Single Source Plagiarism Detection in Intro Level Computer Science Classes Using Keystroke Metadata

Abstract

Plagiarism in intro level computer science courses is a problem. The ability for students to pass classes by copying assignments without learning the material is hurting the industry as a whole. Previously available tools for combating plagiarism are failing as online resources increases in quality and availability. These can range from publicly accessible solutions, to pay-walled solutions, and even to bespoke custom made solutions individual to each student. Freely available generative AI capable of solving intro level problems further exacerbates this problem. Thus we propose a novel approach where plagiarism is determined from metadata embedded by a specialized IDE into student submissions. We find this approach is generally more reliable than traditional plagiarism detection methods in both theory and practice.

1. Introduction

Introductory Computer Science classes are often students first look at programming. In these classes people who dont know how to program, learn to program for the first time. Or that is the assumption. However, due to the nature of computer science it is often possible to pass an intro to computer science course without writing a single line of code by plagiarizing assignments.

1.1. Plagiarisms Logical Consequences

Plagiarism is particularly troublesome in intro classes. Since the intro classes are often a students first experience programming, if a student passes an intro class by plagiarizing it is likely that student does not understand part or all of the course. Since higher programming courses assume you know the basics, these students have only 2 choices going forward. Firstly they can work hard and put in extra time to teach themselves the missed information. This is unlikely, as this student has already demonstrated they are unwilling to or unable to learn the basics by cheating on them to begin with. Or Second, they must keep plagiarizing. This option ultimately leads to a student who does not know how to program potentially graduating with a Computer Science degree. If this student attempts to use this degree to apply for a job, the interview process, or, even worst, the opening weeks of a job, will reveal this lack of knowledge. Such a revelation is harmful to the entire community of computer science as it erodes confidence in the degree, and raises the bar for "entry level" job experience.

1.2. Prevalence

One may ask, "How prevalent is plagiarism in intro CS classes." So did we, In an single 60 student intro level class during 2020, we observed an assignment where 10 people turned in exactly the same assignment, and nearly a third of the class had recognizable plagiarized assignments. This assignment will be referred to later as the "Motivating Case." [[xxxxx Find papers referencing plagiarism]] Additionally, we have known many high level computer science students who don't seam to understand basic programming. While this lack of understanding may not point to plagiarism, that seams a likely culprit.

2. Definitions

Plagiarism is generally defined as "The practice of taking someone else's work or ideas and passing them off as one's own."[1] However, for this paper we will define it exclusivity as "coping someone else's code to complete an assignment." We exclude copying ideas for several reasons, mainly because it is incredible difficult to determine if an idea has been copied or just independently rediscovered; the general culture of computer science is to iterate on others ideas (often without attribution); and because in the intro level, all the required ideas have been thought of by so many people determining an origin would be next to impossible. Additionally, we exclude the possibility a student could "cite" their copied code to avoid plagiarism because the intro level is attempting to teach coding, not copying existing code. This definition works well for the intro level, however in higher levels it becomes less useful. We will use cheating and plagiarism synonymous in this paper. We would also like to define some theoretical plagiarism "vectors."

2.1. Vectors

- **P2P** Peer to Peer file sharing seems the most obvious vector. This is where one student writes the code and shares it with a peer who then submits the copy. In this instance one student is the author, and the other is the plagiarizer. Although both students are at fault, the plagiarizer is more so than the author.
- **Collaboration** Collaboration is a potential vector for individual assignments where two students work on the assignment together even though it is intended to be a solo exercise. Collaboration is a particularly tricky vector as some level of collaboration is generally acceptable. This vector is distinct from P2P because neither student can be pointed to as a distinct "source".
- **Theft** While technically a subvector of P2P, Theft warrants its own definition. Theft is any form of P2P plagiarism where the author does not know that it is happening. For example, a student (the author) works on a shared computer and leaves a file saved locally, and the plagiarizer finds the file when attempting the same assignment.
- **Search** Search engines, such as Google, are powerful tools, one of their powers is the ability to take an arbitrary string (Say, the title of a homework assignment for example) and get a useful result (like say, a github repo with the solution in 5 different programming languages). We are defining any time a plagiarizer uses an internet search engine to find an existing solution to an assignment to be this vector of plagiarism.
- **Expert** Expert sources also exist. These generally fall into 4 categorizes.
  - Paywalled answer repositories: This include sites like Chegg, or Coursehero. On these (and similar sites), students can upload assignments to be done by expert programmers for a fee, or peruse other expert solutions hidden behind a paywall.
  - Freelance Coders: This includes any professional coder who will write code for student assignments.
  - Unethical Tutors: This includes any person who purports to be a tutor, but completes assignments directly instead of guiding their student.
  - Large Language Models: This includes openly available generative models that can write code such as ChatGPT or Google's Bard. These have gotten to the point where they can generate a bespoke solution to almost all intro level assignments if given an assignment as a prompt.

