Software Plagiarism Detection with Birthmarks based on Dynamic Key Instruction Sequences

Zhenzhou Tian, Qinghua Zheng, *Member, IEEE,* Ting Liu, *Member, IEEE,* Ming Fan, Eryue Zhuang and Zijiang Yang, *Senior Member, IEEE*

Abstract—A software birthmark is a unique characteristic of a program. Thus, comparing the birthmarks between the plaintiff and defendant programs provides an effective approach for software plagiarism detection. However, software birthmark generation faces two main challenges: the absence of source code and various code obfuscation techniques that attempt to hide the characteristics of a program. In this paper, we propose a new type of software birthmark called DYKIS (DYnamic Key Instruction Sequence) that can be extracted from an executable without the need for source code. The plagiarism detection algorithm based on our new birthmarks is resilient to both weak obfuscation techniques such as compiler optimizations and strong obfuscation techniques implemented in tools such as `SandMark`, `Allatori` and `Upx`. We have developed a tool called `DYKIS-PD` (DYKIS Plagiarism Detection tool) and conducted extensive experiments on large number of binary programs.
The tool, the benchmarks and the experimental results are all publicly available.

Index Terms—software plagiarism detection, software birthmark

1 Introduction

OPEN source software allows its usage, modification and redistribution under certain types of licenses. For example, GPL (GNU General Public License) allows users to modify GPL compliance programs freely, as long as the derivative works also follow the tenets of GPL. However, driven by commercial interests, some companies and individuals incorporate third party software without respecting the licensing terms. In addition, many downstream companies integrate into their projects software components delivered in binary form from upstream companies without knowing possible license violations. These intentional or unintentional software license violations lead to serious disputes from time to time. For example, Verizon was sued by Free Software Foundation for distributing Busybox, developed by Actiontec Electronics, in its FIOS wireless routers [1]. A second example is the licensing dispute between Skype and Joltid that almost terminated Skypes voice-over-IP service [2]. Software plagiarism detection techniques are, therefore, welcomed by both programmers who want to protect their code and companies who want to avoid costly lawsuits.

In order to prevent software theft some programmers use semantics-preserving code obfuscations to make their programs obscure. Yet code obfuscations can only prevent others from understanding the underlying logic, but cannot hinder direct copy. Even worse, plagiarists can in turn further obfuscate the source code and distribute it in binary form to evade detection. Software watermarking [3] is one of the earliest and most well-known approaches. By embedding a unique identifier, i.e. a watermark, that is hard to remove but easy to verify in the software before its distribution, it can serve as a strong evidence of software plagiarism. However, besides the need to insert additional data in the original program, code obfuscations can often destroy watermarks. It is believed that a sufficiently determined attacker will eventually be able to defeat any watermark [4].

In order to address the problem a different software plagiarism detection technique called software birthmarking [5], [6] has been proposed. Unlike watermarking where unique identifiers are inserted into a program, birthmarking attempts to extract a set of characteristics that can be used to uniquely identify a program. Two programs with same birthmarks indicate software theft. As illustrated by Myles and Collberg [7], a birthmark may identify software theft even after the watermarks are removed by code obfuscations. Although this technique is promising, extracting high-quality birthmarks, unfortunately, turns out to be a very challenging task. State-of-the-art software

- *T.Liu is the corresponding author, he is with the Ministry of Education Key Lab For Intelligent Networks and Network Security (MOEKLINNS), School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China. Email: tingliu@mail.xjtu.edu.cn*
- *Z.Tian, M.Fan and E.Zhuang are with the Ministry of Education Key Lab For Intelligent Networks and Network Security (MOEKLINNS), Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China. Email: {zztian, fanming.911025, zhuangeryue}@stu.xjtu.edu.cn*
- *Q.Zheng is with the Ministry of Education Key Lab For Intelligent Networks and Network Security (MOEKLINNS), Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China. Email: qhzheng@mail.xjtu.edu.cn*
- *Z.Yang is with the Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA, and with the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China. Email: zijiang.yang@wmich.edu*

birthmarking can only provide very limited help due to several reasons. Firstly, many existing approaches [8]–[10] require source code. However, unless other evidence are obtained, programs suspected of plagiarism are usually available only in the form of binary executable to conceal plagiarism. Secondly, many approaches are applicable to specific types of operating systems or programming languages. For example, the API based birthmarks [11], [12] rely on features of Java or Windows system, thus unable to detect cross-platform plagiarism. Thirdly, with the help of various powerful automated semantics-preserving code obfuscation tools [13], [14], plagiarists are able to make significant changes to copied code without changing its functionality. As a result, it is still a daunting task to extract birthmarks that remain the same before and after code obfuscations.

