both these courses are also taken by students from many other STEM programs. CS1 teaches simple procedural programming in Python, CS2 object-oriented programming in Java.

Fig. 1. Rules as expessed on the first interaction page of the CS1 exam

In mid-March 2020, all university campuses in our country were locked down due to Covid-19, with a sudden shift to remote teaching and assessment. The CS2 course, previously assessed by an end-of-course Bring Your Own Device (BYOD) e-exam in supervised exam halls on campus, thus had to shift to an open-book home exam in May 2020. CS1 went the same way in December 2020, and both courses again in 2021. Our university uses the e-exam tool Inspera Assessment but does not use any kind of remote proctoring. Hence, these home exams were completely unsupervised. According to exam regulations, students were allowed to use any books and web sources – but **not** allowed to get help from peers or outsiders. These rules were communicated beforehand, and also included on the first interaction page shown in the e-exam system, where the students had to accept before continuing. Figure 1 shows the English version of this page for the CS1 exam / similar statements were used for CS2. Hence, it would be hard for students found guilty of collaboration to claim ignorance of the rules. However, the unsupervised nature of the exams made it practically impossible to control this.

Since the CS2 course preceded the CS1 course in the shift to home exam - and in the corresponding experiences with cheating - we present the CS2 experiences first, then CS1.

A. CS2: Object-oriented programming (Java)

In the absence of any pandemic, the course in Object-oriented programming (OOP) would have had a BYOD e-exam on campus, supervised by invigilators who would oversee that the students did not break the exam rules (e.g., collaborate, get help from outsiders). Moreover, the e-exam system (Inspera Assessment) would run on top of a lock-down browser (Safe Exam Browser) to prevent cheating via e.g., email or social media. Given a home exam, it was pointless to use the lock-down browser since a student intending to cheat could simply use an extra device for communication. Not using lock-down also had the advantage that students could easily use their preferred IDE, and search the Javadoc or other sources, which made the assessment more authentic. In practice, the exam was conducted in the following way:

1. Candidates would log in to the online e-exam system using their university account.
2. A Java project would be available as a zip file in Inspera. The exam project could also be installed via git prior to the exam, and the new version pulled. The exam questions were written using markup files.
3. In each java file, **//TODO** comments would indicate places where code was supposed to be written. They would work on this in their preferred IDE, outside the e-exam system.
4. The code also included a test harness with a main-method creating objects of classes and calling the student-written methods so that they could check if their code worked as intended. These tests would not include all requirements described in the question text or Javadoc for the different methods, but a subset of these.
5. At the end of the exam, students would then compress the same files to a zip-file which they delivered via the online e-exam system.

The fact that students (at least the successful ones) produced running code was also helpful for censors grading the answers, as it was much easier to grade based on a combination of automated testing and manual reading of code, than to just rely on manual reading. Using a more elaborate test suite than those available to the students at the

exam, we provided the censors with insights into where each student had issues needing special attention in the manual grading.

The investigation of cheating was mainly pursued by checking code similarity using the tool **JPlag**. A similar tool, MOSS[1], was briefly tested. MOSS requires you to upload code, and we considered this to be impractical for a course with nearly 700 students. The JPlag checks were provided with the predefined code, so the tool only considered similarities in the code added by students. One of the interesting features of tools like JPlag is its ability to ignore subtle changes in code, such as variable names and nuances in coding style. These were flagged as similar, enabling us to manually inspect the rest of the code.

In some cases, a high degree of code similarity between students could be plausibly explained by other factors than cheating. For a straightforward solution to a simple coding problem, it would be unsurprising if many students had similar solutions. The first 25% of the exam consisted of such small, simple problems, where only one of the problems was complex enough that a high degree of similarity might be suspicious. This was a string joiner method for which the teacher's model solution is shown in Figure 2. The students' answers would display a lot of variety here. For instance, a lot of students pursued the task in a more cumbersome way, transforming the iterable to a list and checking its length.

```
public static String join(final Iterator<String> strings,
        final String mainSeparator, final String lastSeparator) {
    final StringBuilder builder = new StringBuilder();
    while (strings.hasNext()) {
        final String item = strings.next();
        if (builder.length() > 0) {
            final String sep = (strings.hasNext() ||
                lastSeparator == null ? mainSeparator : lastSeparator);
            builder.append(sep);
        }
        builder.append(item);
    }
    return builder.toString();
}
```

Fig. 2. Example solution for CS2 task 1a, May 2020

The remaining 75% of the exam consisted of one bigger program featuring several collaborating classes, with problems that could be solved in a variety of ways. Here, a high degree of similarity would be even more suspicious than for the task shown in Figure 2. For space reasons, we do not show any model solutions or examples of student code for these bigger tasks.

