A Combined Static and Dynamic Software Birthmark Based on Component Dependence Graph

Xiaoming Zhou[1] Xingming Sun[1] Guang Sun[1,2] Ying Yang[1]
1. School of Computer & Communication, Hunan University, China, 410082
2. Hunan Finacial & Economic College, China, 410082
{linghuchong168, sunnudt, simon5115, ying.yung}@163.com

Abstract

Software birthmarking provides an effective approach to detect software theft by computing the similarity of unique characteristics between the suspected program and the original. In this paper, we present and empirically evaluate a novel birthmarking technique which uniquely identifies a program based on static and dynamic component dependence graphs of it. To argue the advantage of the technique, the credibility and reliability against semantics-preserving transformations are evaluated. Experimental results show that our technique is more stable than the WPP birthmark proposed by Myles and Collberg. Additionally, it complements the previously proposed birthmarking techniques which are only static or dynamic.

1. Introduction

Currently, a number of techniques have been used to prevent, discourage, and detect theft. Among the effective techniques for detecting, software birthmarking is still a relatively new area. A software birthmark relies on a unique characteristic, or set of characteristics, that is inherent to a program to uniquely identify it[1]. Similar birthmarks of two programs suggest that one is copy of the other. For existing birthmarking techniques, birthmarks can be classified as two categories. *Static* birthmarks extract the statically available information in the program code, for example the types or initial values of the fields[2]. Myles and Collberg have shown that the details of code are easily altered by using simple code obfuscation techniques like code removal or splitting of variables[3]. *Dynamic* birthmarks, in contrast, rely on information gathered from the execution of the program[3,4]. Thus, it is more difficult to foil in a semantics-preserving way.

Rather than gathering the static characteristics of the program, Myles *et al.* have represented an advisable dynamic birthmark WPP, and evaluated its performance on a small Java program[3]. The WPP birthmark is a slightly modified version of Whole Program Paths technique which is used to compact a program's dynamic control flow graphs (DCFG)[5]. More specifically, it collects all the compact DCFG and regards them as a program's birthmarks. To compute the similarity of two programs' birthmarks, the distance of each pair of DCFGs between them must be computed by calculating the maximum common subgraph (MCS) of the DCFGs. However, calculating MCS for general graphs is a NP-complete problem[6], which leads the WPP birthmark to extremely low efficiency for the large programs. Moreover, as mentioned by the authors, it is fragile to program optimization as well, including loop transformations and inline functions. Therefore, the WPP birthmark still lacks sufficient practicability and resiliency.

In this paper, we present a novel software birthmarking technique based on component dependence graph (CDG). CDG birthmark employs both static and dynamic component dependence graphs in the program. In particular, we abandon to compute the distance of each pair of dynamic CDGs with graph isomorphism algorithm. Instead, we predigest them to unordered trees in order to facilitate the computation. Thus, our technique is more efficient and convenient for large programs. In addition, such component level graphs are less affected by code obfuscation techniques than code level graphs. So, our CDG birthmark holds highly resiliency to various semantics-preserving transformation attacks.

The rest of the paper is organized as follows. Section 2 discusses the related works. Section 3 introduces the basic idea of our birthmark technique and then describes its implementation in detail. Experimental results are also presented and discussed in Section 4. Section 5 draws the conclusion.

978-0-7695-3278-3/08 $25.00 © 2008 IEEE
DOI 10.1109/IIH-MSP.2008.145

2. Related Works

In order to protect the intellectual property for software producers, many software protection techniques have been proposed. Among them, software watermarking is a well-known technique[7-9]. Its basic idea is to identify the originator by imperceptibly embedding a copyright notice into a program. Unfortunately, watermarking is not always feasible because adversaries also can embed their own watermarks. As a result, it will fail to identify who is the originator. Moreover, the performance loss of the watermarked program and the constraint of program size should be taken into account after the insertion of the watermark information.

Comparing to software watermarking, software birthmarking is very different. It does not like a watermark to add code to a program in order to indicate copyright. Instead a birthmark relies on an inherent characteristic of the program to show that one program is a copy of another, and cannot determine original author. To the best of our knowledge, some birthmarks have been proposed so far. Firstly, Tamada, *et al.* proposed four typical static birthmarks aiming for Java class files: constant values in field variables (CVFV), sequence of method calls (SMC), inheritance structure (IS), and used classes (UC)[2]. Unfortunately, Collberg and Myles have demonstrated such static birthmarks even tend to be vulnerable to basic program transformations in their WPP birthmark. However, as we will see in Sect. 4 the WPP birthmark also is weak to commercial obfuscators.

