2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing

DKISB: Dynamic Key Instruction Sequence Birthmark for Software Plagiarism Detection

Zhenzhou Tian, Qinghua Zheng, Ting Liu, Ming Fan
MOEKLINNS Lab
Department of Computer Science and Technology
Xi'an Jiaotong University, 710049, China
E-mail: zztian@sei.xjtu.edu.cn, qhzheng@mail.xjtu.edu.cn, tingliu@mail.xjtu.edu.cn, fanming.911025@mail.xjtu.edu.cn

***Abstract*—With the burst of open source software, software plagiarism has been a serious threat to the healthy development of software industry. Software birthmark reflecting intrinsic properties of software, is an effective way for the detection of software theft. However, most of the existing software birthmarks face a series of challenges: (1) the absence of source code; (2) diversity of operating systems and programing languages; (3) various automated code obfuscation techniques. In this paper, a dynamic key instruction sequence based software birthmark (DKISB) is proposed. By introducing dynamic data flow analysis into birthmark generation, we are able to produce a high quality birthmark that is closely correlated to program semantics, making it resilient to various kinds of semantic-preserving code obfuscation techniques. Based on the Pin instrumentation framework, a DKISB based software plagiarism detection system is implemented, which generates birthmarks for both the plaintiff and defendant program, and then make the plagiarism decision according to the similarity of their birthmarks. The experimental results show that DKISB is effective to either weak obfuscation techniques like compiler optimization or strong obfuscation techniques provided by tools such as SandMark.**

Keywords*—*software plagiarism; dynamic key instruction sequence; software birthmark; similarity comparison;

I. INTRODUCTION

Free or open source software projects allow users to use, change and distribute software under certain types of license. For example the most popular GPL (GNU General Public License) allows users to modify GPL compliance programs freely, and requires derivative works must also be under the GPL. However, some companies or persons incorporate third party software into their products without respecting their licensing terms for commercial interests. Also there are many companies, especially large companies who usually integrate into their projects software components submitted in binary form by upstream companies, and thus cannot assure these components are free of license violations. And thus, cases about software license violations are brought to court from time to time. For example a former Goldman Sachs programmer was found guilty of code theft [1] and the various disputes between Apple and Samsung [2]. Additionally due to their weak code protection awareness of most software developers and the appearance of various powerful automated code obfuscation tools, making software theft a much easier to implement but difficult to detect thing. Besides, since most of the time software is distributed in binary form especially for commodity software, the detection of plagiarism becomes even harder due to the absence of source code on which level is otherwise easier to detect with mature techniques and tools like [16].

So a series of methods are proposed to prevent and detect software plagiarism, and software watermarking is one of the most well-known and earliest approaches. By embedding a unique identifier (watermark) which is hard to remove but easy to verify in the protected software before its distribution, it can serve as strong evidence when filing a lawsuit related to intellectual property. However it is believed by Collberg et al. that "a sufficiently determined attacker will eventually be able to defeat any watermark" [6]. Besides many software developers prefer to use semantic-preserving code obfuscations to make their source code obscure and difficult to reverse rather than utilizing watermarking which requires inserting additional data into the original code. Yet code obfuscations can just prevent others from understanding the underlying logic of the source code but does not hinder direct copying of them. Even worse, plagiarists can in turn further obfuscate the source code and distribute it in binary form to evade detection.

As such, a relatively new software theft detection technique called software birthmark is proposed recently. A birthmark is a characteristic that reflects intrinsic properties of a program, and can be used to uniquely identify the program. In the literature, software theft problem is translated into the problem of comparing the similarity of two programs whose similarity is further measured based on their birthmarks. The key techniques include extraction of high quality birthmark which really can represent the inner property of program and proper similarity comparison methods corresponding to the birthmark. Although the existing birthmark based techniques help to detect software plagiarism to some extent, they are limited in that: (1) many of the methods [16] require the existence of source code which may never be available until strong evidences are collected while generally suspicious plagiarized programs present themselves in the form of binary executables; (2) applicability of these methods is limited to specific type of operating systems or programing languages such as the API based birthmarks [11, 7] rely on features of Java or Windows system, thus failing to detect software theft making using of the platform dependent shortcomings; (3) most of them are weak to code obfuscation techniques implemented in various automated semantic-preserving obfuscation tools.

In this paper, a new kind of dynamic software birthmark called DKISB is introduced to address the above limitations. Based on the key instruction sequence captured during the execution of a program under certain input, we firstly generate the DKISBs for both the plaintiff and defendant program making use of k-gram algorithm, and then calculate the similarity between their birthmarks to decide whether they're plagiarized or not. By combing with dynamic data flow analysis when generating DKISB, our birthmark is closely correlated to the semantics of the program, making it more resilient to semantic-preserving obfuscation techniques; Besides, DKISB operates on binary executables directly rather than source code; Furthermore the bottom object to be analyzed is each assembly instruction, thus freeing our birthmark from operating system or programming language dependent limitations. On the basis of the famous Pin instrumentation framework, we further implemented a DKISB based software plagiarism detection system. Finally, we evaluated the quality of DKISB using programs of different kinds and versions varying from compression software to image processing software; and using various obfuscation techniques including weak obfuscations provided by different compilers and optimization levels, and strong obfuscations provided by special obfuscators. Our experimental results show that the system is able to identify all 34 obfuscated versions (including 2 deeply obfuscated versions using multiple obfuscators) generated with the SandMark [9] tool. This indicates that DKISB is robust against semantic-preserving code transformations.