In this paper, we propose to compute birthmarks based on execution sequences. Extracting birthmarks from assembly instructions not only enables us to detect plagiarisms without source code, but also makes our approach independent of programming languages and operating systems. The dynamic approach also make it possible for us to focus on the program logic, i.e. how the inputs are processed, rather than its syntax. Instead of considering complete execution sequences, our birthmark is based on a subset of executed instructions. Complete execution sequences are not desirable due to several reasons. Firstly, an execution sequence of a non-trivial program is usually very large. A birthmark consisting of large number of instructions is computationally expensive to process. Secondly, many instructions, such as `mov` that moves data between memory and registers, are added by compilers and are irrelevant to the program logic. Complete execution sequences with large amount of such common instructions bear similarity even they are obtained from different programs. Since plagiarism is measured by similarity instead of exact match in practice, birthmarks based on complete execution traces leads to high false positive[1] rate. Therefore, we consider only the key instructions executed under a given input vector for the computation of birthmarks. The ideal key instructions are inherent to program logic and any change to such instructions leads to malfunction of copied code. In our implementation, we consider key instructions as those that both generate new values (value-updating instructions) and propagate taints from input (input-correlated instructions). These instructions reflect how inputs are processed, thus are inherent to program logic. Since a particular execution is an abstraction of the whole program, we use multiple executions and the k-gram algorithm [7] to compute the similarity of birthmarks.

The contributions of this paper are summarized as follows:

- A new dynamic birthmark called DYKIS (DYnamic Key Instruction Sequence) is proposed to enrich the birthmark-based plagiarism detection family.
- We have implemented algorithms to extract DYKIS birthmarks from binary executables and to detect software plagiarisms by comparing such birthmarks. Our tool is publicly available for download. To the best of our knowledge there are very few birthmark-based plagiarism detection tools that are publicly available.
- We have conducted experiments on 342 versions of 28 different programs. Our empirical study shows that our approach is not only able to detect cross-platform plagiarisms but also resilient to almost all the state-of-the-art semantics-preserving obfuscation techniques. Our benchmarks are publicly available.
- We illustrate five possible whole program plagiarism scenarios, against all of which DYKIS is tested for effectiveness. We believe such study is beneficial for researchers to design experiments and present their findings.
- We compare our approach against SCSSB, an existing birthmark-based approach. The comparison indicates that our approach achieves higher accuracy and superior performance with respect to metrics including `URC` (Union of Resilience and Credibility), `F-Measure` and `MCC` (Matthews Correlation Coefficient).

The remainder of this paper is organized as follows. After describing necessary background on software birthmarks in Section 2, we present our DYKIS birthmark in Section 3 that includes its definition, generation and comparison. Section 4 gives an extensive evaluation on resilience and credibility of DYKIS birthmarks. The section also includes a comparison between DYKIS and an existing birthmark. In Section 5 we discuss threats to the validity of our approach. Section 6 reviews related work and finally Section 7 concludes the paper.

2 Preliminaries

The goal of software plagiarism detection is to determine whether a *plaintiff* is a *copy* of a *defendant*. In this paper, a plaintiff refers to a program that is suspected of plagiarism. A defendant is a program used to compare with the plaintiff. If convicted, the plaintiff is a copy of the defendant program. Otherwise the plaintiff is independently developed.

A software birthmark is a set of characteristics extracted from a program that reflects intrinsic properties of the program. It can be used to uniquely identify

1. In plagiarism detection literature, false positive refers to the case that an independently developed program is considered a copy of another program, and false negative refers to the case that a copied program is considered an independently developed program.

a program. Definition 1 gives its typical definition that has led to several works that extract birthmarks statically by analyzing the syntax of a program.

Definition 1: **Software Birthmark [15].** We say $p_{\mathcal{B}}$ is a birthmark of the program p if and only if both of the following conditions are satisfied:

- $p_{\mathcal{B}}$ is obtained only from p itself.
- Program q is a copy of $p \Rightarrow p_{\mathcal{B}} = q_{\mathcal{B}}$.

Since static birthmarks are usually ineffective against semantics-preserving obfuscations that can modify the syntactic structure of a program, dynamic software birthmarks, as defined in Definition 2, were introduced to remedy the problem.

