Journal of Software Engineering and Applications, **2012, 5, 249-254**
http://dx.doi.org/10.4236/jsea.2012.54029 Published Online April 2012 (http://www.SciRP.org/journal/jsea)

Measuring Whitespace Pattern Sequences as an Indication of Plagiarism

Nikolaus Baer, Robert Zeidman

Zeidman Consulting, Cupertino, USA.
Email: Nik@ZeidmanConsulting.com

Received February 7th, 2012; revised March 10th, 2012; accepted April 8th, 2012

ABSTRACT

There are several methods and technologies for comparing the statements, comments, strings, identifiers, and other visible elements of source code in order to efficiently identify similarity. In a prior paper we found that comparing the whitespace patterns was not precise enough to identify copying by itself. However, several possible methods for improving the precision of a whitespace pattern comparison were presented, the most promising of which was an examination of the sequences of lines with matching whitespace patterns. This paper demonstrates a method of evaluating the sequences of matching whitespace patterns and a detailed study of the method's reliability.

Keywords: Plagiarism; Source Code; Source Code Similarity; Whitespace; Obfuscation; Indentation; Maintainability; Copyright Infringement; Intellectual Property; Litigation; Open Source

1. Introduction

Source code development and traditional source code analysis focus on the visual elements of the code, and the whitespace (space, tab, newline, and other blank characters) is simply left as a matter of individual style and preference [1]. Yet, when attempting to discover copied source code or analyzing identified sections of code for specific evidence of copying, the style decisions gain importance [2-4].

Most state-of-the-art source code comparison methods disregard whitespace patterns. Several methods reduce the source code to individual tokens, and others examine the similarity of how the tokens are parsed, such as by a source code compiler's parser, thereby eliminating the whitespace from further consideration [5-9]. A method proposed by H. T. Jankowitz analyzes the sequence execution of procedures in the code, which does not take whitespace into account [10]. A method proposed by E. Merlo compares metrics of a programs structural elements, without consideration for the whitespace [11]. Previous methods described by R. Zeidman utilize the source code elements, such as statements, comments, strings, and identifies, but not the whitespace itself [12, 13].

Methods that do consider whitespace, often analyze the total amount of whitespace used in a piece of source code as one of many metrics, which are then compared to detect similarities [14-16]. Academic plagiarism detections systems have been developed that introduce and evaluate whitespace signatures to identify student work [17]. S. Aliefendic described an approach for identifying similar sections of source code based upon unusual whitespace patterns common to both pieces of software [18]. However, little work has been done to examine the usage of naturally occurring normal whitespace patterns to detect similarities.

Whether trying to determine if correlated code is copied in an academic or industrial setting, there are two major obstacles: locating the areas of correlation and determining if the correlated areas of source code are the result of copying or not. There can be many reasons for code from different programs to be correlated, such as the common use of third-party code or concepts. However, the stylistic elements of a body of code are rarely if ever dictated by external requirements, so their similarity may reduce the possible reasons for the overall similarity to the point where copying is the only logical explanation [12]. Since the stylistic elements represented as whitespace patterns are less dictated by the external requirements than the visual elements of source code, it has long been thought that correlated whitespace patterns could be used to provide further evidence of copying [19]. In fact, surveys have found that academics often examine the whitespace patterns when evaluating possible student plagiarism [2,3]. Furthermore, a comparison of whitespace patterns may provide a new method of identifying correlated source code in which other elements have been modified to disguise coping, such as through the use of global replacements of variable names [20].

Copyright © 2012 SciRes.

We set out to discover whether whitespace patterns can be used to differentiate between similar and dissimilar source code. Unfortunately, we found that the whitespace patterns of individual lines are not unique and are often repeated throughout a body of code, so comparing the whitespace patterns of individual lines of code has proven to be too imprecise to adequately measure the level of copying or identify specific sections of copying [21].

Although the whitespace patterns of individual lines are not unique enough, groupings of such lines with matching whitespace into large scale patterns may prove unique enough to identify sections of copying. By developing a method of examining the sequences of lines of matching whitespace patterns, we determine whether comparing these large scale whitespace patterns is a method of reliably measuring source code similarity and identifying source code copying.

