5-2024

A Review of Student Attitudes Towards Keystroke Logging and Plagiarism Detection in Introductory Computer Science Courses

Caleb Syndergaard
Utah State University, a02268117@usu.edu

A REVIEW OF STUDENT ATTITUDES TOWARDS KEYSTROKE LOGGING AND PLAGIARISM DETECTION IN INTRODUCTORY COMPUTER SCIENCE COURSES

by

Caleb Syndergaard

A proposal submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

John Edwards, Ph.D. Major Professor	Vicki Allan, Ph.D. Committee Member
Hamid Karimi, Ph.D. Committee Member	D. Richard Cutler, Ph.D Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2024

ABSTRACT

A REVIEW OF STUDENT ATTITUDES TOWARDS KEYSTROKE LOGGING AND PLAGIARISM DETECTION IN INTRODUCTORY COMPUTER SCIENCE COURSES

by

Caleb Syndergaard

Utah State University, 2024

Major Professor: John Edwards, Ph.D.
Department: Computer Science

The following paper addresses student attitudes towards keystroke logging and plagiarism prevention measures. Specifically, the paper concerns itself with changes made to the "ShowYourWork" plugin, which was implemented to log the keystrokes of students in Utah State University's introductory Computer Science course, CS1400. Recent work performed by the Edwards Lab provided insights into students' feelings towards keystroke logging as a measure of deterring plagiarism. As a result of that research, we have concluded that measures need to be taken to enable students to have more control over their data and assist students to feel more comfortable with keystroke logging. This paper introduces the work done to enhance student privacy and feelings of security and an evaluation of the effectiveness of the changes made to ShowYourWork.

(77 pages)

PUBLIC ABSTRACT

A REVIEW OF STUDENT ATTITUDES TOWARDS KEYSTROKE LOGGING AND PLAGIARISM DETECTION IN INTRODUCTORY COMPUTER SCIENCE COURSES

Caleb Syndergaard

The following paper addresses student attitudes towards keystroke logging and plagiarism prevention measures. Specifically, the paper concerns itself with changes made to the “ShowYourWork” plugin, which was implemented to log the keystrokes of students in Utah State University’s introductory Computer Science course, CS1400. Recent work performed by the Edwards Lab provided insights into students’ feelings towards keystroke logging as a measure of deterring plagiarism. As a result of that research, we have concluded that measures need to be taken to enable students to have more control over their data and assist students to feel more comfortable with keystroke logging. This paper introduces the work done to enhance student privacy and feelings of security and an evaluation of the effectiveness of the changes made to ShowYourWork.

To those who have and will continue to assist in the labor of enhancing Computer Science education for the upcoming generations.

ACKNOWLEDGMENTS

- First, my spouse for being incredibly supportive and patient throughout my academic endeavors.
- Second, Dr. John Edwards for his insights and instruction.
- Third, Joseph Ditton for his contributions to the ShowYourWork tool. Without which, this research would not have been possible.

Caleb R. Syndergaard

CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

This research project serves as a continuation of research performed by the EdwardsLab at Utah State University. In the Fall semester of 2021, Utah State University began to track and collect keystroke data for students enrolled in the introductory Computer Science Course, CS1400 [1]. Students enrolled in CS1400 are required to use the PyCharm IDE, created by JetBrains. A plugin to the IDE, named ShowYourWork has been developed to perform the keystroke logging. The impetus behind the decision to collect keystroke data was a desire within the faculty to understand how students were writing code and solving the problems administered through homework assignments. Beyond a desire to understand student work, discouraging plagiarism was also a driving force behind the implementation of keystroke logging. The data collected from keystroke logging has been invaluable to the research of the EdwardsLab, however, we recognize that research into the effects that keystroke logging has on students is limited, and mostly indicates that there are negative side effects of tracking students [1]. The primary negative aspect of keystroke logging is that it induces anxiety in students [1]. In the previous set of survey results, one student stated, "Even though I didn't cheat, I was constantly stressed out that my assignments were being marked for plagiarism. This caused A LOT of anxiety throughout the semester, so much so that I wish I didn't take this class because of the emphasis on cheating....It would be really helpful if my professor was more clear about what the plugin is, especially for students who are prone to feelings of anxiety."

In an attempt to mitigate these negative effects, I have made modifications to the plugin that seek to return control to the students over their data. The original implementation of this plugin allowed students to view the history of their project at the granular level of individual keystrokes and also to revert their working file to any point in its history. I have enhanced the plugin by adding the following operations:

- A delete history option, which allows students to delete any keystrokes recorded between two intervals.
- A check for plagiarism utility, which allows students to click a button and receive a prompt detailing whether or not evidence of plagiarism was found within their document.

This paper focuses on the changes in student perception of keystroke logging given the addition of the aforementioned features to the plugin. We anticipate that these features will have a positive impact on students' feelings towards keystroke logging and that it will reduce the number of students who suffer from anxiety tied to the ShowYourWork utility.

Logging student data at the keystroke level introduces an ample amount of information for a Computer Science program at any university and with the addition of these tools we provide an avenue for academic institutions to begin collecting keystroke data without violating students' rights and privacy. This work is impactful because it will allow more universities to begin logging student keystrokes, and thus greatly enhance the quality of instruction at these organizations. In order to ascertain students' feelings towards the ShowYourWork plugin, a survey was administered at the end of the semester.

The results of the survey in the Spring 2022 semester will serve as a benchmark or control for the results of this survey. I will be making direct comparisons between the results of the surveys administered pre and post addition of the delete history and check for plagiarism functionalities.

My overall contributions are two-fold. First, I introduce novel algorithms that were necessary to the implementation of the delete history and check for plagiarism features and second, the survey data that will allow us to measure changes in student attitudes [2].

1.1 Organization of Thesis

This thesis is organize in a manner commensurate with the chronological order in which the research occurred. Each section will be expounded upon in detail and will depend on information given in previous sections. Thus, an in-order reading of this thesis is imperative to achieve a comprehensive understanding of my methods and results. The paper will seek to primarily address the following:

- A review of prior research and existing knowledge.
- Implementation details for the plugin and survey changes.
- An analysis of the administered survey.
- A review of outcomes and conclusions.

CHAPTER 2
RELATED WORKS

2.1 Introduction

In recent years there has been a significant increase in research publications relating to Computer Science education. Upon searching for the term "CS Education" in the ACM Digital Library, one will see that publications relating to this have increased exponentially since 1975. Dozens of recent publications are concerned about the rampant plague of plagiarism that has swept through the Computer Science community in recent years. With the advent of the "Internet age" accessibility to tools assisting with plagiarism has increased dramatically. While different measures of plagiarism deterrence have been implemented, Computer Science remains a disproportionately strong contributor to the yearly count of academic violations [3, 4].

Possible explanations for Computer Science students' propensity for plagiarism include the ease with which solutions are shared, the availability of solutions on the internet, and the rigorous nature of Computer Science courses in general [4]. In many respects, the causes for academic dishonesty in any field are simply more prevalent in the field of Computer Science. Regardless of the reasons for the prevalence of cheating in the field, solutions for mitigating the frequency of plagiarism are necessary. Proposed methods of deterring plagiarism include using plagiarism detection software, administering severe punishments for plagiarism, keystroke tracking, and a myriad of other techniques. In this paper we seek to expound on research regarding keystroke tracking as a deterrent.

2.2 Psychology Behind Plagiarism

Research into the psychology behind plagiarism is abundant. With the rising availability of information and ease with which students can cheat, understanding the impetus behind cheating is becoming increasingly important. Beyond this, understanding what avenues academic institutions should pursue in order to deter cheating is also extremely important. In a study performed in 2014 by Eric Beasly, he found that out of 298 students who cheated, 99 of them stated that “ignorance of consequences” was a major contributor to why they cheated and 87 of them claimed “ignorance of rules” as their reason for cheating [5]. In many regards, universities and other academic institutions unwittingly contribute to dishonesty by not making what constitutes cheating readily available and by not being more explicit about the consequences of plagiarism. One student in Beasly’s study stated, “If I knew what I was doing was wrong I wouldn’t have done it plain and simple. I wish I could say I wish I never cheated but it is not that simple because at the time of my incident I was unaware that my behavior was wrong” [5]. Because the definition of plagiarism is not always well defined, students fall victim to being penalized for actions they did not realize were incorrect in the first place. The weight of ignorance does not fall squarely on the shoulders of the institutions, however, as many students fail to put forth sufficient effort to understand the university’s guidelines. One student went as far as to say, “I think if i had read the part about academic dishonesty closer mayeb i would have taken more time to lookup about plagirism and what it really is” [5]. Responses like these demonstrate a disconnect between the information academic institutions are trying to convey about plagiarism and their enrolled populous. Many institutions require that a section addressing academic dishonesty be included in every course syllabus, however from the responses collected by Beasly, it is clear that this method of delivering information about cheating is ineffective and needs to be revisited.

Ignorance is not the only driving factor behind plagiarism. In the article “Motivational Perspectives on Student Cheating:Toward an Integrated Model of Academic Dishonesty”, the authors seek to provide a framework under which cheating can be classified [6]. The authors, Murdock and Anderman, devise three categories for reasons behind cheating and phrase them as questions that students may ask themselves before deciding to cheat.

The first question the authors evaluate is “What are my goals?” [6] Included in this section is an evaluation of previous literature on the subject of plagiarism. Murdock and Anderman submit the following summary of previous research into students who cheated because it enabled them to achieve specific goals: “there is a distinction between students who approach classroom tasks with a genuine desire to understand(i.e., high intrinsic value, strong mastery or learning goals)versus those who are more interested in external indicators of accomplishment (i.e., performance goals, ego goals, extrinsic motivation)” [6]. In this category, factors such as “extrinsic goals” and “performance orientation” are considered, with a students “personal theory of intelligence”, parental and peer pressures for a grade, social comparison in the classroom, and classroom goal structures being cited as the “individual and contextual influences” for cheating. The authors conclude this section by stating “as our society continues to emphasize out-comes over learning, many argue that cheating is likely to continue to occur” [6, 7].

The next category considered by Murdock and Anderson is “Can I Do This?” [6]. In this section, factors like self-efficacy and outcome expectations are reviewed with personal ability, personal effort, teacher’s pedagogical skill, and grading standards serving as the individual and contextual influences [6]. In their final summary of this section, the authors state, “Similar to the research linking students’ goals to cheating, there is more evidence that students’ individual self-efficacy beliefs and their perceived outcome expectations are related to cheating, and there is less evidence for the effects of the environment on cheating, via their impact of these beliefs systems. When students have high self-efficacy beliefs and expect to succeed at an academic task, cheating is probably neither a necessary nor useful strategy”. Simply stated, students who have a lower self-perception of their abilities to learn a given skill are more likely to attempt some form of cheating or plagiarism.

Finally, the authors consider the question “What are the costs?”. Here, the authors consider students perspectives on “getting caught and punished” and a “negative view of self” [6]. This section of the paper provides a deeper insight into how students perceive themselves after cheating and offers the insight that, “these costs are not only the direct costs of being caught and punished, but the psychological costs that come from being seen, or seeing oneself, as a person who does something unethical" [6]. Despite how invaluable an understanding of the human psyche before and after cheating would be, without being able to collect objective, quantitative data, it is exceptionally difficult to determine the overall effect that self-perception has on a student's desire and disposition to cheat. In the conclusion of this section of the paper, Murdock and Anderson mention that, "as with much of the other scholarship in this area, one of the major limitations to our understanding results from an almost exclusive reliance on one-time correlational studies using self-report data" [6].

It is interesting to note that students attempt to reduce this psychological toll that cheating takes through means of justification. A study performed by Agustus Jordan in 2001 found, unsurprisingly, that "Cheaters...justified cheating behavior to a greater extent than did noncheaters" [8]. Despite the discrepancy between cheaters and non-cheaters in how justifiable they deem cheating, there are many factors that influence whether a population as a whole feels as if cheating is warranted. In the article by Murdock and Anderson they found that "When blame for cheating is shifted to others, the most common target is the teacher and his or her instructional practices" [6]. To further expound on this point, "cheating was rated as more justifiable and more likely in classrooms where teachers emphasized performance goals as compared to mastery goals and where teachers were perceived as less versus more competent" [6]. In addition to teacher ideology and competency, another major factor influencing a student's propensity to cheat was their peer's perception of cheating as a whole. In the research performed by Jordan, they found that "cheating behavior was related to perceptions of the behavior of peers and to attitudes about cheating. Both factors were significant predictors of cheating rates. Cheaters believed that more students engaged in cheating behaviors than did noncheaters...In addition, the more the participants cheated, the higher their estimate of cheating on campus" [8].

As many students seem to be unaware of the consequences of cheating and what actually constitutes cheating in the first place, it is unsurprising that cheating occurs so frequently. Beyond this, if a student anticipates their peers are going to cheat, they are more likely to cheat themselves. It should then come as no surprise that Computer Science finds itself at the forefront of academic violations, as the stigma that "Computer Science students cheat the most" is constantly perpetuated. In order to mitigate the negative effects of plagiarism, it seems that students' confidence in one another to not cheat, and their confidence in their institution's ability to find cheaters needs to increase dramatically. A proposed manner of accomplishing exactly that is to keep a more granular record of the code that students produce in foundational Computer Science courses.