Definition 2: **Dynamic Software Birthmark [6].** Let p be a program and I be a input to p. We say $p_{\mathcal{B}}^{I}$ is a dynamic birthmark of p if and only if both of the following conditions are satisfied:

- $p_{\mathcal{B}}^{I}$ is obtained only from p itself when executing p with input I.
- Program q is a copy of $p \Rightarrow p_{\mathcal{B}}^{I} = q_{\mathcal{B}}^{I}$.

The two definitions give conceptual description without offering any implementation details. In addition, there are many practical issues not addressed by the definitions. For example, even if q is a copy of p, in practice the birthmarks of the two programs are not identical. As a result, various algorithms for generating and comparing software birthmarks have been derived from the definitions.

Let p and $p_{\mathcal{B}}$ be the plaintiff and its birthmark, and q and $q_{\mathcal{B}}$ be the defendant and its birthmark. In practice the plagiarism is decided by a threshold ε and a function sim that computes the similarity score between $p_{\mathcal{B}}$ and $q_{\mathcal{B}}$. The range of a similarity score is between 0 and 1, and a typical value of ε is 0.25. Equation 1 gives a conceptual definition of sim that returns a three-value result: positive, negative or inconclusive. We say sim gives a *false classification* if it reports false positive or false negative, and we say sim gives *incorrect classification* if it reports false positive, false negative or inconclusiveness.

$$sim\left(p_{\mathcal{B}}, q_{\mathcal{B}}\right) = \begin{cases} \geq 1-\varepsilon \quad positive: \ p \ is \ a \ copy \ of \ q \\ \leq \varepsilon \quad negative: \ p \ is \ not \ a \ copy \ of \ q \\ otherwise \quad inconclusive \end{cases} \tag{1}$$

There are different ways to implement the function depending on the format of birthmarks. For example, for birthmarks in the format of set, the widely used methods for the implementation of sim include Cosine distance and Jaccard index. More details regarding similarity computation are given in Section 3.2.

A high quality birthmark must have a low ratio of incorrect classifications for a given ε. However, high precision is not necessarily required, because, birthmark technique is not a proving technique but rather a detecting technique. In other words, false negative is more critical than false positive because further investigation is conducted once a program is considered a copy. Two properties, resilience and credibility, are widely adopted in the literature to evaluate the quality of a birthmark. The existing definitions of resilience [7] and credibility [16] are absolute in the sense that they do not consider programs or inputs. In practice birthmarks are often relative. For example, birthmarks extracted using the same algorithm are resilient to code transformations for p under input I_1, but are not resilient for q under input I_2. In this paper we make the definitions relative in order to match the dynamic nature of our approach.

TABLE 1
Average similarity scores between a program and its compiler-obfuscated version utilizing whole and key instruction sequences

Program	WholeTrace	KeyTrace
bzip2	0.642	1.00
gzip	0.397	0.899
md5sum	0.30	0.789

Definition 3: **Resilience.** Let p be a program and q be a copy of p generated by applying semantics-preserving code transformations τ. A birthmark is resilient to τ and input I if $sim\left(p_{\mathcal{B}}^{I}, q_{\mathcal{B}}^{I}\right) \geq 1-\varepsilon$.

Definition 4: **Credibility.** Let p and q be independently developed programs that may accomplish the same task. A birthmark is credible under input I if it can differentiate the two programs, that is $sim\left(p_{\mathcal{B}}^{I}, q_{\mathcal{B}}^{I}\right) \leq \varepsilon$.

3 Software Plagiarism Detection Based on DYKIS

3.1 Dynamic Key Instruction Sequence Birthmarks

A high quality birthmark must be closely related to the semantics of a program in order to be resilient to semantics-preserving code transformations. An obvious candidate is the instruction sequences recorded during a program execution, as they clearly reflect how an input vector is processed by the program. However taking the whole sequence as a birthmark is often too large for further analysis. For example, for the three programs `bzip2`, `gzip` and `md5sum` that are used in our experiments, the number of instructions recorded during an execution are 1.73M[2], 4.54M and 0.32M, respectively, with a tiny 5K-byte text file as the program input. Even after filtering out system library instructions, 1.53M, 4.4M and 0.05M instructions still remain for the three executions.