The contributions of this paper are summarized as follows:

- We proposed a new kind of dynamic software birthmark called DKISB which can be used for software plagiarism detection.
- By combing with dynamic data flow analysis when generating DKISB, our birthmark is closely correlated to the semantics of the program, making it more resilient to semantic-preserving obfuscation techniques.
- Based on the Pin instrumentation framework, we implemented a DKISB based software theft detection system, and evaluated the performance of the system on various kinds and versions of programs and obfuscation techniques.

The remainder of this paper is organized as follows: Section 2 reviews related work in software plagiarism detection literature and program characterization. Section 3 introduces the main idea of DKISB based software theft detection method, including the specific definition and extraction method of DKISB, and the means to compare similarity and make final plagiarism decision. The design overview of the DKISB based theft detection system on the basis of Pin is also presented in this part. The quality of DKISB is evaluated in Section 4 through plenty of experiments. Finally conclusions are drawn in Section 5.

II. Related Work

There are similar research areas that are related to our work in that they all characterize software to identify it uniquely, including software watermarking, plagiarism detection, clone detection, malware identification and so on.

Software watermarking [10] is one of the earliest ways to protect and detect software theft. By embedding a unique identifier (watermark) which is hard to remove but easy to verify in the protected software before its distribution, it can serve as strong evidence when filing a lawsuit related to intellectual property. Different from birthmark based method, additional codes need to be added to the program; besides, watermark can contain program owner information while birthmark only reflects similarity between two programs. Nagra et al. classified watermarks into four types according to their functionality: Authorship Mark, Fingerprinting Mark, Validation Mark and Licensing Mark.

Comparing to watermarking techniques, software birthmark is a relatively new technique for software theft detection. We'll group them into two categories: static and dynamic.

Static source code based birthmarks: Tamada [13] et al. proposed four types of static birthmarks consisting of constant values in field variables, sequence of method calls, inheritance structure and used classes. The detection result depends on the average similarity from the four birthmarks. However their birthmarks are vulnerable to obfuscations and are only available to Java programs. In [16] a source code theft detection system is implemented by mining program dependency graphs (PDGs) and by calculating similarity between PDGs through subgraph isomorphism algorithms.

Static binary code based birthmarks: Myles and Collberg [17] proposed a k-gram based static birthmark for Java. Set of Java bytecode sequences of length k are taken as the birthmark, and similarity between birthmarks are calculated through set operations while ignoring frequency of each element. They compared their birthmark with Tamada's using several tiny java programs and shows better robustness, but is still vulnerable to code transformation attacks. A static API birthmark is put forward by Seokwoo Choi et al. [19] to detect plagiarism of windows applications. They firstly define a function birthmark as a set of API calls within k depths of the call tree rooted at the function. Then program similarity is defined as the maximum value among all possible function matchings. Thus transforms the problem of calculating the similarity between two programs as finding a maximum weighted bipartite matching. They evaluated their birthmark with several obfuscated versions generated with different compilers, and it shows pretty well resilience. However we've no idea of its robustness against special obfuscators.

Dynamic software birthmarks: Myles and Collberg [20] suggested a dynamic birthmark based on whole program path generated by compressing a whole dynamic control flow trace into WPP form to uniquely identify program. Schuler [11] takes Java standard API call sequences at object level as a dynamic birthmark for java programs. And it exhibits better performance than WPP birthmark. Tamada [21] introduced two API based birthmarks for windows executables extracted at runtime: Sequence of API Function Calls (EXESEQ) and Frequency of API Function Calls (EXEFREQ). However these methods are all language dependent no matter they are windows API based or java API based. So Wang et al. [7] proposed two dynamic birthmarks based on system calls: System Call Short Sequence birthmark (SCSSB) and Input Dependent System Call Subsequence birthmark (IDSCSB). However both birthmarks have limited applicability to software that has few system calls such as computation-centered programs. Later in [12], by introducing data flow and control flow dependency analysis, they proposed a system call dependency graph based birthmark (SCDG). Recently [5] they suggested to characterize software with core values and applied it to software theft detection and algorithm plagiarism detection.

There also exist several birthmarks based on dynamic instructions. In [18] a whole dynamic instruction trace is recorded during program execution from which a dynamic birthmark is directly extracted applying the k-gram algorithm. But this birthmark cannot even identify two versions generated from the same program with different compiler optimization levels. By treating the dynamic slices generated with dynamic slicing techniques rather than the whole instruction sequence as program characterization, Bai et al. [14] proposed a dynamic birthmark for java based on MSIL instructions rather than assembly instructions. Their birthmark is also compared with Myles's k-gram birthmark through a single small program and exhibits better robustness. Our DKISB differs from their's in that it's operating-system and language free, and it combines dynamic data flow analysis to make it more robust against various semantic-preserving code transformations.

One of the most close research field to software plagiarism detection is clone detection that aims to find duplicate code (or to say clones) within a single program to help improve software maintenance, program comprehension, and software quality. Most existing clone detection algorithms operate on source code only. And there have been many mature systems [15, 3, 4] which implement efficient and accurate clone analysis on large scale software. Similar to theft detection, clone detection identifies cloned fragments by firstly translating the program into a set of characteristics based on which clone detection techniques can be categorized into the following types: String-based, Token-based, AST-based, PDG-based, and Memory-State-based.

III. DKISB BASED SOFTWARE PLAGIARISM DETECTION

In this section, several important concepts and definitions are introduced first, followed by description of our DKISB based software plagiarism detection method.

A. Dynamic Key Instruction Sequence Based Birthmark

Before introducing the main idea of DKISB, definition of software birthmark and dynamic software birthmark which we borrowed from Tamada et al. [21] and Myles et al. [20] are presented first to ease further discussion. They are the first formal definitions and have been restated in most subsequent papers in the literature.