Code clone is another technique which could be used for the copy detection of programs[10]. For a software product, a code clone is a set of code fragments in its source files. The theft is doubted when the code clone is found in other software products. Unfortunately, just like plagiarism detection technique, the drawback is that they are only suitable for the source code level. Nevertheless, software products are often distributed without the source code. In addition, the technique operates statically and without considering the presence of sophisticated obfuscation techniques.

3. Birthmark based on component dependence graph

3.1. Main idea of the birthmark

No matter how complex the program is obfuscated by semantics-preserving transformations, usually, the component dependent graphs of it are unmodified[11,12]. Therefore, the birthmark, resulted from the component dependence graphs, will holds highly resiliency to code transformation techniques. Before making the birthmark more clearly, two component dependences and component dependence graphs are described at first:

Definition 1 *Data Dependence* (*DD*): Let p, q be different components of a program, *DD* (p, q, N, V) denote p has a *DD* on q with the following three conditions:

1) V is a variable or type defined in component q, and N is a method of component p,
2) N uses V when executing,
3) V dose not belong to the inherited class which is in other components.

Definition 2 *Control Dependence* (*CD*): Let p, q be different components of a program, *CD* (p, q, N, F) denote p has a *CD* on q with the following three conditions:

1) F is a method defined in component q, and N is a method of component p,
2) N calls F when executing,
3) F dose not belong to the inherited class which is in other components.

Definition 3 *Local Dynamic Component Dependence Graph* (*LDCDG*): Let each component as a vertex, a *LDCDG* of a program will be a 4-tuple *LDCDG* = (V, E, *CD*, *DD*), where

- V is a set of finite vertices which are traversed by a method in the invoked procedure,
- $E \subseteq V \times V$ is the set of edges,
- *CD*: denotes control dependence defined as the above assigning to the edges,
- *DD*: denotes data dependence just as *CD*.

Definition 4 *Global Static Component Dependence Graph* (*GSCDG*): Differing from the *LDCDG*, the *GSCDG* unify the *DD* and *CD* as component dependence, and a *GSCDG* of a program is a general 3-tuple *GSCDG* = (V, E, D), where

- V is a set of finite vertices of all components just lied in the folder,
- $E \subseteq V \times V$ is the set of edges,
- D: denotes the data dependence or the control dependence.

The *LDCDG*s and the *GSCDG* are regarded as the birthmark of the program, which is called combined static and dynamic component dependence graph birthmark (CDG birthmark). For the original and the suspected program, the *GSCDG* of them provides a bijective function for their components, and the *LDCDG*s decide their similarity. The cooperative relationship of the *GSCDG* and the *LDCDG*s will be introduction in next section.

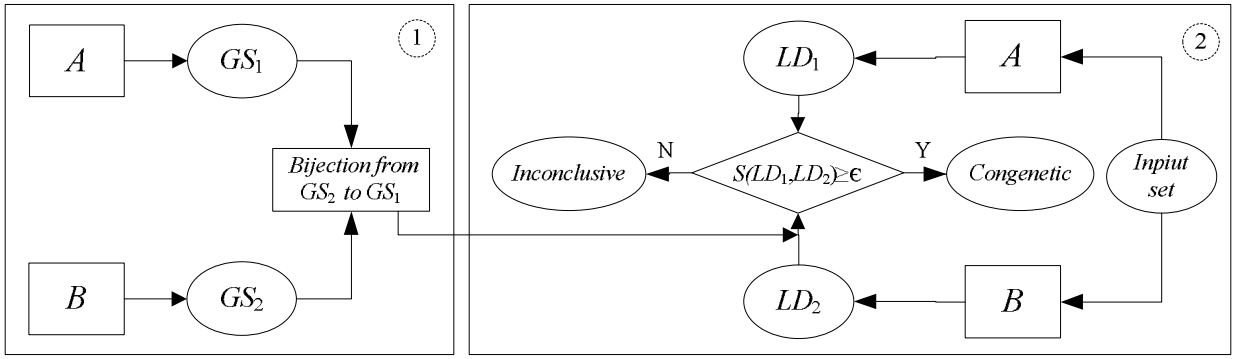

Figure 1. The framework of the CDG birthmark used to detect the software theft

3.2. Detection framework of the birthmark

Figure 1 illustrates the framework of software theft detection system based on component dependence graph. For two programs A and B, GS_1 and GS_2 denote their *GSCDG*, LD_1 and LD_2 express all their *LDCDG*s respectively. In the detection procedure, we extract GS_1 and GS_2 at first, and then build a bijective function for the vertexes of their maximal common subgraph with existent graph isomorphism algorithm[13]. Thus, the vertexes of a *LDCDG* in LD_2 will be mapped to the form in GS_1 if they accord with the bijective function. Under such condition, the similarity of the two programs is obtained by computing the ratio of common part found in LD_1 and LD_2 versus LD_1, which will be defined as S in Sect 3.4. A theft has occurred only the similarity value is larger than the pre-determined threshold ϵ, otherwise, it is not. The detailed processes will be expanded in next two sections.