Besides being computationally expensive, birthmarks extracted from whole instruction sequences are easily defeated by simple code transformations due to the equal treatment of all kinds instructions [17]. For

2. In this paper M refers to 10^6 and K refers to 10^3.

the three programs, we generate a set of weakly obfuscated versions for each of the program by utilizing different compilers and optimization levels to compile the source code. We then generate birthmarks from the whole execution sequences after removing Linux library instructions, and compare the similarity between birthmarks with the approach to be discussed in Section 3.2. Table 1 gives the similarity scores of the compiler-obfuscated versions of the same program. As indicated by Column *WholeTrace*, the similarity scores between the compiler obfuscated versions of the same program are all below 0.65. Considering the most common value of ε is 0.25, whole execution sequence based birthmarking fails to detect any plagiarisms. Besides the issue of false negatives, birthmarks extracted from whole instruction sequences can also report false positives. For example, the similarity between birthmarks of `md5sum` and `cksum` using the whole instruction sequence is as high as 0.9. This is due to the fact that the common system libraries used by the two applications constitute significant portion of the execution sequences.

The aforementioned issues can be solved if we consider only the key instructions in an execution sequence. Ideally the key instructions should constitute a small portion of a whole execution sequence and must be relatively unique. By studying the execution sequences at assembly level, we have found that there exist a large number of data-transfer instructions, such as `mov`, `push` and `pop`, in all the applications. These instructions can be discarded because they usually facilitate computations rather being part of program logic. On the other hand, instructions whose execution will generate new values, such as `add` and `shl`, reflect the inherent logic of a program. These so called value-updating instructions [18] usually represent program semantics. In addition, program semantics is a formal representation of how inputs are processed by the program, so instructions not handling inputs are usually dispensable. Based on dynamic taint analysis [19], we can obtain the correlation between instructions and inputs. The dynamic taint analysis treats program inputs as tainted data. Initially a taint label is associated with the register or memory unit where an input resides to indicate that the location is tainted. During the execution, taint labels are propagated according to taint policies. A variable (register or memory unit) is marked tainted if it is data dependent on a variable that has already been associated with a taint label. We call instructions whose execution introduce new taint labels to registers or memory units as *input-correlated* instructions. Definition 5 summarizes the types of key instructions.

Definition 5: **Dynamic Key Instruction.** Let $trace(p, I)$ be a sequence of executed instructions of Program p under input I. For each instruction c in $trace(p, I)$, we say c is a key instruction if both of the following conditions are satisfied:

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

We use $key(p, I)$ to denote the sub-sequence of $trace(p, I)$ that consists of key instructions only.

After removing non-key instructions, the size of dynamic key instruction sequences of `gzip`, `bzip2` and `md5sum` are 10K, 1.6K and 28K respectively, which are about 173 times, 2800 times and 11 times less than their corresponding whole sequences. Moreover, as illustrated by Column *KeyTrace* in Table 1, there exists strong similarity between the DYKIS birthmarks extracted from dynamic key instruction sequences. This indicates that the key instructions are superior to whole trace for code transformations provided by different compilers and optimization levels.

Despite the fact that key instructions greatly reduces the number of instructions to analyze, the size of reduced key instructions is still unbound. In order to address the problem, we adopt the k-gram algorithm, also used in [7], [11], [12], to bound key instruction sequences with a length k window.

Definition 6: **k-gram.** Let $opcode = \langle e_1, e_2, \cdots, e_n\rangle$ be a sequence of executed operation codes. Given a predefined length k, a subsequence $opcode_j(k) = \langle e_j, e_{j+1}, \cdots, e_{j+k-1}\rangle$ $(1 \leq j \leq n-k+1)$ can be generated by sliding the window over t with stride one each time. We refer to $opcode_j(k)$ as a k-gram.

Finally we give the definition of DYKIS birthmarks.

Definition 7: **DYKIS birthmark.** Let $key(p, I) = \langle ins_1, ins_2, \cdots, ins_n\rangle$ be a key instruction sequence recorded during execution of program p under input vector I. Let $opcode(p, I) = \langle e_1, e_2, \cdots, e_n\rangle$ be the corresponding sequence of operation code. That is, e_i is the operation code of ins_i. Let $gram(p, I, k) = \langle g_j | g_j = \langle e_j, e_{j+1}, \cdots, e_{j+k-1}\rangle\rangle$ $(1 \leq j \leq n-k+1)$ be a sequence of k-grams. We call the key-value pair set $p_{\mathcal{B}}^{I}(k) = \{\langle g_m, freq(g_m)\rangle | g_m \in gram(p, I, k) \wedge \forall_{m_1 \neq m_2} . g_{m_1} \neq g_{m_2}\}$, where $freq(g_m)$ represents the frequency of g_m occurred in $gram(p, I, k)$, as the DYnamic Key Instruction Sequence based (DYKIS) birthmark of p under the input I.

