Techniques for detecting or deterring cheating in home exams in programming

Guttorm Sindre
Department of Computer Science
NTNU
Trondheim, Norway
guttorm.sindre@ntnu.no

Børge Haugset
Department of Computer Science
NTNU
Trondheim, Norway
borgeha@ntnu.no

***Abstract*— During the Covid-19 pandemic, the use of home exams became widespread, with a perceived increase of cheating. Pandemic or not, on-campus or off-campus assessment – 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, each student sitting alone at home or elsewhere, 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]. A systematic review covering other works about plagiarism in computing education can be found in [19], observing 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.

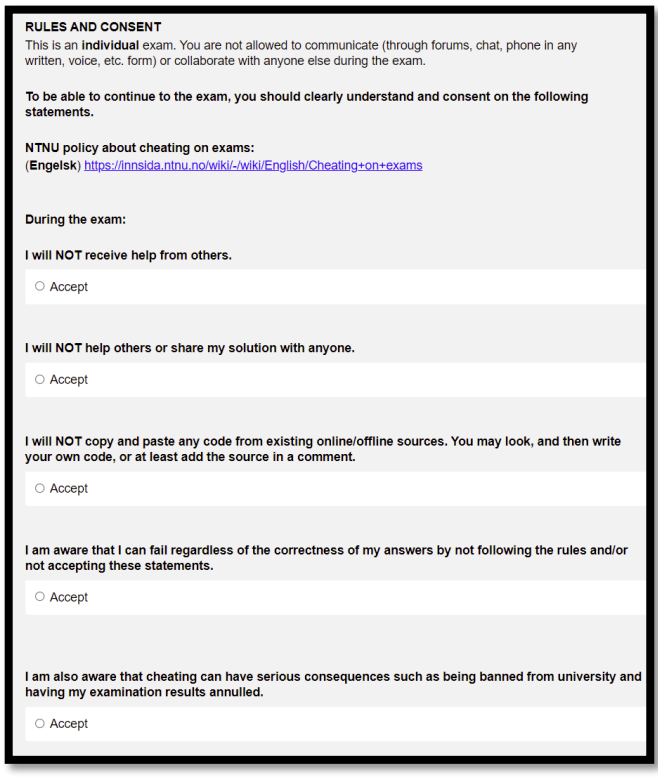

Fig. 1. Rules as expessed on the first interaction page of the CS1 exam

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 BYOD e-exam in supervised exam halls on campus, thus had to shift to an open-book home exam in May 2020, and 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. There, they would download a folder consisting of markdown files with the assignment text, and java files which they would edit to write their answer.
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.
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.

The investigation of cheating was mainly pursued by checking code similarity using the tools **jplag** and **MOSS**[1]. The check only considered the code added by students, not the predefined 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, and the first 25% of the exam consisted of such small, simple problems. 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 more suspicious. However, 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. Since we also had timestamps indicating when the students committed their last changes to each java file, this could also be used as corroborating evidence in some cases – for instance seeing that two students who had suspiciously similar code, also had completed the various tasks in the same sequence and almost simultaneously – for instance, typically less than a minute apart. Figure 1 shows an example of two candidates with closely matching time stamps for a sequence of tasks. Similar timestamps would not be evidence of cheating by itself (e.g., with clearly different code, no suspicion would be raised), but if suspicious similarity was found in the code, closely matching timestamps could strengthen the suspicion of collusion between students.

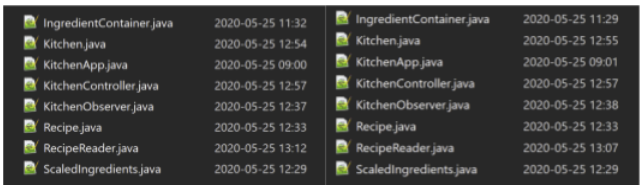

Fig. 2. Two candidates (left and right) with similar timestamps

All in all, of 700?? students taking the exam, ?? were flagged as suspicious for cheating, of which ?? were formally pursued. Of these, ?? came up with plausible explanations, so in the end 60 ?? were convicted of cheating, about 8% of the students taking the exam.