However, even with such bigger tasks, two students might independently have found and adapted the same code from a web source for parts of the problem (which was allowed, since it was an open book exam). Hence, a manual check of similar code was necessary to determine whether similarity could have legitimate causes rather than sharing of answers among students. Manual inspection would then look for evidence in addition to the similarity percentage found by JPlag, such as similar mistakes, peculiar solution approaches, or similarities in variable names and comments, which could corroborate a suspicion of collusion rather than coincidence. We also had timestamps indicating when the students committed their last changes to each java file. Figure 3 shows an example of such timestamps for one student. Another student had very similar code, with the same order of file commitment, and timestamps a minute or less apart. Similar timestamps would not be evidence of cheating by itself (e.g., with clearly different code, no suspicion would be raised), but with suspicious similarity of the code, closely matching timestamps could strengthen the suspicion of collusion between two or more students.

IngredientContainer.java	2020-05-25 13:36
IngredientContainerTest.java	2020-05-25 10:00
Ingredients.java	2020-05-25 10:57
Kitchen.java	2020-05-25 13:06
KitchenApp.java	2020-05-25 10:00
KitchenController.java	2020-05-25 13:28
KitchenObserver.java	2020-05-25 12:59
KitchenTest.java	2020-05-25 10:00
Recipe.java	2020-05-25 13:59
RecipeReader.java	2020-05-25 12:18
RecipeTest.java	2020-05-25 10:00
ScaledIngredients.java	2020-05-25 13:50

Fig. 3. A series of timestamps indicating when a candidate committed various parts of the code. Note: the figure contains no information identifying the student.

All in all, of nearly 700 students taking the exam, 61 were formally pursued for cheating.

B. CS1: Introduction to programming (Python)

The CS1 exam mainly consisted of many short programming tasks. As an example, consider task 2g of the exam in December 2021, where the task was to write a function receiving L, a list of numbers, to be changed in place so that any adjacent duplicates of the same number are removed, for instance [1,1,2,2,2,1,3,3,2] → [1,2,1,3,2]. A straightforward solution to the task is shown in Figure 4. There are also a couple of other easy ways to solve it, so most of the correct solutions would likely be among 3-4 typical approaches, each having 4-6 lines of code.

```
def remove_dup(L):
    for i in range(len(L)-2,-1,-1):
        if L[i] == L[i+1]:
            del L[i]
```

Fig. 4. Example solution for CS1 task 2g, December 2021

With 2000+ students taking the exam, many would plausibly have similar code for a task with such short solutions. Each of the 3-4 typical solution approaches might have been used by a triple digit of exam candidates, only with small variations, for instance, some using the operator **del** to remove an element, others using list methods like **pop()** or **remove()**. Likewise, there would be many answers with similar mistakes, such as for-loops where the index goes out of range, or functions that build a local list without duplicates but fail to mutate the original list. Yet another frequent but wrong solution was conversion to a set, for instance L[:] = list(set(L)), which erroneously removes *all* duplicates, not just adjacent ones. Hence, for many solution attempts – right or wrong – high similarity score in JPlag would not be suspicious since many candidates could plausibly come up with such solutions, and mistakes,

[1]Information about these tools can be found at
https://jplag.ipd.kit.edu/
https://theory.stanford.edu/~aiken/moss/

independently. With short tasks such as this one, there would have to be something more peculiar about the code for similarity to be really suspicious.

Hence, for the CS1 exam, a supplementary approach was devised to detect cheating by collaboration, namely having several different variants of some of the tasks, namely for 2h (17 variants) and 2i (48 variants). The reason it was done only for two tasks was that it was somewhat cumbersome to handle such variants. Our e-exam tool lacked support for making many variants of the same question. Hence, the following work-around was used: (i) write a template task in the e-exam system, with variable names instead of the content to be parameterized, (ii) export this template task as a QTI file, (iii) have a self-made Python script generate a list of value tuples for concrete questions, (iv) have another Python script read the QTI file and generate numerous QTI files (one per variant), replacing variables with actual values, and (v) import the resulting QTI files back into the e-exam system as a question pool from which one variant could then be randomly drawn for each student. Exporting a template and then importing back instantiated variants guaranteed that the QTI files were in the exact dialect used by our e-exam system, whereas generating the QTI files in some other way might have run into problems with different implementations of the standard [25].