Example 1. Suppose the dynamic key instruction sequence $key(p, I)$ obtained from an execution of program p is as the following:

```
add  ecx,ebx
rol  ecx,0x7
and  ebx,ecx
add  ebx,esi
rol  ebx,0xc
```

The corresponding sequence of operation codes is therefore $opcode(p, I) = \langle add, rol, and, add, rol\rangle$. When $k = 2$ the k-gram sequence is $gram(p, I, 2) = \langle\langle add, rol\rangle, \langle rol, and\rangle, \langle and, add\rangle, \langle add, rol\rangle\rangle$. After counting the element frequencies, we get DYKIS: $p_{\mathcal{B}}^{I}(2) = \{(\langle add, rol\rangle, 2), (\langle rol, and\rangle, 1), (\langle and, add\rangle, 1)\}$.

3.2 Similarity Calculation

In the literature of software birthmarking, different methods of similarity measurements are used depending on different birthmark formats that in general are sequences, sets or graphs. Similarity of sequences can be computed by pattern matching methods, such as calculating the longest common subsequences [12], [18]. There are many methods for calculating similarity of sets, including Dice coefficient [16], Jaccard index [11], Cosine distance [20], [21]. Computing the similarity of graphs is relatively more complex. It is conducted by either graph monomorphism [22] or isomorphism algorithms [10], [23], or by translating a graph into a vector using algorithms such as random walk with restart [24].

Our birthmark is a set composed of key-value pairs. The types of keys are not as rich as other static k-gram birthmarks [7], [25] or dynamic birthmarks [17], [26] extracted from the whole instruction sequences. Therefore DYKIS birthmarks have higher probability to be similar if we adopt calculation methods such as Jaccard index [11] and Dice coefficient [16] that ignore element frequency in a sequence. Besides, the execution frequency of an instruction, which is not available to static methods, indicates how inputs are processed and thus should be an integral component of a birthmark. Based on these considerations, we transform the birthmarks into frequency vectors, and then make use of the Cosine distance, the most commonly used similarity metric for vectors, to measure the similarity of two birthmarks.

Formally, given two DYKIS birthmarks $p_{\mathcal{B}} = \{(k_1, v_1), \cdots, (k_n, v_n)\}$ and $q_{\mathcal{B}} = \{(k_1', v_1'), \cdots, (k_m', v_m')\}$, let $U = kset(p_{\mathcal{B}}) \cup kset(q_{\mathcal{B}})$ where $kset(p_{\mathcal{B}}) = \{k_1, \cdots, k_n\}$ and $kset(q_{\mathcal{B}}) = \{k_1', \cdots, k_m'\}$. We convert set U to vector $\overrightarrow{U} = \langle t_1, \cdots, t_{|U|}\rangle$ by assigning an arbitrary order to the elements in U. Let vector $\overrightarrow{p}_{\mathcal{B}} = \langle a_1, \cdots, a_{|U|}\rangle$. For each element in $\overrightarrow{p}_{\mathcal{B}}$ we have

$$a_i = \begin{cases} v_i, & if\ t_i \in kset(p_{\mathcal{B}}) \\ 0, & if\ t_i \notin kset(p_{\mathcal{B}}) \end{cases}, \tag{2}$$

where v_i is the value of key t_i in $p_{\mathcal{B}}$ $(1 \leq i \leq |U|)$. Likewise we define $\overrightarrow{q}_{\mathcal{B}} = \langle b_1, \cdots, b_{|U|}\rangle$. The similarity of two DYKIS birthmarks $p_{\mathcal{B}}$ and $q_{\mathcal{B}}$ is calculated with $sim(p_{\mathcal{B}}, q_{\mathcal{B}}) = \frac{\overrightarrow{p}_{\mathcal{B}} \times \overrightarrow{q}_{\mathcal{B}}}{|\overrightarrow{p}_{\mathcal{B}}||\overrightarrow{q}_{\mathcal{B}}|}$, where $|\overrightarrow{p}_{\mathcal{B}}| = \sqrt{\sum_{a_i \in \overrightarrow{p_{\mathcal{B}}}} a_i^2}$, $|\overrightarrow{q}_{\mathcal{B}}| = \sqrt{\sum_{b_i \in \overrightarrow{q_{\mathcal{B}}}} b_i^2}$.

