Detecting Plagiarism based on the Creation Process

Johannes Schneider, Avi Bernstein, Jan vom Brocke, Kostadin Damevski, and David C. Shepherd

***Abstract*—To this date, all methodologies for detecting plagiarism have focused on investigating the final digital "outcome", eg. a document or source code. Our novel approach takes the creation process into account using logged events collected by special software or by macro recorders found in most office applications. We look at interaction logs of an author with the software used for creation of the work. Detection relies on comparing histograms of command usages of multiple logs. A work is classified as plagiarism, if its log deviates too much from logs of "honestly created" works or if its log is too similar to another log. The technique supports detecting plagiarism for digital outcomes stemming from *unique* tasks such as thesis as well as *equal* tasks such as assignments where the same problem sets are solved by many students. Evaluation focuses on the latter case using collected logs by an interactive development environment (IDE) from more than 60 students for three programming assignments.**

***Index Terms*—plagiarism detection, log analysis, distance metrics, histogram based detection, outlier detection of logs**

I. Introduction

Prominent cases of plagiarism like the one of the former German secretary of defense Guttenberg, who copied large parts of his Ph.D. thesis, have helped in creating more public awareness for this serious problem. In particular, persons with decision power in politics and industry should display high forms of integrity. To this end, a systematic eradication of immoral behavior is needed already during education, eg. reliable detection of plagiarism. Even though software is available to support identification of plagiarism, it can often be defeated by simple manipulation techniques such as substituting words by synonyms, ie. rogeting. The reason being that plagiarism detection software often only detects exact matches of text. Therefore, we introduce a novel mechanism that supports identification of fraudulent works such as thesis or assignments. Our idea is to capture the creation process and compare the generation process of individual works among each other rather than just focusing on the final products. The creation process is represented by a log comprising of a sequence of events which are collected automatically during the making of a digital product.

J. Schneider is with the Institute of Information Systems, University of Liechtenstein, Liechtenstein.
E-mail: johannes.schneider@uni.li
A. Bernstein is with Department of Informatics, University of Zurich, Switzerland.
E-mail: bernstein@uzh.ch
J. vom Brocke is with the Institute of Information Systems, University of Liechtenstein, Liechtenstein.
E-mail: jan.vom.brocke@uni.li
K. Damevski is with the Department of Computer Science, Virginia Commonwealth University, Richmond, VA, 23284, U.S.A.
E-mail: damevski@acm.org
D. Shepherd is with ABB Corporate Research, Raleigh, NC, 27606, U.S.A.
E-mail:david.shepherd@us.abb.com

```
Application.Move Left:=0, Top:=0
Selection.TypeText Text:="Hello,"
Selection.TypeParagraph
Selection.TypeText Text:="how were you?"
Selection.MoveLeft Unit:=wdCharacter, Count:=7
Selection.TypeBackspace
Selection.TypeText Text:="a"
Selection.MoveUp Unit:=wdLine, Count:=1
Selection.MoveLeft Unit:=wdCharacter, Count:=7, Extend:=wdExtend
Selection.Font.Bold = wdToggle
Selection.MoveDown Unit:=wdLine, Count:=1
Selection.TypeBackspace
Selection.MoveRight Unit:=wdCharacter, Count:=3, Extend:=wdExtend
Selection.Font.Italic = wdToggle
Selection.MoveRight Unit:=wdCharacter, Count:=2
Selection.MoveRight Unit:=wdCharacter, Count:=4, Extend:=wdExtend
Selection.MoveLeft Unit:=wdCharacter, Count:=1, Extend:=wdExtend
Selection.Font.Grow
Selection.Font.Grow
Selection.Font.Grow
```

Hello,

how *are* YOU?

Fig. 1. Log created using the macro recorder in Microsoft Word. The outcome is shown on the right.

User behavior and, sometimes, internal processes of the software used to make the digital product are tracked by recording events. A simple example is shown in Figure 1. It illustrates that logs typically contain much more information than the final digital product, e.g. it contains the entire change history of a document in chronological order. A log also is more machine friendly to process, since it usually consists of a sequence of events given in raw simple text format, whereas a final product could be a text with different fonts and colors or even a graphic. These characteristics lead to several opportunities for plagiarism detection. Strengths and limitations are presented in the discussion in Section VIII. There are multiple architectures and technical options for log creation (Section III) that lead to different detection and cheating strategies (Section IV). For automatic plagiarism detection, we propose mechanisms based on histograms of events of a log (Section V). Though automatic detection is the focus of this work, it might not yield definite results. Thus, manual inspection might be necessary for which the main ideas are stated in Section VI. For evaluation we focus on programming assignments in software engineering in Section VII. To the best of our knowledge the only reliable way to avoid detection by our creation process based technique, requires either detailed knowledge of the inner workings of the creation software or a significant degree of manual work. Thus, cheating is less attractive.

II. Related Work

(Software) plagiarism has been discussed from the point of view of students (and university staff) in [11], [17]. They give a definition of plagiarism and discuss student awareness of plagiarism. The work [11] also mentions that more than 80% of surveyed stuff check for plagiarism and about 10% use dedicated software, while the others relies on manual inspection.

The comprehensive survey [21] discusses plagiarism in general and also three detection methods focusing on texts: document

source comparison (such as word stemming or fingerprinting), search of characteristic phrases and stylometry (exploiting unique writing styles of persons). A taxonomy for plagiarism focusing on linguistic pattern is given in [3]. They conclude that current approaches are targeted to determine "copy-paste" behavior but fail to detect plagiarism due to presentation of ideas in different words. Though [21] mentions some (electronic) tools for detection, there are explicit surveys covering these tools and systems [20], [5], [23], [25]. Discussed techniques involve computation of similarity for documents, ie. attribute counting or cosine distance [16]. The relationship between paraphrasing and plagiarism has been looked at as well, eg. [23], [38]. Citation-based plagiarism has also been studied widely, eg. [15], [24], [23]. This approach uses proximity and order of citations to identify plagiarism. Other techniques cover parse tree comparison (for source code) or string tiling [35]. More recent work has identified programmers using abstract syntax trees with a surprisingly high success rate despite obfuscation of code [8]. None of these tools and methods takes the creation process into account.

For source code there are a variety of special techniques at hand, eg. [4] introduces plagiarism detection of source code by using execution traces of the final program, ie. method calls and key variables. An API-based control flow graph is used in [9] to detect plagiarism. A control flow graph represents a program as nodes(statements) with an edges between two nodes if there is a transition in the graph between the nodes. API-based control flow graphs merge several statements to an API call, which is used for feature extraction and then in turn to build a classifier to detect plagiarism. The paper [37] focuses on detecting plagiarism for algorithms. The paper relies on the fact that there exist some runtime values which are necessary for all implementations of the algorithm. Therefore, any modification of the algorithm will also contain these variables. Detection of plagiarism for programming assignments stating similarity metrics of source code has been investigated in [28] together with an extensive experience report. Similarity is based on counts and sequences of reserved words in common among assignments.

A comprehensive survey of statistical methods to detect cheating is given in [7]. In a multiple choice test a student takes more time to answer a difficult question than a simple question. Statistical response time methods [7] exploit this effect, which is hard to capture for cheaters. We believe that this also holds for programming and writing a thesis. Methods addressing the intrinsic aspects of a given task have also been proposed, eg. using latent semantic indexing for text documents [2] and source code [13]. Such techniques might also be applicable for log analysis. More generally, existing insights on the knowledge generation process could be used in our work as well, eg. general findings such as in [27] or findings for specific tasks and practitioners such as the writing process of novice researchers [31]. Our proposed approach is more inductive, ie. driven by data, rather than deductive relying on general insights of cognitive processes in knowledge creation.

There are a variety of logging tools, eg. [36], [32], [18], [33]. We decided in favor of Fluorite [36], since it gives fine grained logging of events. Other tools [33] provides less fine grained logging with better privacy guarantees and gamification approaches, which is preferable for anonymous usage data collection and evaluation [14]. Mouse movements and events outside the IDE are not logged. Some tools can replay sequences of events [32].

Papers such as [34], [1] use logs to analyze the learning process of students, eg. [34] looked at novice students interacting with a complex IDE. They investigated how certain characteristics like compilation errors evolve with experience. Students in need of help are identified in [1]. They employ machine-learning using a variety of features. The only feature of the code snippets they use is the number of steps taking to solve a task of an assignment. For instance, they computed what compilation errors occurred and how their distribution changes with (more) experience of students. They identify states during the evolution of the source code and also correlate student performance with overall course performance. Other work [6] identifies patterns using frequency and size of code updates. They also perform in depth analysis of an assignment using a simple language for robot programming.

The paper [30] shows that it is possible to identify users based on the usage of Linux shell commands. Their techniques could be helpful to detect copying of partial logs. The timing between keystrokes [10] can serve as biometric data that allows to identify users. This could be valuable in our context as well. Authorship can also be traced by extracting a multitude of features from text such as word richness, punctuation etc.[19]. The work [26] provide an evaluation framework for plagiarism detection focusing on text, eg. inserting random text, changing semantic words. We also use random variations in our creation of artificial logs.