  These are distinct from the Search vector for 2 reasons, first it is generally not free (or at least not freely accessible), and second the solution is tailored by the expert to the exact assignment given. This renders any attempt to "Scrape" answers futile. This appears to have been the primary vector in the Motivating Case, the 10 identical submissions were all sourced from a Chegg "Expert Answers" Pay-walled solution repo.

It should be noted that all of these vectors can affect anything from single lines of code to entire assignments, and multiple vectors might be present in a single assignment. This makes it difficult to detect when they are happening, not only because it involves detecting plagiarism, but also because students are encouraged to use many of these vectors to aid their own work. As such we will be focused more on substantial cases where a potential plagiarizer acquires a significant amount of an assignment from the vector. For example, if a single line of code or a non required helper function is copied, probably not a problem; but if an entire required function is copied, that is a problem.

3. Current Approach

There are many existing research projects attempting to detect Plagerism The general method of plagiarism can be summarized as follows. Given a set A of submissions find a set P such that $\forall p \in A$, if $\exists q \in A$ such that $q \neq p$ and $D(p, q)$, then $p \in P$ and $q \in P$ for some plagiarism detection function D. Defining D efficiently can be tricky, however several researchers have done so.

Stanford university developed a program called MOSS (Measure of Software Similarity) which is capable of giving each assignment pair a similarity score. This approach is robust against name changes, code reorganization, and white space changes by analyzing a semi random subset of k-gram hashes based on each assignment. k has been manually tuned for each language.[**?**]

There are many other efforts to define D using a machine learning model.[[citations]]

Unfortunately, regardless of the definition of D used, all of these models have similar results when attempting to detect a given vector.

3.1. Theoretical Per Vector Performance

3.1.1. P2P. P2P is handled fairly well by these models. This vector is well suited for a source code analysis so long as D is resistant to minor changes and all sources used are submitted by someone. However, these models can not detect the author specifically.

3.1.2. Collaboration. Similarly to P2P, Collaboration is often detected by these models as the source code is quite similar. D however is harder to design for these as the "minor" changes can become more extreme with collaboration than with copying. Additionally, when they do detect plagiarism, these models can not discriminate between P2P and Collaboration.

3.1.3. Theft. Theft is as easy to detect as P2P and these models detect it well, however they can not discriminate between the plagiarizer and the victim, nor can they discriminate between P2P and Theft.

3.1.4. Search. Search is where existing models begin to substantially fail. No matter how sophisticated D is, it requires all potential sources to be present for consideration. Given the vast number of potential sources that can be found on the internet, detecting this vector is extremely difficult for D and generally requires a human to put in the leg work to scrape together the list of potential sources. Additionally, these searches generally take $O(s^2 + se)$ Time (where s is student submissions, and e is external examples), for large values of e, this can be prohibitively long.

3.1.5. Expert. Expert is where existing models unambiguously fail the most. A properly executed Expert vector is indistinguishable from a legitimate attempt for all possibly D. As this content is Bespoke or behind a Paywall, it is genuinely impossible to provide the model with all relevant sources. The only way these models can detect an Expert vector is if 2 students both submit the same Expert source by accident.