Example 2. Assume the DYKIS birthmark of another program q executed with the same input I as the program p in Example 1 is: $q_{\mathcal{B}}^I(2) = \{(\langle add, rol\rangle, 1), (\langle rol, and\rangle, 1), (\langle and, sub\rangle, 2)\}$. We have $\overrightarrow{U} = \langle\langle add, rol\rangle, \langle rol, and\rangle, \langle and, add\rangle, \langle add, sub\rangle\rangle$. Correspondingly the two DYKIS birthmarks can be converted into the following vectors: $\overrightarrow{p_{\mathcal{B}}^I} = \langle 2, 1, 1, 0\rangle$ and $\overrightarrow{q_{\mathcal{B}}^I} = \langle 1, 1, 0, 2\rangle$. Finally the similarity score of the two DYKIS birthmarks is $sim(p_{\mathcal{B}}^I(2), q_{\mathcal{B}}^I(2)) = 0.5$.

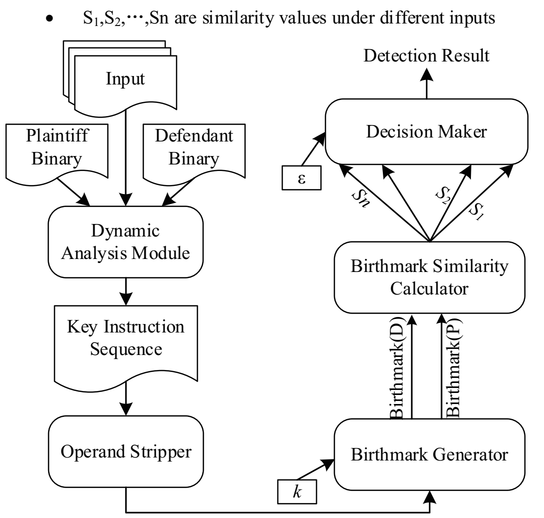

Fig. 1. Overview of DYKIS-PD

3.3 Plagiarism Detection

Although a single execution of a program p faithfully reveals the behavior of p, it is an abstraction and thus may lead to false positives in plagiarism detection. For example, two independently developed programs adopting standard error-handling subroutines may exhibit identical behavior under error-inducing inputs. In order to alleviate this problem, the calculation of the similarity score between two programs is based on various birthmarks obtained under multiple inputs.

Let p and q be the plaintiff and defendant. Given a series of inputs $\{I_1, \ldots, I_n\}$ to drive the execution of the programs, we obtain n pair of DYKIS birthmarks $\{(p_{\mathcal{B}1}, q_{\mathcal{B}1}), \ldots, (p_{\mathcal{B}n}, q_{\mathcal{B}n})\}$. The similarity score between program p and q is calculated by $sim(p, q) = \sum_{i=1}^{n} sim(p_{\mathcal{B}i}, q_{\mathcal{B}i})/n$, whose value is between 0 and 1. That is, the existence of plagiarism between p and q is decided by the average similarity scores of their DKYIS birthmarks and the predefined threshold ε.

3.4 System Design and Implementation

Figure 1 depicts the overview of our DYKIS-based software plagiarism detection tool called DYKIS-PD that consists of five modules: the dynamic analysis module, the operand stripper, the birthmark generator, the similarity calculator and the decision maker. DYKIS-PD has been demonstrated at a conference [27] and is now publicly available for download.

- **Dynamic Analysis Module.** The dynamic analysis module, as depicted in Figure 2, is implemented as a `PIN` [28] plugin called DKISExtractor. There are two sub-modules that are implemented on top of the libdft [29] data flow analysis framework. TaintAnalyzer performs dynamic taint analysis, which includes the recognition of taint sources and the propagation of taints to

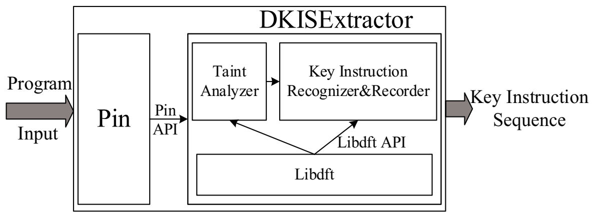

Fig. 2. Structure of the dynamic analysis module

assist the identification of input-correlated instructions. The input-correlated instructions are then processed by the sub-module of Key Instruction Recorder to produce the key instruction sequence. In order to compare with the existing system call based birthmark SCSSB [12], we have also implemented another `PIN` plugin called SysTracer for capturing system calls.