Figure 5 shows an example solution for one variant of task 2h, whose function receives a text string, and is supposed to return a list of characters, namely those that have 'd' 3 characters in front, and 'b' 2 characters after. For example, from the string 'abcdefSabcdefOabcdefSabcdef' the function should return the list ['S', 'O', 'S'].

```
def list_str(s):
    result = []
    for i in range(3, len(s) - 2):
        if s[i-3] == 'd' and s[i+2] == 'b':
            result.append(s[i])
    return result
```

Fig. 5. Example solution for CS1 task 2h, December 2021

The variation in this task implied that different candidates got different characters and distances to look for, e.g., another candidate might be asked to check for 'f' 1 character in front, 'd' 4 characters after. If such a candidate had not written own code but instead copied code from a peer who had a different variant (such as the 'd' 3 'b' 2 variant shown above), this would be quite revealing, then having wrong numbers and letters in the if-test, as well as wrong numbers for delimiting the range.

Since tasks 2h and 2i were in the genre "Code-Compile" in Inspera Assessment, where the student was able to check the code against a test suite during the exam, it was important that this test suite would not reveal that a wrong version had been copied (otherwise it would be too easy for a student simply to copy something from a peer and then just replace the values to have working code). Much like Moodle CodeRunner, the Code-Compile genre allowed hidden tests (not visible for the student) in addition to the visible tests. The tasks 2h and 2i were developed in a way that the visible tests were identical for all variants (and would thus work for correct solutions of all variants), whereas the hidden tests were different for each variant. Hence, a student who copied code from a peer with a different variant might seemingly have all tests showing green, but the code would fail at several of the hidden tests.

Approximately 30 of 2000 students were flagged as suspect of cheating due to this scheme, circa 1.5%. As cases are still pending final decision higher up in the system, it is impossible to say at this point exactly how many will end in a cheating verdict.

IV. Discussion

In addition to detecting cheating, it is even better to deter students from cheating in the first place, which includes making clear to them what the rules for academic integrity are [12]. For both our courses, there was clear information beforehand as shown in Figure 1, yet many students chose to cheat. As indicated in the previous section, the ratio of students caught cheating was much bigger in the CS2 exam in the spring semester (8%) than in the subsequent CS1 exam in the autumn semester (1.5%). It is hard to know whether this was because fewer actually cheated on the CS1 exam, or because the catch ratio was lower for other reasons. Some factors may indeed have caused fewer to cheat in the CS1 exam:

- Students at the CS1 exam may have been more scared of cheating, knowing that many were caught at the previous CS2 exam.
- While the CS2 exam had A-F grading, the CS1 exam only had Pass/Fail, so only weak students fearing failure would have substantial gain from cheating.

On the other hand, there may also be factors that could explain a lower catch ratio:

- Plagiarism checking was much less helpful for small programming tasks with short and straightforward solutions. In the CS2 exam, 75% of the weight was for a task with long and complex code, where plagiarism checking proved much more effective.
- The CS1 exam had only two tasks with variants, together accounting for 10% of the exam weight. Students who colluded on other tasks – but not on 2h and 2i – may have gone undetected.

Even some cases of collusion on tasks 2h and 2i could have gone undetected. Some colluding students could incidentally have the same variants of tasks, but only 1/17 for 2h, 1/48 for 2i – hardly explaining the difference between 8% and 1.5%. However, if two or more students sat together during the home exam, they may have discovered differences between the variants – after which the clever peer could easily explain to the others how to adapt the code. There may also have been cases of collaboration that did not entail direct copying of code, e.g., a student giving partial help to another, ("start by making an empty list, then iterate through the string by index, …"), but where the student receiving help also made some partial own effort, thus ending up with the correct values.

