Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Credible, resilient, and scalable detection of software plagiarism using authority histograms

Dong-Kyu Chae[a], Jiwoon Ha[a], Sang-Wook Kim[a,*], BooJoong Kang[b], Eul Gyu Im[a], SunJu Park[c]

[a] Department of Computer and Software, Hanyang University, 17 Haengdang-dong, Seongdong-gu, Seoul 133-791, Republic of Korea
[b] School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast BT3 9DT, United Kingdom
[c] School of Business, Yonsei University, Sinchon-dong, Seodaemun-gu, Seoul 120-749, Republic of Korea

ARTICLE INFO

Article history:
Received 12 March 2015
Revised 25 November 2015
Accepted 20 December 2015
Available online 31 December 2015

Keywords:
Software plagiarism detection
Birthmark
Similarity analysis
Static analysis

ABSTRACT

Software plagiarism has become a serious threat to the health of software industry. A *software birthmark* indicates unique characteristics of a program that can be used to analyze the similarity between two programs and provide proof of plagiarism. In this paper, we propose a novel birthmark, *Authority Histograms* (*AH*), which can satisfy three essential requirements for good birthmarks—*resiliency, credibility*, and *scalability*. Existing birthmarks fail to satisfy all of them simultaneously. *AH* reflects not only the frequency of APIs, but also their call orders, whereas previous birthmarks rarely consider them together. This property provides more accurate plagiarism detection, making our birthmark more resilient and credible than previously proposed birthmarks. By *random walk with restart* when generating *AH*, we make our proposal fully applicable to even large programs. Extensive experiments with a set of Windows applications verify that both the credibility and resiliency of *AH* exceed those of existing birthmarks; therefore *AH* provides improved accuracy in detecting plagiarism. Moreover, the construction and comparison phases of *AH* are established within a reasonable time.

1. Introduction

*Software plagiarism*is developing software using someone else's source code or open source code without a license and disguising it as original software [1]. As software plagiarism has been increasing significantly, serious economic loss in the software industry has also been increasing. According to the Business Software Alliance report,[1] the financial damage caused by software plagiarism in the USA is about 95 million dollars, and the damage in China is about 77 million dollars. To mitigate such economic losses, software developers need methods to detect software plagiarism.

Recently, *software birthmarks* (*birthmarks*) have come under study. A program's birthmark represents unique characteristics that can be used to identify the program [2]. The similarity between two birthmarks represents how much one program is likely to be a copy of another. Birthmarks permit a compact analysis of the similarity between a pair of programs without requiring extra data, such as the source code of the programs in question.

Existing birthmarks can be categorized according to the following criteria. First, depending on the extraction scheme, birthmarks are categorized as *static* or *dynamic*: a static birthmark is extracted by disassembling a program without execution, and a dynamic birthmark is extracted from a program's runtime behavior and can be obtained by executing it [1]. Second, depending on the form of the birthmarks, they are classified as *set-based, frequency-based*, and *sequence-based*.

Regardless of these categories, birthmarks need to be designed to meet the following requirements [3]:

- *Resiliency*: Birthmarks should be robust even if plagiarizers modify the structure of a program slightly or reorder the source codes statements while preserving the semantics of the program.
- *Credibility*: Birthmarks extracted from independently developed programs should be dissimilar even if they accomplish similar tasks.
- *Scalability*: Birthmarks should be applicable to even large programs.

* Corresponding author. Tel.: +82 10 6749 6392.
E-mail addresses: kyu899@agape.hanyang.ac.kr (D.-K. Chae), oneofus@agape.hanyang.ac.kr (J. Ha), wook@hanyang.ac.kr (S.-W. Kim), b.kang@qub.ac.uk (B. Kang), imeg@hanyang.ac.kr (E.G. Im), boxenju@yonsei.ac.kr (S. Park).

[1] BSA Global Software Piracy Study, http://globalstudy.bsa.org/2010.

http://dx.doi.org/10.1016/j.knosys.2015.12.009
0950-7051/© 2015 Elsevier B.V. All rights reserved.

However, existing birthmarks fail to satisfy all the above requirements at once. We briefly explain, for each category, which properties are not satisfied and why.

- *Static set-based birthmarks*: Their credibility is unsatisfactory because they cannot distinguish two independently developed programs that inadvertently use several APIs in common. Choi et al. [4] improved the credibility by separating all APIs into subsets and labeling each subset with the name of the function that calls the APIs in the subset. However, that improvement makes such birthmarks vulnerable to function structure transformations, such as function in-lining and dividing one function into multiple functions, as demonstrated in our experiments. They also suffer from the scalability issue because they use the *maximum weighted bipartite matching* algorithm, which has a time complexity of $O(n^3)$ for taking all user-defined functions into account when computing similarity.
- *Static frequency-based birthmarks*: They are more credible and scalable than the original set-based birthmarks because they reflect API call frequencies. However, they are still likely to lack credibility if two different programs use many APIs in common with a similar frequency distribution or use two or three common APIs whose frequencies are much higher than those of any other APIs in each program [5]. These drawbacks are also shown in our experiments.
- *Static sequence-based birthmarks*: The main problem with these birthmarks is poor resiliency because assembly instruction or API sequences can be easily changed by switching some statements in the source code (e.g., function calls or mathematical operations) while preserving the semantics of the program [6]. Scalability is also a problem because an exponential number of possible traces exist according to the branches defined in the program, all of which need to be considered in a similarity computation of the two birthmarks.
- *Dynamic birthmarks*: Regardless of their form, it is questionable whether dynamic birthmarks can capture the unique characteristic of a program, which is the fundamental objective of designing birthmarks [7,8]. Because dynamic birthmarks are extracted during the execution of *some* pre-defined scenarios, only a small part of the program is reflected in the birthmarks. It does not seem appropriate to consider dynamic birthmarks as characteristics of a program because they inherit only a small part while completely ignoring the rest [4,9,10].