- **Operand Stripper.** Given the same source code, the register and memory usage in the binary executable can be totally different if different compilers or optimization levels are adopted. Even with the same compiler and optimization level, due to the difference of memory layout during compilation, the operands across binary files corresponding to the same source code can be different. Therefore, the operands in the key instructions are not helpful in plagiarism detection and we use operand stripper to remove them.
- **Similarity Calculator and Decision Maker.** Similarity calculator measures similarity of two DYKIS birthmarks by their cosine distance with a value between 0 and 1. Decision maker decides plagiarism by the average value ψ of multiple similarity scores against a predefined threshold ε. We adopt a default value of $\varepsilon = 0.25$ as in previous studies [11], [16], [30], [31]. That is, two programs are classified as independent if $\psi \leq 0.25$, as plagiarized if $\psi \geq 0.75$.

4 Evaluation

We have conducted extensive experiments to evaluate the resilience and credibility of DYKIS birthmarks. Our first empirical study is to identify a proper value of k, because k-gram algorithms with different k values may produce different birthmarks even for the same instruction sequence. Once the value of k is fixed we evaluate DYKIS on large number of programs, various compilers, different implementation languages, and multiple semantics-preserving code obfuscators.

Table 2 lists the names and some other basic information of our benchmarks. Column *#Versions* gives the number of versions that include the original programs, their successive releases and the obfuscated versions. *L* and *W* mark whether there exist Linux or Windows binaries, respectively. Column *Size* shows the number of bytes of the largest version, with its version number listed in column *Version*. In the following we give a summary of our testing environment.

- The benchmarks consist of 342 versions of 28 programs implemented in C or Java, including:
  - four image processing software: `sxiv`, `feh`, `pho` and `qiv`;
  - five compression/decompression software: `gzip`, `bzip2`, `zip`, `rar`, and `lzip`;
  - seven Java applications: `JLex`, a lexical analyzer generator; `JavaCUP`, a LALR parser generator for Java; a `Calculator` implemented using the JavaCUP specification; as well as `Avrora`, `Antlr`, `Luindex` and `Lusearch` from the Dacapo benchmark;
  - six security libraries: `md5sum`, `openesslSHA`, `openesslSHA1`, `openesslMD4`, `opensslMD5`, `openesslRMD160`.
  - five text-based web browsers: `elinks`, `links`, `links2`, `lynx` and `w3m`;
- We use two compilers `gcc` and `llvm` with various optimization levels to generate binaries from source code.
- We apply many publicly available code obfuscators, including `SandMark`, `Zelix KlassMaster`, `Allatori`, `DashO`, `JShrink`, `ProGuard` and `RetroGuard`.
- We utilize many packing tools including `UPX`, `AS-Protect`, `Fsg`, `Nspack`, `PECompact` and `WinUpack`, as well as a specialized binary obfuscator `binobf` to obfuscate the binaries.

The quality of DYKIS birthmark is further compared with an existing type of birthmark called SCSSB [12]. Finally we evaluate the sensitivity of DYKIS to inputs.

4.1 Impact of Parameter k

The value of k used in the k-gram algorithm plays an important role, as its variation leads to different DYKIS birthmarks even with the same key instruction sequence. In this section we study the impact of k on efficiency and effectiveness of our plagiarism detection tool. Intuitively, larger value of k incurs additional overhead but gives better picture of program semantics. Surprisingly our empirical study contradicts the intuition: there is no clear benefit using large k values even not considering computational cost. Based on our experiments we are able to determine a fixed k value that is appropriate for all our experiments.

4.1.1 *Impact of k on efficiency*

We conduct three groups of experiments. In the first group we extract DYKIS birthmarks from the programs within the same category, and then compute

TABLE 2
Benchmark programs.

Name	Size (bytes)	Version	#Versions	Name	Size (bytes)	Version	#Versions
gzip	1,953,792	1.2.4	25/LW	JLex	282,624	1.4	44/L
bzip2	2,011,136	1.0.6	19/LW	JavaCUP	421,888	0.10k	39/L
zip	189,256	3.0	2/L	Calculator	421,888	0.10k	41/L
rar	577,796	5.0	2/L	Avrora	5,827,287	Dacapo09	17/L
lzip	87,784	1.13	2/L	Luindex	3,661,824	Dacapo09	26/L
sxiv	46,980	1.0	2/L	Lusearch	3,678,208	Dacapo09	25/L
feh	489,287	2.2	8/L	Antlr	2,469,888	Dacapo06	25/L
pho	63,720	0.9.8	2/L	md5sum	1,728,512	8.13	3/L
qiv	60,612	2.2.4	3/L	opensslMD4	510,224	1.0.1	9/LW
elinks	1,200,128	0.12	2/L	opensslSHA	510,224	1.0.1	9/LW
links	1,011,712	2.1	2/L	opensslSHA1	510,224	1.0.1	9/LW
links2	2,797,568	2.6	2/L	opensslRMD160	510,224	1.0.1	9/LW
lynx	1,396,736	2.8.8	2/L	opensslMD5	510,224	1.0.1	9/LW
w3m	1,282,048	0.5.3	2/L	UPX	319,292	3.91	2/LW

