Aleksander Wojdyga[1]

TOWARDS INTELLECTUAL PROPERTY THEFT PREVENTION: ECONOMIC SIGNIFICANCE OF AUTOMATIC SOFTWARE PLAGIARISM VERIFICATION

Software plagiarism can have destructive effects on company's economy or one's personal carrier. Easy access to online source code databases encourages such behaviour. Authorship of a unit of software should be easily determined. This is a problem in education at the university level, especially for the off-site or e-learning courses. Verification by human effort is tedious and error-prone, therefore not acceptable. This article presents expectations and vulnerabilities of automatic plagiarism verification and presents a general method for solving this problem. A proof-of-concept code in Haskell functional language implements such an algorithm.

Keywords: *automatic plagiarism verification, software plagiarism, intellectual property, theft prevention.*

Олександр Войдига

ЗАПОБІГАННЯ РОЗКРАДАННЮ ІНТЕЛЕКТУАЛЬНОЇ ВЛАСНОСТІ: ЕКОНОМІЧНЕ ЗНАЧЕННЯ АВТОМАТИЧНОЇ ПЕРЕВІРКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ПЛАГІАТ

У статті показано, що плагіат у програмному забезпеченні може мати руйнівні наслідки для компанії та приватних осіб. Легкий доступ до онлайнових баз даних вихідного коду заохочує таку поведінку. Авторство конкретного програмного забезпечення має бути легко визначуване. Це проблема у сфері освіти на університетському рівні, особливо в разі віддаленого або електронного навчання. Перевірка вручну втомлива і ненадійна, тому неприйнятна. Описано очікування і вразливості автоматичної перевірки на плагіат і представлено загальний метод вирішення цієї проблеми. Такий алгоритм застосований для коду, який доводить правильність концепції, на функціональній мові Haskell.

Ключові слова: *автоматична перевірка на плагіат, плагіат у програмному забезпеченні, інтелектуальна власність, запобігання розкраданням.*

Александр Войдыга

ПРЕДОТВРАЩЕНИЕ ХИЩЕНИЙ ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ: ЭКОНОМИЧЕСКОЕ ЗНАЧЕНИЕ АВТОМАТИЧЕСКОЙ ПРОВЕРКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ НА ПЛАГИАТ

В статье показано, что плагиат в программном обеспечении может иметь разрушительные последствия для компании и частных лиц. Легкий доступ к онлайновым базам данных исходного кода поощряет такое поведение. Авторство конкретного программного обеспечения должно быть легко определяемо. Это проблема в сфере образования на университетском уровне, особенно в случае удаленного или электронного обучения. Проверка вручную утомительна и ненадежна, поэтому неприемлема. Описано ожидания и уязвимости автоматической проверки на плагиат и представлен общий метод решения этой проблемы. Такой алгоритм применен для кода, доказывающего правильность концепции, на функциональном языке Haskell.

[1] Institute of Computer Science, Lublin University of Technology, Poland.

Ключевые слова: *автоматическая проверка на плагиат, плагиат в программном обеспечении, интеллектуальная собственность, предотвращение хищений.*

1 Software plagiarism. Software development is a process involving human effort and machine work. Number of different tools and integrated development environments (IDE) such as Microsoft Visual Studio [9], The Eclipse Platform [16] or QtCreator [10] improve editorial tasks, binary package generation or even refactoring tasks. Software design patterns allow for component reuse and solving general problems. However, there is always a part of software, unique in every system, which must be handwritten. Elimination of human work may be possible in future due to advancements of model-driven software engineering [13].

Education at the university level in computer science or related courses (mathematics, physics) usually involves classes in computer programming languages, algorithms or software development engineering. During examinations or laboratory classes students are required to produce complete programs, modules or other appropriate programming language parts in a conditions of controlled independence. Cheating, involving communication with other students or copying part of solutions, results in failed exams and, possibly, legal repressions.