III. Definitions and Scope

Plagiarism is commonly defined as the "unjust appropriation", "theft and publication" of another author's creation including "language, thoughts, ideas, or expressions" and the representation of it as one's own. This could mean copying paragraphs from the Internet without citation, slightly modifying parts of a work of a fellow student for an assignment, paying a third party to do the actual work and so on. We mainly focus on the case that a creator attempts to copy significant parts of the digital outcome by stealing from another source. He or she might copy literally or create a modified version of the original to avoid detection, eg. by word substitution or even by changing the content in a more complex way, eg. document structure or semantics. This scenario covers the most appealing approach for a plagiarist, ie. copying as much as possible with little modification. It also includes more subtle approaches such as rewriting a given outcome and thereby avoiding the (time-consuming) process of deriving a solution for the given task. We do not attempt to detect copying of small portions of a work except if it contributes significantly to overall effort. For instance, we handle the typical scenario, where a student attempts to copy most of a course assignment or a thesis. Generally, we do not detect copying of a small paragraph. However, if the

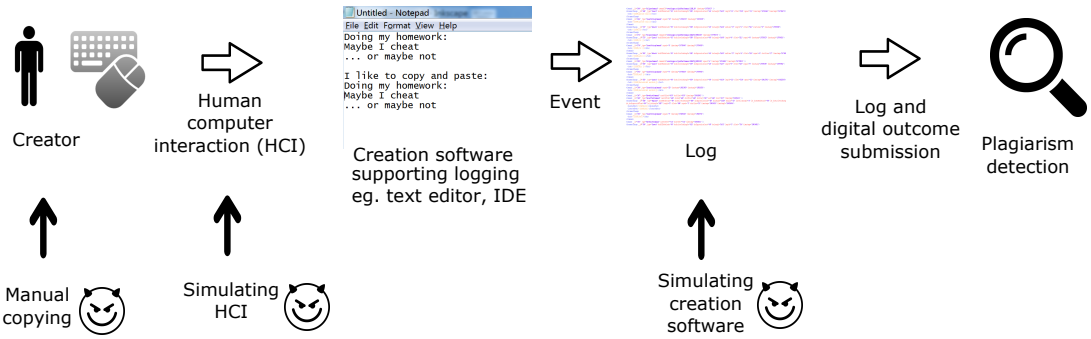

Fig. 2. Log creation process and three possibilities for cheating

paragraph is responsible for a large amount of work (eg. the most difficult part of a programming assignment) and its lack manifests in significant less time and effort (and thus in a shorter or different log), we are likely to detect it.

A *digital outcome* for an assignment or a thesis consists of files containing the final product such as formatted text of a thesis or source code. It is made by a *creation software* such as a programming IDE or a text processing software. The digital outcome is accompanied by a *log* file, documenting the creation process by a sequence of automatically logged events. A log often allows to reconstruct the digital outcome, i.e. the events in the log describe the creation process step by step and they resemble all manual editing. In this case, "replaying" the log yields the digital outcome and therefore, it is not strictly necessary to submit a log as well as the digital outcome. We assume both are submitted, since this helps in plagiarism detection by comparing the digital outcome stemming from replaying the event log and the submitted digital outcome. The creation process can be captured either by the creation software emitting events or by a general purpose human computer interaction(HCI) recorder tracking mouse and keystrokes. Both options are shown in Figures 2 and 3. The setup in Figure 3 for tracking HCI is more

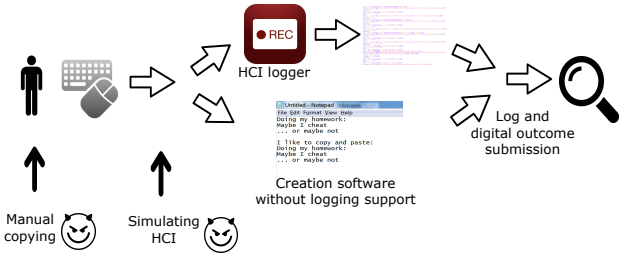

Fig. 3. Log creation by tracking human computer interaction directly

general and more widely applicable than relying on logs from the creation software, since not all applications support logging. However, many widely used tools such as Microsoft Office and OpenOffice applications come with a recorder for macros, i.e. logs, or they allow installation of plugins to record user activity, e.g. programming IDEs such as Eclipse or Visual Studio. Generally, logs stemming from the creation software contain human computer interaction categorized into specific events. For instance, a mouse click at a certain position might become a command execution event if the click was on a menu item. Additional events stemming from internal processes might also be logged. Logs from a creation software might contain more information than a direct HCI recording. Thus, they are somewhat preferable for detection, though both setups, i.e. tracking HCI (in Figure 2) and logging done by the creation software (Figure 3) allow for similar detection techniques. On the downside tracking HCI raises privacy concerns, in particular, if interaction with other applications than the creation software is also logged. Therefore, we mainly discuss the first setup as shown in Figure 2 and refer to a log as events stemming from the creation software.

```
<DocumentChange __id="368" _type="Insert" docASTNodeCount="108" docActiveCodeLength=
"658" docExpressionCount="53" docLength="1478" length="34" offset="725" repeat="33"
timestamp="19842218" timestamp2="19850590">
  <text><![CDATA[servSock = new ServerSocket(pirt);]]></text>
</DocumentChange>
<Command __id="369" _type="InsertStringCommand" repeat="34" timestamp="19842223"
timestamp2="19850601">
  <data><![CDATA[servSock = new ServerSocket(pirt);]]></data>
</Command>
<Command __id="435" _type="EclipseCommand" commandID=
"eventLogger.styledTextCommand.COLUMN_PREVIOUS" repeat="4" timestamp="19851010"
timestamp2="19851456" />
<DocumentChange __id="439" _type="Delete" docASTNodeCount="108" docActiveCodeLength=
"657" docExpressionCount="53" docLength="1477" endLine="24" length="1" offset="754"
startLine="24" timestamp="19851953">
  <text><![CDATA[i]]></text>
</DocumentChange>
<Command __id="440" _type="EclipseCommand" commandID=
"eventLogger.styledTextCommand.DELETE_PREVIOUS" timestamp="19851968" />
<DocumentChange __id="441" _type="Insert" docASTNodeCount="108" docActiveCodeLength=
"658" docExpressionCount="53" docLength="1478" length="1" offset="754" timestamp=
"19852302">
  <text><![CDATA[o]]></text>
</DocumentChange>
<Command __id="442" _type="InsertStringCommand" timestamp="19852309">
  <data><![CDATA[o]]></data>
</Command>
```

Fig. 4. Log created using a plugin in the Eclipse IDE

An *event* can either be triggered immediately by a human interacting with the creation software, e.g. by executing a specific command or by internal processes of the creation software. Commonly used commands are related to navigation, editing text (insert, replace, delete), selecting menu entries (opening or saving files) etc. Exemplary events generated by the creation software are 'autosave' events (logging that a file was saved automatically), spell check completed (indicating that a background process for checking all words in the document hass finished), breakpoint hit events in a programming IDE (saying that a program suspended execution at a breakpoint). An event often consists of a timestamp, the name of the event (typically being the command type) and details about the event such as parameters. For instance, "2015-12-08 10:00.00, InsertString, 'A', position 100:20" corresponds to a user pressing key 'A' while editing source code at line 100 column 20. The exact content of events as well as the kind of logged events might vary depending on the creation software. Figure 1 and 4 show two examples of log files originating from different applications.

We use the following *notation* throughout this work. The set of all logs is denoted by $\mathbb{L}$. A log $L \in \mathbb{L}$ consists of a

sequence of entries. The i-th entry l_i of log L corresponds to a single logged command or event. It is a tuple consisting of a timestamp t_i, an event or command c_i of a certain type. By $N(L, c)$ we denote the number of occurrences of command type c in log L. We focus on commands triggered by a human (rather than system generated events). The set of all command types is given by T. The types in a single log L are given by $T(L)$. Let $L_U \subset L$ be the subsequence of L that contains only commands of type from the set $U \subseteq T$.

IV. Cheating and Detection

We elaborate on basic detection mechanisms before defining the challenges that they imply for a cheater. We also show differences in detection of plagiarism originating from tasks such as writing a (unique) thesis or doing the same assignment as a set of students.

A. Detection Overview

Detection builds heavily upon two mechanisms. The first relies on investigating whether the distribution of events originate from honest creation or plagiarizing. The second one checks that the validity of the log, ie. that the log can have been indeed produced by the creation software. Ideally, the log can be "replayed" yielding the submitted digital outcome. The key assumption is that plagiarizing a work results in a different sequence and also distribution of events. For example, plagiarism might lead to less edit events but more copy-and-paste events. Our proposed detection relies on two dimensions, namely, checking for frequencies of a command and used command types. The histogram based technique relies on frequency counts of event types extracted from each log. It computes distances between logs for multiple samples of event types. Thus, we verify: (i) whether a cheater uses common event types as non-cheaters do and whether he did not use uncommon command types; (ii) whether the usage is approximately the correct number of times, ie. not too similar to any other log or extremely dissimilar from most other logs. In data mining terminology, we perform a nearest neighbor search for identifying manipulated logs, e.g. by changing a honestly created log. We also perform an outlier detection for finding newly created logs stemming from plagiarized work.
Invalid logs are classified as originating from attempted manipulation, eg. plagiarism. For a log to be valid it must fulfill semantical and syntactical correctness. Syntactical correctness refers to obeying specific rules that determine how events are represented in the log. We say that a log is *semantically* correct, if it could have been created by the creation software. For example, in an empty text editor an event for deletion of a character 'A' is likely to yield a semantically incorrect log, since the text editor is empty and, thus, the creation software could not have emitted such an event. Generally, a cheater must know the *state* of the creation software, which determines feasible events as well as the impact of events. For example, pressing the "return" key might move the cursor to a new line. But, in case the software displays a dialog with focus on the "cancel" button, such a key press might close the dialog. It might be possible to verify a log by 'replaying' its events. Invalid logs are likely to result in an error message while replaying. For instance, standard office tools come with a recorder for macros. These macros can function as logs that can be replayed.