3.3. Extraction of *GSCDG & LDCDG*s

For the *GSCDG*, since some existent tools are sophisticated for component dependence analysis, we obtain it easily with the analysis tool *Reflector*[14]. The extraction procedure for the *LDCDG*s is slightly difficult, because all the dynamic execution traces with a given input must be analyzed to obtain a *LDCDG*. However, since no .NET code can hide from the profiling API in the .NET Framework 2.0. Moreover, Microsoft has developed a profile tool with it for .NET program, which is called CLR Profiler[15]. The call graph analysis of the Profiler provides us a visual graph for the library calls and all the calls information can be exported to a local file. With the call graph information, Figure 2 shows the schematic construction process for a *LDCDG*. The call graph record is shown on the left, and the *LDCDG* is shown on the right. In that way, therefore, the *LDCDG*s of a program are obtained handily.

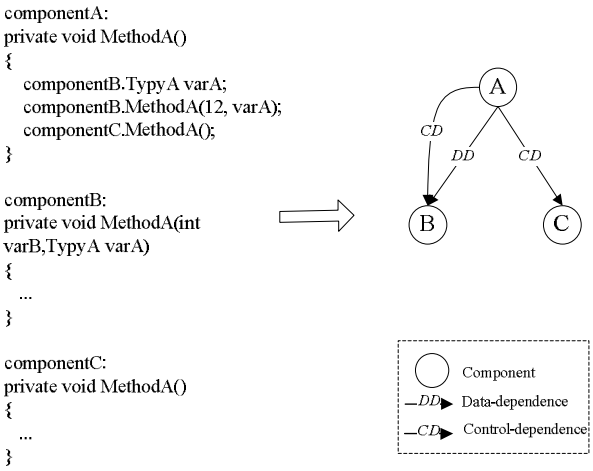

Figure 2. A schematic process turns a call graph into a *LDCDG*

3.4. Similarity of *LDCDG*s

As we have seen in Figure 1, the central operation for theft detection is to compute the similarity of the *LDCDG*s. It seriously impacts on the quality of the CDG birthmark. Let LD_1 and LD_2 be the all *LDCDG*s for the original and the suspected program respectively, we compare the similarity of them by computing the ratio found in both of them versus the LD_1 which can be described as follows:

$$S(LD_1, LD_2) = \frac{|LD_1 \cap LD_2|}{|LD_1|} \times 100\%$$

To find the common part of LD_1 and LD_2 is to detect some *LDCDG*s in LD_2 whether also reside in LD_1, which is a graph matching process. However, due to the graph matching problems usually are NP-hard[6], moreover, LD_1 and LD_2 are enormous as a rule, the matching process will be time-consuming even unattainable. Therefore, we have taken two strategies to facilitate the matching computation. Let g_1 and g_2 be

*LDCDG*s in LD_1 and LD_2 respectively, the detailed process is described as follows:

First, instead of only considering the structure similarity for g_1 and g_2, the vertex names of them are taken for matching. At first, the vertexes in g_2 should be mapped to g_1 with the bijective function introduced in Sect 3.2, and then if a vertex name of $g_1$is different to g_2 in matching process, the current matching is failed at once.

Second, we compress g_1 and g_2 to two unordered trees with depth no more than three since they are simple directed graphs and their scale is small generally. In order to offset the losing information in the compact trees, we name the nodes for system components (SC) with their names, but for developer's own components (DC) with exclusive labels. Thus, a *LDCDG* that starts with a DC p will be compressed to a tree T by using the following five steps.

step 1 Initialize T with p.

step 2 If p has a *DD* on a DC q for the first time, let q_d be a child node of T.

step 3 If p also has a *CD* on a DC q with a method f for the first time, and f holds a direct *DD* and *CD* set S with other vertices, let q_c be another child node of p, every element of S be the child node of q.

step 4 If p has a *DD*/*CD* on a SC r with u, where u is a data type or method with a real name, let u a child node of p.

step 5 Repeat Steps 2 through 4 until all the *DD* and *CD* of p are traversed.