2.3 Student Tracking Datasets

The concept of using programs to detect plagiarism is not a novel one, with work dating back to at least 1988 with the formulation of the Plague program which compared programs for similarity in structure [9]. More recently, there has been a shift towards evaluating a single assignment submission at multiple points in its history to get a better idea of how the student arrived at the code they are turning in [10] . The major shortcoming with these datasets is granularity, which is what the EdwardsLab at USU has sought to fix with the ShowYourWork plugin.

2.4 Research Performed by the Edwards Lab

In recent years, the EdwardsLab recognized the need to collect student data in order to facilitate new research, improving the quality of instruction in the Computer Science department at USU. In order to collect temporal data with a higher resolution than previous systems, the EdwardsLab created the ShowYourWork plugin. ShowYourWork is a plugin implemented for the PyCharm integrated development environment produced by the JetBrains company. The plugin currently uses an SQLite database to collect keystrokes. In order to understand students' feelings towards the plugin, a survey was administered at the end of the first semesters during which the plugin was being used. Hart et. al. evaluated the results of this survey in the paper "Plagiarism Deterrence in CS1 Through Keystroke Data" [1]. The data collected previously by the EdwardsLab will serve as a control for the data collected in this study.

CHAPTER 3
IMPLEMENTATION METHODS

3.1 Introduction

In this section we analyze the implementation methods for the ShowYourWork plugin. We begin with an overview of the plugin as it stood during the time in which Hart et al. collected their survey data [1]. After this we will review the substantial changes I have made to the inner workings of the plugin since that paper's publication. Finally, we will review the addition of two new features that form the foundation of this literary work. The changes that are most pertinent to this study are the addition of the "Delete History" and "Check for Plagiarism" utilities.

3.2 Previous Plugin Versions

The ShowYourWork plugin consists primarily of the following two components, the inner mechanisms required for the keystroke logging itself and a student facing UI designed to allow students to "rewind" the history of their file and see all previous changes keystroke by keystroke. The keystroke logging uses the event bus managed by the integrated development environment to collect the keystrokes. Keystrokes were then written out to a comma delimited file for storage. After using the plugin for multiple semesters, the EdwardsLab determined that it needed to be more efficient. The plugin now uses an SQLite file to store the keystrokes and has a more streamlined manner of collecting user events. While updating the plugin to use SQLite instead of CSV files, the Edwards lab also rewrote the main user interface. Originally, the plugin was built in the GUI library provided by the JetBrains community, however, the plugin has now been translated to JetPack compose, the new Android GUI application programming interface. The application now exists in a more extensible state that will be conducive to future additions and changes.

3.3 Delete History

The last time a survey about ShowYourWork was administered, the majority of students reported viewing the plugin as a positive thing, with 11 percent of students saying that they forgot it was there at all [1]. While most students agreed that it was a good thing, the number of students who felt the plugin was invasive was still significant at 6 percent [1], with one student going as far as to say, “I did not like the plugin at all. I would have never voluntary downloaded a key logger to my computer no matter what its intended purpose was if it wasn’t a requirement for any of my assignments to get graded” [1]. Our hope was that in implementing the delete history utility, we would decrease the proportion of students who feel uncomfortable with keystroke logging. The primary purpose of this addition is to enhance the student experience while using the plugin and protect their privacy, even if that comes at the cost of losing some file history data that could be useful in other research avenues. Within this section I will discuss a total of five algorithms. These are the simple rewrite, masking, LCS, build edit list, and smart delete algorithms. Figure 3.1 shows a reference of how those algorithms relate to one another.

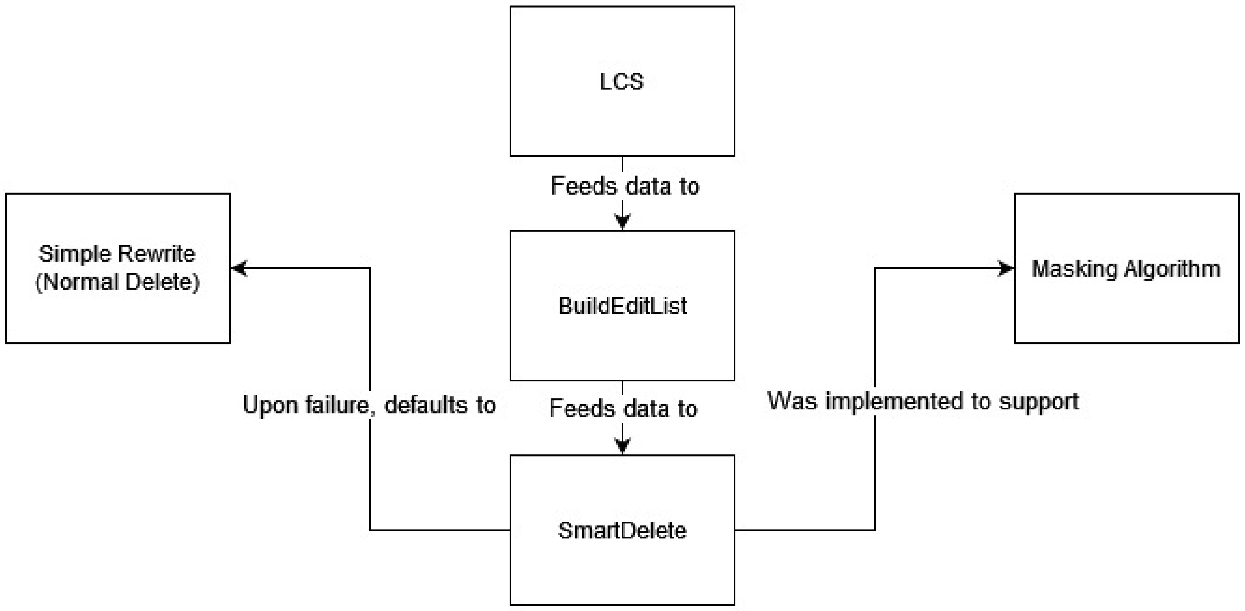
LCS
Feeds data to
Simple Rewrite (Normal Delete)
BuildEditList
Masking Algorithm
Upon failure, defaults to
Feeds data to
Was implemented to support
SmartDelete

Fig. 3.1: Relationships between the delete and masking algorithms.

I added a button titled “Delete History” to the ShowYourWork plugin. This button allows users to remove the record of all keystrokes logged between two points in their file history. The user selects the first point in their file history by dragging a slider back in time to the moment they would like to delete from. The user then selects the “Delete History” button and is presented with a second slider that they use to select the point in history that they would like to delete to. Once the user clicks “Delete”, the keystrokes are deleted, and an algorithm is run in order to reconcile the differences between the first state of the file and the final state of the file. For example, if a file’s beginning state is the following:

```
def printNumber(x):
    print(x)
print("The number is: ")
printNumber(4)
```

and ending state is:

```
def printNumber(x):
    print(f"The best number is: {x}")
printNumber(4)
```

The algorithm determines what changes need to be made made to get from the first file state to the second, and then add those changes synthetically to the keystroke history. The changes made in reconciling a user’s history are flagged as “SYNTHESIZED” in the database, to help the application determine which changes originated from the user and which changes were created by the ShowYourWork plugin.

I have employed two different algorithms in order to reconcile the differences in file states. The first algorithm, called the “smart delete” algorithm, is an algorithm that seeks to fix the flaws in the original delete algorithm, which is called the “simple rewrite” method. The original delete algorithm, or simple rewrite, was inspired by the code used to reconcile differences between the code a student had written and the information in the keystroke log. For example, if a student were to paste in a chunk of text and then quickly close the application, the database writer may fail to save the information out to the database. In this case, when the IDE reopens, the information in the database does not match the code in the file. In order to rectify this, a single edit is inserted to the database, where the edit consists of deleting all of the text in the last file snapshot and then inserting all of the text that exists in the file as it stands. To better illustrate this point, consider the case where the ShowYourWork plugin had a list of edits that would produce the following code:

```
def printNumber(x):
    print(x)
```

Now, if a student were to quickly update their code to the following, the database may not have time to save the edits:

```
def printTen():
    print(2 * 5)
```

Now, in order to resolve the difference, a single edit would be added to the database where the deleted string would be “def printNumber(x):\nprint(x)” and the inserted text would be “def printNumber():\nprint(2 * 5)”, rather than a series of insertions and deletions. Modeling this change in the database may look something like the table defined in the following figure:

Insert Text	Delete Text	Index	Edit Type
def printTen():\nprint(2 * 5)	def printNumber(x):\nprint(x)	0	synthesized

Table 3.1: Simple Rewrite Method

This method preserves the code written by the student when the file history and the current file do not match.

This process worked well until I began to prototype a masking algorithm that would help protect student data. The algorithm would allow data sanitizers, researchers tasked with replacing keystroke data containing identifying information with non-sensitive material, to look at a student's code, select text they would like to mask, and replace that text with a mask character or a series of random characters. Random characters would produce the best results, as a data sanitizer may see fit to mask a variable name, in which a chosen mask character, such as "@", would break the code's functionality. This would step beyond the bounds of data masking, as the goal would simply be to obfuscate any identifying or personal information without altering the data in a compromising way.

The masking algorithm that I produced, together with Utah State University professor, Joseph Ditton, uses two custom data containers called a "MaskObject" and a "Selection", respectively. A "MaskObject" is simply a pair of text, a string value, and index, an integer value. An example of which may look something like "{text: 'a', index: 4}". A "Selection" is simply a list of "MaskObjects" constructed from a given string and a starting index. For example, if given the string "code" and a start index of 2, the resulting selection would be a list of the following mask objects: "{ text: 'c', index: 2 }, { text: 'o', index: 3 }, { text: 'd', index: 4 }, { text: 'e', index: 5 }". The "edit" value that appears in the algorithm represents a value in the database which stores logged keystrokes. For the sake of brevity, "MaskObject" has been shortened to "mo" and "Selection" to s. The definition of the rudimentary mask algorithm is on the subsequent page.

```
procedure MASK($s$)
    for each $edit$ in $edits$ (reversed) do
        for each $mo$ in $s$ (reversed) do
            if $edit.deleteText$ is not empty then
                if $mo.index \geq edit.sourceLocation$ then
                    $mo.index \leftarrow mo.index + length(edit.deleteText)$
                end if
            end if
            if $edit.insertText$ is not empty then
                if $mo.index = edit.sourceLocation$ and $mo.text = edit.insertText$ then
                    $edit.insertText \leftarrow$ random char from a-z or a masking character
                    toRemove.add(mo)
                    break
                else if $mo.index \geq edit.sourceLocation$ then
                    $mo.index \leftarrow mo.index - length(edit.insertText)$
                end if
            end if
        end for
        for each $mo$ in $toRemove$ do
            remove mo from s
        end for
        if $s$ is empty then
            break
        end if
    end for
end procedure
```

The algorithm functions by using the current position of the characters that need to be masked as a starting point to derive the original position in which the characters were inserted. Beginning at the end of the edit history, one simply needs to iterate backwards through the list of edits altering the index of the characters until they find an edit that matches in both text and index. Take, for example, a small program that looks like " `a=('Caleb')` ". Now consider what would happen to the edit history if a user were to change the statement from an assignment statement to a print statement like the following: " `print('Caleb')` ". The following table demonstrates what the edit history may look like:

Insert Text	Delete Text	Index	Edit Type
a		0	insert
=		1	insert
(		2	insert
‘		3	insert
C		4	insert
a		5	insert
l		6	insert
e		7	insert
b		8	insert
’		9	insert
)		10	insert
	a=	0	delete
print		0	insert

Table 3.2: Small Program Edit History

Now, suppose that a data sanitizer were to select the text "Caleb" for masking in the final program, " `print('Caleb')` ". The following selection list would be created from the selection: [{ text: ‘C’, index: 7}, { text: ‘a’, index: 8 }, { text: ‘l’, index: 9 }, { text: ‘e’, index: 10}, { text: ‘b’, index: 11}]. Now, we would simply iterate through the edits in the edit history in reverse order and apply the changes specified in the algorithm.

For the first edit in the history, we have an edit consisting of the text "print", at index 0, and the edit is of type insert. Reviewing the algorithm, we begin iterating over the selection list in reverse order as well and see that the initial mask object { text: ‘b’, index: 11 } has a higher index than that of the edit to which it is being compared. This means that we need to decrease the index of this mask object by the length of the inserted text, “print”, which in this case, is 5. The resulting mask object would be { text: ‘b’, index: 6 }. We now proceed to perform this operation to each of the object in the selection list, as all indices in the list are over 0. The resulting selection list would be: [{ text: ‘C’, index: 2}, { text: ‘a’, index: 3 }, { text: ‘l’, index: 4 }, { text: ‘e’, index: 5}, { text: ‘b’, index: 6}].