B. Creating Forged Logs

We discuss the feasibility and effort for plagiarizing depending on the system architectures as well as availability of honestly created logs that could be used for forging. To begin with, we elaborate on all three possibilities of cheating as shown in Figure 2. Other options such as manipulating the creation software or plagiarism detection software itself seem significantly harder. Therefore, they are not discussed.

i) Manual Copying: The easiest method of cheating is by manual interaction with the creation software. For instance, a student might create a thesis by retyping large parts of relevant text from a website with some rephrasing or just copy-and-pasting. This is easy to do and it always yields a semantically and syntactically correct log, but it might come with a significant workload for the cheater and/or a high risk of detection, if done carelessly – even if the digital outcome itself does not show any signs of plagiarism. The distribution of events in the log might deviate from logs of honestly created works and, thus, disguise the cheater. Therefore, a successful cheater must anticipate the distribution of events that is typical for honestly created digital outcomes. He must forge a log that is close to such a distribution. Essentially, a cheater must perform roughly the same amount of interaction with the creation software as a non-cheater and he must interact in a similar way as non-cheaters. For example, for a programming assignment he must perform a similar degree of program testing and "debugging" that would not be needed if he just wanted to change the structure of the code. For writing a thesis he must perform a similar degree of navigation, redo's, undo's and editing.
ii) Simulating HCI: A log can also be created indirectly by simulating human computer interaction(HCI) with the creation software. This requires special software that can emit mouse movement and key press events. The goal for the cheater is to create a log that appears non-forged in a (partially) automatic manner rather than performing all interaction manually. For example, a student might copy a large part of text from another source. He could paste it into the creation software resulting in a single "Paste" event in the log that might appear suspicious. He might also use a special tool that takes the copied text as input and simulates HCI by sending key press and mouse movement events to the creation software resulting in a more credible log. Automating HCI is possible and an essential part of GUI testing [22] and automation (as showcased by the macro functionality in office applications). Recording screen coordinates of mouse movements and clicks is sensitive to screen resolutions and software configuration which reduces portability. As of now, even by using an available recording and replay tool, a cheater would have to configure the tool, e.g. to replace a single paste command of text by single key presses of the letters of the text, or to specify how navigation and editing events

should be created by such a tool. A key challenge is that the cheater must be aware of the state of the creation software and the events that are feasible in that state as well as their effect. For instance, a ”cut text” operation is only available, if indeed text was selected priorly. Typically, non-availability would be indicated by a grey “cut” icon and menu item in the GUI of the text editor. Clicking on such an icon or menu item does not have any impact, in particular, the creation software does likely not generate an event that is appended to the log. Thus, a cheater attempting to simulate cut operations, must make sure that the command is available, otherwise he does not achieve his goal of appending to the log. In a programming IDE a click on ”Start/Run program” results in different user interfaces depending on whether the program is compilable and executable or not. If it is not executable, a dialog with an error message might be shown. If it is, the program might run in the foreground and a toolbar with several debug commands might appear, such as pausing the program to investigate its internal state. If a cheater wants to simulate debugging behavior, he must make sure that the program is compilable.
Besides, though simulating HCI always yields a semantically and syntactically correct log, replaying the log might yield a digital product that is different from the submitted digital outcome. Thus, detection software would identify the cheater. A human can determine the availability of a command easily by visual inspection of the GUI, but tools cannot easily do so and rely on keeping track of the internal state of the creation software. Though these challenges might be addressable, they make forging a log automatically rather complex.

iii) Simulating the creation software: A cheater might also create or alter the log of the creation software directly, if he has access to it. In the simplest case the log is a human readable text file that allows for straight forward manipulation. A cheater might add events or change events in the log that mimic sequences of events emitted by the creation software for honestly created logs. Analogous for simulating HCI, the cheater wishes to automatically manipulate the log file rather than to edit it by hand. As of now, to the best of our knowledge there are no tools that would allow automatic creation or modification of such logs. The challenges are analogous as for simulating HCI, i.e. the cheater must anticipate the internal state of the software, otherwise the log does not yield the final digital outcome when replayed. Even more, a manipulated log might not even be semantically or syntactically correct, resulting in an error message when being replayed. This immediately reveals a forged log.

There are different types of system *architectures* determining whether logs are collected online or offline and whether logs are created by software running on a device under the creator’s control or not. The architecture determines the ease with which logs can be changed. In the least secure setup logs are created offline at a student’s laptop. If logs are not encrypted and in text format the student can alter them before submission. Thus, in such a setup a student might use all three forms of cheating in Figure 2. In the most secure technical configuration, a student uses the creation software on a device, where he can neither access logs nor install any software that allows simulating human computer interaction.

The availability of logs or recordings of HCI facilitates two kinds of forging: i)‘Copy and Modify’: A cheater has access to a valid log, eg. at least one (honestly created) log that he manipulates before submitting it. To minimize effort, a cheater might take a log of a fellow student and only perform a small amount of changes. He might also use a recording of HCI of a fellow student and alter the recording and then replay it to generate a forged log. From the perspective of detection both methods result in plagiarized logs that are similar to the original log. ii) Generation from scratch: The cheater does not have access to a log but potentially to (parts of) the digital outcome, eg. source code from a fellow student or the Internet, that he tries to plagiarize by incorporating it in his own work.

C. Unique vs. Equal Tasks

Our detection relies on comparing multiple logs among each other. If the same exercises are solved by many students, we expect similar logs with some variation. Given that a task is unique such as writing a thesis, we must rely on comparing logs of multiple students each treating a different topic in their thesis. Detection capability relies on the assumption that writing a thesis conforms to a different process than plagiarizing a thesis that is reflected in the logs, eg. less editing. Detection is likely to be better if the thesis’ are similar in expected effort and stem from people with a similar level of education.

V. Automatic Detection based on Histograms

We compute frequency statistics for each log. These are compared among each other. The counts of events for each event type yield a histogram that is used for distance computation among logs. If two logs have very similar statistics or a log shows very dissimilar frequencies of usage of commands from all others then the log is more likely belonging to a cheater. The first case is likely a result of copying a log and modifying it. The second is a result of creating a log from scratch while copying (parts of) the final digital outcome. We employ different metrics to detect pairs of similar logs and outlying logs using some general design considerations. The detection should be as robust as possible to potential modifications of a cheater. In particular, for similarity, changing a single command type should not impact the metric too much, since it is relatively easy and fast to (manually) increase the counts of certain command types to an arbitrary length, eg. moving the cursor left and right. For outlying logs, low usage of commands that are very commonly found in honestly created logs are a strong indicator for cheating.

We first discuss general aspects related to data preparation followed by metrics for plagiarism detection.

A. Data preparation

Data preparation might involve data cleaning and data transformation. Data cleaning encompasses removing entire logs as well as cleaning the content of individual logs. It is quite common that assignments or thesis are only partially finished and handed-in either way. Such logs corresponding might be removed, since a cheater is not likely to copy an incomplete assignment resulting in a non-satisfactory grade. They are often characterized by being of short length. If left untouched, they could distort outlier detection, since they do not resemble a proper complete solution process of the assignment. For example, we provided a code skeleton. Several students looked at the skeleton and did minor changes but were far from any serious attempt to solve the task. Thus, the logs contained a significant proportion of navigation but relatively little editing or debugging. A cheater creating a log with similar properties might escape detection if these logs remain present.

The content of individual logs typically requires no cleaning, since it is generated automatically. Data transformation could involve renaming of events, eg. we shortened several very long command names. We also neglected all information such as the timestamp of an event or parameters. We focused only on the command type. We computed a histogram for each log capturing the frequency of each command type. As we shall see and discuss in the evaluation (see Figure 9) the distribution of the usages $N(L,c)$ of a specific command c across logs $L \in \mathbb{L}$ is skewed. We transformed the data using the Box-Cox transform to get a more symmetric distribution. We added the value '1' to all values before transformation to handle the case of zero counts, ie. the Box-Cox transform might otherwise compute $\log 0$, which is undefined. By $B(L,c)$ we denote the transformed value of $1 + N(L,c)$. Figure 10 gives some insights showing the transformed distribution for frequent command types. Less commonly used command types have a peak at 0.