In Section 2 we describe our hypothesis regarding how to test whether comparing large scale whitespace patterns is a reliable method of evaluating source code similarity. In Section 3 we discuss the methodology that we used to perform these tests. In Section 4 we present the resulting data. Our conclusions regarding the reliability of comparing large scale whitespace patterns as a method of evaluating source code file similarity are presented in Section 5.

2. Hypothesis

If comparing large scale whitespace patterns, seen as sequences of lines of matching whitespace, is a reliable method of evaluating source code file similarity and thus a good method of detecting copied code, then a file copied from another file will have long corresponding sequences of lines with matching whitespace patterns while two independently created files will have very short, if any, corresponding sequences. Measuring the longest common sequence length should reliably differentiate files pairs and bodies of code with similarity from file pairs and bodies of code without similarity.

To test this hypothesis we compared and calculated the sequence score between file pairs from different bodies of code. The sequence score is the percentage of the longest sequence of lines divided by the smaller file's length. For example, if the longest common sequence of instructions is 80 lines when comparing a 100 line file to a 200 line file then the sequence score for the file pair is 80% [13].

We modeled identical source code by comparing a body of code against itself, which should produce the obvious 100% similarity between each file and itself. We modeled a project that has some copied code amongst mostly new original code by comparing two distant versions of a single software project, which have limited similarities [22].

Based upon the assumption that a file compared to itself should produce a 100% similarity score, and comparing a file to a completely different file should produce very few, if any, similarities and a very low similarity score, a reliable method of comparison should demonstrate the following behavior:

- When comparing the source code files from one body of code against itself, the resulting scores for all of the file pairings should produce a bimodal distribution with the majority of scores being distributed close to a very low local maximum (a large peak close to zero), and the pairings of identical files producing the second local maximum at the top of the range (a second peak at the maximum possible score, 100);
- When comparing the source code files from one project against another completely separate project the results should always produce a distribution with a very low mean and a small standard deviation;
- When comparing the source code files from one version of a software project against another version of the same project the results should show a distribution with a higher mean and larger standard deviation than the comparison of completely different projects. The results should still be skewed toward the low end because most file pairs will have little to no similarity.

3. Methodology

Building upon the whitespace comparison tools from our last paper, the steps to measure the sequences of whitespace are:

- Convert each file in the source code (*.c, *.cc, *.cpp, *.h, *.hh, *.hpp) to a whitespace file format;
- Compare whitespace formatted files;
- Analyze and properly filter the results.

3.1. Whitespace Conversion

To convert each file into a whitespace format we integrated our previous tool into the new Whitespace Extractor custom application, which performs the conversion according to the following rules:

- Every continuous sequence of printable characters is converted to the character "x";
- Every space is converted to the character "S";
- Every tab is converted to the character "T";
- Newline characters are not converted.

The output file contains only the characters "T", "S", "x" and the original newline characters, which are considered the line separators. For example, the following line of C code:

int var = prevValue + 5;

can be thought of as:

Copyright © 2012 SciRes.

(*T*)*int*(*S*)*var*(*T*) = (*S*) (*S*) *prevValue* (*S*) + (*S*)5;

where (*S*) and (*T*) represent space and tab characters, so Whitespace Extractor will translate the line into:

TxSxTxSSxSxSx

3.2. Sequence Comparison

We used a source code comparison tool to compare the sequences in each file pairing. The program compares the files in pairs—one file from the first directory and one file from the second directory. When more than ten lines of matching whitespace are found in a file pair, the file pair is reported. The comparison scores of all file pairs are compiled into a single database.

The comparison tool only determines that a series of lines is a sequence if there are more than 10 matching lines in a row. Therefore files with fewer than 10 non-blank lines are not counted as having any similarity. However, when proving plagiarism did not occur, small files should not be dismissed based only on their size.

3.3. Analysis Tools

In addition to the Whitespace Extractor, two other custom programs were required to manipulate the comparison tool's result databases:

- DB Skimmer: parses the database and removes all entries except the n^{th} highest scoring file pairs for each file in the first directory.
- Filter DB: removes files from a database, based on their extension, size, or the number of lines that they contain.

The comparison tool produces summary spreadsheets from the distribution of scores in a database. It also calculates the mean and standard deviation for the scores in a database.

3.4. Analysis

The comparisons were performed on open source code that was written in the C programming language, namely two versions of the Linux kernel, from www.kernel.org, and the Apache HTTP server, from http://d.apache.org/download.cgi. We ran the Whitespace Extractor on the source code and converted every single file to whitespace format. Then we ran the comparison tool in multi-processor mode with the following parameters:

- 512 file threshold;
- Only analyze sequences;
- Record sequences.