The next iteration of the algorithm would happen with a delete edit that consists of the text “a=” at index 0. We would once again iterate over the selection list in reverse order making any changes specified in the algorithm. In this case, we would start with the edit { text: ‘b’, index: 6 } and see that the index of the mask object is greater than that of the edit. Differently than last time, however, we are dealing with a delete edit, meaning that we need to increase the index of the mask object by the length of the edit text rather than decreasing in. This particular case would require that we decrease the index of the mask object by the length of the string “a=”, which is 2, resulting in { text: ‘b’, index: 8 }. After iterating through the rest of the selection list, the result would be: [{ text: ‘C’, index: 4}, { text: ‘a’, index: 5 }, { text: ‘l’, index: 6 }, { text: ‘e’, index: 7}, { text: ‘b’, index: 8}].

Now, we consider the next edit in the history which is inserts ‘(’ at index 10. Iterating over the selection list reveals that all mask objects have an index that is less than 10, meaning no changes need to be made. Similarly, for the iteration with the insert edit at index 9 with text “ ’ ”, the index is higher than that of any item in the mask list, meaning that, once again, no changes need to be made. The next edit in the history inserts the text ‘b’ at index 8 of the file. Upon iterating through the selection list this time, we see that the mask object { text: ‘b’, index: 8} matches the edit’s text and index. This means that the mask object in our selection list is pointing to this edit in the file history, and that a masking operation needs to be performed. We generate a random character, in this case ‘z’ and replace the inserted text in the file resulting in an edit with insert text ‘z’ at position 8.

The mask object { text: ‘b’, index: ‘8’ } would then be removed from the selection and the process would continue. The subsequent iterations may result in ‘e’ being masked with ‘q’, ‘l’ being masked with ‘r’, ‘a’ being masked with ‘w’, and ‘C’ being masked with ‘m’. Now, the selected occurrence of the word “Caleb” in the file history would be masked, the final program would read “ print(‘mwrqz’) ”, and the edit table would look like the following:

Insert Text	Delete Text	Index	Edit Type
a		0	insert
=		1	insert
(		2	insert
‘		3	insert
m		4	insert
w		5	insert
r		6	insert
q		7	insert
z		8	insert
’		9	insert
)		10	insert
	a=	0	delete
print		0	insert

Table 3.3: Small Program Edit History After Mask

It is important to note that this algorithm is still in its infancy and is not intended to serve as a polished solution to the selection based masking problem. It is included here to demonstrate the work I completed on the masking problem and to help illustrate the need for a delete operation that functions differently from the “Simple Rewrite Method”. An ideal, selection based, masking algorithm would not only cloak characters at their insertion point, but also their deletion point, if one exists. While the following algorithm fails to do the latter, it would not be difficult to extend in order to accomplish this. Another of the algorithm’s primary deficiencies happens when multi-character inserts contain the text that needs to be masked. In such cases, the algorithm fails to find the character that needs to be masked, as it does not iterate through inserted strings to perform character by character operations, rather, it considers inserts and deletes holistically. This is to say, that, for example, if a single insert generated by a paste event contained the text "My university id is: abc123", and a user decided to mask "abc123", the algorithm would fail to mask it. If a user were to paste "My university id is: " and then proceed to type 'a', 'b', 'c', '1', '2', and '3', the algorithm would successfully detect and mask the identifying information. The algorithm would simply need to iterate over each character in the insert and delete text, treating them as individual edits with single character text fields and unique indices, in order to be able to mask text that appears inside of paste events. Further work and discussion on the masking problem lies outside of the purview of this research, but it is important to note that if the suggested changes to the masking algorithm were implemented, the simple rewrite method would disrupt the masking functionality when used to delete a section of file history. The reason for this lies in how the enhanced masking algorithm, with the functionality of masking specific characters in multi-character inserts, finds the characters that it needs to mask. Take, for example, a program that looks like the following:

```
print('Caleb')
print(10)
```

A subsection of its history may look like:

If a student decided to delete the history of edits that were recorded when they typed out " `print(10)` ", the file history would be updated to look like the following:

If a data sanitizer were to mask the occurrence of the word "Caleb" that appears in the final file, only the final line would be changed, producing the following history:

Note that only the final line of the edit history is altered. The word "Caleb" still appears in the file history and has only been masked up to the point that the delete operation took place. This means an individual masking data would have to mask the instance of "Caleb" both before and after the delete event to ensure that all instances of the word would be masked. Not only is this tedious, but also error prone, as a masker may fail to notice that a delete event occurred in a student's history, and thus fail to mask the file correctly. In order to counteract this, I developed the "smart delete" algorithm, which takes a more granular and manual approach to editing the file history.

Insert Text	Delete Text	Index	Edit Type
...			
C		7	insert
a		8	insert
l		9	insert
e		10	insert
b		11	insert
’		12	insert
)		13	insert
\n		14	insert
p		15	insert
r		16	insert
i		17	insert
n		18	insert
t		19	insert
...			

Table 3.4: Simple Program Edit History

Insert Text	Delete Text	Index	Edit Type
...			
C		7	insert
a		8	insert
l		9	insert
e		10	insert
b		11	insert
’		12	insert
)		13	insert
print(‘Caleb’)\nprint(10)	print(‘Caleb’)	0	synthesized

Table 3.5: Simple Program Edit History After Deletion

The smart delete algorithm functions by comparing the history of a file at the two states between which a student wants to perform a delete operation. The algorithm performs a string comparison and determines the minimum number of single character edits required to get from the starting state of the file to the ending state. Taking less than or equal to the same number of edits that a user took to produce the same file history is important because using more edits would result in a state where the subsequent edits in the database need to be reindexed. This is time consuming and would require the application to stop the user from continuing to make edits. Thus, the single character edits are then compressed into

Insert Text	Delete Text	Index	Edit Type
...			
C		7	insert
a		8	insert
l		9	insert
e		10	insert
b		11	insert
’		12	insert
)		13	insert
print(‘xdfge’)\nprint(10)	print(‘Caleb’)	0	synthesized

Table 3.6: Simple Program Edit History After Deletion

the longest possible multi-character edits to ensure that the total number of synthesized edits is equal to or less than the number of edits originally taken to produce the second file state from the first. In exceedingly rare situations, the smart delete algorithm may fail to produce a list of edits that is the same length or shorter than the original set of edits. This will be discussed after the algorithm’s definition.

The primary “Smart Delete” algorithm has requires the definition of two other algorithms, an slightly altered version of the longest common subsequence algorithm, which will be shortened to LCS, and another algorithm that has been deemed the “Build Edit List” algorithm, which forms a list of single character edits from the table produced by the LCS algorithm. Finally, the “Smart Delete” method is responsible for calling the LCS and Build Edit List algorithms and then compressing and reindexing the list of edits produced by Build Edit List into multi-character edits. The algorithm definitions are as follows:

```
**function** LCS($str1, str2$)
    Create the table $lcsTable$ to store the LCS.
    **for** $i \leftarrow 1$ to $length(str1)$ **do**
        **for** $j \leftarrow 1$ to $length(str2)$ **do**
            **if** $str1[i-1] = str2[j-1]$ **then**
                $lcsTable[i][j] \leftarrow lcsTable[i-1][j-1] + 1$
            **else**
                $lcsTable[i][j] \leftarrow \max(lcsTable[i-1][j], lcsTable[i][j-1])$
            **end if**
        **end for**
    **end for**
    $lcsLength \leftarrow lcsTable[length(str1)][length(str2)]$
    $lcs \leftarrow$ an empty list
    $i \leftarrow length(str1)$, $j \leftarrow length(str2)$
    **while** $i > 0$ and $j > 0$ **do**
        **if** $str1[i-1] = str2[j-1]$ **then**
            Append $str1[i-1]$ to $lcs$
            $i \leftarrow i - 1$
            $j \leftarrow j - 1$
        **else if** $lcsTable[i-1][j] > lcsTable[i][j-1]$ **then**
            $i \leftarrow i - 1$
        **else**
            $j \leftarrow j - 1$
        **end if**
    **end while**
    Reverse $lcs$ to get the correct order.
    **return** $lcs$
**end function**
```

```
function BuildEditList($str1$, $str2$, $lcs$)
    $editList \leftarrow$ empty list
    $i \leftarrow 0$
    $j \leftarrow 0$
    for $k$ in 0 to $lcs.length$ - $1$ do
        while $i < str1.length$ and $str1[i] \neq lcs[k]$ do
            $editList.add(TempEditFromDelete(i, deleteText = str1[i].toString()))$
            $i \leftarrow i + 1$
        end while
        while $j < str2.length$ and $str2[j] \neq lcs[k]$ do
            $editList.add(TempEditFromDelete(j, insertText = str2[j].toString()))$
            $j \leftarrow j + 1$
        end while
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
    end for
    while $i < str1.length$ do
        $editList.add(TempEditFromDelete(i, deleteText = str1[i].toString()))$
        $i \leftarrow i + 1$
    end while
    while $j < str2.length$ do
        $editList.add(TempEditFromDelete(j, insertText = str2[j].toString()))$
        $j \leftarrow j + 1$
    end while
    return $editList$
end function
```

For the definition of "Smart Delete", "it" will stand for insert text, "dt" will stand for delete text, and "sl" will stand for source location

```
function SmartDelete($str1$, $str2$)
    $originalEdits \leftarrow LCS(str1, str2)$
    $editList \leftarrow BuildEditList(str1, str2, originalEdits)$
    $offset \leftarrow$ empty map
    $i \leftarrow 0$
    $prevEdit \leftarrow$ null
    $currentEdit \leftarrow editList[0]$
    while true do
        $totalOffset \leftarrow 0$
        for $(offsetKey, offsetValue)$ in $offset$ do
            if $offsetKey \leq currentEdit.sl$ then
                $totalOffset \leftarrow totalOffset + offsetValue$
            end if
        end for
        if $currentEdit.dt \neq$ null then
            $currentEdit.sl \leftarrow currentEdit.sl + totalOffset$
        end if
        if $!offset.containsKey(currentEdit.sl)$ then
            $offset[currentEdit.sl] \leftarrow 0$
        end if
        if $currentEdit.it \neq$ null then
            $offset[currentEdit.sl] \leftarrow offset[currentEdit.sl] + 1$
        else
            $offset[currentEdit.sl] \leftarrow offset[currentEdit.sl] - 1$
        end if
        if $i \neq 0$ then
```

```
            **if** $prevEdit.it = $ **null and** $currentEdit.it = $ **null then**
                **if** $prevEdit.sl = currentEdit.sl$ **then**
                    $prevEdit.dt \leftarrow prevEdit.dt + currentEdit.dt$
                    $editList.remove(currentEdit)$
                    $currentEdit \leftarrow prevEdit$
                    $i \leftarrow i - 1$
                **end if**
            **else**
                **if** $prevEdit.it \neq$ **null and** $currentEdit.it \neq$ **null then**
                    **if** $prevEdit.sl + length(prevEdit.it) = currentEdit.sl$ **then**
                        $prevEdit.it \leftarrow prevEdit.it + currentEdit.it$
                        $editList.remove(currentEdit)$
                        $currentEdit \leftarrow prevEdit$
                        $i \leftarrow i - 1$
                    **end if**
                **end if**
            **end if**
        **end if**
        $i \leftarrow i + 1$
        **if** $i < length(editList)$ **then**
            $prevEdit \leftarrow currentEdit$
            $currentEdit \leftarrow editList[i]$
        **else**
            **break**
        **end if**
    **end while**
    **return** $editList$
**end function**
```

The practical implementation of these algorithms has a final step in which there is a check to ensure that the number of edits created by the smart delete algorithm does not exceed the number of edits taken to get from a starting file state to a final file state. In the rare case that the smart delete algorithm proposes a list of edits that is longer than the existing list of edits, the ShowYourWork plugin defaults to using the simple rewrite method. Using a combination of the smart delete and simple rewrite methods helps to mitigate the negative effects that simple rewrite has when it comes to masking while ensuring that student code is not lost in a delete operation.

Now that I have defined the algorithms, I will proceed to demonstrate some concrete examples of the algorithm in process by showing what the output values for each step of the algorithm would be:

1. Simple insert changes.

   To demonstrate this, suppose that a student wants to delete all edits from the file history that take the file from its starting state of:

   ```
   print("Hello")
   ```

   To the ending state:

   ```
   print("Hello world")
   ```

   This change could have been produced in a number of ways including the student typing ‘ ’, ‘w’, ‘o’, ‘r’, ‘l’, ‘d’ character by character, or simply as a result of the student pasting the text “ world”. Upon running the smart delete algorithm, we would see first that the output of the LCS function would be the array of characters that make up the shared text between the two file states.

   ```
   [‘p’, ‘r’, ‘i’, ‘n’, ‘t’, ‘(’, ‘"’, ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘"’, ‘)’]
   ```

Next, the “build edit list” portion of the algorithm would run taking the longest common subsequence produced by the LCS function and the two file states as inputs. This portion of the algorithm would yield the initial list of single character edits that would be required to get from the starting file snapshot to the final file snapshot. The output of “build edit list” in this case is the following list of edits:

```
[
 {index: 12, insert: ‘ ’},{index: 13, insert: ‘w’},
 {index: 14, insert: ‘o’},{index: 15, insert: ‘r’},
 {index: 16, insert: ‘l’},{index: 17, insert: ‘d’}
]
```

This list is now fed to the rest of the “smart delete” algorithm in order to produce a list of condensed edits by finding subsequent insertions and deletions and combining them into single edits when possible. The final result of the entire process, in this case, would the single edit:

```
[{index: 12, insert: ‘ world’},]
```

The algorithm recognized that a space being inserted at index 12 directly followed by a ’w’ inserted at index thirteen could be combined into a single, multi-character insert starting at index 12. It proceeds to do this with the remaining edits produced by “build edit list” until the smallest possible list of edits that can resolve the differences between the starting file snapshot and the ending file snapshot has been formed.