3.2. General Shortcomings

In addition to the per vector performance, all of these models suffer from frequent false positives especially in the intro level. Consider Hello World in java. There is effectively 1 way to write hello world in java, you have your class, your main, and your print. This is similar for nearly all other languages. Since nearly all intro classes start with hello world, if the first assignment is checked with the above methods, it is likely that every student will be marked as plagiarized, even if they all attempted the assignment properly and individually.

3.3. Notable Alternatives

There are other existing models that can potentially address these problems. One research group used machine learning to detect stylistic variation in essays. [**?**] This approach has potential, especially when applied across a student's entire code corpus, and might be able to detect a change of Author compared to other assignments. However, this vector is still weak to a serial plagiarist using a single source consistently. Additionally our target is intro level students who may not yet have a defined "style." We believe it would be reasonable to suppose an intro level student's style would drift significantly during their first few courses. As such, we did not consider this approach.

There are also hyper-focused detection models that attempt to exclusively a detect single LLMs output.[**?**] However, these are not particularly reliable at present[**?**] and have a high false positive rate. [**?**] At the time we started this research, LLMs were not a vector, and the hyper-focused and unreliability of these have prevented us from giving serious credence to this approach.

4. Our Approach

Single Source Plagiarism Detection For our approach, we decided that it is more useful to be able to determine if an individual file its self is plagiarized, rather than a collection. To enable this, we created a unique IDE based on the Processing IDE. When the IDE first launches it creates a persistent UUID on the machine that installed it. This UUID acts as a machine or user ID, and is referred to as the InstallID and is assumed to be constant throughout the class. Additionally, whenever a project is created with the IDE, a second UUID is created unique to that project, we refer to this as the ProjectID. Both UUIDs are saved in a special hidden metadata comment in the program source file, we will refer to this as the metaComment.

Whenever a file is opened, the IDE checks the InstallID of the file and machine. If they differ, the IDE notes the new InstallID in the metaComment in an ordered list called the InfectionStack. A similar stack is also used to track paste events. As these stacks are effectively one stack with a way to discriminate between event types, we will use InfectionStack to refer to both.

Additionally, when any part of the code is copied zero width spaces are interspersed with the normal letters to encode as much data as possible. The exact amount of data varies based on the size of the copy, but it can include several copies of the InstallID, ProjectID, and InfectionStack. When pasted into a project, the encoded data is compared to that of the project reviving the code, and any mismatch UUID's are added to the InfectionStack. This encoding survives being sent over most messaging software such as Discord, and is generally resilient to partial pastes. Any paste without encoded data is logged as originating outside the IDE.

Regrettably all email clients we tested strip out the zero-width spaces, as such emailed code copies register as originating outside the IDE.

In addition to copy tracking, the IDE also logs the time, location, and content of all edit events including copy, paste, cut, delete, and typing. Any mismatched UUIDs are included in the log for the respective paste event. This data is also included in the metaComment. Using this keylog, it is possible to reconstruct the entire coding process and trace all code pastes.

Lastly, if the IDE loads a file without a metaComment, it notes this as the first edit and logs the initial state of the file.

While it is can be enough to use a single file to detect Plagiarism, the UUIDs can be used to trace code authorship through multiple student submissions. Additionally, analyzing an entire class at the same time can be useful to identify false positives such copies from previous assignments on the students machine.

4.1. Theoretical Per Vector Performance

This method was designed with our vectors in mind, and performs well on all of them.

4.1.1. P2P. As with existing methods, P2P is easy to detect. There is no easy way to submit someone else's unmodified file without the UUIDs being a clear red flag. Any attempt to edit the file will mark it as infected. Intro classes frequently have students add an "Author" comment to the top of their files. This would be need to be edited by anyone attempting to cheat, and the original comment is preserved in the log. Finally any code shared via online messaging will be flagged at they very least, and generally, traced. If a student copies from a previous assignment, it will be traceable to which, and who authored that assignment.

Even simple files such as Hello World would be detected, as the detection does not rely on code comparison. Additionally, our method is immune to traditional false positives as organically created code is never considered plagiarized regardless of similarity to another student.

4.1.2. Collaboration. Collaborating students will have similar code with overlapping timestamps at each edit. Additionally, collaborating students tend to send code back and forth, meaning both side's InfectionStack will be populated with each other's UUIDs. This back and forth makes it easy to distinguish from P2P.