Our cheating mitigation approach for CS2 course was similar to other published approaches such as Hellas et al. [13] and Jeffries et al. [17], also relying on plagiarism checking and time-stamps. Moreover, similarities in mistakes, peculiarities, unusual variable naming, comments, etc. were investigated manually – but not for all students,

only for pairs of students who had already been flagged as suspicious by automated analysis. For the CS1 course, with much shorter code snippets, plagiarism checking was less effective – as also observed by Karnalim et al. [20], so students were mainly caught by having solved other variants of particular tasks than the ones they actually got. Of the 30 students caught cheating in CS1, only a couple were caught by plagiarism checking alone, while for the rest, the tasks with different variants were essential in establishing convincing evidence.

All the cheating cases revealed in our CS2 and CS1 exams were about collusion between candidates during the exam. Anecdotes suggest that there may also have been another approach to cheating taking place, namely getting help from a third party (not an exam candidate). There are various ways that students could acquire help from a more competent third person during a home exam. Some may get it for free, due to friendship or family relations. Others may buy it as a service, either from an acquaintance who might be physically present during the exam, or anonymously through a web service – so called contract cheating [26]. In a case study for computer science courses, Manoharan and Speidel found that good quality solutions for programming tasks could be bought on short notice, cheaply and easily [27].

The mitigation approaches that we employed would not be effective in discovering cheating by means of a third party helper. Table I shows a number of mitigation approaches, with the middle column "Collusion" indicating whether the approach can be effective against illegitimate collaboration between two or more peers who are taking the same exam at the same time, whereas the column "3p Helper" indicates whether it could be effective against cheating where the exam candidate gets help from somebody who is not a candidate for the exam. Some rows are marked as grey because the mitigations were not available to us for technical or legal reasons.

TABLE I. MITIGATION APPROACHES VS COLLUSION AND THIRD PARTY ASSISTANCE

Cheating mitigations, assumed effectiveness		
Mitigation approach	***Collusion***	***3p Helper***
Plagiarism checking	√	--
Time-stamps, traces	√	(√)
Variants of tasks	√	(√)
Time-slotting	√	--
Stylometry	√	√
Screen-casting	√	√
Remote proctoring	√	√
Post-exam vivas	√	√

Plagiarism checking is potentially effective against collusion, but not so much against third party help, since then the cheating candidate is not getting code from a peer, but from someone who is not delivering an own answer to the exam. The exception, of course, would be if this third party is helping several exam candidates in parallel, reusing solutions across candidates – or if the third party is a web service and several exam candidates have purchased help from the same web service.

Time stamps are only effective for collusion between two or more candidates. Instead getting outside help, there is no reason why a cheater should have time stamps that were suspiciously similar to those of other students (again with the exception of several cheaters getting help from the same outsider). The only exception would be if a candidate is observed to have solved a task impossibly fast, e.g. due to several outside helpers working in parallel.

Different variants of tasks are similarly geared towards mitigating collusion. The typical trap is that one student reads the task and solves it, then provides the solution to a peer. Not realizing that there are different variants, this peer then ends up submitting an answer to a question he did not get. With dedicated help from a third party, the helper will read the student's own version of the task, hence not fall in the trap of solving the wrong task. As argued by Jeffries et al. [17], different variants of tasks can in some cases help to detect contract cheating, for instance if cheaters use services such as Chegg where answers are visible also to other users, some of whom might then erroneously believe that the provided answer also solves their task. If teachers also register as members and look through answers – and variants are unique per student – one might even deduct from a provided answer who has asked for help. However, with less transparent services, or if a student gets solo help from a dedicated third person, variants of tasks will not help against this. Another problem with the approach of random drawing among variants of tasks is if the tasks turn out to have variation in difficulty. This is discussed by Fowler et al. in [30], who tried exams with tasks that were variants derived from the same template according to the approach described by Fowler and Zilles [23]. Variation in difficulty was found in some cases, though within ranges considered acceptable. Our approach to variation was rather limited compared to the one used by Fowler and Zilles. While they varied several aspects of the code, such as parameter names, function names, constant values, and "polarity" (e.g., finding the biggest / smallest, positive / negative, odd / even, …), we only applied changes to constant values. For instance, all students got task 2h as shown in Figure 4, solvable by the exact same Python function with a for-loop, except that the two integer constants (3 and 2 in the example) and two string constants (d and b) would vary from 1-6 and a-f, respectively. With such minor variations, the effect on difficulty would likely be negligible. With the CS1 exam we gave, changes to parameter order, variable names, function names, and polarity would anyway have been difficult – at least when using the Code-Compile question genre. These kinds of changes would then have failed the tests, making it too obvious to the student who had copied code that some minor adaptations would be necessary. Restricting ourselves to only varying constant values, it was possible to design the test suites such that all variants would pass all the student tests, but not the censor tests.