2. Simple delete changes.

   For this series of edits, suppose that a student wants to delete all edits from the file history that take the file from its starting state of:

   ```
   a = 5
   ```

```
print(a)
```

To the ending state:

```
a = 5
```

As we have already established the manner through which the algorithm functions, we will simply show what the output of each step would be. First, the LCS function would produce the following array of characters:

```
[‘a’, ‘ ’, ‘=’, ‘ ’, ‘5’]
```

Next, the “build edit list” function would create the list of single character edits.

```
[
 {index: 5, delete: ‘\n’},{index: 6, delete: ‘p’},
 {index: 7, delete: ‘r’},{index: 8, delete: ‘i’},
 {index: 9, delete: ‘n’},{index: 10, delete: ‘t’},
 {index: 11, delete: ‘(’},{index: 12, delete: ‘a’},
 {index: 13, delete: ‘)’}
]
```

Finally, the “smart delete” algorithm would finish the process by combining these edits into multi-character changes:

```
[{index: 5, delete: ‘print(a)’}]
```

3. Complex insert and delete changes.

   This would be the most common use case of the delete functionality within the ShowYourWork plugin, and also the most difficult. This use case helps demonstrate the process of combining sequential single character edits of the same type into single multi-character edits more clearly. We begin with the program:

```
print(‘My name is Caleb’)
```

   And we will end with the program:

```
print(‘His name isn’t Caleb’)
```

   The LCS portion of the algorithm would produce the following list:

```
[
 ‘p’, ‘r’, ‘i’, ‘n’, ‘t’, ‘(’, ‘’", ‘ ’, ‘n’, ‘a’, ‘m’, ‘e’, ‘ ’,
 ‘i’, ‘s’, ‘ ’, ‘C’, ‘a’, ‘l’, ‘e’, ‘b’, ‘’", ‘)’
]
```

   This list would then be fed to “build edit list”, which gives use the following list of edits:

```
[
 {index: 7, delete: ‘M’},{index: 8, delete: ‘y’},
 {index: 7, insert: ‘H’},{index: 8, insert: ‘i’},
 {index: 9, insert: ‘s’},{index: 18, insert: ‘n’},
 {index: 19, insert: ‘’’},{index: 20, insert: ‘t’}
]
```

Finally, these edits would be combined into larger edits by the "smart delete" algorithm producing the final list of edits:

```
[
 {index: 7, delete: ‘My’},{index: 7, insert: ‘His’},
 {index: 18, insert: ‘n’t’},
]
```

This case helps to illustrate that by combining sequential single character edits into multi-character edits, we are able to create the smallest series of edits to get from the starting file snapshot to the ending file snapshot without simply deleting all of the text and pasting in the new file contents. Note that the final edits need to be performed sequentially, just like a file history produced by manually typing and deleting code. This is to say that the indices of subsequent operations in the list of edits depend on the edits prior to them being performed. The final insert happens at index 18, but the text `‘n’t’` would need to be inserted at index 17 if performed first. The final list of edits shows 18 as the index for this insertion, because the insertion of `‘His’` has added an additional character to the string proceeding the location of the final insert.

To reiterate, the idea behind this algorithm was to allow the program to resolve the changes between two file snapshots using the smallest possible number of organic changes, with organic meaning simple inserts and deletes, such as the ones students may produce. While this algorithm performs flawlessly in almost every situation, I have identified a single instance in which it fails to make the edits in the simplest way possible. Thus, the catch to verify that the algorithms edits can fit in the index of the edits of the first file snapshot and the index of the edits of the final file snapshot is necessary. This is not to say that there are not more instances where the algorithm fails to perform as expected, only that this is an instance of which I am aware. In the following case, it would be possible for the student to produce a list of edits shorter than that created by the "smart delete" algorithm.

I will begin by providing a concrete example, and then I will seek to generalize. Take the following to be the starting file snapshot:

`‘# ABCD CDEF GHIJ’`

The final snapshot is:

`‘# CDEF GHIJ’`

To the casual observer, it is simple to see that the quickest possible way to get from the first file state to the second, is simply to delete `‘ABCD ’`, resulting in only a single edit being written to the database. However, in this case, the final output of the algorithm is the following:

```
    [{index: 2, delete: ’AB’},{index: 4, delete: ’ CD’},]
```

The algorithm first deletes the characters `‘AB’`, resulting in the string `‘ CD CDEF GHIJ’`, it then proceeds to delete the space and the `‘CD’` that is adjacent to `‘EF’`, allowing the first appearance of `‘CD’` to remain. To generalize, the algorithm produces an extra edit any time a portion of string is deleted that terminates with the starting portion of the next string. Other examples of this would be:

1. `‘XYZ YZA RST’` to `‘YZA RST’`
2. `‘Caleb leb’` to `‘leb’`
3. `‘print int’` to `‘int’`

Given the data on how students have chosen to utilize the delete tool, which I will discuss later on, I find this instance to be somewhat of a nonissue. It would be highly unlikely that a student would write code following the shortest possible route to produce a given program. Students typically delete large swaths of changes, many of which would contain multiple deletions and insertions that could happen in a smaller, more compressed fashion. This is a result of students typically writing code character by character, and not pasting complete portions of text into their program. This means that many sequential single character edits would be taking place between almost any two file snapshots, resulting in a larger set of indices to work with in the database. While it is improbable that students would produce such a program and then delete the file history between two snapshots that would result in the aforementioned issue to take place, it is not impossible. For this reason, I preserved the simple rewrite method in the code of ShowYourWork and employ it any time that the smart delete algorithm fails to produce a small enough list of edits to fit within the space provided by the delete.

Now, having defined the manner in which the delete feature functions, it is of worth to mention that allowing users to delete their file history, much, if not all of the keystroke data we wish to collect is at risk. If a student decides to delete all of their history on every assignment, we have one less piece of data that could be used to analyze student coding patterns. In addition to this, one of the purposes of ShowYourWork is to serve as a deterrent for cheating and as a method to detect plagiarism; allowing users to delete any record of their wrongdoings may render the plugin ineffectual. That being said, we did not see abuse of the "Delete History" feature in this study, nor did most of the students report that they even used it at all.

3.4 Check for Plagiarism

The second change of note to the show your work plugin is the addition of a "Check for Plagiarism" button that students may use at any time to analyze their code. The "Check for Plagiarism" button is a utility to help students preemptively speak with an instructor if they have made changes to their file that may seem incriminating and explain the manner in which they approached the assignment. A student can receive two responses upon clicking on the "Check for Plagiarism" button:

1. "There are no concerns in your edit history!", if the plugin does not find evidence of plagiarism.

2. "There are one or more concerns in your edit history. We recommend that you meet with the instructor.", if the plugin finds evidence of plagiarism.

In either case, I have added “Note: This tool is not perfect and should not be used as definitive proof.” to the end of the message. I added this footnote in order to assuage students fears in the case that they have not plagiarized, but the plugin detects evidence that they have. This also serves as a reminder, to any instructors that open up student file history using the plugin, that our methods for detecting plagiarism are fallible. There are two main methods through which we determine if evidence of plagiarism is present.

The first method is to analyze the student’s file history for any “large paste” events. Any paste that incorporates the newline character is considered to be a “large paste”. The algorithm that performs this operation searches through the file history in reverse order until it comes across a large paste event. Upon finding a large paste event, the algorithm looks through all cut operations prior to that event to see if the source of the pasted text is the file itself. If there are any cut operations in the file history prior to the large paste event that have matching text, the event is considered a nonissue and is not counted as evidence of plagiarism. If no cut operations are found, the algorithm uses all previous edits to build the file up until the point at which the paste event was detected. Then, the algorithm proceeds to reverse the edits performed to the file and at each iteration of the file, the algorithm looks for text matching the large paste event. This is done in order to detect text that may have been copied from the file itself, but not removed like in the case of a cut operation. Because copied text may have been written over the course of many keystrokes, it is important that the algorithm checks each file state prior to the paste event. If text matching the paste event is found at a previous point in the file, the large paste event is not counted as evidence of plagiarism. However, if there is no such text in the file history, this means that the text in the paste event originated from an external source. This could come from a myriad of sources that would not be plagiarism, such as another IDE that a student prefers to use, an assignment description, or another document written by the student. However, this could also be text that was found on a website, generated by a language model, or copied from another student. As there is no way to determine the origin of the text without invading student privacy and students are expected to write all of their own code, the large paste event is deemed as evidence of plagiarism. This, of course reiterates the importance of the note displayed to the user stating that the “Check for Plagiarism” tool is not perfect.

The second manner through which the plugin detects plagiarism is by looking at the proportion of keystrokes that resulted in text being inserted and text being deleted. A recent analysis of keystroke data performed by the EdwardsLab yielded the information that “58% of characters entered into a source code file are eventually deleted” [11]. Because the proportion of deletions to insertions is so high, a check on this proportion has been added. If less than 10 percent of recorded operations are deletions, the plugin determines that the file history contains evidence of plagiarism. There is a threshold, however, of 200 keystrokes, meaning that regardless of the proportion of insertions to deletions, if a file contains less than 200 keystrokes, this manner of detecting plagiarism is not used. I added this check to counteract the students who recognize that large paste events are flagged as evidence of plagiarism and begin typing out, character by character, work that they did not produce. Students who are copying information from another source rather than producing it themselves will have a much lower ration of deletions to insertions in their file history.

The algorithm is defined on the following page. Once again, “deleteText” has been shortened to “dt”, “insertText” to “it”, “editType” to “et”, and “sourceLocation” has been shortened to “sl”.

```
function FILECONTAINSEVIDENCEOFCHEATING
    $numInserts \leftarrow 0$
    $numDeletes \leftarrow 0$
    $hasEvidence \leftarrow false$
    for $(index, edit)$ in edits do
        if edit.it $\neq$ null then
            if edit.it.contains(“\n”) and edit.it $\neq$ “\n” and edit.et $\neq$ “Synthesized” then
                $prevEdits \leftarrow$ edits with a smaller timestamp than edit
                $possibleCuts \leftarrow$ prevEdits with non-empty deleteText
                $pasteCount \leftarrow 0$
                for $possibleCut$ in $possibleCuts$ do
                    if $possibleCut.dt \neq edit.it$ then
                        $pasteCount \leftarrow pasteCount + 1$
                    end if
                end for
                $state \leftarrow$ snapshot of file at index
                $copyCount \leftarrow 0$
                if not $state.contains(edit.it)$ then
                    for $e$ in $prevEdits.reversed()$ do
                        if $e.dt \neq$ “” then
                            $stateSub \leftarrow state.substr(0, e.sl)$
                            $state \leftarrow stateSub + e.dt + state.substr(e.sl)$
                        end if
                        if $e.it \neq$ “” then
                            $state \leftarrow state.substr(0, e.sl) + state.substr(e.sl + e.it.length)$
                        end if
                        if not $state.contains(edit.it)$ then
                            $copyCount \leftarrow copyCount + 1$
                        end if
                    end for
                end if
                if $pasteCount = possibleCuts.size$ and $copyCount = prevEdits.size$ then
                    $hasEvidence \leftarrow true$
                    break
                end if
            end if
            if edit.it $\neq$ “” then
                $numInserts \leftarrow numInserts + 1$
            end if
        end if
        if edit.dt $\neq$ “” then
            $numDeletes \leftarrow numDeletes + 1$
        end if
    end for
    if $numDeletes/(numDeletes + numInserts) < 0.1$ and $edits.size \geq 200$ then
        $hasEvidence \leftarrow true$
    end if
    if hasEvidence then
        Show the “Evidence found” message
    else
        Show the “No evidence found” message
    end if
    Show the “This tool is not perfect” message
    end function
```

Although I discuss student perception of the tool in a following section, it is important to note here that this algorithm has one, simple, defining assumption. In order for this algorithm to work, we assume that the students taking the introductory Computer Science course at Utah State University are not provided with source code from their instructor. Given the stringent definition of plagiarism in this course, that each student must entirely write their own code, getting nothing from external sources, I neglected to take into account that for many assignments, instructors would provide their students with starter code. Because of this, if students used the starter code provided by the instructor, their work would be marked as having evidence of plagiarism, as a large paste event to create the starter code would be the first edit in every file history. I regret to say that my failure to account for this may have impacted the survey results and students' sentiments toward the ShowYourWork plugin this semester. This tool should only be employed, as is, in environments where students will write the entirety of their code. Simple modifications to the algorithm should allow for situations in which starter code is provided.