1) Software Birthmark.

A software birthmark is a set of characteristics extracted from a program that reflects intrinsic properties of the program and can be used to identify the program uniquely. It can be categorized into two types: static birthmark and dynamic birthmark, where the former is generated mainly by analyzing syntactic features of a program and thus is weak to sematic-preserving obfuscations while the latter is extracted based on runtime behavior that reflects how inputs are processed by a program and thus is more correlated to program semantics and robust against obfuscations. Now we give their definitions.

Definition 1. (Software Birthmark) Let *p,q* be two programs or components. Let *f* be a method for extracting a set of characteristics from the programs or components. We say *f(p)* is a birthmark of *p* if both of the following conditions are satisfied:

- *f(p)* is obtained only from *p* itself.
- Program *q* is a copy of $p \Rightarrow f(p) = f(q)$.

Definition 2. (Dynamic Software Birthmark) Let *p,q* be two programs or program components. Let *I* be an input to *p* and *q*. Let *f(p,I)* be s set of characteristics extracted from *p* when executing *p* with input *I*. Then *f(p,I)* is a dynamic birthmark of *p* only if both of the following conditions are satisfied:

- *f(p,I)* is obtained only from *p* itself when executing *p* with input *I*.
- Program *q* is a copy of $p \Rightarrow f(p,I) = f(q,I)$.

2) DKISB.

Our birthmark is based on dynamic instruction sequence and so it belongs to the dynamic birthmark category. It's believed that a high quality birthmark should be closely related to the semantics of a program. Computer state get updated with the execution of each single instruction, so instruction sequence recorded during program execution is a reflection of how inputs are processed by the program and is closely related to program's semantics, thus making it a good birthmark candidate. However taking the whole sequence as a birthmark is too large or even impossible for further analyze. So we propose the concept of dynamic key instruction based on which can greatly reduce the size of instruction sequence from which DKISB is extracted further.

So what is required for an instruction to make itself a key instruction? Firstly we believe that they should be relatively unique, so instructions of specific types (for example mov, push, etc.) which exist widely in most programs and which constitute a large part of the dynamic instruction trace are just noises which do not represent unique behavior of a program, and so should be eliminated confidently. Besides they should be closely related to program semantics, so comparing to instructions like mov etc. whose functionality is just to transfer data between memory or CPU or both while no new values are generated but just get migrated, instructions whose execution will generate new values (such as add, shl, etc.) or so called value-updating instructions as defined in [5] reflect how the program computes, and thus are better reflections of program semantics; At last, semantics is a formal representation of how inputs are processed by the program, so instructions related to inputs are more related to semantics. Based on dynamic taint analysis, we can acquire the correlation between instructions and inputs. And we call instructions whose execution will change the taint labels of registers or memory units as input-correlated instructions.

Based on the above discussions, we now give the definition of key instruction, DKISB etc.

Definition 3. (Dynamic Key Instruction) Let $trace(p,I)$ be an execution trace composed of dynamic instructions executed during program run under input I, then for each instruction c that belongs to $trace(p,I)$, we say c is a key instruction under input I if the following two conditions are satisfied:

— c is a value-updating instruction.
— c is a input-correlated instruction.

Definition 4. (K-Gram) Let $t=\langle e_1,e_2,\cdots,e_n\rangle$ be a sequence of which each element can be a word, a character, an object, or in our case an instruction (specifically, mnemonic of the instruction). Define a sliding window of length k, and generate a subsequence $sub(t)_j=(e_j,e_{j+1},\cdots,e_{j+k-1})$, $j\in\{1,2,\cdots,n-k+1\}$ by sliding the window over t with stride one each time. Then we refer to $sub(t)_j$ as a k-gram.

Definition 5. (DKISB) Let $s(p,I)=\langle ins_1,ins_2,\cdots,ins_n\rangle$ be a key instruction sequence recorded during runtime of program p under input I. Let $t(p,I)=\langle e_1,e_2,\cdots,e_n\rangle$ be the mnemonic sequence by extracting mnemonic of each instruction in $s(p,I)$. Let $Set(p,I)=\{g_j \mid g_j=(e_j,e_{j+1},\cdots,e_{j+k-1})\}$, $j\in\{1,2,\cdots,n-k+1\}$ be a set of k-grams. Then we call the set of key-value pairs $Birth_p^I(k)=\{\langle g'_m, freq(g'_m)\rangle \mid g'_m\in Set(p,I)\, and\, \forall m_1\neq m_2, g'_{m_1}\neq g'_{m_2}\}$ where $freq(g'_m)$ represents frequency of g'_m in $Set(p,I)$ as the dynamic key instruction sequence based birthmark for p under input I, briefly called DKISB.

B. DKISB Based Software Plagiarism Detection

1) Similarity Calculation.

In the literature of birthmark based plagiarism detection, the similarity between two programs is measured by the similarity of their birthmarks. According to the manifestations of birthmarks, different methods should be chosen to properly calculate the degree of similarity. Generally speaking, birthmarks mainly exist in three forms: sequences, sets and graphs. Similarity of sequences can be computed with pattern matching methods, such as calculating the longest common subsequences (LCS) [7, 5] and so on. There are many methods to calculate similarity for sets that are widely adopted in the field of information retrieval, for example the Dice coefficient [19], the Jaccard index [11] etc. Computing the similarity of graphs is relatively complex where graph or subgraph isomorphism algorithm [16, 12] such as the VF graph matching algorithm can be used.