Copying can be a problem for e-learning online courses involving programming languages. Except standard online tests, students sometime have to produce computer programmes. As all major e-learning platforms use login and password authentication scheme, it is impossible to check communication between students. They can work together, creating one unit of software and submitting it to the online system. Moreover, author is aware of at least one students' online forum with restricted access (unavailable to non-students). Users of this forum share solutions, especially computer programs. Solution containing alleged plagiarism has to be checked manually and compared to other programs. Depending on student's skills such comparison can be tiresome and error-prone. Program author can use various camouflage techniques, as described in following subsection.

Computer programme source code can be stored in many online hosting services such as SourceForge [15], Google Code [4] or Savannah [12]. Although these services allow for strict access control, most of the projects have granted the read-only access to general public. Therefore, one can browse and download any file. Quality of this software varies, simply copying the software also copies the bugs, unwanted features, performance problems etc. As the author committing plagiarism will not reveal the origin of source code, the correction of these errors is made very difficult. Computer science student may receive unsatisfactory grade. A software development company cannot allow this, for it results in increased costs. Despite copying reduces programming effort and results in shorter development time, it can result in subtle errors or incomprehensible source code. The maintain costs increase, as bugs are duplicated throughout the system.

It must be noted that this article does not cover software patents. The author does not see any possibility for automation, as these issues, though very interesting, are usually treated on a very high design and analysis level. This paper tries to confront with the, so called, copy-paste development method. In fact, there are software licences that allow for this, consider for example donating the source code to public

domain, the most accessible introduction is contained in [6]. The intellectual property theft happens in other place — by denying the authorship of original software. While the citations are welcome either in software development, education or scientific work, appropriation of other people's work may be an interesting psychological issue.

1.1 Software clones camouflage techniques. Probably the most extensive survey on software plagiarism and verification methods is the technical report for Queen's University, Canada [11]. Reader is referred to that document, as it contains well written classification and rich bibliography. This section briefly presents the basic notions.

A code fragment, copied and pasted without or with minor modifications is called a software clone or a code clone. If modifications result in different behaviour — different output for the same input - one can assume it is a new, original piece of software. In computer science education, however, a student may wish to hide the fact of copying and pasting. Here follows a list of common camouflage techniques and possible countermeasures.

1. Indentation changes and white spaces deletion or addition. Trivial to perform; at first glance two dubious programmes will look very different. However, with a standard Unix program indent it is possible to translate any given source code to standard indentation rules.

2. Comments modifications. Parsers and compilers discard comments by default.

3. Variable renaming. Easy to perform with a help of refactoring tools. Needs algorithmic treatment.

4. Function parameters reordering. Again one of the refactoring issues.

5. Redundant instructions insertion. Probably the most interesting approach, covered thoroughly in this paper.

The last point requires some originality and practical programming skills. It cannot be accomplished by an inexperienced or undereducated student. This list is, clearly, not closed. The author will be happy to know other camouflage techniques.

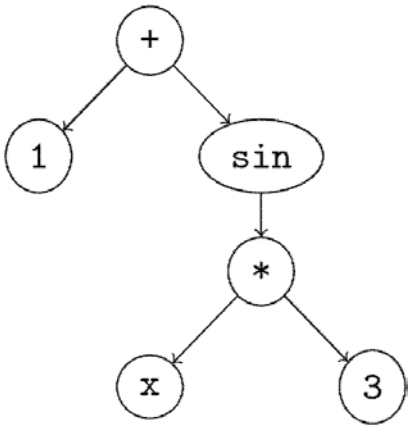

Figure 1. **AST for expression 1 + sin(x * 3)**

1.2 Verification methods. There is a number of software plagiarism verification methods. They work on different levels of source code: textual, syntactical or semantical.

The textual approach will compare character strings in two alleged programmes. This method is very simple and very efficient, but applicable only to pasting without modifications. Standard Unix tool diff is capable resolving this kind of software clones. However, simple variable renaming causes false negative answer.

The token analysis lies somewhere between the textual and syntactical levels. The tokens are lexical entities of the programming language like operators, variable occurrences, keywords etc. Source code is first preprocessed and translated to token stream and substring match is applied to that stream. The most prominent example of software implementing this method is CCFinder, see [5] for details.

An abstract syntax tree (AST) is an intermediate compiler representation of expressions. Every node of the tree represents variable, operator, function application etc. In general, AST is structure of an expression, comparing syntax trees is effectively a comparison of expression in an algorithmic way. Figure 1 contains an example AST for expression 1+sin(x*3).