3.5 Survey

The administered survey for this research is an expanded version of the survey administered by Hart et al. [1]. The original survey was designed to collect student opinions about the ShowYourWork plugin and a few pieces of demographic information. As the purpose of the plugin has not changed, rather only new features have been added to enhance user privacy, it is appropriate to administer the same survey. The survey was administered over Qualtrics, the same avenue through which it was administered previously. I added the following questions regarding the delete history and check for plagiarism utilities:

- I think the delete history tool enables students to cheat more easily [strongly disagree to strongly agree]
- I felt more comfortable with my keystrokes being logged knowing that I could, at any time, delete my history. [strongly disagree to strongly agree]

- I used the delete history tool on the ShowYourWork Plugin [yes/no]

  If yes:

  - I used the ShowYourWork delete history feature:
    - 0 times
    - 1-3 times
    - 4-10 times
    - More than 10 times
  - The ShowYourWork delete history feature was useful. [strongly disagree to strongly agree]
  - Why did you use the delete feature? [optional question, textbox]
- Please share any additional thoughts or insights you have about the delete history feature. [textbox]
- I used the check for plagiarism tool on the ShowYourWork Plugin [yes/no]

  If yes:

  - I used the ShowYourWork check for plagiarism feature:
    - 0 times
    - 1-3 times
    - 4-10 times
    - More than 10 times
  - The ShowYourWork check for plagiarism feature was useful. [strongly-disagree to strongly agree]
  - The ShowYourWork check for plagiarism feature gave me a better understanding of what plagiarism looks like [strongly disagree to strongly agree]
  - Do you feel like the ShowYourWork check for plagiarism feature accurately assessed your work? [yes/no]

        * Please explain: [textbox]

- Please share any additional thoughts or insights you have about the "check for plagiarism" feature.

In order to avoid priming the subjects by asking their feelings towards the new features of ShowYourWork before they submit their answers for the other survey questions, I administered the survey as it was administered the first time. Upon completing the original survey, the subject advances to a new section of the survey, from which they cannot return to their answers in the original survey. The new section is comprised of the questions that pertain to the new tools. In cooperation with course instructors, I offered fifteen points of extra credit to any student who participated. We provided an alternative route to secure the extra credit points to students who decided not to participate in the survey. The complete survey can be found in appendix A.

3.6 Coding

After I administered survey, I created a code book for the open ended questions. The following list contains each question along with the associated codes:

1. Please share any additional thoughts you have about the ShowYourWork plugin.
    - Not useful
    - Buggy
    - Useful/good idea
    - Encouraged original work
    - Did not know about it/Did not understand how to use it
    - Forgot about it
    - Worried about plagiarism
2. Why did you use the delete feature?
    - Did not use it

- Restore old code
- Remove code
- Just to test it out

3. Please share any additional thoughts or insights you have about the delete history feature.

    - Hard to use
    - Did not know about it
    - Defeats the purpose
    - Like knowing that it is there

4. (In relation to the preceding question, "Do you feel like the ShowYourWork check for plagiarism feature accurately assessed your work?") Please explain

    - Worked Ok
    - False Positive
    - Did not know about it/did not use it

5. Please share any additional thoughts or insights you have about the check for plagiarism feature.

    - Useful/Good Idea
    - Did not work/Bad execution
    - Did not know about it/did not use it

I created the codebook by analyzing the first ten responses to each question. After I finished the codebook, two separate individuals from the EdwardsLab coded every response to each of the five free response questions. Once all questions were coded, the codes were compared using Cohen's kappa to determine inter-rater reliability [12]. The following table shows the Cohen's kappa value for each question:

Question	kappa
Please share any additional thoughts you have about the ShowYourWork plugin.	0.7935
Why did you use the delete feature?	0.9556
Please share any additional thoughts or insights you have about the delete history feature.	0.7365
(In relation to the preceding question, “Do you feel like the ShowYourWork check for plagiarism feature accurately assessed your work?”) Please explain	0.8746
Please share any additional thoughts or insights you have about the check for plagiarism feature.	0.7496
Collectively Assessed	0.8285

Table 3.7: Kappa values determined from coding results for each question. The final score reflects an aggregate Cohen’s kappa value where all questions are evaluated together.

As each Cohen’s kappa value demonstrated at least a substantial strength of agreement, with some even reaching almost perfect agreement, we determined to move forward and analyze the data with the codes assigned by the researchers [13].

CHAPTER 4
OUTCOMES AND ANALYSIS

4.1 Introduction

This section seeks to provide an overview and analysis of the collected data. I will begin by reviewing the background of the students from the survey administered for this paper and drawing comparisons between these students and the ones who participated in the research previously performed by Hart et al. [1]. I will continue by reviewing the survey responses that discuss the delete history and check for plagiarism features. Finally, I will look at the students' perception of the ShowYourWork plugin as a whole.

4.2 Survey Results

The purpose of the addition of the delete history and check for plagiarism tools was to protect user privacy and enhance their experience. Although we did not expect that a substantial proportion of the subjects would use the delete history functionality, we did expect that its presence in the tool will decrease the proportion of students who feel uncomfortable with keystroke logging. We also anticipated that the proportion of students who believe that ShowYourWork will deter plagiarism will decrease, due to this functionality. In addition to this, we expected that students would respond with "agree" or "strongly agree" to the question "I think the delete history tool enables students to cheat more easily".

As there has not been any instances of students receiving immediate feedback concerning whether their file history contains evidence of plagiarism or not, we had little expectations for the results of the section of the survey concerning the check for plagiarism tool. This portion of our research was exploratory and is anticipated to lead to new avenues of research. We hope to take the results of the check for plagiarism section to enhance the algorithms that are being used to detect cheating and tune them to student's needs and data.

We will analyze the questions in the following order:

1. Questions pertaining to a student's background.

2. Questions pertaining to the Delete History feature.

3. Questions pertaining to the Check for Plagiarism feature.

4. Questions pertaining to the ShowYourWork plugin as a whole.

Only the results of the first two sections will be compared to survey results from a previous semester (Spring 2022), as the final two sections were new to this semester's survey.

4.2.1 Student Background

In the student background section of the survey, I highlight a few questions. In figure 4.1, we can see that the distribution of experience for incoming students was largely similar to that of the previous year.

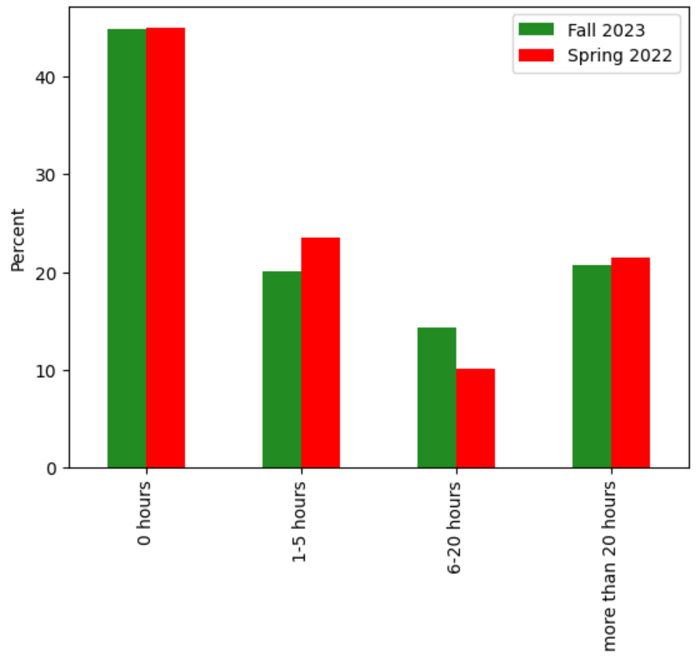

Fig. 4.1: How much computer programming experience did you have before taking CS 1400?

This shows a level of consistency between the data points collected in 2022 and those collected in this survey.

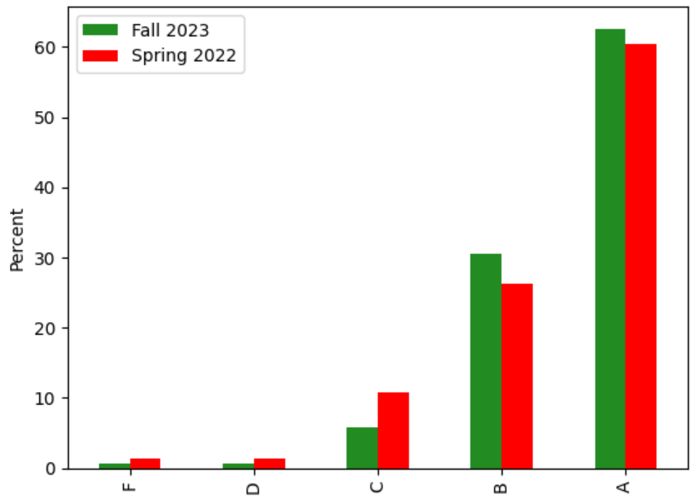

Fig. 4.2: How much computer programming experience did you have before taking CS 1400?

In figure 4.2, we do not see a large variation in student responses from year to year. While this normally would not be very noteworthy, other than to help reinforce the previous year's data, it is interesting to see little or no change given the students' responses to specific questions later on in the survey. This graph indicates that student's perceived personal performance was unaffected by the changes made in the ShowYourWork plugin.

Another question of interest, in figure 4.3 deals with the duration of time that passed before students forgot about the plugin.

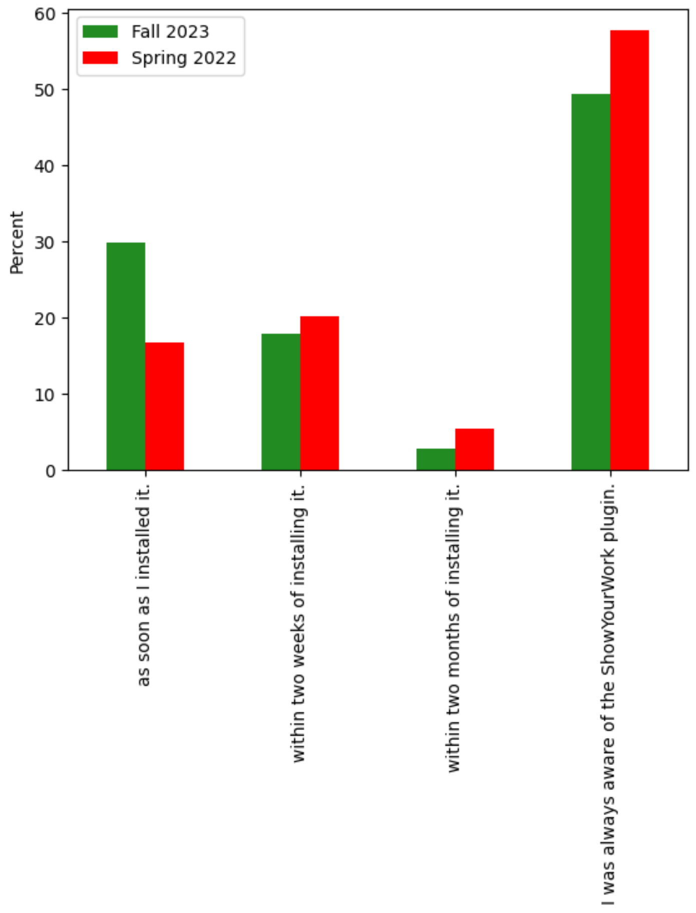

Fig. 4.3: I forgot about the ShowYourWork plugin

The responses in figure 4.3 seem to indicate that more students forgot about the plugin in the Fall 2023 semester than did in the Spring 2022 semester. While the data serves as a strong indication that the new changes to the plugin were effective at decreasing anxiety, this data may also be a result of many students reporting bugs with the plugin this semester. It is possible that students simply gave up on using it and forgot about the plugin entirely as a way to avoid dealing with the bugs. The bug issues may be a compelling source for more students forgetting about the plugin, however the results of the question “I used the ShowYourWork playback feature”, detailed in figure 4.4, help demonstrate that a larger proportion of students interacted with the plugin in the Fall 2023 semester than in the Spring 2022.

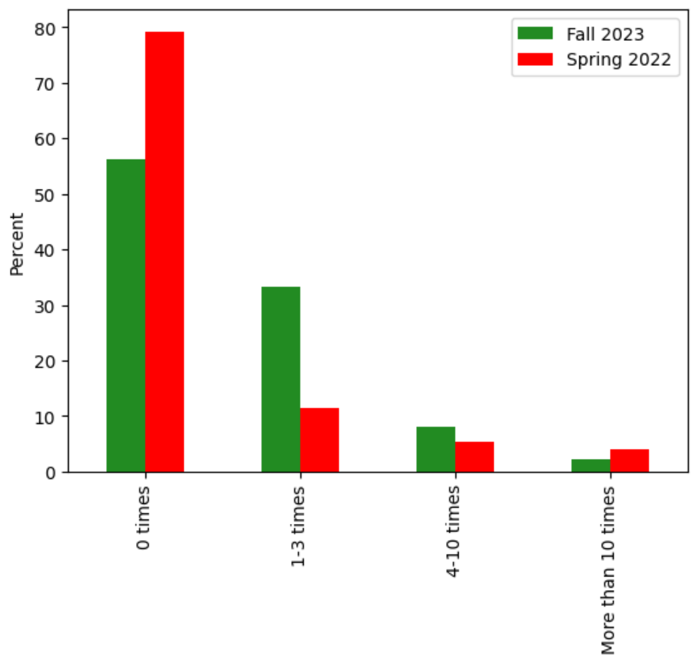

Fig. 4.4: I used the ShowYourWork playback feature

While the results that more students forgot about the plugin and more students interacted with the plugin may seem counter intuitive, it indicates that students were able to interact with the plugin to a degree that satisfied their curiosity, and then forget about it. The delete history feature had the goal to decrease the anxiety that students felt about the plugin and the keystroke logging process, and this data is an indication that this was the case.