B. 'Copy and Modify' Detection

The goal of 'Copy and Modify' detection is to determine if one log is an altered version of another. This is done by computing the similarity between two logs, ie. the Pearson product-moment correlation coefficient of two logs. Using all command types for computation of the correlation might not be robust against insertion or deletion of a few event types. In the most extreme case, a cheater might alter the frequency of a single command type to such an extent that an otherwise equivalent log is not regarded as similar. Therefore, we compute the correlation for randomly chosen subsets of commands. For a pair of logs to be very similar it suffices that the correlation for just one of the subsets is very similar. There is a risk that if we pick too small subsets, ie. too few event types or strongly dependent event types that by mere chance the distance between two logs is small. However, as discussed in the evaluation, this seems to be a minor concern. There are only relatively few commands that occur in most of the logs. Thus, it is important to choose command types for correlation computation separately for each pair of logs rather than just one subset for all logs. Otherwise, if a subset of rare commands is chosen that only occurs in a few logs then many logs will be classified as very similar, since many logs might not contain any of the rare commands at all. Thus, to compute the similarity of a pair of logs we only select commands that occur in at least one of the two logs. Furthermore, choosing the subset from one of the logs only, ignores the number of different command types occurring in the other log. This might lead to logs being judged as similar even though one of them contains a significant number of additional event types. Thus, we pick half of all command types from each log.

More formally, a subset S of commands is chosen as follows for a pair L, L' of logs: We pick half, ie. $|S|/2 = s_{sam.}/2$, of all command types $U \subset S$ uniformly at random from log L, ie. $T(L)$, and the other half $U' \subset S$ from $T(L') \setminus U$. The entire subset S of command types contained in the histogram is given by $S = U \cup U'$. The Pearson-product correlation $\rho(L, L', S)$ for a subset $S \subset T(L) \cup T(L')$ of command types is given by:

$$\overline{n(L)} := \frac{\sum_{c \in S} n(L,c)}{|S|}$$

$$\overline{n(L')} := \frac{\sum_{c \in S} n(L',c)}{|S|}$$

$$\rho(L, L', S) := \frac{\sum_{c \in S} (n(L,c) - \overline{n(L)}) \cdot (n(L',c) - \overline{n(L')})}{\sqrt{\sum_{c \in S} (n(L,c) - \overline{n(L)})^2} \cdot \sqrt{\sum_{c \in S} (n(L',c) - \overline{n(L')})^2}}$$

The choice of a subset and the similarity computation for it is illustrated in Figure 5. In the figure, we chose two commands of each log. The correlation was computed using just four values per log, ie. one for each of the chosen command types.

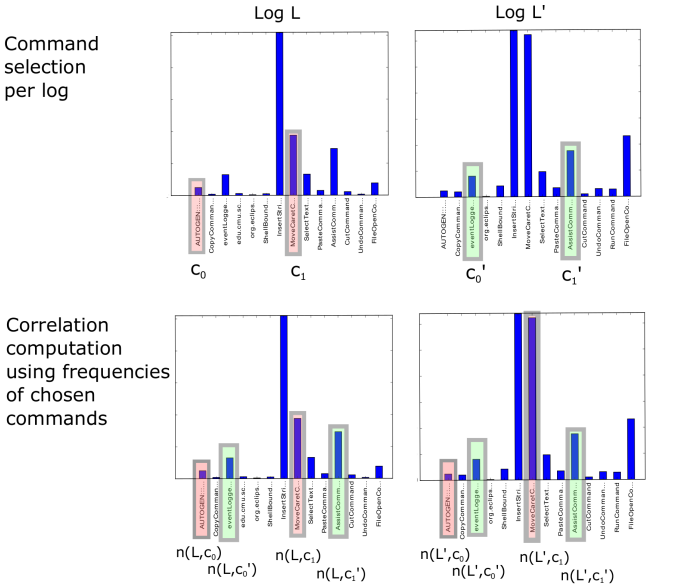

Fig. 5. Illustration of similarity computation of two logs for 'Copy and Modify' detection for one subset

Let $\mathbb{S}(L, L')$ be the set of all chosen subsets S for a pair of logs L, L'. The correlation $cor(L, L')$ of two logs is given by the maximal Pearson-product correlation $\rho(L, L', S)$ of any set $S \in \mathbb{S}(L, L')$:

$$cor(L, L') := \max_{S \in \mathbb{S}(L,L')} \rho(L, L', S) \tag{1}$$

Potential cheating candidates are indicated by larger correlation, eg. we might report a fixed number of pairs of logs having largest correlation among all pairs of logs from $\mathbb{L}$.

C. 'Generation from Scratch' Detection

A cheater might not use an existing log for plagiarism, but copy a final outcome (without the log). He might also enter a modified solution manually or copy-and-paste it and then edit it. In both cases, he creates a log for his plagiarized work. Clearly, given enough effort and time any final outcome such as thesis can be changed to another so that its structure appears very dissimilar. However, the process of changing such an outcome is typically different from solving the task in an honest manner. For example, plagiarizing might be characterized by activities such as a lot of word substitutions or reordering, changing the layout of a work, and, in software engineering, also permuting commands. We might expect less incremental changes, rework, navigation, and, in software development, also less debugging or navigating between files. Therefore, if a log contains event of some command types much more (or less) often than most other logs, this might be an indicator for plagiarism.

Our measure, ie. the outlier score, is a weighted sum of scores of individual commands. The weights are higher for commands occurring in many logs at least once. The score for the 'PasteCommand' is illustrated in Figure 6 together with the Box-Cox transformed frequency distribution of the number of usages per log. If the number of usages of the command is common, ie. within one standard deviation from the mean, then the outlier score is zero. It raises rapidly to almost one when moving another two standard deviations away from the mean. Next, we explain the outlier score in more detail and afterwards motivate each choice for its computation.

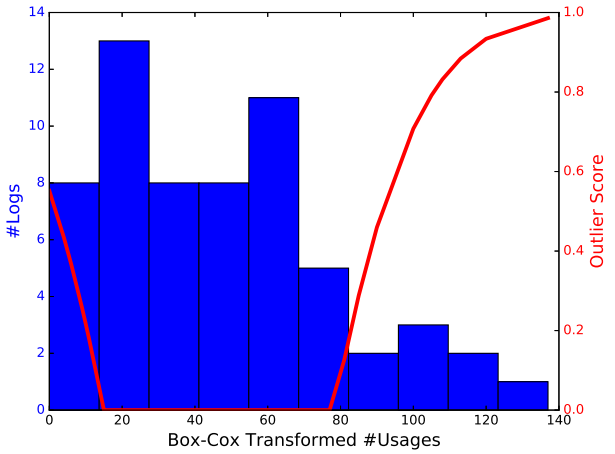

Fig. 6. Outlier score contribution of a single command

Roughly speaking, the count of a single command of some log $N(c, L)$ is outlying if it is much larger or smaller than most other logs. To get a quantitative estimate, we used the distribution of the Box-Cox transformed counts given by $B(c, L)$ and we computed the probability of obtaining a count $B(c, L)$ or an even less likely count. For that purpose, we assumed that the resulting distribution of the transform is normal. We computed the raw outlying probability $p_{raw}(c, L)$ of command type c of log L as follows: We assumed that the count random variable $B(c, L)$ is normally distributed with probability density function p using the mean $\overline{B(L)}$ and the standard deviation $\sigma(B(L))$. For a count value X we computed the probability that a count is at least (or at most) as large as X: Given that $B(c, L)$ is larger than the average, ie. $\overline{B(L)} := \sum_{L\in\mathbb{L}} B(c, L)/|\mathbb{L}|$ we computed the probability $p_{raw}(c, L) := p(X > B(c, L))$ for random variable X and otherwise $p_{raw}(c, L) := p(X \leq B(c, L))$.

$$\overline{B(L)} := \sum_{L\in\mathbb{L}} B(c, L)/|\mathbb{L}| \tag{2}$$

$$\sigma(B(L)) := \sqrt{\frac{\sum_{L\in\mathbb{L}}(B(c, L) - \overline{B(L)})^2}{|\mathbb{L}|}} \tag{3}$$

$$p_{raw}(c, L) := \begin{cases} p(X > B(c, L)) \text{ if } B(c, L) > \overline{B(L)} \\ p(X \leq B(c, L)) \text{ otherwise} \end{cases} \tag{4}$$

Figure 10 depicts the transformed distribution for frequent command types. Assuming a normal distribution is generally not correct for all command types though the majority of commonly used commands seem to follow roughly a normal distribution. For rarely used command types, which constitute a significant proportion of all commands (see Figure 7), there are many logs exhibiting a zero count and a few with counts larger than zero. Fitting a normal distribution does not assign enough probability mass to the zero value itself as well as values far from zero. It might be better to model the distribution using a random indicator variable X stating whether the count is zero ($X = 0$) or not ($X = 1$) together with a conditional distribution $p(Y|X = 1)$ that could be normal or more heavily tailed. However, we found that this complexity in modeling is not needed, since rarely used command types have only little influence on the overall outlier score.

When looking at the count of a specific command $B(c, L)$, some derivation from the expected value is supposed to occur, eg. for a standard normal distribution the standard deviation from the mean is one. Therefore, one might not judge a log as being more of an outlier than another log given that both are within some limit of the mean. More precisely, we said that all counts within a distance of one standard deviation do not increase the outlier score, ie. the outlier score for a count $X \in [\overline{B(L)} - \sigma(B(L)), \overline{B(L)} + \sigma(B(L))]$ is zero. The cumulative probability density of a value being smaller than the lower limit or larger than the upper limit is equal, ie. given by 0.16. We defined the unweighted outlier score $out(c, L)$ of command c of log L as follows:

$$out(c, L) := \begin{cases} 0 \text{ if } B(c, L) - \overline{B(L)} \in [-\sigma(B(L)), \sigma(B(L))] \\ 1 - p_{raw}(c, L)/0.16 \text{ otherwise} \end{cases} \tag{5}$$

The division by 0.16 serves as normalization so that all values between $[0, 1]$ are possible.