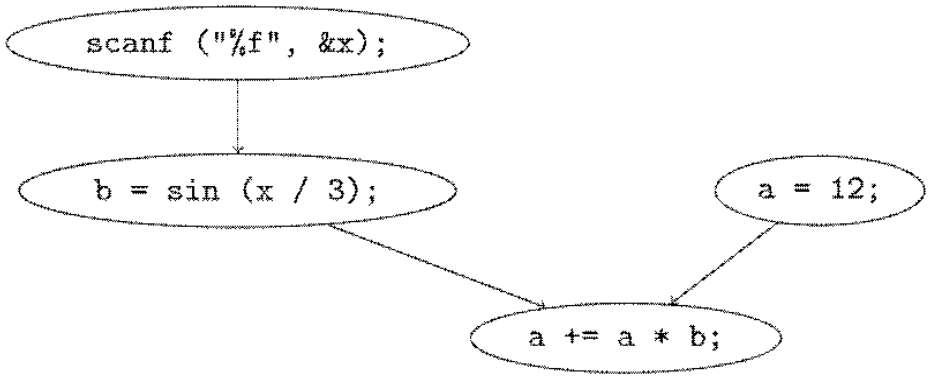

Figure 2. **Program dependency graph**

Tree matching algorithms with parameterized hashing functions were used in a tool called CloneDR. It allows for concurrent work and can manage with variable renaming, see [2] for details.

```
1. scanf ("%f", &x);
2. a = 12;
3. b = sin (x / 3);
4. a += a * b;
```

Program dependency graph (PDG) is a directed graph, whose nodes are instructions and edges are computational dependencies. Node B is computationally dependent over node A if variable modified in instruction A is used in instruction B. Reader may be more familiar with a notion of data flow. This technique is very robust to handle variable renaming or instruction insertion and deletion. The drawback of this approach is the non-polynomial computation complexity. Depending on data structures applied, one may reach the isomorphic subgraph-matching, classical NP-complete problem. There are possible performance optimisations, consider for example GPLAG [8] and node thresholds applied there. This subject in under active research, please refer to [3].

Figure 2 illustrates a PDG for the C-source snippet on Listing 1: An example C source code. The instruction in the second line can get exchanged with the previous one or the next one, as there are no connections with appropriate nodes on Figure 2.

2. Plagiarism verification algorithm and its implementation. The verification algorithm is expected to handle the camouflage techniques presented in previous section. Despite the lack of any exhaustive studies in software plagiarism in education, the author considers insertion of dummy instructions and identifier renaming as the two most common in students' assignments. Please, note that a change in instruction sequence or instruction deletion should be considered as creating new, original unit of software. When done at random, it is expected to result in a failed exam.

There are some non-obvious notions used in the algorithm. The computational result of a function is built from the following objects (variables, arrays etc):

1. objects used in the return statement;
2. objects used in I/O functions (including, e.g., database function);
3. objects modified in the function body, either global or passed as the argument.

In a general-purpose imperative programming languages (such as C, C++, Java) there exists a main() function or its substitute. The computational result of a main() function does not contain the third point from the enumeration above. The necessary input are variables used to determine the computational result. They either appear on the right-hand-side of assignment instructions or are passed as argument to function modifying one of the computational result.

Nodes in a PDG, not reachable for one another, allow to change the order of relevant instructions. Such instructions are called computationally independent.

The algorithm. Here follows the complete algorithm, it is applied for every function in a given computer program.

1. Construct a abstract syntax tree
2. Construct a program dependency graph
3. Identify the computational result of the function
4. Identify the necessary input
5. Remove surplus instructions
6. Identify and group computationally independent consecutive instructions
7. Compare instructions and groups

Comparison of instructions (original and copied) is applied to normalised expressions. The normalisation procedure is a translation to prefix notation preserving operator priorities (including function application). For example, both expressions 1 + sin(x * 3) and sin(3 * x) + 1 are translated to + sin * x 3 1. Comparison of normalised expressions results in a identifier renaming scheme. This scheme will be void - no identifier substitution possible - for structurally different expressions. In this case, the comparison stops at first operator mismatch. Theoretical computer scientists call this property the α-equivalence, probably the best (conceptually nearest) example is the α-equivalence for the λ-calculus. Reader is referred to classical positions [1] or [14] for full explanation of this notions.