4.2.2 Perception of the Delete History and Check for Plagiarism Features

The survey gave us a better understanding of how students feel about the delete history feature. As previously stated, the entire second half of the survey was solely directed towards these changes.

The first question that was given in this section of the survey was designed to determine students' feelings about cheating and the delete history feature. The results are shown in figure 4.5.

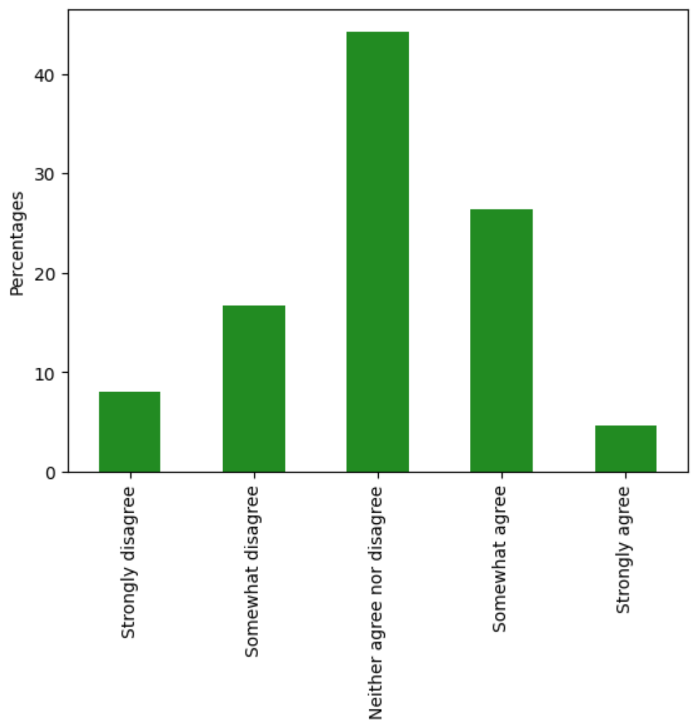

Fig. 4.5: I think the delete history tool enables students to cheat more easily.

We can see that the results are fairly evenly distributed. There does not appear to be a strong consensus among the students on this issue. Interestingly, the distribution of responses to the question “I felt more comfortable with my keystrokes being logged knowing that I could, at any time, delete my history” is almost identical to that of the previous question.

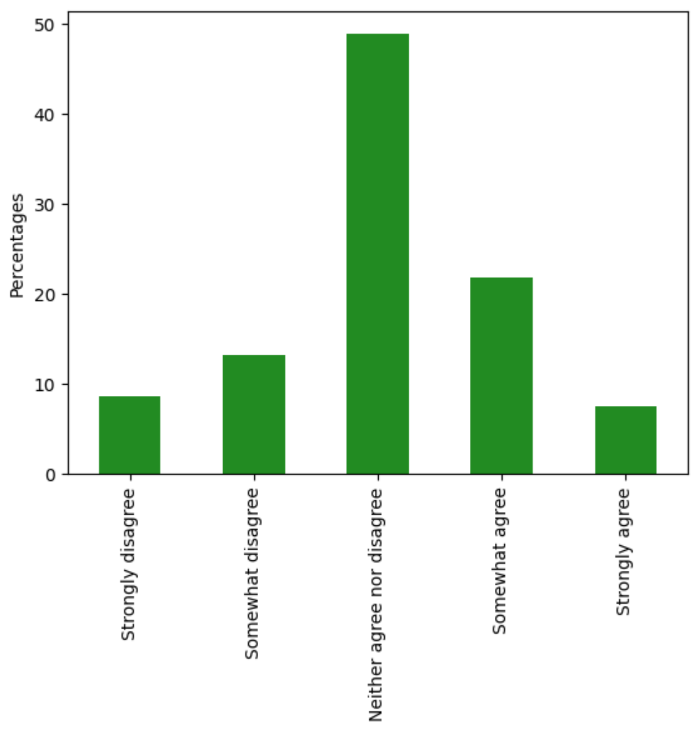

Fig. 4.6: I felt more comfortable with my keystrokes being logged knowing that I could, at any time, delete my history.

The number of people who used the delete history feature was very limited, and no one reported using the delete history feature more than four times.

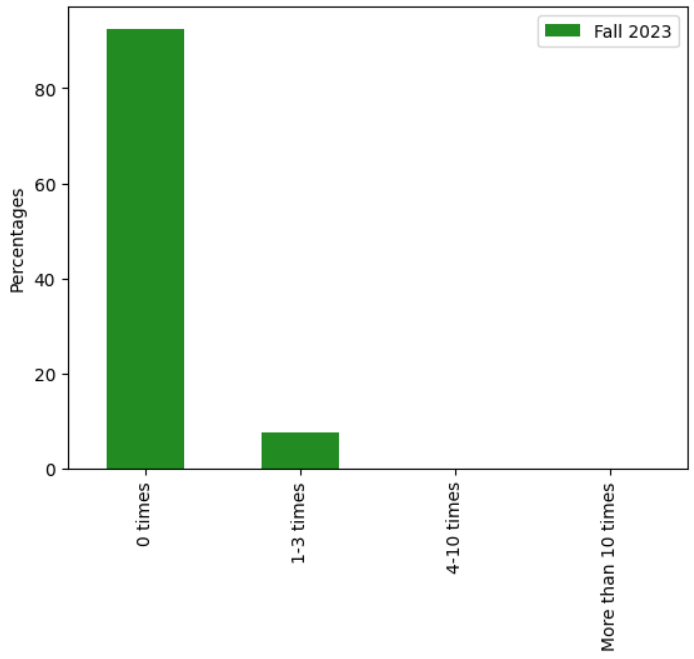

Fig. 4.7: I used the ShowYourWork delete history feature

We anticipated that the majority of students would not feel the need to use the feature at all. We also anticipated that the delete history feature would act much like a placebo, with the positive reactions to the feature mostly coming directly from the fact that it exists, rather than a result of its use. This data does not reflect the true proportion of students who would have used the feature, as many reported that they did not know about its existence. When asked the reason behind using the delete feature, most of the students that used it said they either used it to restore old code or to feel like they had a fresh start on a problem. One student said, “I used it because I had started coding and decided to restart because I felt I was going the wrong way. I didn’t want that first attempt to be shown.”

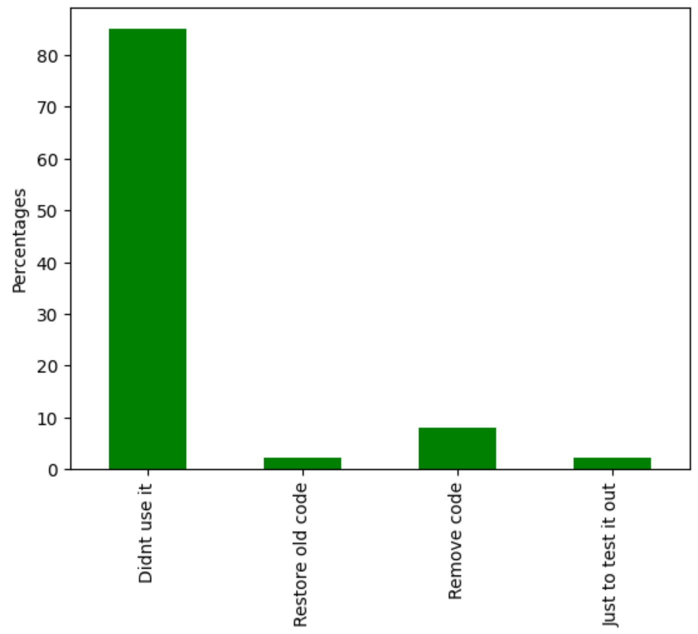

Fig. 4.8: Why did you use the delete feature?

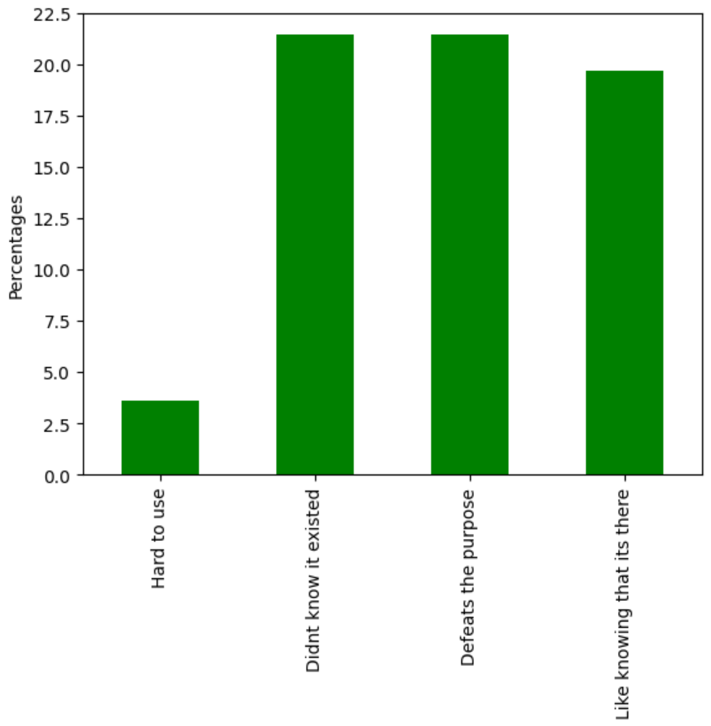

Fig. 4.9: Please share any additional thoughts or insights you have about the delete history feature.

Figure 4.9 shows the results to the open ended question, “Please share any additional thoughts or insights you have about the delete history feature”. These results coincide with what we have already established, students felt as though the delete history feature defeated the purpose of the plugin. With comments like, “It was strange to include it”, “It could probably make it easier to get away with cheating but I don’t know.”, and “I think it’s useful to have but could kind of defeat the purpose of the program.”, it is clear to see that students predominantly feel like the ShowYourWork plugin is designed to catch cheaters. One of the positive responses under the “Like knowing that it’s there” section helps illustrate that students recognize the delete history feature as a positive thing for privacy, while still acknowledging that cheating would be an issue, “I like knowing that there is a possibility to delete history in case I did get frustrated and keyboard smashed or wrote something I didn’t mean. I can see how it could be used to cheat, but overall plagiarism can be caught in other ways too”. To continue on the thought that privacy is important, another student said, “I think it is good to have, because it allows for a little more privacy for when you a make big mistake or something like that”. This helps demonstrate that, while the tool can be used to remove evidence of plagiarism, students felt that it was important to be able to remove evidence of messing up.

Moving into a discussion of the check for plagiarism feature, I reiterate that this tool did not function as planned. This, however, has given us new insights into how this feature might be implemented in the future, and also highlights the need to discuss how the checker works with the students. Because of the stringent definition of plagiarism in the introductory Computer Science course at Utah State University, which is “any code that was not written by you is considered plagiarism”, I wrote the algorithm to detect any code that may not have been written by the student. The obvious exception to this, of course, would be starter code provided by the teachers themselves. I failed to consider that nearly all assignments would contain one form of starter code or another, and thus the tool was rendered useless at its intended purpose. All of this, however, is not to say that we did not learn from the experience. In figure 4.10 we can see that nearly all students who used the check for plagiarism feature reported that it gave false positives.