Our birthmark is a set composed of key-value pairs, the type of keys may not be so rich as other static k-gram birthmarks [17, 18] or dynamic birthmarks extracted from the whole instruction sequences [18, 14] according to the definition of DKISB. So the computed similarity values have a high probability to be the close with each other if we adopt calculation methods such as Jaccard index, Dice coefficient etc. that ignore frequencies of elements in the set. Besides, the frequency of each element which is not available to static methods also reflects how inputs are processed, and should be an important part of the birthmark. So we make use of the Cosine distance to measure the similarity of two birthmarks. The formal description is as follows:

For software birthmarks $A=\{\langle k_1,v_1\rangle,\langle k_2,v_2\rangle,\cdots,\langle k_n,v_n\rangle\}$ and $B=\{\langle k'_1,v'_1\rangle,\langle k'_2,v'_2\rangle,\cdots,\langle k'_m,v'_m\rangle\}$, let $S=keySet(A)\cup keySet(B)$. Then construct a vector $\vec{A}=(a_1,a_2,\cdots,a_l)$ of which each element $a_i=\begin{cases}v_i, if\ S_i\in keySet(A)\\ 0, if\ S_i\notin keySet(A)\end{cases}$, where $1\le i\le l$ and v_i is the value of key S_i in A. Likewise $\vec{B}=(b_1,b_2,\cdots,b_l)$ can be constructed, and then similarity of two birthmarks A and B can be calculated with $sim(A,B)=\dfrac{\vec{A}\bullet\vec{B}}{|\vec{A}||\vec{B}|}$.

2) *Plagiarism Detection.*

The purpose of extracting birthmarks and calculating their similarity is to eventually make a decision of whether two programs are plagiarism. Since our DKISB belongs to dynamic birthmark category, so multiple similarity scores are calculated by providing multiple inputs to exclude the influence of random factors, and average of the scores is taken as evidence to make the final decision.

Formally, Let P_A and P_B be two programs to be analyzed. The DKISBs extracted from each of them by providing a series of inputs $I_1, I_2, \cdots, I_n$ are $A_1, A_2, \cdots, A_n$ and $B_1, B_2, \cdots, B_n$. Then the similarity between program P_A and P_B can be calculated with $sim(P_A, P_B) = \frac{\sum_{j=1}^{n} sim(A_j, B_j)}{n}$ whose value is between 0 and 1. Then we determine whether two programs are copies according to their similarity score and a threshold ε as follows:

$$sim(P_A, P_B) = \begin{cases} \geq 1-\varepsilon & P_A, P_B \text{ are classified as copies} \\ \leq \varepsilon & P_A, P_B \text{ are classified as independent} \\ \text{otherwise} & \text{inconclusive} \end{cases}$$

In our plagiarism detection system, we choose the value of ε to be 0.2 as adopted by Schuler et al. [11], which means: two programs whose similarity score is in the range of [0, 0.2] are classified as independent, (0.2, 0.8) as inconclusive, and [0.8, 1] as copies. A smaller threshold value is desired but it may lead to many false classifications.

C. *System Design*

Fig.1 shows the overview of our dynamic birthmark system. The plaintiff binary represents the original program owned by its developer while the defendant binary represents suspicious program that may have plagiarized the plaintiff. The system comprise five modules: dynamic analysis module where key instructions are recognized and recorded, pre-processor that aims to extract mnemonics and peel off operands, birthmark generator where DKISB is generated, similarity calculator where the similarity score of two DKISBs are computed, and decision maker that outputs final detection result.

Given two programs plaintiff and defendant and a series of inputs, our system runs the two programs with the same input one by one. Meantime, the dynamic analysis module monitors the execution of each program in a fine granularity of instruction level, it identifies and records key instructions in real time and finally outputs a key instruction sequence. After the two sequences of both plaintiff and defendant program are ready, they are feed into the pre-processor module to remove operands of each instruction. Then birthmark generator will generate two DKISBs whose similarity score is further calculated by the similarity calculator. Finally detection result of whether is plagiarism or not is made by the decision maker according to the similarity scores computed under different inputs and the given threshold ε.

Of all the modules, dynamic analysis module is one of the most important part of the whole system that is in charge of the monitoring of a program, and performs dynamic taint analysis to identify and record key instructions. It consists of Pin [8] and DKISExtractor, where the former is a well-known dynamic instrumentation framework provided by Intel for its rich API and high efficiency, and the latter is a plugin distributed as a pintool that we developed based on libdft which is a data flow analysis framework implemented on the basis of Pin. DKISExtractor and Pin work together to identify key instructions and generate a key instruction sequence per run.

IV. Evaluation

A high quality birthmark manifests in that the ratio of false classifications (both inconclusive and incorrectly classified are treated as false classifications) should be rather low for a given ε. Specifically, the similarity scores between a program and its derivation versions generated by applying various semantic-preserving transformation techniques should be high enough so as to recognize copies, while scores between independently developed programs should be low enough to distinguish them. Generally, the following two properties of a birthmark should be evaluated in the literature. We restate them by referring to descriptions of Myles et al. [17] and Seokwoo et al. [19].