Time slotting would be to enforce different orders of tasks, e.g. candidate X must solve Task B 9:00-9:30, then task B 9:30-10:00, etc. while candidate B has a different order. It would then be harder for a candidate to get help from a more clever peer because the peer might be busy solving another task than the one the candidate needs help with. However, this only applies if the helper is indeed busy with any other task – a third partly helper who is not tasked

with any own exam, can dedicate all effort to helping the cheating candidate.

Stylometry could be used for an approach to cheating mitigation that is sort of inverse to plagiarism checking. Instead of investigating whether the material delivered by the student is suspiciously similar to something written by somebody else, it can be investigated whether the material delivered by the student is suspiciously different from what the same student has delivered before (and where the previous material was maybe produced under more controlled circumstances, so that its authenticity is certain). Alin [28] suggests a "Doping Test" approach where text sample 1 (known to be authentic) is compared to sample 2 (the work delivered for grading). If these are suspiciously different according to various style properties, the student is summoned to a test where sample 2 is provided to the student with various parts redacted and the student is asked to fill in the blanks. If the resulting sample 3 is again suspiciously different from sample 2, it may be prosecuted as a cheating case. While the approach is reported as promising, it may be more suitable for deliverables in natural language text than for introductory programming. Whereas most students have written text in natural language for many years, they may be beginners in programming – thus not having developed a style yet. As novices, they may improve a lot during the course, so it may not necessarily be suspicious if they have delivered clumsy program code early on but then deliver much better code for the exam. Moreover, if students gradually learn that being summoned to such an extra test means suspicion of cheating, they might memorize all their exam code – and filling in blanks in code is easier than writing the code from scratch.

Post-exam vivas, as also discussed by Alin [28] could still be an option for programming exams, having students explain orally some of the code they had written during the exam and asking them follow-up questions to investigate whether they really understood the code. Due to the pandemic, such oral examinations might also have had to take place remotely, yet it is more difficult to cheat in oral examinations [29] than in written examinations. The asynchronous nature of a written exam, where a question is received at one point in time, then answered at a later point (after some thinking, problem solving), makes it easier for the test-taker to forward the problem to the helper, who then looks at it and returns hints or a complete solution for the candidate to deliver. In an oral exam, there is less time to think – which means less time for the helper to produce any answers or hints in the first place, and less time for the candidate to digest the hints, since the answer must be spoken in the candidate's voice, not simply copy-pasted as could be done for writing. The challenge with oral examinations, is that they are hard to scale to huge classes – and thus out of the question for many CS1 and CS2 courses. Our faculty had a tentative plan that some students could be randomly drawn for post-exam vivas – not just in our course but also in other courses such as mathematics, where it was felt that home exams gave too easy opportunities for cheating. However, the faculty had to back down from this proposal due to concerns of legality and fairness. The proposal might also have been difficult to implement in practice. Some students may be good at coding, yet clumsy in oral interviews, so it would be hard to determine where to draw the line: Assume a student's interview response is of poor quality compared to the written code. How much poorer must it be to be considered evidence of cheating? It would be terrible if some students were unjustly accused of cheating just due to interview nerves and social awkwardness.

Remote proctoring could mitigate both collusion and third party help. For instance, surveillance through the webcam could see whether another person is sitting beside the test taker, and surveillance of audio through the computer's microphone could hear if somebody is speaking in the room. Also, the test-taker might be flagged if looking frequently away from the screen – and the screen could be surveyed and flagged if it shows anything else than the test interface (e.g., student communicating via email or social media about answers). However, such remote proctoring is far from 100% secure. For instance, the helper need not be in the same room, but could be in the next room or far away, collaborating remotely with the test taker. For programming exams – especially if rather long and complex code is to be written – it will be preferable to have candidates use IDEs rather than a generic e-exam system. IDE's like Visual Studio Code have extensions for online collaboration (Live Share), making it possible for a third party to write code directly in the student's editor. To hinder this, the remote proctoring system would need to be able to control the IDE settings, making sure that sharing features were disabled during the exam. Admittedly, most remote proctoring systems include features to either monitor or limit communications to and from the exam PC (except communication with the exam server), but the student could use a cable splitter to make the exam interface visible on an extra monitor in the next room – or to be forwarded elsewhere – so that the hired helper can also see the questions. Instructions from the helper on what to type could be relayed back to the student for instance via a hidden wireless earpiece – easily audible to the test-taker but not to the surveillance through the PC mic if there is some ambient noise in the room (e.g., traffic from the outside street, music from the apartment next door). Hence, even with sophisticated remote proctoring it is impossible to prevent that some students may get help from others, in the worst case having that other person solve the entire exam [5].