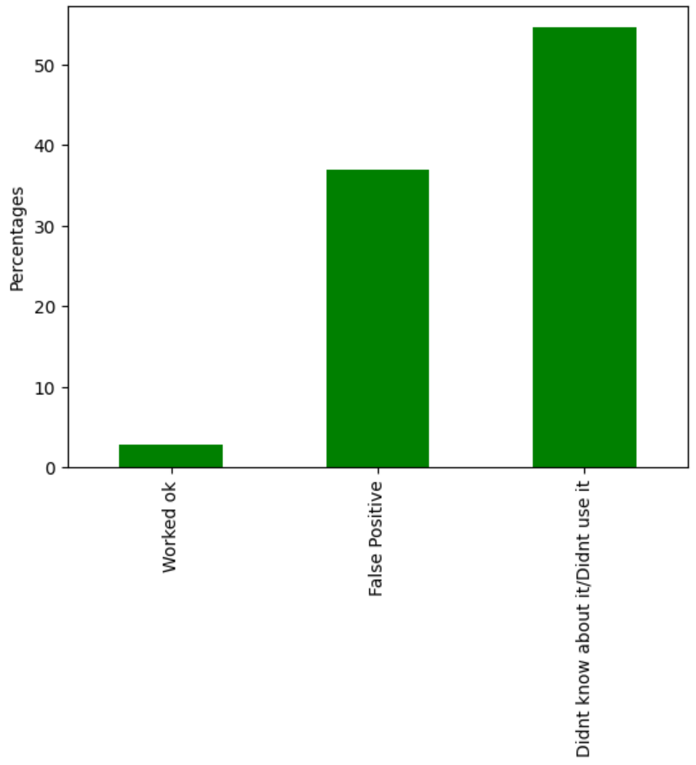

Fig. 4.10: (In relation to the preceding question, "Do you feel like the ShowYourWork check for plagiarism feature accurately assessed your work?") Please explain

The feedback on why the students felt that the checker was inaccurate is where the valuable information resides. The following quote sums up the students' overarching sentiment with brevity, "It is not always accurate and it is dangerous to accuse students of plagiarism based on what a computer program says". Many students reported that they felt uncomfortable with the checker, not because it told them that they had evidence of plagiarism, but because it did not tell them what evidence it found. One student mentioned, "It kept flagging me one the task 1 for assignment nine and for the life of me I couldn't figure out why or what it was flagging. It also flagged me on task three, but I wasn't plagiarizing, and I couldn't think of a reason why it was flagging me. If it showed exactly what code was being flagged or why, I think it would be a lot more useful." Because the tool did not tell the students what evidence it found, that left them to try and determine what it was checking for on their own. Many students arrived at the correct conclusion, finding that the started code was the source of the issue. However, a few students made comments indicating that they believed their work was being compared to other students' work in order to determine possible sources of plagiarism. One student made the comment, "I never used it. However, I think that in programming, that there are a lot of "right ways" to do certain things, and so just because my code is the same as/similar to another code doesn't necessarily indicate plagiarism, but maybe there's a way to do something that is better than any other way to do it. But maybe I don't understand the feature works or how it detects plagiarism." This response, and others like it help show that students believe that the plugin was making extrinsic comparisons rather than simply performing an internal analysis of the code in the file.

The tool may have been more well received if it had not only allowed for starter code, but also if we had a chance to talk with the students about what the tool was doing. An explanation, such as the following, may go a long way to help calm students fears about this tool, "The check for plagiarism tool only looks at the file you are currently editing, and will not compare your work to other peoples work. Beyond this, the check for plagiarism tool does not consider any signs of deleted history as evidence when looking for plagiarism. Finally, the tool allows you to use code that you have written within the file itself. Meaning that if you wrote code on one line and copied it to the next, it will not count that as plagiarism." It is important to note that the explanation above mostly outlines what the tool does not do, rather than delving into how the tool checks for plagiarism. This was intentional but may not be the only way to go about informing the students of this tool. We may see benefit in telling the students exactly what the tool is looking for, which was large paste events not originating from the source code, and a proportion of insertions to deletions. As stated in the previous works section, many students simply do not understand what plagiarism looks like and do not cheat knowingly [5]. Providing students with an instantaneous way of knowing whether what they did was right or wrong could help prevent cases of plagiarism of this nature. In addition to an explanation of what the tool is doing, it would also be beneficial, for the reasons listed above, to highlight what the checker considered to be "evidence". While this tool failed to complete the purpose we had intended in this round of research, I recognize that were able to better understand how students think because of it.

4.2.3 Perception of the ShowYourWork Plugin

The coded responses to the question "Please share any additional thoughts you have about the ShowYourWork plugin" help to establish general trends in the perception of the ShowYourWork plugin. Figure 4.11 shows the coded responses and the proportion of responses that fell into each category.

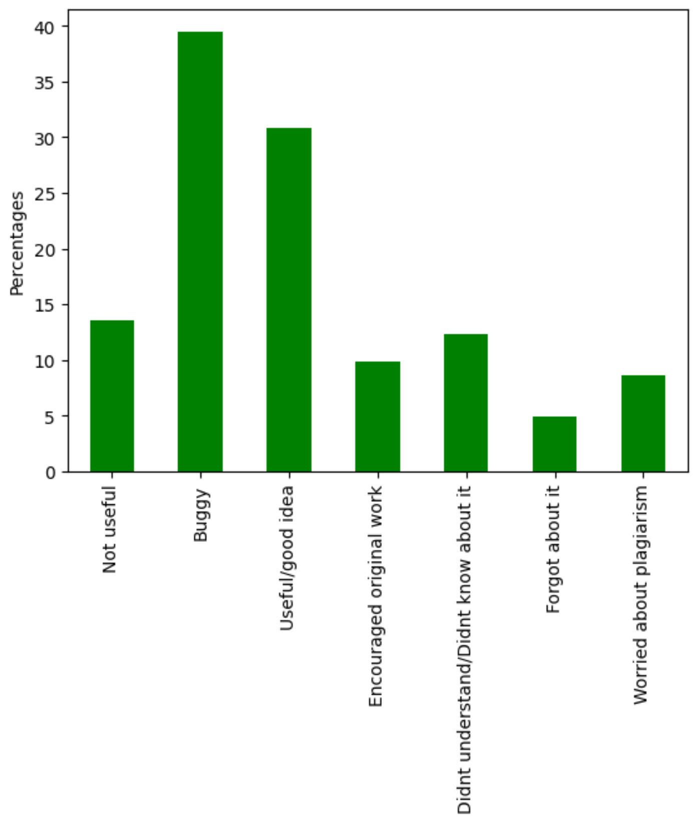

Fig. 4.11: Please share any additional thoughts you have about the ShowYourWork plugin.

The most commonly cited topic in the responses to this question was that the plugin was "buggy". Through the course of this semester the plugin had to undergo substantial changes that enhanced performance and made the plugin easier to use. Many of these changes were forced to be released because the CS1400 classes elected to use the newest version of PyCharm, which made significant changes to how plugins were developed. For a significant portion of the semester, students who chose to use the plugins features had to deal with issues such as PyCharm crashing while using the plugin, long load times for file history, and file histories that did not reflect what students had written. Fixes for all of these issues are now finalized and the majority of the bugs have been resolved. We hope that in gathering future data we can avoid the problems we experienced this year by encouraging the classes to use a version of PyCharm that supports the plugin.

Despite this, we were still able to get some wonderful feedback from students this semester. As noted in the graph, many students felt like the plugin was a good idea and that it helped to encourage original work with one student saying, "The plugin was very useful while learning to code, especially when small mistakes are made often. The replay feature was very helpful for me. I believe that it is a necessary requirement in order to ensure original work". Not all of the useful feedback was positive though. For example, one student said, "Honestly I am sometimes insecure when coding because I don't want it to seem like I am dumb when I have to retry a code multiple times(I'm also not a cs major or anything I took this class for fun). Show Your Work plug-in just made it a little scarier to mess up even if it showed progress and learning". Responses like this demonstrate a perfect use case for the delete history feature and one of the reasons that I chose to implement it. It was disheartening to see that many students, such as the one above, did not know about the delete history feature and many students reported not understanding how the plugin worked. Many of the responses to the free response questions mirrored the sentiment shared in this student's answer, "I did not know that there were features that I could have used and wished that I knew that before now." The fact that students simply didn't understand how to use the plugin surprised me, as I created an instructional video that detailed how to use the features that the plugin offered. The instructors of the class also introduced the plugin and discussed its use throughout the semester. In order to combat this problem, a more comprehensive version of the instructional video should be made and and a greater effort to discuss the plugins uses needs to be put forward. While the instructors are not responsible for the plugin, nor should they feel obligated to discuss it in class, their help moving forward will be instrumental in enabling the students to use the tools that we provide for them. It would be beneficial to add a portion to the CS1400 syllabus that discusses the use of the plugin and references places that students can go to learn more about how to use it.

As expected, less students reported feeling as though they would be caught if they plagiarized in the Fall 2023 semester than in the Spring 2022 semester. In figure 4.12, we see a massive drop in students who responded with “Strongly agree” to the question “I believe that I would get caught if I plagiarized in CS 1400.”

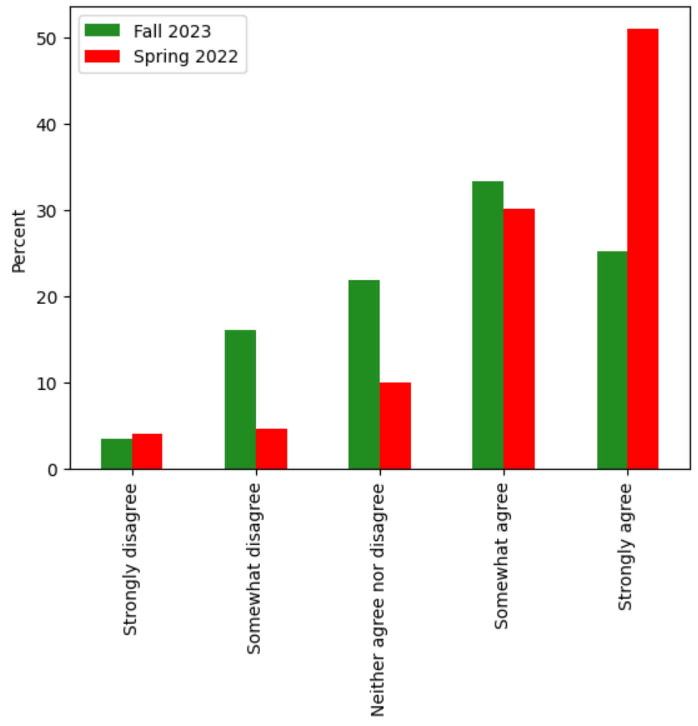

Fig. 4.12: I believe that I would get caught if I plagiarized in CS 1400.

There can be multiple sources for this decrease. The addition of the delete history tool would be one contributing factor, with another possible contributor to this being the advent and rise of large language models. With more tools to facilitate cheating, it is likely that students feel more capable of getting away with plagiarism than ever before, and this is shown figure 4.12. Beyond this, students seem to have less confidence in their instructors abilities to detect plagiarism as well. In figure 4.13 we can see that the confidence that students have in their instructor's ability to detect plagiarism has also decreased.

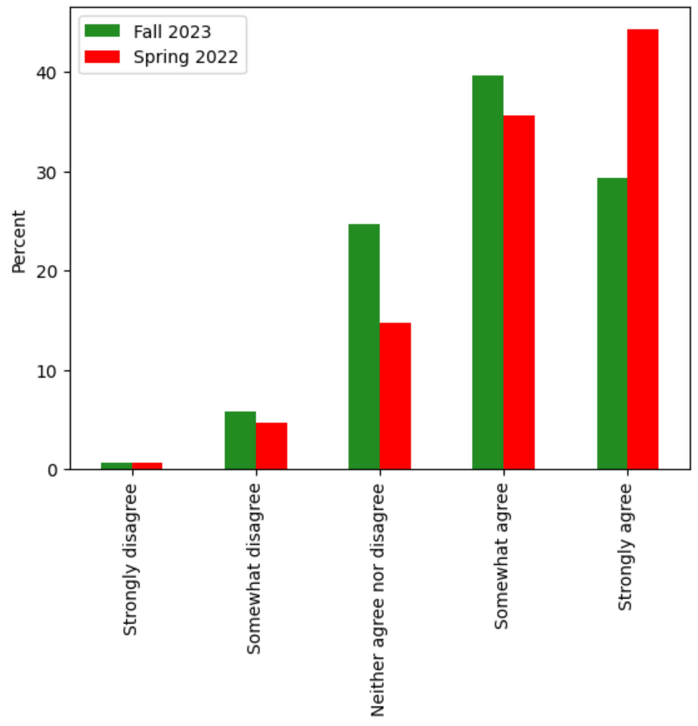

Fig. 4.13: The CS 1400 instructor is capable of finding cases of plagiarism.

These results were expected and deemed an appropriate sacrifice in order to help students feel more comfortable with the ShowYourWork plugin and the concept of keystroke logging. Despite these changes, students still overwhelmingly report that they believe that the ShowYourWork plugin helps instructors identify plagiarism more easily. This is shown in figure 4.14.

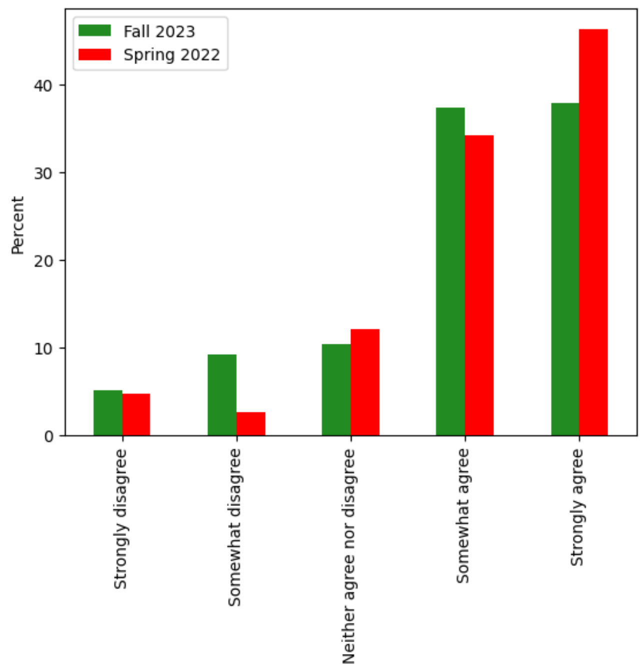

Fig. 4.14: The ShowYourWork plugin that logs keystrokes would make it easier for the CS 1400 instructor to find cases of plagiarism.