The distribution of command type usages (Figures 7 and 8) shows that it is not uncommon that a command type is

used only by a few students. Intuitively, a log being the only one using some command types, is likely to be an outlier and, therefore, potentially part of plagiarism. We found that also honestly created logs often have some unique command types. In contrast, for forging a digital produce (e.g. rephrasing words, changing layout etc.), standard editing operations seem to suffice, i.e. the corresponding logs do not necessarily contain special commands not contained in honestly created logs. But, some command types that are common in honestly created logs are not or less needed to disguise a plagiarized work and vice versa. Therefore, we added weights for each command stating how much a deviation from the mean indicates cheating. Generally, the more often a command occurs at least once in some log the larger its weight. Thus, a strong deviation of counts from the mean of a command that occurs in all logs seems a strong indicator for an outlier. We computed the weight for a command c as the squared fraction of logs containing the command. The squaring emphasizes the fact that commands that are used not so often should not impact the outlier metric heavily. The computation uses an indicator variable I being one if the command c occurs in log L, i.e. $n(c, L) > 0$:

$$I(n(c,L) > 0) := \begin{cases} 1 \text{ if } n(c,L) > 0 \\ 0 \text{ otherwise} \end{cases} \tag{6}$$

$$w(c) := \left(\frac{\sum_{L \in \mathbb{L}} I(c,L)}{|\mathbb{L}|}\right)^2 \tag{7}$$

The definition yields weights in $[0, 1]$.
The total outlier score for a log L is given by the normalized sum of the contributions of each command:

$$out(L) := \frac{\sum_c w(c) \cdot out(c,L)}{\sum_c w(c)} \tag{8}$$

The larger the score the more likely a log is an outlier and, thus, stemming from a dishonestly created work.

VI. Manual Detection

Automatic detection only gives cheating candidates with varying degree of confidence. Manual inspection of logs can improve clarity for doubtful cases. Our focus is mainly on automatic detection but we also briefly mention possibilities for manual inspection that we show-case in the evaluation. They can be classified into using more sophisticated analysis of the event logs, inquiring the creator and cross-validating potential plagiarism using other techniques.
The histogram-based techniques based in Section V is very general with little domain knowledge. It could be enhanced by performing manual analysis, e.g. by visually assessing histograms of several students and look for deviations or similarities. Some commands are typically correlated, eg. a larger number of "move cursor right" usually also implies many "move cursor left" commands, "Insert string" correlates with "Deletes" as well as relative counts of several commands, eg. the amount of navigation should grow with the amount of editing. Furthermore, one can check if necessary event types were missing, extremely low or high, or if there are event types that should not be in the log. In short, a more holistic look at the histogram might be beneficial. One might also compare histograms based on a subset of the entire log. This increases sensitivity to spot certain behaviors and it also allows to compare different phases of the creation process that are characterized by different activities. For example, assume a forged log is a result of appending to a valid log by editing of a digital produce, e.g. by performing extensive rephrasing of most of the text to mislead conventional plagiarism detection tools. When looking at the distribution of the last part of such a log, it might be characterized by a large amount of text replacement events that was performed in a short amount of time. A non-forged log is likely to contain relatively less replacements, but more navigation or inserts, since it seems more common that either a student appended to the thesis to just finish it or to read through it and fix typos or change individual sentences. Aside from looking at the histograms, one might use additional analysis using logs as mentioned in Section VIII, e.g. looking for common subsequences to identify copies, looking for pasted content from external applications, etc. Contacting the creator of the work to inquire him about the creation process is another option. For example, he should be able to answer questions such as where we spent most time on editing, in which order content was created etc. In case, keystroke timings are available, one might also conduct simple tests, like keystroke pattern analysis, e.g. check if the timing between keystrokes is similar during the supervised test is similar to timing seen in the submitted log [10]. A further option would have been to also consider existing techniques for verification, eg. to compute the similarity among the final source code using the MOSS tool [29] or for a text document one could check consistency of multiple works of the same author by extracting features from both texts [19]. Finally, one might also use techniques not related to creation logs to validate that a potential cheater is indeed cheating. We refer to the related work for such techniques.

VII. Implementation And Evaluation

A. Collected Data

We obtained logs using the Eclipse IDE with the Fluorite logger plug-in [36] from more than 60 students for three programming assignments. We conducted the same analysis for all three assignments. We discuss the first of them in detail, since all gave similar results. In Assignment 1 students were given a skeleton of multiple files. They were supposed to write roughly 100 lines of code.

Figure 7 shows the frequencies of commands of a certain type. There are a few commands that are used very frequently, related mainly to editing and navigation. Figure 8 shows how often students use a certain command. About 10% of all commands are used by all students and about 50% are used by at most three students. Command usage might also be partially unintentional, eg. by choosing a menu item by accident. The usage of a particular command type per student varies a lot. The box plot of the 15 most frequent event types is shown in Figure 9. For a specific command the student at the 25% percentile used it roughly a factor 2-15 less than the student at the 75% percentile. Some of the submitted logs are rather

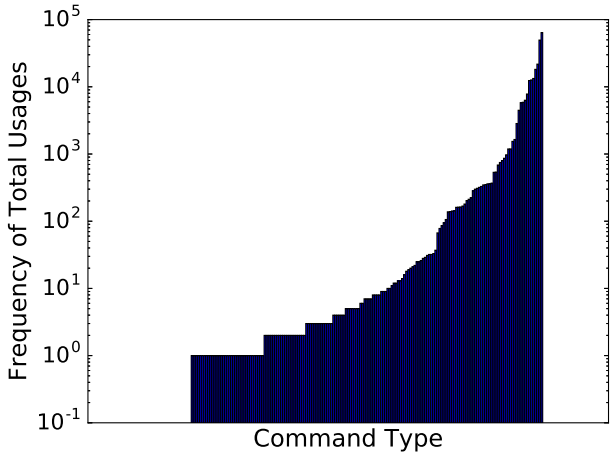

Fig. 7. Distribution of total usages (by all students) for command types

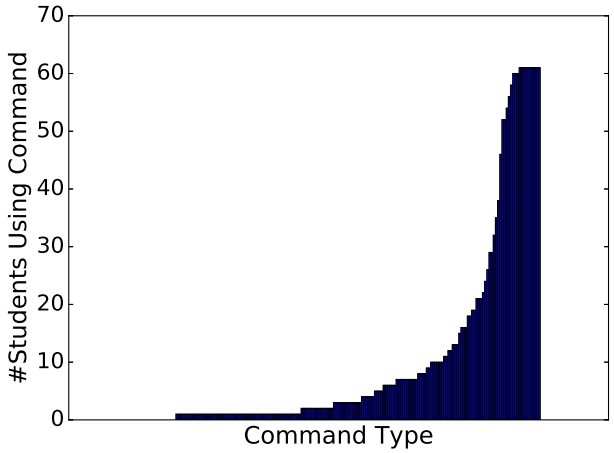

Fig. 8. Distribution of number of students using a command types

short and stem from incomplete assignments. Some submitted logs also contain work not part of the assignments. We did not filter these though they are likely to have an unpleasant effect on the outlier detection result.

B. 'Plagiarism' Data

Dataset 1: All collected logs from students. It is used to conduct a semi-automatic detection, i.e. identify potential cheaters automatically and check if they are really cheaters manually.

Dataset 2: An extension of Dataset 1 with synthetic data created through modification from original logs without using the IDE. The dataset should support addressing two questions. First, we want to assess robustness of our detection with respect to modification of logs, ie. what is the minimum required change so that a log is not detected? Second, we want to get an understanding of how much a normal log has to be modified to be considered an outlier. To this end, each single log was modified in two ways:

- *Event type* change: We vary the number of event types, ie. we remove a certain percentage of all events of some types (chosen uniformly at random) from the log. This corresponds to the case, where a cheater is not using all kinds of events, eg. due to copy-and-pasting of the final outcome.

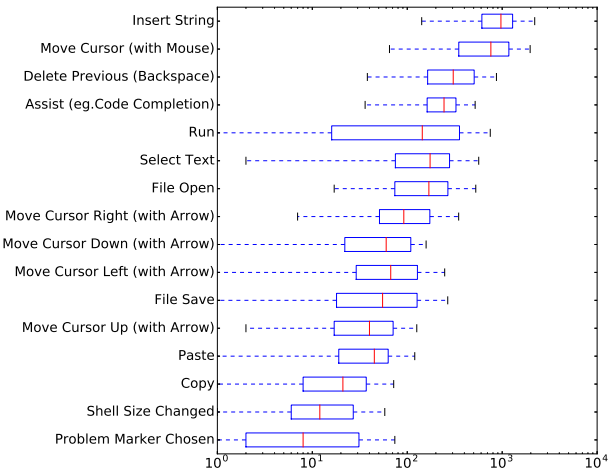

Fig. 9. Box plot of counts of 16 most frequent event types.