For example, Figure 3 shows a schematic transformation from a *LDCDG* to an unordered labeled tree that only the vertex y is discarded. Our experiment in Sect.4 has showed that the impact on the veracity loss for the theft detection is negligible.

Furthermore, the matching computation for g_1 and g_2 is transformed into unordered tree matching. Many algorithms have been proposed for unordered tree matching. However, we consider that the breadth-first traversal algorithm is suitable for our small-scale trees.

4. Performance Evaluation

A good birthmarking technique should show low similarity between independently written programs, as well as indicate high similarity between same source programs. More precisely, the quality of it depends crucially on two properties introduced by Myles[3]. We just repeat them as follows:

Property 1(Credibility): Let A and B be independently written programs which accomplish the same task. Then we say f is a credible measure if $f(A) \neq f(A')$.

Property 2(Reliability): Let A' be a program obtained from A by applying semantics-preserving transformation T. Then we say f is resilient to T if $f(A) = f(A')$.

Apart from the two properties, the yardstick to decide whether a software theft has taken place or not must be concerned as well. For this problem, Myles (2006) assumed that a software theft takes place between two programs A and B with birthmarks S_A and S_B and a critical value ϵ, only the following two conditions are satisfied:

- A and B with the same external behavior
- $1-S(S_A, S_B) < \epsilon$

In our experiment, just as Myles let $\epsilon = 0.2$, we also set it to 0.2. However, smaller values are desirable but may lead to more false judgments.

4.1. Credibility

To evaluate the credibility of the DGB birthmark, we employed four pairs of large applied programs in our experiment and each pair of them with similar functions. The reason for choosing such programs is that an eligible birthmark not only can distinguish different programs, in addition, it should be applied in practice. Therefore, the programs in our experiment all are famous software in .NET like the *CodeSmith* (a famous code generator)[16]. Moreover, to ensure the experimental results are fully convincing, we thoroughly executed each pair of programs with the varied and thousands of same inputs. Table 1 shows comparison results. From the table, for the MonoDevelop and CSharpStudio, their birthmarks show the lowest similarity (only 0.01). The highest similarity between two distinct programs NLucene and DotLucene is 0.29. Subsequently, however, we have discovered that NLucene is just the .NET implementation of Lucene which is a full-text search engine written in Java. At the same time, DotLucene is based on the Lucene.NET, which was also a .NET extended edition of the Lucene. This implies that our DGB birthmark can effectively distinguish the origin of a program even though they offer the similar functionality.

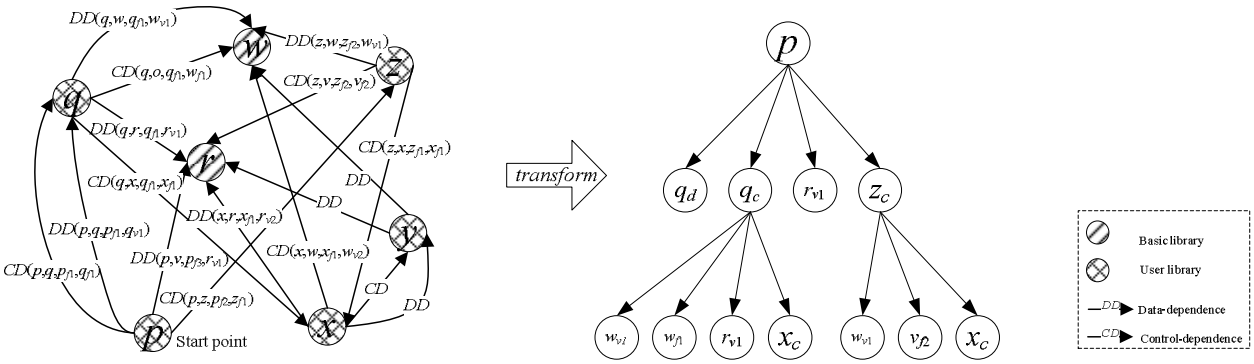

Figure 3. An illustration of the procedure involved in predigesting a *LDCDG* to an unordered labeled tree

Table 1. Similarity percentage found using CDG birthmark in each pair of similar programs

Function	Subject I	Subject II	Similarity
IDE for C#	MonoDevelop	C# Studio	0.01
Search Engine	NLucene	DotLucene	0.29
Unit Testing	dotunit	NUnit	0.09
Code Generator	CodeSmith	Codematic	0.16

4.2. Reliability

To evaluate the resistance ability of the DGB birthmark against semantics-preserving transformations, we conducted a study with two commercial obfuscators. Xenocode is a famous commercial code obfuscation tool for .NET programs. Its control flow obfuscation converts compiled IL code into "spaghetti code" by inserting decoy branches and re-ordering instructions, confusing hackers and crashing decompilers[11]. Dotfuscator is another commercial obfuscator, and the manufacturer of it has been the sole supplier of obfuscation technology for Microsoft[12]. Hence, they are strongly enough to handle all our subjects.