3.5. Filtering

As long as an operation is applied equally to every file or directory, it does not bias the results and qualifies as a valid step of the method.

We found that header files are often similar in both original content as well as the sequence of whitespace. Therefore, one of the steps of our methodology involved filtering out the scores of the header files.

We also found that #include statements have very similar whitespace patterns and are typically listed together at the start of the file, so the sequences of matching whitespace patterns that are caused by the #include statements are false positives, which can be eliminated. The regular expression ".*#include.*$" was used to remove the standard #include statements from every file.

Finally, when attempting to identify copied code one would typically look at the highest scoring file pairs, so we also filtered out all but the top score for each file.

What remains after these filtering steps are file pairs that have the highest probability of containing substantially similar code.

4. Results

We performed three comparisons of code from open-source projects. The first was a comparison of a program against itself, then a comparison of two completely different programs, and finally a comparison between two different versions of the same program. The following results for these three comparisons include charts of the score distributions, and the mean and standard deviation values for each filtration step.

4.1. Comparison of a Program to Itself

The open-source Linux Kernel version 1.0 was selected for the comparison of a program to itself.

We expected to observe 100% similarity when comparing each file to itself. The comparison of the 488 source code files of version 1.0 of the Linux Kernel against themselves did demonstrate that identical files had similarity scores of 100%, while other file combinations had lower scores.

4.2. Comparison of Two Different Programs

Comparing two different programs should result in low similarity scores. Using the comparison tool, we compared the Apache HTTP version 2.0.35, which contained 653 files to the Linux Kernel version 1.0, which contained 488 files.

We expected to observe only low scores, as the code should be completely unrelated. The comparison of the two different programs, the Apache HTTP server project against version 1.0 of Linux, is shown in **Figure 1** with the y-axis scaled linearly and on a logarithmic scale in **Figure 2**. The y-axis is the number of file pairs, the x-axis is the sequence similarity scores, and the z-axis is the different filters that were applied to the data.

Copyright © 2012 SciRes.

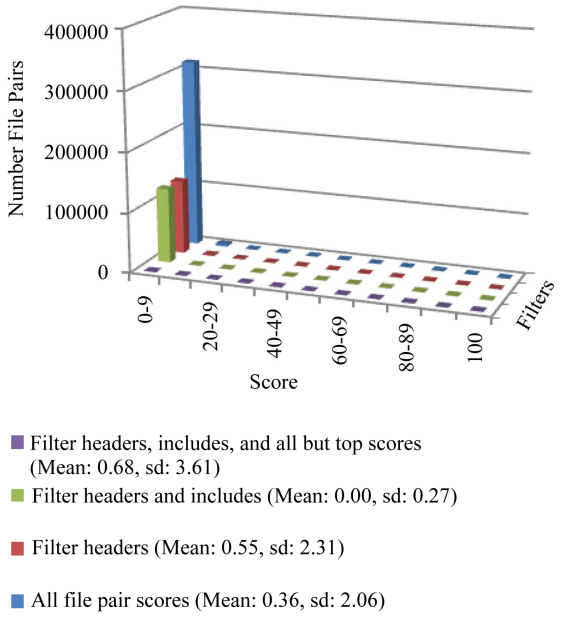

Figure 1. Comparison of two different programs on a linear scale.

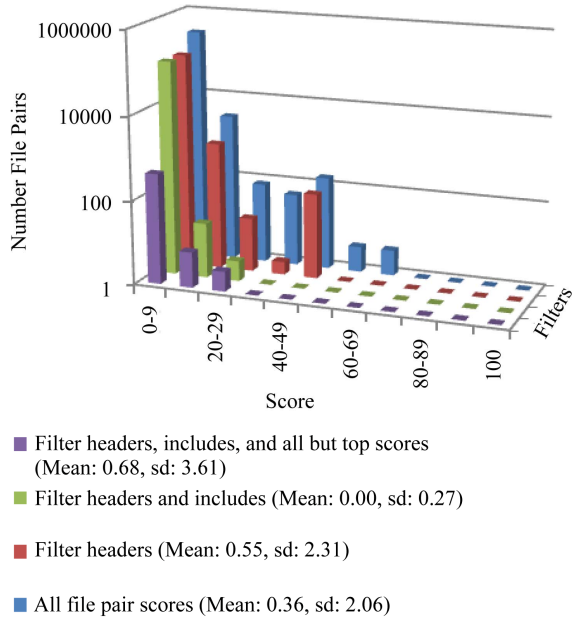

Figure 2. Comparison of two different programs on a logarithmic scale.