In this paper, we propose a novel static birthmark, *Authority Histograms* (*AH*). *AH* satisfies all three of the requirements above. Our birthmark is a histogram whose dimension is an API used in the program, and the value of each dimension indicates an *authority score* that represents how prominently the corresponding API is used in the program. We measure the importance of APIs by analyzing the program's structural characteristics to figure out which APIs are in a core position in the program structure and which are not.[2]

Specifically, we first construct an *API-labeled control flow graph* (*A-CFG*), which is a graphical representation of a program. *A-CFG* has APIs as vertices and call orders among APIs as edges. *A-CFG* represents the full structure of the program, having all possible control flows from the start of the program to its termination. Next, we measure the authority score of each API based on the structural characteristics of *A-CFG*. To compute the authority scores, we use *random walk with restart* (*RWR*), a probabilistic model for a random surfer to reach a web page on a web graph after a given number of iterations. *RWR* captures which web pages are authoritative in the graph and gives high authority scores to those nodes by analyzing the structural characteristics of the graph [11]. With respect to our work, *RWR* figures out which APIs are popularly called in a program by analyzing *A-CFG* and gives high authority scores to those APIs. At this step, the authority scores are affected not only by the number of incoming edges of API nodes (i.e., call frequency of APIs) but also by the orders of API nodes [12]. The resulting histogram of the authority scores over all the APIs becomes *AH*. Two programs with similar *AH*s are highly suspected of plagiarism because they not only use similar important and minor APIs but also have similar structural characteristics.

Our design ensures that *AH* satisfies the three essential requirements as follows. First, by reflecting both the frequency and call order of APIs, we remedy the deficiencies of the previous set-based, sequence-based, and frequency-based birthmarks,making our birthmark more resilient and credible. Generally, API calls are a common way for a program to request resources or services provided by the OS. API calls are tightly related to the main functionalities of a program. The importance distribution of APIs for each program can be unique. Moreover, it is difficult for a plagiarist to manipulate the overall call frequency of APIs, the call order of APIs, or to replace APIs with something else, while maintaining the program's original semantics [3]. Second, by generating an n-dimensional histogram that inherits the structural characteristics of *A-CFG*, our birthmark makes the problem of comparing two *A-CFGs* a quite simple and scalable task. We compute the similarity between *AH*s efficiently by using the *cosine similarity*, a simple and widely used similarity measure.

Based on our proposal, we implemented the *AH*-based software plagiarism detection method. Given original and suspicious programs, we first constructed *A-CFGs* for each program. Then we generated *AH*s from each *A-CFG* and computed their similarity. Based on the similarity value, we determined whether the suspicious program was copied from the original one.[3] We used a set of Windows programs and performed extensive experiments. We observed that *AH* outperforms previous state-of-the-art static birthmarks in terms of resiliency and credibility, thereby providing improved accuracy in detecting plagiarism. We also observed that the extraction and comparison of *AH* birthmarks could be completed within a reasonable time.

The initial idea of this paper was presented with some preliminary experimental results at ACM CIKM 2013 as a short paper [17]. This paper is an extended version of it. The main differences can be clarified as follows. In the extended version, we explain both our proposed method and previous work in more detail. Also, we have conducted more extensive experiments than described in our previous paper: (1) we additionally implement *Flow-Path birthmark*, a state-of-the-art sequence-based birthmark, and perform comparative experiments using it and our method; (2) we generate two kinds of real plagiarized samples (one with changed compiler optimization options and the other manually transformed by human experts), and perform experiments with the plagiarized samples; (3) we perform scalability testing by measuring execution times for every method according to the size of the target programs.

The rest of this paper is organized as follows. Section 2 briefly reviews some methods for detecting software plagiarism. Section 3 explains in detail *AH*'s definition, generation procedure, and similarity computation. Section 4 evaluates our *AH* based software plagiarism detection method by comparing it with previous methods

[2] We assume that programs in question are large in size and use a number of APIs inside (i.e., commercial programs released by software companies). If programs are small and use few APIs (i.e., toy programs), our proposed method tends not to work well.

[3] We note that our scope is to provide the similarity between two programs in question. Deciding which program is the original one is out of the scope of this paper.