For our study, first, we obfuscated each the above mentioned program to two different versions by using Xenocode and Dotfuscator. Then, for each original and its two obfuscated versions, we orderly executed them with same inputs over thousands times and each time with different input. Results showed that the similarity of *LDCDG*s extracted from each pair of original and obfuscated version was 100%. Hereby, it indicates that our DGB birthmark holds highly resilience to semantics-preserving transformations.

In addition, the WPP birthmark, which collects compact code control flow graphs of a program and regards them as birthmarks, is similar to our CDG birthmark. Thus, we also implemented the same evaluation of it to testify which birthmark holds a better performance to code obfuscation. The experimental results show that the WPP birthmark can hardly tell apart any an original from obfuscated versions. Moreover, the highest similarity measured for the original NLucene and an obfuscated version was only 0.05, whereas our CDG birthmark always yields 1.0. Therefore, our CDG birthmark performs more resilience over the WPP birthmark.

5. Conclusions and Future Work

In this paper, we expand on the idea of a combination of static and dynamic birthmarking technique, which is based on component dependence graph. The credibility and reliability experiments against semantics-preserving transformations have been evaluated. We have shown in the first evaluation that the dynamic dependence graph set is highly characteristic for large programs. The birthmark therefore holds appropriate credibility to distinguish independently written programs. Unlike prior works, the CDG birthmark does not operate at the code level. Instead, it works at module level. It is thus much harder to foil by code obfuscation. In particular, the second experiment has demonstrated that it is more efficient to immune to state-of-the-art obfuscators than the WPP birthmark.

However, since the CDG birthmark works at altitudinal abstract module-level, we know that some independently written small programs with only one or two components may show very similar birthmark. Besides, it may be weak to detect library theft. For example, only a part of important components of the original program are stolen (e.g., modules) to use in other products. In that case, the similarity of the original and the suspected program may be a low value. Nevertheless, our work in this paper is only preliminary and our future work will concentrate on the improved detection ability for small programs and library theft. In addition, we will conduct a more

extensive evaluation of the CDG birthmark with more combinations of obfuscations.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No. 60573045; the National Research Foundation for the Doctoral Program of Higher Education of China No.20050532007.

6. References

[1] G. Myles and C. Collberg, "k-gram based software birthmarks", In Proceedings of SAC, 2005.

[2] H. Tamada, M. Nakamura, A. Monden and K. Matsumoto, "Detecting the theft of programs using birthmarks", Information Science Technical Report NAIST-IS-TR2003014 ISSN 0919-9527, Graduate School of Information Science, Nara Institute of Science and Technology, Nov 2003.

[3] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks", In Information Security, 7th International Conference, 2004.

[4] D. Schuler, V. Dallmeier and C. n. Lindig, "A Dynamic Birthmark for Java", ASE'07, November 49, 2007.

[5] James R. Larus, "Whole program paths", In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 99), 1999.

[6] M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman and Company, 1979.

[7] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp, "Dynamic path-based software watermarking", In ACM SIGPLAN Conference on Programming Language Design and Implementation, 2004.

[8] J. Nagra, C. Thomborson, "Threading software watermarks", In 6th International Information Hiding Workshop, 2004.

[9] D. Curran, M. O. Cinneide, N.J. Hurley, and G.C.M. Silvestre, "Dependency in software watermarking", In First International Conference on Information and Communication Technologies: from Theory to Applications, 2004, pp.311-324.

[10] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multi-linguistic-based code clone detection system for large scale source code", IEEE trans. on Software Engineering, 2002, pp.654-670.

[11] http://www.xenocode.com/

[12] http://www.preemptive.com/products/dotfuscator/

[13] L. P. Cordella, P. Foggia, C. Sansone and M. Vento, "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs", IEEE Transactions on Pattern Analysis and Machine Intelligence, October 2004, p.1367-1372.

[14] http://www.aisto.com/roeder/dotnet/

[15]http://msdn.microsoft.com/msdnmag/issues/03/09/NETProfilingAPI/

[16] http://www.codesmithtools.com/downloadrequest.asp