4.1.3. Theft. Theft is detected in the same way as P2P. Unfortunately it can be tricky to discriminate it from P2P, but it is now theoretically possible with meta knowledge of the computer setup (i.e. class computers would have a known InstallID) and the InfectionStack combined with timestamped log make it possible to isolate 1 copy as the origin IF it is edited on the same or a different computer.

4.1.4. Search. All Search vectors are soundly defeated. No matter how a file is found, once it is loaded by the IDE, it will be marked as external. If it is submitted without being opened by the IDE, it will lack a metaComment. If the source code is copied into an existing project, it will be marked as external. The only viable source that this does not detect is physically typing out the code from a reference. This is easily detected by analyzing the timestamp data. Organically written code has a lot of back and forth. For example, Java's Hello World tends to be typed as

```
class HW{

}
[UP ARROW]
  public static void main(String[] args){

  }
[UP ARROW]
    System.out.println("Hello World");
[RUN]
```

Where as a plagiarist copying it by typing would likely write it top to bottom without backtracking significantly.

4.1.5. Expert. Expert sources are detected in the same way as Search. The only way an Expert can avoid detection as an external source is by installing the IDE and using that, in which case *they* will have a different InstallID from the student. This may require multiple assignments to detect, but is likely reliable over a full class unless the student finds 1 consistent Expert, or the Expert uses the student's own computer.

4.2. Evasion

There are notable ways to avoid detection by this method. In general we have dismissed them as "being more work than just *doing* the assignment." The metaComment is a plain text java comment appended to the file. Any text editor *other* than our IDE will show this comment and allow it to be edited. Convincingly faking the edit log would be incredible difficult, and deleting the entire string is easily detected.

A more viable method would be to share the InstallID file with an Expert, however, you would have to know that the InstallID exists, and where it is saved.

A corrupted metaComment is another possibility. The comment could be tampered with in such a way that it is incomplete. Sadly, this occasionally legitimately does happen so this is a legitimate concern.

We have dismissed these weaknesses reasoning that if an intro level student discovers *and* is able to exploit them, they likely know enough to not *need* to plagiarize in the intro level. And as the program is currently not well known, there will be no external assistance in doing so.

A potential bypass is to complete the assignment via any means that doesnt add to the InfectionStack, and then copy and paste the file content into another copy of the IDE, this then gets logged as an internal paste events between assignments and all keystroke logging is lost. This method we dismiss as there is little reason to do it (other than IDE bugs) and it is easily detectable as an event of interest.

4.3. Automated Detection

It is all well and good to wax eloquently about the theoretical ways the data can be used to detect plagiarism, and quite another to do so in an automated fashion suited to someone grading dozens of assignments. As such, we also developed a script that takes all student submissions for a given assignment and analyzes them. This script was developed after manual analysis of the Case Study, and as such, we did not implemented detection of the features not present in the case study. Notably, we did not implement the timestamp based Collaboration detection as there was no Collaboration. Additionally we did not implement the linear coding detection as only 1 student used it and there were not enough students who had not plagiarized to define a threshold for how linear is too linear.

The Script will take in a directory of submissions. First it will build a graph using the InfectionStack in an attempt to trace shared files. At the same time it builds a list of shared machines, which is used later. Any connections discovered at this point will be noted. Next the script goes student by student and checks all paste events. Purely internal paste events are ignored. From there pastes fall into 4 categories: pastes originating from a project that was turned in by another student, or pastes originating from a different students machine; pastes originating from the same machine, but a different project; pastes that originate from an InstallID that is not associated with any project turned in; and pastes of external origin.