B. CS1: Introduction to programming (Python)

The CS1 exam mainly consists of many short programming tasks. As an example, consider task 2g of the exam in December 2021, where the task is 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, e.g. [1,1,2,2,2,1,3,3,2] → [1,2,1,3,2]. A straightforward solution to the task is shown in Figure 2 – as the function mutates the list, it does not need to give a return value. The for-loop iterates the list back to front, since the similar front-to-back loop would have the index i go out of range as the list shrinks due to deletion of elements. A front-to-back while-loop is viable, though, deleting the element if it is a duplicate, else leaving it in place and incrementing i by 1. Yet another typical solution would be to make a local list variable to which non-duplicate elements are iteratively appended, then after the loop, mutate the new list into the old one, e.g., L[:] = new_list.

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

Fig. 3. 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 three alternative solution approaches mentioned above might easily have been used by a triple digit of exam candidates, only with small variations, e.g., 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 does go out of range, or functions that build a local list without duplicates but fail to mutate the original list by the contents of the new list, perhaps in the end returning the result instead. Yet another frequent but wrong solution was conversion to a set, e.g. 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, 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 3 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. E.g., from the string 'abcdefSabcdefOabcdefSabcdef' the function should return the list ['S', 'O', 'S'].

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

```
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. 4. 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 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. Anyway, it can be noted that 1.5% for the CS1 course is much smaller than the 8% earlier caught for CS2. It is hard to know whether this was because fewer actually cheated on the CS1 exam, or because the catch ratio was lower. There are some differences between the two exams that could plausibly lead to less cheating:

- Students at the CS1 exam may have been more scared of cheating, knowing that many were caught at the preceding CS2 exam.
- While the CS2 exam had A-F grading, the CS1 exam only had Pass/Fail, so the possible gain from cheating would be limited, expect for the weakest students.

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. Obviously, 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.

IV. Discussion

As explained in the previous section, our approach against cheating differed between the two courses. In the CS2 course, plagiarism detection was effective in many cases, as 75% of the exam tasked the students with writing long and complex code. Similar to other published approaches such as Hellas et al. [13] and Jeffries et al. [17], we also incorporated time-stamps as additional evidence for collusion. 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 then ones they actually got.

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. Likely, there were more students who cheated than the number that were caught, as more sophisticated code sharing would easily go beneath the radar of plagiarism detection. Moreover, beyond code sharing there is another approach to cheating which is hardly addressed by the countermeasures discussed here, namely getting help from a third party who is not a student taking the exam in the extreme case having somebody else take the entire exam for you. The mitigation approaches discussed here are not necessarily effective against this as they mainly target cheating by collaboration:

- Collaborating students easily end up with similar code and timestamps, while a student who has a dedicated helper need not have similar code to anybody else.
- Collaborating students reveal themselves by delivering code solving other variants than they got of certain tasks. The student with a dedicated outside helper would however have the right variants all the time.
- The approach with sequencing and time/slotting of tasks as presented by is effective against collaboration because other candidates are busy with their own tasks. A helper who is not burdened with an own exam at the time, will not be hindered by this.

There are various ways that students could acquire help from a third person during a home exam. Some may get it for free, for instance being in a friendly or romantic relationship

with someone more competent in the subject matter. 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 with programming tasks could be bought on short notice, cheaply and easily [27]. 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 other types of services, or if a student gets solo help from a dedicated third person, variants of tasks will not help against this.

Alin [28] discusses various other mitigation attempts against contract cheating, such as post-exam vivas where students are asked to explain their answers, or comparing the text the student delivered from the home-exam with previously delivered text which is known to be written by the student (e.g., having been delivered in more controlled circumstances). Specifically, a "Doping Test" approach is suggested, 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.

As mentioned in the introduction, remote proctoring could mitigate assistance from any helper. E.g., surveillance through the webcam could see whether another person is sitting beside the test taker – whether this is a peer student, parent or hired expert – 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 (e.g., a close friend) or far away (a remote helper). 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 anyway some ambient noise in the room (e.g., traffic from the outside street, music from the apartment next door). Hence, even with 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].