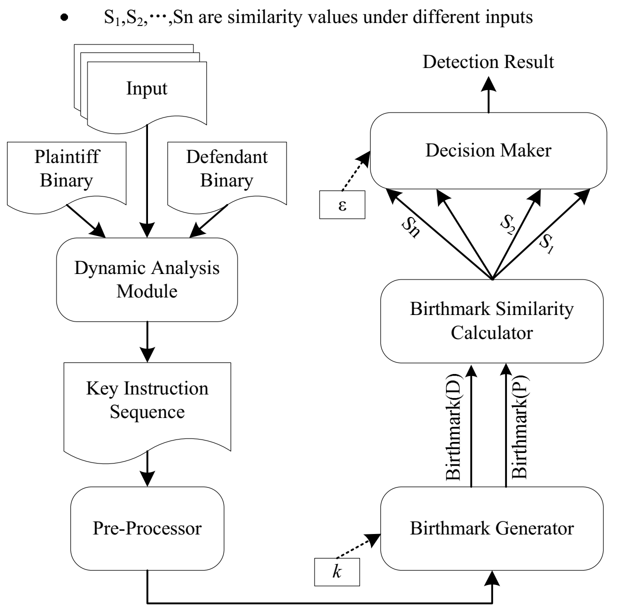

Fig. 1 DKISB based software plagiarism detection system

Property 1. (Resilience) Let p be a program and p' be a derivative version generated by applying semantic-preserving code transformations τ to p. Then we say a birthmark B_p is resilient to τ if $sim(p,p') \geq 1-\varepsilon$.

Property 2. (Credibility) Let p and q be independently developed programs which may accomplish the same task. Then we say a birthmark B_p is credible if $sim(p,q) \leq \varepsilon$.

It should be noticed that the extracted DKISB is different even for the same instruction sequence each time we choose a different value for the parameter k of k-gram algorithm, thus causing the similarity scores to change with k. Hence it is necessary to study whether there exists a k value that makes the similarity scores between plagiarized programs are high while the similarity scores between independently implemented programs are low. So we'll firstly study the impact of the value of k to the similarity calculation between programs, then the quality of DKISB will be evaluated against the two properties mentioned above with a fixed k value.

A. Impact of Parameter K

1) Impact of k to similarity scores between software of different categories.

We selected four image processing software: sixiv, feh, pho, and qiv which are widely used in Linux system to compare their similarity with programs [1] of other categories including compression, encryption etc. under different k values. Firstly key instruction sequences were extracted for all of them, followed by the generation of DKISBs by varying the value of parameter k. Then cosine distance was computed for each pair of the birthmarks to measure similarity. Determining the similarity based on a single input may not be credible, so multiple and various types of inputs [2] (including jpg, png, bmp, gif etc.) were provided, and average score of similarity was calculated.

The blue area in TABLE I illustrates how similarity changes between qiv and other types of programs by varying k. Plagiarism does not exist between image processing software and others, which means the similarity scores should be rather low to make the birthmark effective. As we can see, the similarity scores between qiv and programs of other categories are rather high when DKISBs are generated with 1-gram, and the scores decrease sharply by increasing the k value, then remain almost unchanged when a certain k value (here we say 4 or 5) is reached. There are similar results for the other three image processing software, they're not listed in the table but contribute to the calculation of average scores (shown in the dark grey areas) due to space limitations.

[1] The benchmark consists of representative programs widely used in previous papers and several newly added experimental objects.

[2] All the experiments conducted below will be feed with multiple inputs by default.

2) Impact of k to similarity scores between software of same categories.

The functionalities for software of same categories overlap to a great extent, for example bzip and gzip are two very popular and widely used software that both implement compression and decompression. So it's necessary to study how similarity scores change with k between programs that are in the same category but are independently implemented. As illustrated in the green area and corresponding dark grey areas of TABLE I, similarity scores decrease sharply until k increases to a certain value (say 4 or 5 here). It shows that birthmarks generated with k value of four or five are enough to recognize independently developed programs.

3) Impact of k to similarity scores between plagiarized software.

Here we treat binaries compiled from the same source code but with different compilers or optimization levels, and the ones generated with semantic-preserving obfuscation techniques as copies, which means the similarity between them should be rather high. It can be observed from the orange area and corresponding dark grey areas of TALBE I that, similarity scores between plagiarized program pairs are indeed very high and present a slightly rather than sharply (as illustrated in the last two experiments) decreasing trend as k increases, which also reflects the robustness of DKISB in a way.

Based on the above observations, we can conclude that similarity scores calculated based on DKISB decrease as the value of k increases, and then remain unchanged when a certain k value is reached between programs no matter plagiarism exists or not. Moreover, we can see that similarity scores between independently implemented programs have been low enough to distinguish them when 4 or 5 is adopted for the value of k, meanwhile scores between plagiarized program pairs are also enough to identify copies. Besides, a larger k doesn't seem to offer more benefit but just introduces more computation efforts. In the reset of the paper, all experiments conducted adopt a default k value of 4 which provides a good tradeoff between efficiency and accuracy.

B. Resilience

1) Resilience to different compilers and compiler optimization levels.

Software plagiarist may try to evade detection by choosing a different compiler or changing compiler optimization levels which can be seemed as a kind of weak semantic-preserving code transformations. Here, two compression software: gzip-1.2.4 and bzip2-1.0.6 are chosen to evaluate the resilience property of DKISB against different compilers and optimization levels. In our experiments, three versions (4.4, 4.5, and 4.6) of gcc compiler are selected and multiple optimization levels (O1-O3) are adopted to compile each of the programs. Then the generated executables are executed against our detection system where DKISBs are generated and similarity scores are calculated.