All steps of the algorithm are linear referring to the size of input. Only the sixth step requires non-polynomial computation. If a function contains many variable declarations, this can be a performance problem. However, it is assumed that the author

hiding a copy would insert surplus instructions quite often. Thus, the sequences of computationally independent instructions will be short.

The implementation. The algorithm has been partially implemented using Haskell functional language with a help of Language.C library. It is obvious that this library and implementation of the said algorithm in only applicable to C programming language. This is probably the most popular, general purpose language [7], it is also present in all computer science studies at the university level. The functional language allows for easy processing the AST through advanced pattern matching; this feature is present in all functional programming languages. It is also easy to manually programme the normalisation of expressions or other recursive properties such as, for example, the computational result of a function.

The current state of implementation does not allow for performance testing. Two major steps are missing: construction of PDG and identification of computationally independent instructions. This in not uncommon, for instance very advanced tool such as GPLAG does not create PDG and depends on external tools. On the other hand, as a consequence of chosen language and library, the functions for computational results and necessary input have been produced with a little effort.

3 Summary and future work. As stated in the previous section, the verification algorithm has not been fully implemented. The short-term goal is complete Haskell module ready for performance tests. More interesting is the presupposed mid-term goal. Imagine online scoring system, which not only compiles and runs students' programmes but also checks all pairs of authors for presence of software clones. Students will receive information about correctness, efficiency and originality of their solutions. Full textual information like: "You have copied function factorial from student John Smith but changed variable counter to i and parameter n to x" should discourage copying and, more important, sharing solutions.

Some parts of the algorithm may have one more possible application. Efficient identification of computationally independent instructions allows for automatic parallelisation. Prevalence of multi-core processors and multi-processor computers entails a considerable use of these technologies. While students are educated towards parallel programming, one could use some tools for automatic parallelisation. This subject is currently under significant development worldwide. It may be possible to achieve some satisfactory results in future on the basis of the algorithm presented in this article.

References:

Barendregt, H. P. (1984). The Lambda Calculus - Its Syntax and Semantics. Vol. 103. North-Holland.

Baxter, I. D. et al. (1998). "Clone Detection Using Abstract Syntax Trees". In: ICSM, pp. 368-377.

Chen, R. et al. (2010). Author Identification of Software Source Code with Program Dependence Graphs. In: Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops. COMPSACW '10. Washington, DC, USA: IEEE Computer Society, pp. 281-286.

Google Code. URL: http://code.google.com/.

Kamiya, T., Kusumoto, S. and Inoue, K. (2002). CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. In: IEEE Transactions on Software Engineering 28, pp. 654-670.

Lessig, L. (2005). Free culture. New York: Penguin Books. URL: http://openlibrary.org.

Lipovaca, M. (2011). Learn You a Haskell for Great Good!: A Beginner's Guide. 1st. San Francisco, CA, USA: No Starch Press.

Liu C. et al. (2006). GPLAG: detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. KDD '06. Philadelphia, PA, USA: ACM, pp. 872-881.

Microsoft Visual Studio. URL: http://www.microsoft.com/visualstudio/.

QtCreator. URL: http://qt.nokia.com/products/developer-tools/.

Roy, C. K. and Cordy, J. R. (2012). A Survey on Software Clone Detection Research. Tech. rep. 2007-541. Accessed June. 25. Queen's University, 2007. URL: http://www.cs.queensu.ca/TechReports/Reports/2007-541.pdf.

Savannah. URL: http://savannah.gnu.org/.

Schmidt, D. C. (2006). Model-Driven Engineering". In: IEEE Computer 39.2, pp. 25-31.

Sorensen M. and Urzyczyn P. (1998). Lectures on the Curry-Howard isomorphism. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.7385.

SourceForge. URL: http://sourceforge.net/.

The Eclipse Platform. URL: http://www.eclipse.org/.

Стаття надійшла до редакції 22.08.2012.