The linear scale demonstrates the very large local maximum for the low scores, and the logarithmic scale shows greater detail for each level of filtering that we performed. The large disparity between the number of file pairs at the lower local maximum and the rest of the data is easily seen in **Figure 1**. **Figure 1** should be kept in mind as one views the following logarithmically scaled charts.

Starting from the back of **Figures 1** and **2** the rows of results show all the file pairs, the filtering out of the header files, the additional filtering out of the #include statements, and finally the additional filtering out of all but the top scoring file pairs for each Apache file that remains.

The comparison of all the file pairs produced a skewed distribution with a mean score of 0.36 and a small standard deviation of 2.06. As expected, a low mean score and small standard deviation were observed through all the filtering, with the final filtering of header files, #include statements, and all but the highest scores producing a skewed distribution with a mean score of 0.68 and a small standard deviation of 3.61.

The final filtering step that one might use to identify copied code is to only look at the highest matching file pair for each file. Once headers and #includes were filtered out there were no file pairs with similarity scores more than 11 times the standard deviation above the mean. A manual examination of the higher scoring file pairs showed that false positives had occurred on a series of common source code statements such as function definitions, variable definitions, and preprocessor commands that were otherwise dissimilar.

4.3. Comparison of Two Versions of the Same Program

Version 1.0 of the Linux Kernel was compared against the much later version 2.6. The 2.6 version was released in late 2003 with 12,412 distinct source code files, which is 9 years after version 1.0 was released in 1994 with only 488 source code files.

We expected to observe lower scores, and few, if any, 100% similarity scores, but we also expected to find some high similarity scores from code that has persisted over the years. The result of the comparison of version 2.6 to version 1.0 of the Linux kernel is shown in **Figure 3** with the y-axis scaled logarithmically for the clearest detail. Starting from the back of the chart, the rows of results show all the file pairs, the filtering out of the header files and the #include statements, and finally the additional filtering out of all but the top scoring file pair for each remaining Linux 2.6 file.

The initial comparison of all the file pairs produced a skewed distribution with a mean score of 0.48 and standard deviation of 2.62, which are both slightly higher than the previous comparison of different projects. The filtering of header files and #include statements produced a skewed distribution with a low mean of 0.2 and a narrow standard deviation of 0.48

The large number of possible file pairs illustrates the importance of performing the final step of filtering all but the highest scores. This final filtering produced a skewed distribution with a mean score of 2.33 and a standard deviation of 5.85. There were several file pairs with simi-

Copyright © 2012 SciRes.

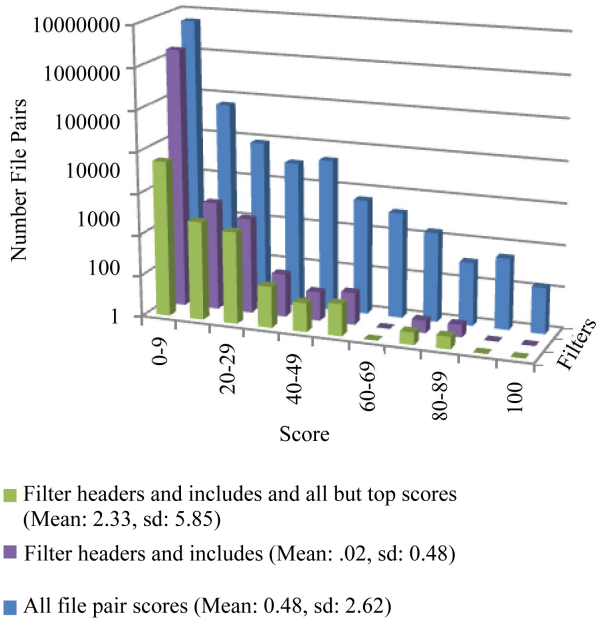

Figure 3. Comparison of two different versions of the same program on a logarithmic scale.

larity scores that were up to 14 times the range of the standard deviation. Many of these statistically significant file pairs did indeed contain substantial similarity.