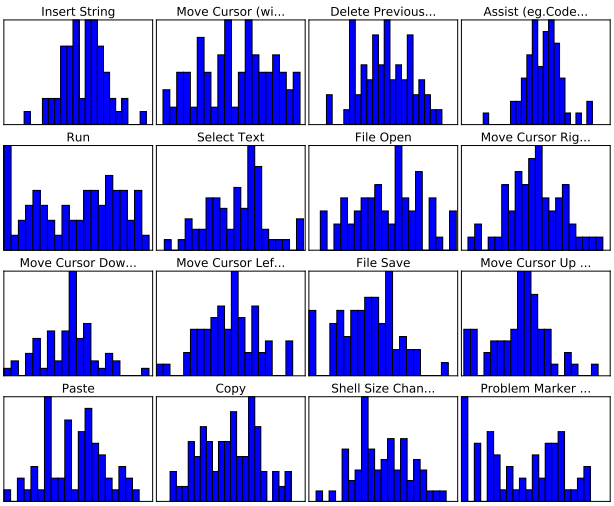

Fig. 10. Distribution of counts of 16 most frequent event types after Box-Cox transform.

- *Event frequency* distribution change: We alter the distribution of events, ie. we increase or decrease the count of events. Given a maximal change factor k we change the count of each command type of a log as follows. We choose a random number $r \in [1, k]$ and with probability one half we increase the count by a factor r with probability one half we decrease the count by a factor $1/r$. This is motivated by the fact that a cheater might use certain commands more often or less often than most of his peers, eg. more copy-and-paste, less undos/editing, less debugging etc. In particular, this covers the most appealing scenario for plagiarism: Taking an honestly create log and appending to it by simply editing the digital produce.

For each log and each change factor k we created 10 modified versions of logs.

Dataset 3: This data set extends Dataset 1 by creating a plagiarized solution of an assignment using the IDE. We considered several strategies to obtain a 'fake' log from scratch, ie. without using an existing log, with increasing effort for the cheater. Our strategies are similar to [12] which focuses on Wikipedia articles.

- Copy-n-paste: Copying the source code (eg. from the Internet) without modification or any additional activity, such as running the code.
- Small Refactoring: Copying the source code, renaming a few variables.
- Medium Refactoring: Copying the source code, renaming several variables, changing some comments and running the code. The amount of work performed is roughly a factor ten less than the average log size.
- Large Refactoring: Copying the source code, renaming most variables, changing a lot of comments, reordering statements, making minor code changes and running the code to test. The size of the log is slightly larger than the average log size.

For each strategy we created three logs manually and we considered 100 variations for each of the twelve manually created logs by changing the event frequency by using a maximal change factor 3 in a manner as described for the event frequency alterations of Dataset 2.

C. Setup and Parameters

We ran our histogram-based detection on a PC (2.7 GHz CPU, 8 GB RAM) on our collected data for all students for all three assignments. For computing similarity between two logs, we chose $n_{sam.} = 100$ subsets of commands of size $s_{sam.} = 2$. We also investigated subset sizes $s_{sam.}$ of 2,4,8 and 16.

Experiment using Dataset 1: Objectives were to set the ground truth for later experiments by identifying inadequate logs and their impact (e.g. incompletely submitted logs) as well as to get an understanding of actual data. In particular, Dataset 2 builds upon the assumption that there are no cheaters in Dataset 1. We identified logs that appear as outliers or very similar due to various reasons ranging from cheating to erroneous handling of logs by students, eg. not submitting all log data or an excessive amount (including work beyond the assignments).

First, we looked for cheaters in the original dataset. A natural choice is to assume that the pair(s) of students with minimal distance (respectively maximal distance) are cheaters for the correlation measure (1) and those with the largest outlier score (8). We manually inspected the ten most similar pairs of logs and the ten most outlying logs.

Experiment using Dataset 2: The goal was to see how sensitive the detection is with respect to deviation of an existing log or the set of all logs. For each log, we created a modified log and then we ran the detection algorithm including all original logs and the modified log. Thus, for each modified log we obtain a correlation score (1) and an outlier score (8). An instructor might have a small time budget to check for plagiarism that he uses to investigate a few high risk candidates for cheating. Therefore, we say that a cheater was successfully detected, if his or her log is among the five most similar logs (for a cheater that copies and modifies a log) or among the five most dissimilar logs (for a cheater that creates a log from scratch). We computed which percentage of faked logs (of all students) are detected. The detection capability was assessed for both kinds of synthetic data, ie. removing some event types completely and changing the distribution of events (more gradually).

Experiment using Dataset 3: The goal was to investigate whether the detection can detect logs that are created from scratch. As for Experiment 2, we added each of the created (cheating) logs to the entire set of logs and tested whether its outlier score or correlation score ranks among the five highest.

D. Results

The entire computation took less than one hour.

Parameters: The results of all assessments was best for $s_{sam} = 2$. This is no surprise, since in case a cheater does not alter the frequency of all command types, it is quite likely that the correlation is maximal, ie. 1 due to the choice of two commands having the exact same counts. If we pick more commands than two then it becomes more likely that command types have different counts, yielding smaller correlation.

Results Experiment 1: The distribution of the distances of original logs is shown in Figures 11 and 12. Figure 11 shows the correlation scores used to identify copies of an assignment, ie. $cor(L, L')$ (1). Outlier scores, ie. $out(L, L')$ (8), are shown in Figure 12. One might observe a large gap between distances of cheaters and non-cheaters. The larger a gap the more suspicious a work. Due to strong variations in behavior among individual students gaps are expected. For the original dataset, we judged the ten most similar pairs of logs to be created by different students. We also found the ten most dissimilar logs to be logs that seem to stem from students solving the assignment in an honest manner. Thus, we concluded that there are very likely no cheaters. Some logs did not provide a proper solution to the assignment, ie. students gave up and submitted incomplete assignments. We did not further check such logs for plagiarism. We employed several checks as stated in Section VI. We looked at the distribution of commands of each log with respect to others, eg. we created a figure for each log L as Figure 9 but containing more event types and highlighted the counts for log L. We did this also for partial logs, but did not find any suspicious patterns.

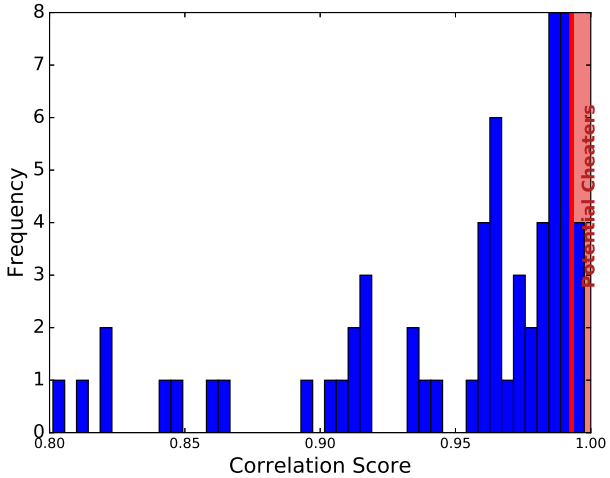

Fig. 11. Histogram of pair-wise histogram distances $cor(L, L')$. The red line shows the threshold for being a potential cheater.

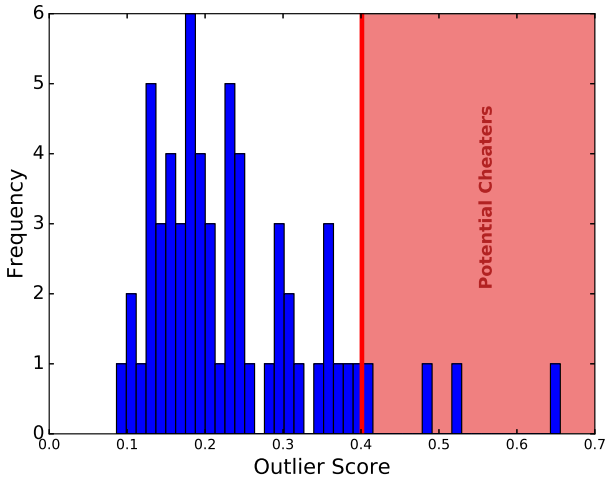

Fig. 12. Histogram of pair-wise histogram distances $out(L, L')$. The red line shows the threshold for being a potential cheater.

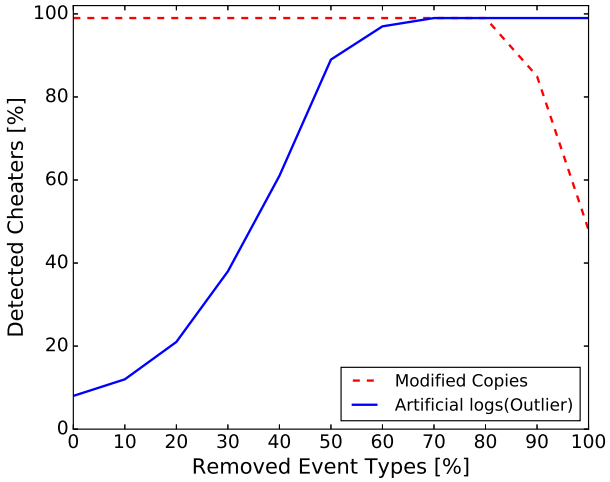

Fig. 13. Plot of fraction of detected faked logs for event type change.