TABLE I. Impact of parameter k

Category	Name	K=1	K=2	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10
Similairty of Software in Different Categories	qiv--VS--bzip2	0.496	0.128	0.056	0.027	0.006	0.005	0.004	0.004	0.004	0.004
	qiv--VS--cksum	0.469	0.033	0.013	0.01	0.008	0.006	0.006	0.006	0.006	0.006
	qiv--VS--gzip	0.409	0.001	0	0	0	0	0	0	0	0
	qiv--VS--md5sum	0.748	0.4	0.058	0.038	0.018	0.017	0.016	0.016	0.016	0.016
	qiv--VS--opensslMD4	0.662	0.32	0.153	0.136	0.135	0.134	0.132	0.131	0.129	0.127
	qiv--VS--opensslMD5	0.72	0.433	0.124	0.124	0.12	0.118	0.116	0.114	0.111	0.109
	qiv--VS--opensslRMD160	0.668	0.281	0.1	0.075	0.073	0.071	0.07	0.068	0.066	0.063
	qiv--VS--opensslSHA1	0.564	0.251	0.202	0.205	0.204	0.202	0.201	0.199	0.197	0.194
Average Similarity Score of Software Listed above		**0.592**	**0.231**	**0.088**	**0.077**	**0.071**	**0.069**	**0.068**	**0.067**	**0.066**	**0.065**
Average Score of All Software in Different Categories Tested		**0.472**	**0.162**	**0.063**	**0.053**	**0.049**	**0.048**	**0.047**	**0.047**	**0.046**	**0.045**
Similarity of Software in the Same Category	bzip-VS-gzip	0.496	0.009	0.006	0.004	0.004	0.004	0.004	0.003	0.003	0.003
	bzip--VS--zip	0.544	0.01	0.005	0.004	0.004	0.003	0.003	0.003	0.003	0.003
	cksum--VS--md5sum	0.592	0.084	0	0	0	0	0	0	0	0
	opensslMD5--VS--opensslRMD160	0.837	0.569	0.302	0.166	0.091	0.063	0.051	0.038	0.024	0.012
	opensslMD5--VS--opensslSHA	0.963	0.645	0.409	0.196	0.097	0.052	0.033	0.026	0.026	0.026
	opensslMD5--VS--opensslMD4	0.955	0.745	0.488	0.312	0.178	0.15	0.126	0.099	0.086	0.071
	opensslMD5--VS--cksum	0.683	0.121	0.004	0.002	0.001	0	0	0	0	0
	pho--VS--feh	0.653	0.402	0.306	0.301	0.298	0.296	0.293	0.292	0.29	0.29
Average Similarity Score of Software Listed above		**0.507**	**0.18**	**0.090**	**0.069**	**0.058**	**0.054**	**0.053**	**0.050**	**0.049**	**0.048**
Average Score of All Software in the Same Category Tested		**0.741**	**0.372**	**0.186**	**0.115**	**0.084**	**0.074**	**0.07**	**0.066**	**0.064**	**0.062**
Similarity Between Plagiarized Software	Jlex--VS--Jlex_ClassSplitter	1	1	1	1	1	1	0.999	0.999	0.999	0.999
	Jlex--VS--Jlex_SplitClasses	1	1	1	1	1	1	0.999	0.999	0.999	0.999
	bzip(gcc44_o0--VS--gcc45_o0)	1	1	1	1	1	1	1	1	1	1
	gzip(gcc46_o0--VS--gcc44_o0)	1	1	1	1	1	1	1	1	1	1
	bzip(gcc44_o2--VS--gcc44_o1)	0.966	0.958	0.946	0.946	0.946	0.946	0.946	0.946	0.946	0.945
	bzip(gcc44_o3--VS--gcc44_o1)	0.969	0.962	0.951	0.951	0.951	0.951	0.951	0.951	0.95	0.95
	gzip(gcc44_o1--VS--gcc44_o3)	1	1	1	1	1	1	1	1	1	1
Average Similarity Score of Software Listed above		**0.991**	**0.989**	**0.985**	**0.985**	**0.985**	**0.985**	**0.985**	**0.985**	**0.985**	**0.985**
Average Score of All Plagiarized Software Tested		**0.986**	**0.983**	**0.978**	**0.978**	**0.978**	**0.978**	**0.977**	**0.977**	**0.977**	**0.977**

It is observed that all the similarity scores are rather high (much higher than the detection bound 0.8), indicating that DKISB is resilient to different compilers and optimization levels. Similar results are also observed among all the binaries of gzip.

2) Resilience to special obfuscation tools.

In this section, we'll evaluate the resiliency of DKISB against advanced obfuscation techniques implemented in special tools. Unfortunately no available binary obfuscators (obfuscators such as Upx, WinUpack etc. that only implement code compression, encryption and packing obfuscations are ignored, since executables processed with them must be decompressed or decrypted during runtime, and our dynamic birthmark has innate immunity to them) are found except the commercial obfuscator CloakWare Security Suite.

So here, we choose to use the Java byte code obfuscation tool SandMark [9], which implements a series of advanced semantic-preserving transformation techniques including 15 application obfuscations, 7 class obfuscations, and 17 method obfuscations to generate a group of obfuscated versions. Since our system works on binary executables, so obfuscated versions of Java byte code are converted to x86 executables first with GCJ, the GNU ahead-of-time Compiler for Java.

JLex, a lexical analyzer generator written in Java, is selected as the experimental subject, which is also used in [7, 5]. We conducted the same experiments as in [5] to measure the resiliency of DKISB against single and multiple obfuscations.

a) Resilience to single obfuscation

Similarity scores are calculated between the original JLex and its obfuscated versions (32 successfully obfuscated versions[3]) generated by applying a single obfuscation technique a time. Besides, we compare JLex to programs that are totally different but can share same inputs to show the differences. The experimental results show that the similarity scores of JLex to its obfuscated ones are all as high as 1.0 (much higher than 0.8) while scores of JLex to totally different programs are all below the threshold 0.2, which means no false classifications exist. It indicates that our DKISB is resilient to single semantic-preserving code transformations.

b) Resilience to multiple obfuscation

A plagiarist may attempt to evade detection by applying multiple obfuscation techniques to a single program so as to generate a deeply obfuscated version. However, applying many obfuscators to a single program could raise practical issues of correctness of the target program and efficiency [7]. So we adopted the method as used in [7, 5] where all obfuscators are classified into two categories: data obfuscators and control obfuscators. Then obfuscators of same category are applied to

[3] Seven obfuscators of SandMark failed to transform JLex, so we can't test them all.