5. Conclusions

Examining the sequences of matching whitespace patterns produced by a comparison of a single version of a program against itself, different versions of a program against each other, and two completely different programs demonstrated that this can potentially be a reliable method of measuring source code similarity and precisely identifying similar sections of code.

The comparison of a single version of a program to itself showed that a file compared with itself has a whitespace sequence similarity score of 100.

The low mean and small standard deviation from the comparison of two different programs, Apache and Linux, demonstrates that this method accurately scores file pairs without similarity. Two different programs should have very little similarity, and low scores, which was confirmed by the low mean.

When examining two different versions of the same program, version 2.6 and 1.0 of Linux, a skewed distribution with a mean and standard deviation that was higher than the comparison of different programs was detected. Therefore this method appears to reliably differentiate between similar and dissimilar projects.

Although the mean of the comparison of different version was still larger than that from the comparison of different programs, we had expected more of a difference between the means and standard deviations for both the raw and filtered data. Version 2.6 of Linux is a relatively large project, which provides many possible file pairs when compared with version 1.0, a small project, which only allows for a limited number of correlated file pairs. The final filtering, which limits the number of possible file pairs did show a larger mean, and the difference between the similar and dissimilar comparisons was more easily detectable.

The comparison of different versions of the same project produced local maxima of high scores far above the range of the standard deviation and the comparison of different projects did not. Upon manual inspection the higher scoring statistically significant file pairings, those which were far above the range of the standard deviation, generally demonstrated similarities, unlike the file pairings inside a few standard deviations.

It is further observed that if the similarity score data cannot be filtered to provide a standard deviation small enough to allow for the possibility of statistically significant similarity scores then the method is not reliable. This supports the finding in our previous paper that the whitespace patterns of individual lines were not unique enough to make accurate and precise analysis of software similarity, because the high mean and large standard deviations observed did not allow for the occurrence of statistically significant file pairs [21].

However, the method of comparing large scale whitespace patterns by looking for sequences of matching lines of whitespace did differentiate between a comparison of a program against another version of itself, and a comparison of a program against another program. The examination of sequences of lines of whitespace patterns appears to be a good method for determining some source code similarities and the higher scoring statistically significant file pairings were found to be indicative of copied files.

It is not possible to directly compare our method of examining actual whitespace patterns with other methods, because the state-of-the-art source code comparison methods disregard the whitespace patterns, introduce artificial whitespace fingerprints, or only compare whitespace metrics. We do not think that our method should be used in place of existing methods, but since our method examines aspects of source code that are often dismissed and has been shown to successfully identify similar source code files it could provide some advantages when used in conjunction with existing methods.

One very useful possibility for whitespace sequence detection is to compare very large programs where an in-depth comparison tool may take too long to run. Whitespace sequence matching could be used to locate small sets of files that appear similar. The in-depth comparison tool would then be run on those files in a reasonable amount of time to detect copying.

Copyright © 2012 SciRes.

Detecting similar code after the function and variable names have been replaced or type definitions have been altered is a challenge for many plagiarism detection methods, so another possible use for whitespace sequence detection could be the detection of source code where the similarity has been obfuscated by global find and replacements [3,6,16].

6. Future Work

Although this method has been shown to be good it has not been tested for completeness. Many file pairs with high scores do show copying, and file pairs with low scores did not show similarity. Also the method has not been tested to see if it can miss similarity, producing false negatives. Since false negatives can be as detrimental as false positives, this method may be especially useful for detecting similarities between obfuscated code, but may not be as useful to eliminate the possibility of similarities between two bodies of source code. It would be interesting to see what particular types of copying this method can and cannot detect through further testing. It would also be interesting to directly compare this method against methods that rely upon whitespace metrics.

7. Acknowledgements

Thank you to Jim Zamiska for his insight and valuable feedback.

Thank you to John Pfeiffer for his valuable validation.

REFERENCES

[1] E. Brady and C. Morris, "Whitespace," 2004. http://compsoc.dur.ac.uk/whitespace

[2] G. Cosma and M. Joy, "Source-Code Plagiarism: A UK Academic Perspective," *Research Report*, University of Warwick, Coventry, 2006, pp. 116-120.

[3] G. Cosma, "An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis," Ph.D. Thesis, University of Warwick, Coventry, 2008.