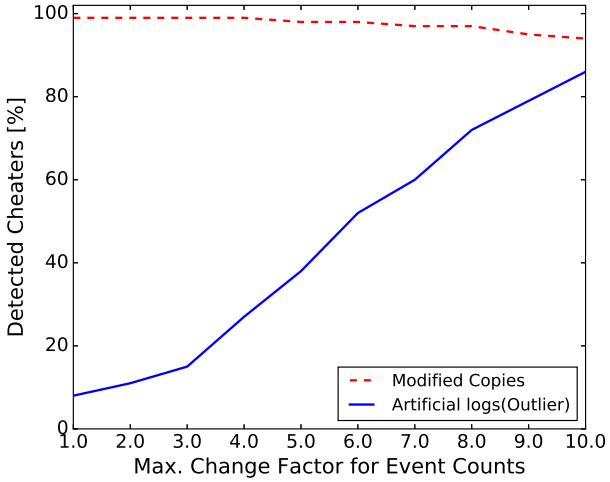

Fig. 14. Plot of fraction of detected faked logs for event frequency distribution change).

Results Experiment 2: When adding synthetically created logs, we found the following. In summary, for both data sets we witnessed an expected trade-off: The more a log of a student is distorted the less likely it is detected as being a copy but the more likely it is being detected as outlier.

Detection of modified copies: When removing 90% of all event types (see Figure 13) or change event counts by a factor of up to 10 (Figure 14), we still detect about 90% of copied and modified logs as plagiarism. The high detection rate stems from the fact that it suffices to find a pair of command types that were not modified. This chance is relatively high even if many command types are altered: Say, we removed 90% of command types for a log and left the count of the other 10% unchanged. For computation of similarity we pick one command type from the original and modified log with count larger zero. The one from the modified log occurs in the original, but the one from the original log only occurs in modified log with 10% probability. This gives a 10% probability for one subset that the counts are identical, i.e. perfect correlation. If we modify counts by a random factor $[0, k]$ then there is always a certain probability that a few command types are altered by a rather small factor which results in a high detection rate.

Detection of artificially created copies: Outlier detection requires relatively large changes of frequencies (see Figure 14). When modifying a log created by an honest student, we must change it beyond the variance across all logs first before it becomes an outlier. Given the fact that logs vary strongly, partially due to incomplete or incorrect logs (see Section VII-A), the change required is also significant, ie. when changing command execution counts by a factor of 5, we detect about 40% of logs as outliers. For removing event types similar reasoning applies, but detection seems to work better because removing a few frequent event types that occur in most logs has a strong impact on the outlier score. We classify about 90% of logs as outliers, if they lack 50% of events, while being otherwise identical to one of the logs.

Results Experiment 3: Detection worked very well in all cases (Figure 15). This is not surprising, since the logs stemming from plagiarized work might lack some frequently used commands entirely or to a large extend, eg. commands for opening files, edits (delete, undo, selecting of text), navigation (switching between files), saving,running the code.

Though our results seem very encouraging, we do not want to hide limitations as discussed in the next section.

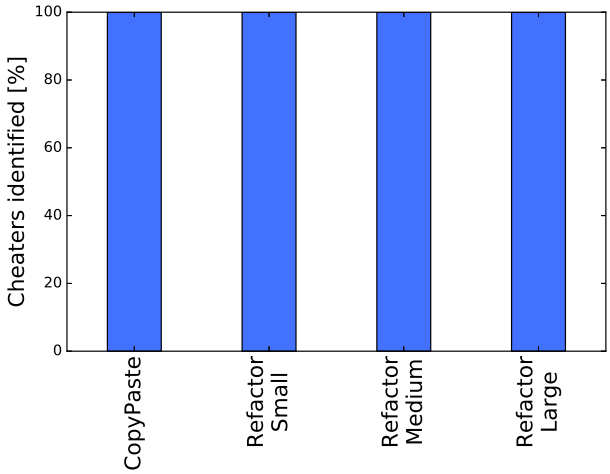

Fig. 15. Detection of cheaters for variations of logs created using different approaches.

VIII. Discussion and Future Work

We mention advantages and disadvantages of using the creation process as well as future work that targets on improving automatic detection and facilitating manual inspection of logs.

A. Strengths and Weaknesses

We identified several reasons, why plagiarism detection based on logs is attractive:

- Amount of information: Logs typically contain much more information than the final product, since they usually implicitly include the final work as well as all actions and changes throughout the making of the product such as saving or printing. Since they rely on more information, logs enable potentially more accurate methods of detection. For that reason forging a log requires more effort than forging the final product. Our approach forces cheaters not only to modify (copied) content as is required for traditional methods, but it also demands them to understand and mimic the creation process.
- Complexity of log forging: Creating a forged log requires in depth understanding of the log creation software. Log entries often depend on internal processes that are hard to predict, e.g. the exact time of automatic software updates or the number of lines that result from inputting a text in a text editor.
- Novel detection techniques: A whole set of new techniques from data mining based on outlier detection or nearest neighbor approaches can support the plagiarism detection based on logs. Our technique is data-driven with direct comparisons among a set of logs rather than relying on expert input or dedicated rules. This makes it very flexible, ie. easily usable for logs stemming from different programs. We might also detect plagiarized work that other techniques focusing on the final product fail to identify.
- Structure of logs: In contrast to the digital outcome that might appear as 'unstructured' data such as text from a data science perspective, logs appear as structured data. This allows for easier analysis.

Other potential advantages are that logs can also be used for other purposes that might increase learning. They could be used to give students feedback in an automated manner that allows them reflect on their behavior and capabilities as well as to recommend functions used by fellow students. The fact that logs are created might also encourage students to work more focused.

Log based detection also comes with certain weaknesses compared to conventional plagiarism detection techniques. If logs are very similar, the odds of a false positive seem rather small. There is a higher risk of false positives for students that behave differently from the majority of students. For example, if a student is very smart and requires much less rework than other students, he risks being classified as potential cheater. The same holds for students that perform an extra-amount of editing to improve the quality of their work significantly beyond average. Therefore, our detection technique often requires manual interaction, eg. examining of logs that seem to be forged. Though this might also hold for conventional techniques, the concept of logs might seem harder to grasp and assess for an instructor – at least in an initial phase.

We require that students collect and submit logs. In our setup, we also require students to install a plugin for recording events. Installation of the plugin took less than 5 minutes and submission per log less than 2 minutes. As of now, students had to disable the logging, if they worked on private projects. Both of these limitations could be reduced (or eliminated). For instance, one might install a private (without the logging plugin) and university version (with the logging plugin) of the software.

Furthermore, students have to use a particular software that supports logging for creating their work. In practice, it is not uncommon that students are already forced to use certain applications, eg. templates for thesis might only be available for a particular text editor or a course might demand to submit project files from a particular editor that was also used in tutorials. In principle one might support multiple applications but this increases effort for the lecturer or use a HCI recorder. Though many applications support logging, the detail of such logs differs significantly. This might limit the possibility of replaying a log to create the digital outcome. For example, whereas the macro recorder in Microsoft Excel logs mouse clicks in a spreadsheet, the recorder in Microsoft Word only logs clicks on Buttons and keystrokes. In the first case, the (final) spreadsheet can be created by replaying the log, in the second case of Word one might check the log for semantical correctness by replaying it, but it might not be possible to create the final document by replaying the log. Still, also in the second case the presented plagiarism detection techniques can be employed using the in-built recorder to create logs. However, a cheater might forge logs more easily, if no additional means are taken to check whether the submitted log corresponds to the submitted digital outcome.

B. Possible Improvements

Our histogram-based detection could be improved. There are many more kinds of techniques such as identifying a specific cheating behavior based on rules. We briefly discuss copy-and-paste and paraphrasing. Typically, copying occurs outside the creation software, eg. in a web-browser, and the pasting of text or images occurs within the creation software. A large amount of pasting (both in frequency and quantity) might be an indicator of cheating. Furthermore, inserting a lot of text in a short amount of time with relatively little editing, eg. delete events, might be an indicator of paraphrasing. Fingerprinting of keystrokes [10] is another way to identify forged logs. More generally, one could improve the detection mechanism to incorporate timing behavior of events. One might also look at (short) sequences of events. For example, one might pick a sequence that seems unique in a log (any sufficiently long sequence is unique) and search for this sequence in other logs.

Though all techniques for plagiarism detection apply to any

kind of digital outcome, we have evaluated our technique on programming assignments conducted in a complex IDE. The process and tools used for programming are arguably more complex than those used to create text documents for assignments that involve merely expressing ideas and summarizing work without implementation. For example, activities such as debugging do not occur in ordinary text processing. Essentially, this results in a reduced set of commands that are used when creating the digital outcome. Thus, forging a log created in a simple tool with a less diverse command set might seem easier. This might be (partially) compensated by using more advanced detection techniques. In particular, histograms using short sequences of events rather than single commands are plausible option.
We focused on assignments that are characterized by many students solving the same task. For thesis, in contrast, students work on different topics. Thus, logs of assignments might appear more homogeneous than logs of thesis. Homogeneous logs imply that a cheater has less options to forge a log that is not detected, since homogeneous logs are characterized by little variance in usage statistics. However, thesis also adhere to a common structure and also have specifics of the creation process. Furthermore, they are significantly longer, resulting in longer log files. Here, an approach that examines sequences of events related to a section of a thesis, such as literature review or introduction, rather than a single large log might improve detection accuracy.