Screen-casting in Table I implies that the student delivers screencast video for the answers to some questions, either just the video, or as supplement to delivered written code. The video could be a straightforward, unedited recording of the student while writing exam code, with some thinking aloud to explain the rationale behind the code. It could mainly show the coding window, but with a small presenter window in one corner. This could be used to determine, by voice and image that it was indeed the right person writing the code. Of course, similar to remote proctoring, this would not prevent the student from getting help from somebody outside the camera angle, or remotely via a hidden earpiece. Moreover, since the student controls the screencast recording (i.e., deciding when to start and stop the recording) – whereas the remote proctoring is outside the student's control – the screencast approach would allow some extra cheating opportunities that are more difficult with remote proctoring. For instance, a third party helper could show the student how to solve the coding task, and coach the student about what to say "thinking aloud" while typing, and then the student could make the video. Or, if getting help from a peer, the student could watch the peer's video, then make an own with similar think-aloud. However, this takes time. With a pure written answer, the student could receive it

from a peer or third party helper and use it immediately. With a requirement to deliver screencast video, the cheating student needs to spend time understanding the received code or being coached what to say about it, and then also spend time making the video. Hence, students who are hardly able to solve tasks themselves and rely on a lot of help would struggle to complete the exam in time.

Compared to remote proctoring, screen-casting has the advantage of being cheaper. Students could use free screen-casting software, or software that the university already has a license for in connection with production of video lectures. Remote proctoring software , on the other hand, can be quite expensive and has no other usage than just remote proctoring. Another advantage is that screen-casting is less intrusive. As opposed to remote proctoring, where the student is under surveillance for the entire duration of the exam, the recording and delivery of a screencast video could be controlled by the student. If something personally sensitive happens during the recording, the student could redact that part or do it anew. Obviously, grading screencast videos would be much more demanding than just grading written code – and especially in cases where the code could be auto-scored against a test-suite, while the think-aloud explanation certainly cannot. However, grading could focus on the written code and the videos could be used mainly for control purposes. It could also be combined with other approaches such as having variants of tasks and time-slotting the various tasks in an exam so that different students get tasks in different orders. With several such mitigations together, cheating would be made increasingly difficult.

V. Conclusion

In our university we largely ended up with unsupervised home exams, and all such exams would be open-book exams, since it was anyway impossible to enforce a closed-book test in a home setting. Thus, one type of cheating (illegal use of textbook, cheat notes, googling for answers, …) was handily eliminated simply by allowing the behavior. Unfortunately, some other types of cheating (collaboration, getting help from an outsider) cannot be eliminated in similar fashion. If you allow students to get substantial help or outsource their exam to a third party, the grade might be totally invalid, reflecting the competence of the helper rather than the student. For courses with a limited number of students, oral examination via videoconference could be a more tempting option, as this makes outsourcing much more difficult. However, our large programming courses have way too many students for oral examination to be scalable. Also, several other mitigations such as time-slotting, remote proctoring, and post-exam vivas were unavailable to us due to limitations in the technical infrastructure, or due to legal and administrative constraints. Hence, we were in a situation where the possibility to prevent such forms of cheating was very limited, yet we had to do *something* to mitigate cheating.

The approach we ended up with, was mainly focused on plagiarism checking. For the CS2 course this checking was supplemented with comparison of time-stamps for committing solutions to various coding tasks. For CS1, a different technical setup meant that we did not have such time-stamps. However, this exam introduced another countermeasure: two exam tasks that existed in several variants. It turned out that these were vital to catching cheating in the CS1 course, as the rather short coding solutions meant that it was hard to derive convincing evidence for collusion from plagiarism checking alone.