Pastes from another turned in ProjectID or InstallID that is asociated Pastes that originate on the same machine are treated differently depending on the machine. If the machine is from a machine that was not shared,

5. Case Study

During the Spring 2023 ICS 111 and 211 Intro to Java classes, we offered an extra credit assignment to students. [[XXX what do I have to do to reference the ethics board here?]] Students were instructed to use the IDE and intentionally plagiarize the assignment. We worked with the professors of the classes to give each class a project that would be slightly too difficult to the average person in the class to encourage plagiarism, but not so difficult that NO students in the class would be able to complete it on their own. The 111 class was given a project to make 3 Polymorphic Shapes objects that move around the screen and bounce off the edges. The 211 class was given the task to create any function defined fractal (such as a Mandelbrot fractal). ChatGPT was able to generate solutions to both without assistance, and a solution for each was seeded on to an assortment of websites. Assignments were collected completely anonymously by distributing and collecting identical thumbdrives containing the IDE which was set to save to the drive its self. Assignment credit was given for turning in a thumbdrive without checking its content. 2 students in 211 caught this loophole and did in-fact turn in completely blank thumbdrives.

Students were also asked to self report the method they used to complete the assignment and include the report on the thumbdrive. As students were asked to “cheat” on this assignment, the frequency of cheating its self carries no meaningful information, it only serves to reveal the Vectors and usability of our method.

As the 111 and 211 classes are separate assignments and cohorts, we will treat them as 2 concurrent Case Studies.

As part of each Case Study, the metaComment was stripped out of the files, and they were given to Stanford MOSS.

The raw data can be found at [XXXX Link to data]

5.1. ICS 111

We received 5 submissions from 111, these have each been arbitrary given a single letter designation between A and F. Of these 5, 3 successfully completed the assignment by some means, and 2 did not. Student A copy and pasted from ChatGPT, but prompted it wrong, and did a similar but wrong assignment. Our method makes it easy to see this, there is a 75 line paste followed by about 38 minor tweaks and formatting changes.

Student D appears to have struggled: the code they copied is a badly mangled AWT example and does not compile without significant modification.

Of the Correct assignments Student B completed the assignment Legitimately and it is shows in the metaComments. There are multiple well organized files each organically coded with no large external pastes or irregularly linear typing sections. There are copies between files where code should be reused and modified, and follow up edits making these modifications.

Student C used the copy paste exploit mentioned previously and did not leave a comment. As such, we can not determine if their code is legitimate or not.

Student E pasted a large section of code from a different project with a matching InstallID and then spent considerable time (around 8 hours spread over a week or so) debugging and improving it. We believe this to be a legitimate completion.

Student F found and submitted the one of the seeded assignment file. It is logged as a large foreign paste.

5.1.1. MOSS.

Despite having 4 files containing plagiarized code, Moss only identified a small connection between Student C and E. This connection is entirely composed of expected driver code.

5.1.2. Observed Vectors.

Between all the submissions metaComment we were able to identify that Student A and F used Expert or Search vectors; Student B and E completed the assignment on their own; Student D used Search vector; and only student C is of indeterminate origin, but has a clean InfectionStack.

[[Add Table]]

5.2. ICS 211

5.2.1. Observed Vectors.

6. Future Work

In the future, we hope to completely automate the detection of plagiarized assignments created with the IDE. UUID tracing is easy enough to automate, but automating the whole process will require several heuristics. For example, how much of an assignment must be copied from the internet for it to be considered plagiarized? How big of a paste is that? What about linear typing? How many lines are required to say “this isnt organic” from just linear typing? All of these potential questions need future research to answer.

Additionally, our primary aim is plagiarism reduction, so it may be equally useful to create a heuristic that represents the general amount of plagiarism present in a class as a whole rather than individual assignments.

However, before any of this can happen the IDE needs to be stabilized. Several students complained about it “freezing” on save and there are a few notable problems. One possible improvement would be to gzip metaComment and the embedded data. Care should be taken in doing so, and it may be advantageous to pair this approach with Reed-Solomon ECC to combat accidental file corruption. This may be advantageous even if there is no net decrease in metaComment size. Additional improvements include embedding extra clipboard data that another copy of the IDE could read separate from the hidden text embedding

7. Conclusion

Our approach has shown promise as a tool in detecting plagiarism within intro level CS courses. The most useful features are paste event tracking and UUIDs. We encourage anyone concerned with plagiarism in an intro level class to provide a modified IDE for their class that uses these tracking features.

References

[1] O. U. press, Ed., *Oxford English Dictionary*.