Whereas simple cheating attempts involving copy-and-paste and some textual changes are easily detected, currently, we do not protect well against the case where a student manually enters text and simulates the creation process more thoroughly. For instance, a cheater might copy-and-paste text and then conduct artificial editing of the text. If the amount of editing and way of editing is similar to other logs, a cheater might well escape detection. However, without automatic generation of logs the manual work a student must invest significantly increases compared to work required to just altering the final outcome. The amount of work (measured in interactions with the tool) seems to be in the order of magnitude of peers that behave honestly.

To further improve detection and to support the inspection of logs, we envision a tool that visualizes the creation process in an intuitive manner. This should allow a skilled person, such as a supervisor of a thesis or a teaching assistant, to easily identify whether a semantically correct log, is likely forged or not, ie. generated or modified by an automatic tool (or manually). For instance, if a tutor detects that a rather difficult part of the work was done fairly quickly but other simple parts required a lot of time, this could make him suspicious. Such a tool might in particular help to avoid false-positives, ie. to gather evidence that people accused of plagiarism are indeed guilty.

IX. Conclusions

Cheating has always been an issue in- and outside education. We have contributed to remedy this problem by proposing automatic identification of 'likely' cheaters. Our novel approach requires collecting logs rather than just the final digital result, which is readily supported by several programs (with appropriate plug-ins). Ideally, our detection technique is combined with other techniques relying on analyzing the final "digital" outcome. Currently, this would make cheating very time consuming and rather unattractive. However, as soon as circumventing log based detection is supported by automatic tools, this might change. Thus, as in other domains, catching offenders will remain a "cat-and-mouse" game. But this is a game that must be played to counteract incentives for cheating. It is necessary to ensure that cheaters are not among the graduates of universities obtaining powerful positions where misconduct can harm large parts of society. We see this work as one step in this direction.

We would like to thank YoungSeek Yoon (author of Fluorite[36]) for valuable contributions.

References

[1] A. Ahadi, R. Lister, H. Haapala, and A. Vihavainen. Exploring machine learning methods to automatically identify students in need of assistance. In *Proc. of Conf. on Int. Computing Education Research*, pages 121–130, 2015.

[2] M. Alsallal, R. Iqbal, S. Amin, and A. James. Intrinsic plagiarism detection using latent semantic indexing and stylometry. In *Int. Conf. on Developments in eSystems Engineering (DeSE)*, pages 145–150, 2013.

[3] S. M. Alzahrani, N. Salim, and A. Abraham. Understanding plagiarism linguistic patterns, textual features, and detection methods. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(2):133–149, 2012.

[4] V. Anjali, T. Swapna, and B. Jayaraman. Plagiarism detection for java programs without source codes. *Procedia Computer Science*, 46:749–758, 2015.

[5] A. Bin-Habtoor and M. Zaher. A survey on plagiarism detection systems. *Int. Journal of Computer Theory and Engineering*, 4(2):185–188, 2012.

[6] P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper, and D. Koller. Programming pluralism: Using learning analytics to detect patterns in the learning of computer progr. *Journal of the Learning Sciences*, 23(4):561–599, 2014.

[7] T. Bliss. Statistical methods to detect cheating on tests: A review of the literature. *National Conference of Bar Examiner (NCBE)*, 2012.

[8] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *USENIX Security Symposium*, pages 255–270, 2015.

[9] D.-K. Chae, J. Ha, S.-W. Kim, B. Kang, and E. G. Im. Software plagiarism detection: a graph-based approach. In *Proc. of Conf. on Information & knowledge management (CIKM)*, pages 1577–1580, 2013.

[10] T.-Y. Chang, C.-J. Tsai, Y.-J. Yang, and P.-C. Cheng. User authentication using rhythm click characteristics for non-keyboard devices. In *Proc. of Conf. on Asia Agriculture and Animal (IPCBEE)*, volume 13, pages 167–171, 2011.

[11] D. Chuda, P. Navrat, B. Kovacova, and P. Humay. The issue of (software) plagiarism: A student view. *IEEE Transactions on Education*, 55(1):22–28, 2012.

[12] P. Clough and M. Stevenson. Developing a corpus of plagiarised short answers. *Language Resources and Evaluation*, 45(1):5–24, 2011.

[13] G. Cosma and M. Joy. An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE Transactions on Computers*, 61(3):379–394, 2012.

[14] K. Damevski, D. Shepherd, J. Schneider, and L. Pollock. Mining sequences of developer interactions in visual studio for usage smells. *IEEE Transactions on Software Engineering*, 99:1–14, 2016.

[15] B. Gipp and N. Meuschke. Citation pattern matching algorithms for citation-based plagiarism detection. In *Proc. of symposium on Document engineering*, pages 249–258, 2011.

[16] M. Jiffriya, M. A. Jahan, H. Gamaarachchi, and R. G. Ragel. Accelerating text-based plagiarism detection using gpus. In *Int. Conf. on Industrial and Information Systems (ICIIS)*, pages 395–400, 2015.

[17] M. Joy, G. Cosma, J. Y.-K. Yau, and J. Sinclair. Source code plagiarisma student perspective. *IEEE Transactions on Education*, 54(1):125–132, 2011.

[18] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. of symposium on Foundations of software engineering*, pages 1–11, 2006.

[19] J. Li, R. Zheng, and H. Chen. From fingerprint to writeprint. *Communications of the ACM*, 49(4):76–82, 2006.

[20] R. Lukashenko, V. Graudina, and J. Grundspenkis. Computer-based plagiarism detection methods and tools: an overview. In *Proc. of the int. conference on Computer systems and technologies*, pages 40–, 2007.

[21] H. A. Maurer, F. Kappe, and B. Zaka. Plagiarism-a survey. *Journal of Universal Computer Science*, 12(8):1050–1084, 2006.

[22] A. M. Memon. Gui testing: Pitfalls and process. *IEEE Computer*, 35(8):87–88, 2002.

[23] N. Meuschke and B. Gipp. State-of-the-art in detecting academic plagiarism. *Int. Journal for Educational Integrity*, 9(1), 2013.

[24] N. Meuschke, B. Gipp, C. Breitinger, and U. Berkeley. Citeplag: A citation-based plagiarism detection system prototype. In *Proc. of Int. Plagiarism Conference*, 2012.

[25] M. Novak. Review of source-code plagiarism detection in academia. In *Conv. on Inf. and Com. Tech., Electronics and Microel.*, pages 796–801, 2016.

[26] M. Potthast, B. Stein, A. Barrón-Cedeño, and P. Rosso. An evaluation framework for plagiarism detection. In *Proceedings of the 23rd international conference on computational linguistics: Posters*, pages 997–1005. Association for Computational Linguistics, 2010.

[27] J. A. Reither. Writing and knowing: Toward redefining the writing process. *College English*, 47(6):620–628, 1985.

[28] F. Rosales, A. García, S. Rodríguez, J. L. Pedraza, R. Méndez, and M. M. Nieto. Detection of plagiarism in programming assignments. *IEEE Transactions on Education*, 51(2):174–183, 2008.

[29] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[30] M. Schonlau, W. DuMouchel, W.-H. Ju, A. F. Karr, M. Theus, and Y. Vardi. Computer intrusion: Detecting masquerades. *Journal of Statistical science*, pages 58–74, 2001.

[31] J. Shah, A. Shah, and R. Pietrobon. Scientific writing of novice researchers: what difficulties and encouragements do they encounter? *Academic Medicine*, 84(4):511–516, 2009.

[32] C. Simmons. Codeskimmer: a novel visualization tool for capturing, replaying, and understanding fine-grained change in software. *http://hdl.handle.net/2142/44125*, 2013.

[33] W. Snipes, A. R. Nair, and E. Murphy-Hill. Experiences gamifying developer adoption of practices and tools. In *Companion Proc. of the 36th International Conference on Software Engineering*, pages 105–114, 2014.

[34] A. Vihavainen, J. Helminen, and P. Ihantola. How novices tackle their first lines of code in an ide: analysis of programming session traces. In *Proc. of the Int. Conf. on Computing Education Research*, pages 109–116, 2014.

[35] M. J. Wise. Yap3: improved detection of similarities in computer program and other texts. In *ACM SIGCSE Bulletin*, volume 28, pages 130–134, 1996.

[36] Y. Yoon and B. A. Myers. Capturing and analyzing low-level events from the code editor. In *Proc. of the 3rd SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pages 25–30, 2011.

[37] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu. A first step towards algorithm plagiarism detection. In *Proc. of the Int. Symposium on Software Testing and Analysis*, pages 111–121, 2012.

[38] M. Zurini. Stylometry metrics selection for creating a model for evaluating the writing style of authors according to their cultural orientation. *Informatica Economica*, 19(3):107, 2015.

Johannes Schneider is an assistant professor in data science at the University of Liechtenstein. His main research interests are data mining applications and methods.

Kostadin Damevski is an assistant professor at the Deparment of Computer Science at Virginia Commonwealth University. His research focuses on software maintenance and empirical software engineering, applied to a variety of domains.

David C. Shepherd is an assistant professor at the Department of Computer Science at Virginia Commonwealth University. His research focuses on software maintenance and empirical software engineering, applied to a variety of domains.

Avi Bernstein is a professor and heads the Dynamic and Distributed Information Systems Group in the Department of Informatics at the University of Zurich, Switzerland. His research interests include supporting organizational processes with focus on data mining, crowdsourcing and the semantic Web.

Jan vom Brocke is Professor for Information Systems and Hilti Chair of Business Process Management. His research focuses on IT-enabled business innovation and IT-driven business transformation.