[4] P. J. Plauger, "Fingerprints," *Embedded Systems Programming*, Miller Freeman, San Francisco, 1994, pp. 84-87.

[5] S. Schleimer, D. Wilkerson and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," *Proceedings of the* 2003 *SIGMOD International Conference on Management of Data*, San Diego, 9-12 June 2003, pp. 76-85.

[6] B. Cui, L. Han, Y. Hao, Z. Li, J. Wang and R. Zhang, "Type Redefinition Plagiarism Detection of Token-Based Comparison," *Proceedings of the* 2010 *International Conference on Multimedia Information Networking and Security of the IEEE Computer Society*, Nanjing, 4-6 November 2010, pp. 351-355.

[7] G. Malpohl, M. Philippsen and L. Prechelt, "Finding Plagiarisms among a Set of Programs with JPlag," *Journal of Universal Computer Science*, Vol. 8, No. 11, 2000, pp. 1016-1038.

[8] M. Wise, "YAP3: Improved Detection of Similarities in Computer Program and Other Texts," *Proceedings of the* 27*th SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, 15-18 February 1996, pp. 130-134.

[9] C. Anderson and M. Ellis, "Plagiarism Detection in Computer Code," Rose-Hulman Institute of Technology, Terre Haute, 2005.

[10] H. T. Jonkowitz, "Detecting Plagiarism in Student Pascal Programs," *The Computer Journal*, Vol. 31, No. 1, 1998, pp. 1-8. doi:10.1093/comjnl/31.1.1

[11] E. Merlo, "Detection of Plagiarism in University Projects Using Metrics-Based Spectral Similarity," *Dagstuhl Seminar Proceedings*, Dagstuh1, Saarland, 2007.

[12] R. Zeidman, "Software Source Code Correlation," *Proceedings of the* 5*th IEEE/ACIS International Workshop on Component-Based Software Engineering*, Honolulu, 10-12 July 2006, pp. 383-392. doi:10.1109/ICIS-COMSAR.2006.79

[13] R. Zeidman, "Multidimensional Correlation of Software Source Code," *Proceedings of the* 3*rd International Workshop on Systematic Approaches to Digital Forensic Engineering*, Oakland, 22-22 May 2008, pp. 144-156. doi:10.1109/SADFE.2008.9

[14] H. Li, Z. J. Li, H. H. Yan and H. Xiong, "BUAA_Anti-Plagiarism: A System to Detect Plagiarism for C Source Code," *Proceedings of the International Conference on Computational Intelligence and Software Engineering*, Wuhan, 11-13 December 2009, pp. 1-5. doi:10.1109/CISE.2009.5366790

[15] U. Bandara and G. Wijayarathna, "A Machine Learning Based Tool for Source Code Plagiarism Detection," *International Journal of Machine Learning and Computing*, Vol. 1, No. 4, 2011, pp. 337-343.

[16] J. Hamblen and A. Parker, "Computer Algorithms for Plagiarism Detection," *IEEE Transactions on Education*, Vol. 32, No. 2, 1989, pp. 94-99. doi:10.1109/13.28038

[17] C. Daly and J. Horgan, "A Technique for Detecting Plagiarism in Computer Code," *The Computer Journal*, Vol. 48, No. 6, 2005, pp. 662-666. doi:10.1093/comjnl/bxh139

[18] S. Aliefendic, "Using Whitespace Patterns to Detect Plagiarism in Program Code," School of Computer Science and Informatics University College Dublin, Dublin, 2003.

[19] R. Zeidman, "The Software IP Detective's Handbook: Measurement, Comparison, and Infringement Detection," Prentice Hall, Boston, 2011

[20] B. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," *Proceedings of the Second Working Conference on Reverse Engineering*, Washington DC, 1995, pp. 86-95.

[21] I. Shay, N. Baer and R. Zeidman, "Measuring Whitespace Patterns as an Indication of Plagiarism," *Proceedings of the ADFSL Conference on Digital Forensics, Security and Law*, St. Paul, 20 May 2010, pp. 63-72.

[22] N. Baer and B. Zeidman, "Measuring Software Evolution with Changing Lines of Code," *Proceedings of the* 24*th International Conference on Computers and Their Applications*, New Orleans, 8-10 April 2009, pp. 264-270.

Copyright © 2012 SciRes.