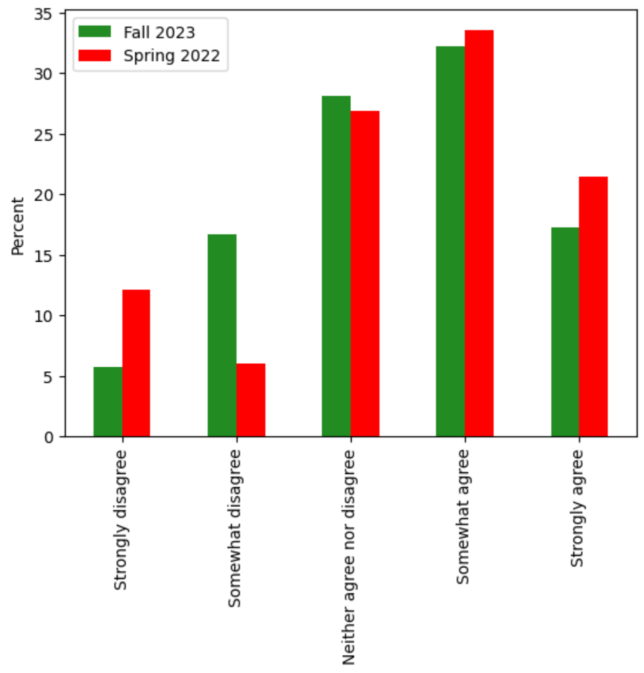

Fig. 4.15: I think the ShowYourWork plugin was a good idea.

The final survey question that I address in this section is shown above in figure 4.15. The results of the Fall 2023 survey show that less students thought the plugin was a good idea than in Spring 2022. While this is true, it illustrates what I consider to be the most important discovery of this survey. Students think that the addition of the delete history feature decreased the effectiveness of ShowYourWork plugin as a method to deter and catch instances of plagiarism. This is correct, however, the effectiveness of the plugin as a whole has been increased. The sole purpose of the ShowYourWork plugin is not just to deter and prevent plagiarism. While this is important, the primary purpose of the plugin is to increase the quality of instruction at our university and enhance student experience. This is all to be done without overstepping and invading the students' right to privacy. We recognized that students felt that the plugin was too invasive, and we responded by implementing the changes outlined in this paper. Now the students report in larger numbers that the plugin is not effective in accomplishing its purpose. This is simply because the students' exposure to the plugin is one dimensional. It is introduced as a tool to stop plagiarism, and that is all they think of it.

Responses like to survey questions such as "I think that it works but at the end of the day it can be turned off. If someone is really dedicated to cheating they could always write up the code then rewrite it in the editor to make it look like it was done from scratch" help demonstrate that the students understanding of the plugin only extends to plagiarism detection. What needs to change, as a result of these findings, is how we present the plugin to students. As students come to understand the purpose of the plugin in its entirety, plagiarism detection and data collection, together, we expect that their perception of the plugin will improve. The driving force behind ShowYourWork is a desire to help students learn better, and that is what students need to understand in order to appreciate keystroke logging and in order to clarify why we are willing to add a tool that seemingly helps cover up plagiarism.

CHAPTER 5
DISCUSSION AND FURTHER RESEARCH

5.1 Threats to Validity

We recognize that our sample population is extremely limited in scope. The conclusions about student attitudes towards plagiarism will be, in large part, limited to students attending Utah State University. Beyond this, there are a number of factors that may influence students' feelings towards the ShowYourWork plugin this semester. One of the most prominent examples of this is that the two sections of Utah State University's introductory Computer Science course did not follow the same lesson plans and assignment structures. One class followed the structure that was used in the semesters in which the survey was previously administered, and one did not. In addition to this, the class that did not follow the previous structure did not use the PyCharm IDE and ShowYourWork plugin for half of the semester, opting to use another IDE instead.

Perhaps the most disappointing portion of this research is the failure of the check for plagiarism utility. Closely behind this was the constant stream of bug reports that needed to be addressed during the semester, resulting in frequent changes and updates to the plugin. It is hard to know if students' overall perception of the plugin has truly dropped because of the changes that we have implemented this semester, or simply because they could not get the program to run on their machine.

5.2 Further Research

While there is much work that still needs to be done as far as the ShowYourWork plugin is concerned, the EdwardsLab has worked diligently to ensure that the next release of the ShowYourWork plugin is stable and bug free. Repeating this survey in a subsequent semester would be wise in order to validate the findings shown in this paper. This would help to mitigate the effects that bugs have on the overall perception of the show your work plugin and provide cleaner data and insights into students' feelings about the delete history and check for plagiarism features.

In addition to this, the check for plagiarism feature has many areas for improvement and retesting. The algorithms used to check for plagiarism in the students' code should be revisited and tested for effectiveness on a previously collected student dataset before being put to use on actual students again. These changes and studies would require a significant amount of work and could be treated as the entire subject for a follow up paper. After achieving accuracy in plagiarism detection within a real student dataset, it would be good to update the tool to highlight the reasons that it flagged a student for evidence of plagiarism. It would be interesting to keep track of these occurrences and look at the frequency with which evidence of plagiarism is reported and how often the delete history is used in conjunction with that report.

Finally, the most pertinent change to the ShowYourWork plugin is not an implementation detail, rather, a presentation detail. As previously stated, we need to do a better job of explaining the purpose of this plugin to the students here at Utah State University. It would be interesting to compare perceptions of the plugin between two groups, where one group is told that the ShowYourWork plugin has the primary purpose of detecting plagiarism, and another group is told that the plugin simply captures data for future research. This would be possible to execute in a single semester, as multiple sections of the course are given at Utah State University each semester.

5.3 Conclusion

Within this research we have sought to decrease student anxiety about keystroke logging and enhance student privacy. We have accomplished the portion of our goal dealing with student privacy, however, the overall perception of the ShowYourWork plugin this semester was lower than that of previous semesters. We recognize that there was a myriad of factors that affected the students perception of the plugin, such as frequent bugs and a poorly implemented check for plagiarism tool. We have learned that students value privacy, but are confused at the seemingly paradoxical addition of the delete history feature in an application they perceive solely as a plagiarism detector. I conclude by saying that the most drastic change in student perception of the ShowYourWork plugin will occur when we can effectively communicate the altruistic goals of the plugin and its holistic function with the research performed by the EdwardsLab.

REFERENCES

[1] K. Hart, C. Mano, and J. Edwards, "Plagiarism deterrence in cs1 through keystroke data," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 493–499. [Online]. Available: https://doi.org/10.1145/3545945.3569805

[2] C. Syndergaard and J. Edwards, "A review of student attitudes towards keystroke logging and plagiarism detection in introductory computer science courses," *In Review*, 2023.

[3] R. Fraser and D. Cheriton, "Collaboration, collusion and plagiarism in computer science coursework," *Informatics in Education*, vol. 13, 09 2014.

[4] E. Roberts, "Strategies for promoting academic integrity in cs courses," in *32nd Annual Frontiers in Education*, vol. 2, 2002, pp. F3G–F3G.

[5] E. M. Beasley, "Students reported for cheating explain what they think would have stopped them," *Ethics & Behavior*, vol. 24, no. 3, pp. 229–252, 2014. [Online]. Available: https://doi.org/10.1080/10508422.2013.845533

[6] T. B. Murdock and E. M. Anderman, "Motivational perspectives on student cheating: Toward an integrated model of academic dishonesty," *Educational Psychologist*, vol. 41, no. 3, pp. 129–145, 2006. [Online]. Available: https://doi.org/10.1207/s15326985ep4103_1

[7] D. Callahan, *The Cheating Culture: Why More Americans Are Doing Wrong to Get Ahead*. Orlando, Fla: Harcourt, 2004.

[8] A. Jordan, "College student cheating: The role of motivation, perceived norms, attitudes, and knowledge of institutional policy," *Ethics Behavior - ETHICS BEHAV*, vol. 11, pp. 233–247, 07 2001.

[9] G. Whale, U. of New South Wales. Department of Computer Science, U. of New South Wales. School of Electrical Engineering, and C. Science, *Plague: Plagiarism Detection Using Program Structure*, ser. DCS report. School of Electrical Engineering and Computer Science, University of New South Wales, 1988. [Online]. Available: https://books.google.com/books?id=ONVXYgEACAAJ

[10] L. Yan, N. McKeown, M. Sahami, and C. Piech, "Tmoss: Using intermediate assignment work to understand excessive collaboration in large classes," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 110–115. [Online]. Available: https://doi.org/10.1145/3159450.3159490

[11] J. Edwards, K. Hart, and R. Shrestha, "Review of csedm data and introduction of two public cs1 keystroke datasets," *Journal of Educational Data Mining*, vol. 15, no. 1, p. 1–31, Mar. 2023. [Online]. Available: https://jedm.educationaldatamining.org/index.php/JEDM/article/view/581

[12] M. Banerjee, M. Capozzoli, L. McSweeney, and D. Sinha, "Beyond kappa: A review of interrater agreement measures," *Canadian Journal of Statistics*, vol. 27, no. 1, pp. 3–23, 1999. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.2307/3315487

[13] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977. [Online]. Available: http://www.jstor.org/stable/2529310

APPENDICES

APPENDIX A

End of Semester Survey

A.1 End of Semester Survey

Survey for Plagiarism Study - Fall 2023

Section 1 of the Survey

- Which ethnicities do you identify as? [Select all that apply]
  - Asian
  - Black or African American
  - Native Hawaiian or Pacific Islander
  - Hispanic or Latinx
  - Middle Eastern
  - Native American
  - White or Caucasian
  - Other (with a field to fill in)
- What is your gender?
  - Man
  - Woman
  - Transgender
  - Non-binary or agender
  - Self-describe [answer]

- Prefer not to answer

- What is your age? [numerical input]
- How much computer programming experience did you have before taking CS 1400?
  - 0 hours
  - 1-5 hours
  - 6-20 hours
  - more than 20 hours
- What grade do you expect to get in CS 1400?
  - A
  - B
  - C
  - D
  - F
- In CS 1400, plagiarism is defined as using someone else's Python code as your own. [The following questions are a five-point Likert-style scale of "strongly disagree" $\rightarrow$ "strongly agree"]
- I was tempted to plagiarize in CS 1400 this semester.
- I believe that I would get caught if I plagiarized in CS 1400.
- If I copied someone else's code, I would be capable of making it look like I didn't copy.
- The CS 1400 instructor is capable of finding cases of plagiarism.

ShowYourWork Study

- I participated in the ShowYourWork research study this semester [yes/no]

- If yes:
  - The ShowYourWork plugin that logs keystrokes makes it easier for the CS 1400 instructor to find cases of plagiarism.
  - I would have been more tempted to plagiarize if I hadn't been required to submit the ShowYourWork log file.
  - I forgot about the ShowYourWork plugin
    - as soon as I installed it.
    - within two weeks of installing it.
    - within two months of installing it.
    - I was always aware of the ShowYourWork plugin.
  - I used the ShowYourWork playback feature
    - 0 times
    - 1-3 times
    - 4-10 times
    - More than 10 times
  - The ShowYourWork playback feature was useful. [strongly disagree $\rightarrow$ strongly agree]
  - I think the ShowYourWork plugin was a good idea. [strongly disagree $\rightarrow$ strongly agree]
  - Please share any additional thoughts you have about the ShowYourWork plugin. [text box]

[End of the first section, upon passing from the first section to the second section, students will not be able to return to the first section]

Section 2 of the Survey

- I think the delete history tool enables students to cheat more easily [strongly disagree → strongly agree]
- I felt more comfortable with my keystrokes being logged knowing that I could, at any time, delete my history. [strongly disagree → strongly agree]
- I used the delete history tool on the ShowYourWork Plugin [yes/no]
  - If yes:
    - I used the ShowYourWork delete history feature:
      - 0 times
      - 1-3 times
      - 4-10 times
      - More than 10 times
    - The ShowYourWork delete history feature was useful. [strongly disagree → strongly agree]
    - Why did you use the delete feature? [optional question, textbox]
  - Please share any additional thoughts or insights you have about the delete history feature. [textbox]
- I used the check for plagiarism tool on the ShowYourWork Plugin [yes/no]
  - If yes:
    - I used the ShowYourWork check for plagiarism feature:
      - 0 times
      - 1-3 times
      - 4-10 times
      - More than 10 times
    - The ShowYourWork check for plagiarism feature was useful. [strongly disagree → strongly agree]

- ∗ The ShowYourWork check for plagiarism feature gave me a better understanding of what plagiarism looks like [strongly disagree $\rightarrow$ strongly agree]
- ∗ Do you feel like the ShowYourWork check for plagiarism feature accurately assessed your work? [yes/no]
- ∗ Please explain: [textbox]
- ∗ Please share any additional thoughts or insights you have about the ”check for plagiarism” feature.

Completion Page

Thank you for participating in the survey. Please submit a screenshot of this page to the “Plagiarism study” assignment in Canvas to receive the extra credit points.