So what if you do not have the opportunity to gather program code guaranteed to be written by the student earlier in the semester (if you had such an opportunity, you might have the exam itself under supervised conditions instead), and you – or your university – do not want to use remote proctoring, but you still want to do something to mitigate cheating by help from a third party? Another possibility lies in the observation that effective cheating (i.e., achieving huge grade advantage) is much more difficult for oral examinations [29] than for written examinations. Of course, a candidate who is examined remotely via video conference, may have a helper in the same room, or a device for communicating with a helper elsewhere – as long as the helper or device is kept outside view of the webcam. However, the resulting gain from such help may be much smaller, due to the live interaction between the student and examiner. A written exam, which is more asynchronous in that a question is received at one point in time, then answered at a later point (after some thinking, problem solving), makes it easy 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 hint 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 big problem with oral examinations, of course, they are hard to scale to huge classes – and thus out of the question for many CS1 and CS2 courses. However, what if we could mimic the immediacy of the oral examination in a written examination? For instance, imagine an exam containing problems with possible solutions such as those shown in Figure 3 and 4, but where the students were tasked not only with delivering the written code, but also were supposed to deliver for each problem a screencast video of 10 minutes duration where they think aloud while writing the code to solve the problem. This would make cheating harder in the following ways:

- Collusion between candidates would be harder. An incompetent candidate who receives code from a stronger candidate must likely also watch the other candidate's video before being able to make an own think-aloud video while retyping the code – adding a delay of 10-15 compared to just copy-pasting for a task that only has written delivery. If needing to cheat like this on many tasks, the weaker candidate will likely run out of time with the latter half of tasks unanswered, thus gaining much less from the cheating.
- Getting help from a third party will similarly be delayed. Although the competent third party helper – who is not burdened with delivering an own exam answer – does not need to make any video, the less competent test-taker will need to pretend to write the received code from scratch, and think aloud while doing so. If the cheater does not understand the

received code, the think aloud will easily be suspicious, unless the competent helper either sends code that includes a natural language think-aloud script or coaches the candidate on what to say. Either way, this scheme increases the burden on the helper (who must provide code + explanation, not just code), and creates delay for the candidate who must first receive this material, then make the video pretending to make the code from scratch.

As opposed to remote proctoring, where the student is under surveillance during the exam, the recording and delivery of a screencast video could be controlled by the student. Hence, if something happens during the recording which the student does not want the rest of the world to see (e.g., a friend busting into the room proposing a drug deal or criticizing the country's authoritarian regime in the midst of recording), the student could redact that part and do it anew. The downside of such a screen-casting approach, of course, would be that 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, e.g., flagging as suspicious (possibly by help of some AI) videos where explanation is thin or in the wrong voice. Also, for students whose auto-score is just below the passing threshold, the voice explanation could be used to assess whether the student – in spite of a wrong solution – demonstrated enough understanding to deserve a passing grade.

Such a screen-casting approach would likely demand more student time per task. Although the think-aloud could be recorded while the student solves the problem anyway, not demanding any extra editing of the video, most students would likely work a little slower if having to speak while coding rather than just code. Hence, it might only be usable for some select tasks, not the whole exam. It can of course also be combined with other approaches such as having variants of tasks and time-boxing 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 – though still not impossible.

A potential problem with the approach of random drawing among variants of tasks is if the tasks turn out to have variation in difficulty. This problem is discussed in [30], where variation was found in some cases, though within ranges considered acceptable. Our approach to variation was anyway somewhat more limited, only amounting to changes of 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 minor. However, a potential problem with such minor variations is that they 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.

TABLE I. MITIGATION APPROACHES

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

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. 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.

ACKNOWLEDGMENT

We thank Dr. Hallvard Trætteberg who was strongly involved in the exam and cheating mitigation approach 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.