TABLE II. Obfuscators used to generate Jlex_contrl and Jlex_data

Control Obfuscation	Data Obfuscation
Transparent Branch Insertion, Simple Opaque Predicates, Reorder Instructions, Dynamic Inliner, Method Merger, Inliner, Insert Opaque Predicates	Array Folder, Integer Array Splitter, Promote Primitive Registers, Variable Reassigner, Duplicate Registers, Merge Local Integers, Boolean Splitter

JLex one by one, and finally we got two deeply obfuscated versions: JLex_control and JLex_data. The specific obfuscators used are listed in TABLE II while the order applied is random. The similarity scores computed of JLex to JLex_control and JLex_data are correspondingly 0.978 and 1.0, both are much higher than 0.8. This indicates that DKISB is resilient even to rather complex obfuscations.

The results show that DKISB has better performance (higher score between obfuscated versions and lower score between JLex and others) than system call based birthmarks [7], and as good performance as value based birthmark [5].

C. Credibility

1) Similarity between different versions of the same program.

To evaluate the credibility of DKISB, we firstly compare different versions of the same program. Here five different versions of gizp are taken as the subjects, which are all compiled with gcc4.6 and optimization level of O2. Then the similarity between different versions is calculated.

As illustrated in TABLE III, the similarity scores between different versions of gzip are all very high (average score is larger than 0.99 for each version pair). It indicates that during the process of software evolution, code of previous versions are usually shared and reused by new versions, which also means that the new versions will generally inherit most of the features of older versions. This also shows that our DKISB is a good reflection of the intrinsic characteristics of a program.

Besides, we also compared similarities between two different versions of image processing software. Specially, feh2.3 is compared with feh2.9.2, and qiv2.23 is compared with qiv2.2.4. They both illustrated high similarity as close as to 1.0. By viewing the upgrade report of qiv2.2.4, we found that the new version just fixed several bugs submitted in the previous version, which means code is modified slightly and that's why the two versions can have such a high similarity.

2) Similarity between programs of the same category.

In this section, programs in the same categories are compared to prove the credibility of DKISB. Although programs in the same categories usually overlap greatly in their functionalities, they can be rather different in the code level if implemented independently due to different algorithms adopted, different design patterns applied, different mode of thinking and coding habits of developers' etc.

In our experiments, three kinds of software are chosen, including 4 image processing software, 3 compression and decompression software, and 7 encryption and decryption software. Multiple inputs are provided for software of same categories to mitigate the impact of causal factors and average similarity scores are calculated.

The experimental results are shown in the blue area of TABLE IV. We can see that most of the similarity scores are relatively low. For programs that are independently implemented, their similarity scores are as close as to 0, as illustrated by the gzip-bzip pair and the cksum-md5sum pair. The similarities between the series of openssl programs are slightly higher but most are still below 0.2. This is because these programs share the same front end for preprocessing parameters etc. while their kernel modules are implemented differently. We believe that lower scores can be acquired by filtering out the instructions executed in the shared module, and this can be accomplished by specifying where to attach and detach Pin.

Besides, we observe that the similarity scores of gzip-zip pair and md5sum-opensslMD5 pair are both as high as 0.9. According to the documentations of gzip and zip projects, they are both based on the compression algorithm *deflate* which is also implemented in the zLib library. And gzip contains code from zLib while zip is dynamically linked to system-wide zLib, which is as also confirmed in [5]. This explains why they have such a high similarity score, and also convincingly demonstrate the credibility of our DKISB in a way.

In addition, the similarity scores between image processing software are relatively high. This is due to that they share many image processing libraries which can be learned by checking the dependencies of each program with the *apt-get depends* command. For example, the dependencies of pho are totally included in the ones of qiv, causing their similarity score to be more than 0.9. As for qiv (which is implemented on the basis of imlib2 and gtk2) and feh (whose implementation is based simply just on imlib2), only part of the dynamic libraries are shared, so their similarity is relatively low (0.302 here). The consistency between the containment of dependencies among these programs and similarity scores calculated with our birthmarks proved the credibility of DKISB once again.

3) Similarity between programs of different categories.

In this section, we calculated the similarity scores between software in different categories. As the experimental result shown in the green area of TABLE IV, all the scores are below the threshold 0.2, which means no false classifications exist.

TABLE III. Similarity of different versions of gzip

name	gzip1.2.4a	gzip1.2.4	gzip1.3.13	gzip1.4	gzip1.5
gzip1.2.4a	1	1	0.995	0.992	0.997
gzip1.2.4	-	1	0.995	0.992	0.997
gzip1.3.13	-	-	1	0.997	0.992
gzip1.4	-	-	-	1	0.995
gzip1.5	-	-	-	-	1