Tasks with minor variations are only likely to catch cheaters if the variation goes unnoticed. If a similar approach is used several years in a row, the student population will probably learn to look for such variations, and thus take care to avoid the trap – either by adapting copied code to one's own variant, or seeking out someone with the same variant when asking for a solution in the first place (which the students can manage quite effectively if setting up a web site where solutions are shared, including notification of which tasks have multiple variants). It will therefore be interesting to see if an approach using variants will be effective several years in a row, or if it will catch gradually fewer cheaters.

The most notable weakness of our approach is that it mainly serves to catch cheating, not prevent it. Catching and prosecuting cheating is expensive, as it takes a lot of teacher time to collect and present the evidence, and then a lot of time to process the cases through the university bureaucracy and complaint boards. There are also legal costs. In our country, students accused of cheating are entitled to legal aid, and regardless of the outcome, the university is obliged to cover the student's lawyer expenses within reasonable limits. For the students found guilty of cheating, there are however big costs in other ways. In addition to having their exam annulled, they are expelled from pursuing any higher education in Norway for the next 6-12 months, so they typically have their studies delayed for a year, with downstream effects in terms of reduced life-time income. Indirectly, this is also a cost to society.

Hence, a natural question to ask is whether it was worth the effort to catch 60 cheaters in the CS2 exam and 30 cheaters in the CS1 exam. Catching cheaters is one of the saddest parts of a professor's job, and it would have been tempting to be able to use the time for research or course improvement instead. On the other hand, if cheating had not been pursued in these courses, the tell-tale experience among students would have been that a lot of students cheated in these exams, yet nobody got caught. This would have created an impression that cheating was not taken seriously in the university, possibly tempting these same students to cheat in their subsequent exams, too, as well as tempting subsequent cohorts of students to cheat in CS1 and CS2 next time around. Hence, we believe that considering long-term effects, it is worth the effort to pursue cheating. After all, the main victims of cheating are not the teachers, but the honest students who did not cheat – but who will see the value of their grades and study credits deflated if other students with lacking competence could achieve the same grades and credits by dishonest means.

Given that it is costly to pursue cheating, it would be much better to have mitigations that deter cheating up front, rather than detecting it after the fact – as deterrence might be less costly both for teachers, students, and society. Of course, this year's detection might to some extent serve as next year's deterrence – fewer students will be tempted to cheat in a course if hearing that quite a number of students were caught the year before. Especially, clever students who could end up cheating by sharing their code with weaker students, would likely think twice if hearing that many such cases were caught the year before. Such clever students have very

little to gain from the cheating unless there is some social or monetary reward from the weaker students they are helping, so they would likely do it only if they think there is minimal likelihood of being caught.

Still, it would be interesting to also have countermeasures that could limit the point of sharing code in the first place. One possible approach would then be to have tasks that differ a lot more than just being subtle variations, and where the students see that the tasks are clearly different, hence realizing that there is no gain from sharing code. However, a problem then might be increasing variation in the level of difficulty, beyond what was observed in [30], so that one would also need an approach to adjust for variation in difficulty when grading.

ACKNOWLEDGMENT

We thank Dr. Hallvard Trætteberg who was a key contributor to the exam design and cheating investigations for the CS2 course.

REFERENCES

[1] Bilen, E. and A. Matros, Online cheating amid COVID-19. Journal of Economic Behavior & Organization, 2021. 182: p. 196-211.

[2] Chen, B., S. Azad, M. Fowler, M. West, and C. Zilles. Learning to cheat: quantifying changes in score advantage of unproctored assessments over time. in Proceedings of the Seventh ACM Conference on Learning@ Scale. 2020.

[3] Gudiño Paredes, S., F.d.J. Jasso Peña, and J.M. de La Fuente Alcazar, Remote proctored exams: Integrity assurance in online education? Distance Education, 2021. 42(2): p. 200-218.

[4] Stapleton, P. and J. Blanchard. Remote proctoring: Expanding reliability and trust. in Proceedings of the 52nd ACM Technical Symposium on Computer Science Education. 2021.

[5] Geiger, G., Students Are Easily Cheating 'State-of-the-Art' Test Proctoring Tech, in Motherboard – Tech by Vice. 2021, Vice.

[6] Colonna, L., Legal Implications of Using AI as an Exam Invigilator. Faculty of Law, Stockholm University Research Paper, 2021(91).