TABLE IV. Similairty between independently implemented programs

Category		Compression And Decomression			Image Processing				Encryption And Decryption						
		Gzip	Bzip	Zip	Qiv	Feh	Pho	Sxiv	Cksum	Md5sum	OL-MD4	OL-MD5	OL-RMD160	OL-SHA	OL-SHA1
Compression And Decompression	Gzip	1.00	0.004	0.914	0.010	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	Bzip	-	1.000	0.004	0.027	0.007	0.051	0.004	0.063	0.000	0.000	0.000	0.000	0.000	0.001
	Zip	-	-	1.000	0.010	0.013	0.010	0.009	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Image Processing	Qiv	-	-	-	1.000	0.302	0.930	0.562	0.010	0.038	0.136	0.124	0.075	0.104	0.205
	Feh	-	-	-	-	1.000	0.301	0.578	0.011	0.001	0.003	0.006	0.008	0.007	0.006
	Pho	-		-	-	-	1.000	0.530	0.053	0.016	0.128	0.104	0.068	0.097	0.191
	Sxiv	-	-	-	-	-	-	1.000	0.005	0.016	0.122	0.101	0.065	0.093	0.194
Encryption And Decryption	Cksum	-	-	-	-	-	-	-	1.000	0.000	0.004	0.002	0.001	0.001	0.013
	Md5sum	-	-	-	-	-	-	-	-	1.000	0.238	0.927	0.159	0.142	0.010
	OL-MD4	-	-	-	-	-	-	-	-	-	1.000	0.312	0.053	0.174	0.285
	OL-MD5	-	-	-	-	-	-	-	-	-	-	1.000	0.166	0.196	0.150
	OL-RMD160	-	-	-	-	-	-	-	-	-	-	-	1.000	0.077	0.068
	OL-SHA	-	-	-	-	-	-	-	-	-	-	-	-	1.000	0.109
	OL-SHA1	-	-	-	-	-	-	-	-	-	-	-	-	-	1.000

V. CONCLUSION

In this paper, a dynamic birthmark called DKISB is proposed based on the key instruction sequence, to solve some of the limitations in birthmark based software plagiarism detection literature. By introducing dynamic data flow analysis into birthmark generation, we are able to produce a high quality birthmark that is resilient to various kinds of obfuscation techniques. Further, based on the Pin instrumentation framework, a DKISB based software plagiarism detection system is implemented which firstly generate birthmarks for both the plaintiff and defendant program, and then make the plagiarism decision by comparing the similarity of their birthmarks. Finally, plenty of experiments are conducted on various programs, and the results show that the resilience and credibility of DKISB are pretty well.

ACKNOWLEDGMENT

The research was supported in part by National Science Foundation of China under Grant Nos. 91118005, 91218301, 61221063; National High Technology Research and Development Program 863 of China under Grant No. 2012AA011003; Cheung Kong Scholar's Program; Key Projects in the National Science and Technology Pillar Program under Grant No. 2011BAK08B02; Doctoral Fund of Ministry of Education of China (20110201120010) and the Fundamental Research Funds for the Central Universities.

REFERENCES

[1] http://www.fbi.gov/newyork/press-releases/2010/nyfo121010.htm

[2] http://www.tuicool.com/articles/j2INVn

[3] Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code[J]. Software Engineering, IEEE Transactions on. 2002, 28(7): 654-670.

[4] Jiang L, Misherghi G, Su Z, et al. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones[C]. In: ICSE '07.Washington, DC, USA: IEEE Computer Society, 2007. 96-105.

[5] Jhi Y, Wang X, Jia X, et al. Value-based program characterization and its application to software plagiarism detection[C]. In: ICSE '11.New York, NY, USA: ACM, 2011. 756-765.

[6] Collberg C, Carter E, Debray S, et al. Dynamic path-based software watermarking[C]. In: PLDI '04.New York, NY, USA: ACM, 2004.

[7] Wang X, Jhi Y, Zhu S, et al. Detecting Software Theft via System Call Bas ed Birthmarks[C]. In: ACSAC '09.Washington, DC, USA: IEEE Computer Society, 2009. 149-158..

[8] Luk C, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation[C]. In: PLDI '05.New York, NY, USA:

[9] Collberg C, Myles G R, Huntwork A. Sandmark-a tool for software protection research[J]. Security & Privacy, IEEE. 2003, 1(4): 40-49.

[10] Collberg C, Thomborson C. Software watermarking: models and dynamic embeddings[C]. In: POPL '99. NY, USA: ACM, 1999. 311-324.

[11] Schuler D, Dallmeier V, Lindig C. A dynamic birthmark for java[C]. In: ASE '07.New York, NY, USA: ACM, 2007. 274-283.

[12] Wang X, Jhi Y, Zhu S, et al. Behavior based software theft detection[C]. In: CCS '09.New York, NY, USA: ACM, 2009. 280-290.

[13] Tamada H, Nakamura M, et al. Design and evaluation of birthmarks for detecting theft of java programs.[C]. In: IASTED 2004. 569-574.

[14] Bai Y, Sun X, Sun G, et al. Dynamic k-gram based software birthmark[C]. In: ASWEC, 2008. 644-649.

[15] Kim H, Jung Y, Kim S, et al. MeCC: memory comparison-based clone detector[C]. In: ICSE '11.New York, NY, USA: ACM, 2011. 301-310.

[16] Liu C, Chen C, et al. GPLAG: detection of software plagiarism by program dependence graph analysis[C]. In: KDD, 2006. 872-881.

[17] Myles G, Collberg C. K-gram based software birthmarks[C]. In: SAC '05.New York, NY, USA: ACM, 2005. 314-318.

[18] Bin L, Fenlin L, Xin G, et al. A Software Birthmark Based on Dynamic Op code ngram[C]. In: ICSC '07.Irvine CA, United states,2007. 37-44

[19] Choi S, Park H, et al. A static API birthmark for Windows binary executables[J]. Journal of Systems and Software. 2009, 82(5): 862-873.

[20] Myles G, Collberg C. Detecting software theft via whole program path birthmarks[M]. Information security, Springer, 2004, 404-415.

[21] Tamada H, Okamoto K, et al. Dynamic software birthmarks to detect the theft of windows applications[C]. In: ISFST 2004.