[7] Kleeman, J., Remote Proctoring: Fairness and Compliance. ITNOW, 2020. 62(4): p. 58-59.

[8] Caines, A. and S. Silverman, Back Doors, Trap Doors, and Fourth-Party Deals: How You End up with Harmful Academic Surveillance Technology on Your Campus without Even Knowing.

[9] Nguyen, J.G., K.J. Keuseman, and J.J. Humston, Minimize online cheating for online assessments during COVID-19 pandemic. Journal of Chemical Education, 2020. 97(9): p. 3429-3435.

[10] Harris, J.K. Plagiarism in computer science courses. in Proceedings of the Conference on Ethics in the computer age. 1994.

[11] Sheard, J., M. Dick, S. Markham, I. Macdonald, and M. Walsh. Cheating and plagiarism: perceptions and practices of first year IT students. in ACM SIGCSE Bulletin. 2002. ACM.

[12] Sheard, J., M. Butler, K. Falkner, M. Morgan, and A. Weerasinghe. 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.

[13] Hellas, A., J. Leinonen, and P. Ihantola. Plagiarism in take-home exams: help-seeking, collaboration, and systematic cheating. in Proceedings of the 2017 ACM conference on innovation and technology in computer science education. 2017.

[14] Apoorv, R., et al. Examinator: A Plagiarism Detection Tool for Take-Home Exams. in Proceedings of the Seventh ACM Conference on Learning@ Scale. 2020.

[15] Misc, M., Z. Sustran, and J. Protic, A comparison of software tools for plagiarism detectionin programming assignments. The International journal of engineering education, 2016. 32(2): p. 738-748.

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

[17] Jeffries, B., T. Baldwin, and M. Zalk. Online Examinations in a Large Australian CS1 Course. in Australasian Computing Education Conference. 2022.

[18] Spanswick, E., M. Kastyak-Ibrahim, C. Flynn, S.E. Eaton, and N. Chibry. Data mining of online quiz log files: Creation of automated tools for identification of possible academic misconduct in large STEM courses. in European Conference on Academic Integrity and Plagiarism. 2021.

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

[20] Karnalim, O., M. Ayub, G. Kurniawati, R.A. Nathasya, and M.C. Wijanto. Work-in-progress: syntactic code similarity detection in strongly directed assessments. in 2021 IEEE Global Engineering Education Conference (EDUCON). 2021. IEEE.

[21] Fendler, R.J. and J.M. Godbey, Cheaters Should Never Win: Eliminating the Benefits of Cheating. Journal of Academic Ethics, 2016. 14(1): p. 71-85.

[22] Rusak, G. and L. Yan. Unique exams: designing assessments for integrity and fairness. in Proceedings of the 52nd ACM Technical Symposium on Computer Science Education. 2021.

[23] Fowler, M. and C. Zilles. Superficial Code-guise: Investigating the Impact of Surface Feature Changes on Students' Programming Question Scores. in Proceedings of the 52nd ACM Technical Symposium on Computer Science Education. 2021.

[24] Li, M., et al., Optimized collusion prevention for online exams during social distancing. npj Science of Learning, 2021. 6(1): p. 1-9.

[25] Piotrowski, M., QTI: A failed e-learning standard?, in Handbook of Research on E-Learning Standards and Interoperability: Frameworks and Issues. 2011, IGI Global. p. 59-82.

[26] Lancaster, T. and R. Clarke, Rethinking Assessment by Examination in the Age of Contract Cheating, in Plagiarism Across Europe and Beyond 2017. 2017: Brno, Czech Republic. p. 215-228.

[27] Manoharan, S. and U. Speidel. Contract cheating in computer science: A case study. in 2020 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE). 2020. IEEE.

[28] Alin, P., *Detecting and prosecuting contract cheating with evidence–a "Doping Test" approach.* International Journal for Educational Integrity, 2020. **16**(1): p. 1-13.

[29] Akimov, A. and M. Malin, *When old becomes new: a case study of oral examination as an online assessment tool.* Assessment & Evaluation in Higher Education, 2020: p. 1-17.

[30] Fowler, M., D.H. Smith IV, C. Emeka, M. West, and C. Zilles. Are We Fair? Quantifying Score Impacts of Computer Science Exams with Randomized Question Pools. in Proceedings of the 53rd ACM Technical Symposium on Computer Science Education. 2022.