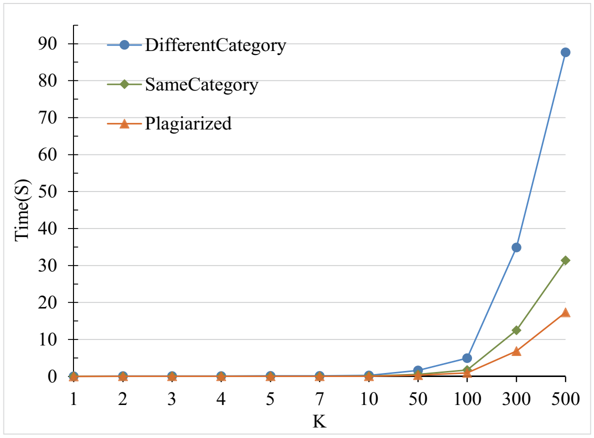

Fig. 3. Impact of k on time consumption of similarity calculation

pair-wise similarity scores. In the second group we conduct pair-wise comparison of software across different categories. Finally in the third group we compare different versions of the same program compiled by different compilers or optimization levels. The programs in the first two groups are considered independent, while the programs in the third group are considered plagiarized.

Figure 3 depicts the time consumption under different values of k with three colored lines. The blue, green and orange lines give average time spent on obtaining and comparing DYKIS birthmarks of software in different categories, software in same categories and plagiarized software, respectively. As expected, time consumption increases as the value of k increases. The time usage is trivial when the value of k is small, and becomes significant when k is more than 100.

4.1.2 Similarity scores of independent software in different categories

Since plagiarism does not exist between image processing software and other types of software, we expect very low similarity scores. Obviously the larger the value of k, the smaller the similarity scores. The question here is whether the benefit of a large k value justifies its cost.

The blue rows in Table 3 illustrate how similarity changes between the imaging processing software `sxiv` and other types of programs by varying the value of k. 1-gram is apparently a bad choice as all the scores are above the default threshold 0.25. The scores decrease sharply starting from 2-gram and there is no false positive when $k = 3$. Under 3-gram the largest similarity score is 0.168 and the average is 0.075. Although $k = 500$ gives lower average score 0.004, there is no clear benefit over $k = 3$ since the average score of the latter is also well below 0.25.

We have conducted experiments on the other three image processing software against other types of programs, and have obtained similar results. The grey row gives the average similarity scores on all our comparisons.

4.1.3 Similarity scores of independent software in the same category

The functionality of programs in the same categories overlap to a great extent, even they are developed independently. Therefore a low value of k may have greater risk of false positives. In this group of experiments we study the impact of k values on the similarity scores between independently developed programs in the same category. In particular, we compare DYKIS birthmarks between compression/decompression software `bzip2`, `gzip`, `zip`, `rar`, `lzip`, between encryption/decryption software `md5sum`, `openesslSHA`, `openesslSHA1`, `openesslMD4`, `openesslRMD160`, and between image processing software `pho`, `feh`, `qiv`, `sxiv`. Table 3 randomly chooses 8 pairs of comparisons. As expected, the average similarity scores decreases as k increases. However, the speed of decrease becomes trivial once k reaches 3. For example, the average similarity scores are 0.116 and 0.101 for $k = 4$ and $k = 500$,

TABLE 3
Impact of parameter k on similarity scores between independently developed programs in different categories (blue) and in same categories (green), and between programs and their copies (orange).

ComparisonPairs	K=1	=2	=3	=4	=5	=7	=10	=50	=100	=300	=500
(sxiv,bzip2)	0.414	0.074	0.03	0.018	0.006	0	0	0	0	0	0
(sxiv, gzip)	0.578	0.062	0	0	0	0	0	0	0	0	0
(sxiv, md5sum)	0.614	0.308	0.03	0.016	0	0	0	0	0	0	0
(sxiv,opensslMD4)	0.552	0.242	0.168	0.163	0.162	0.16	0.15	0.071	0.01	0	0
(sxiv, opensslMD5)	0.564	0.322	0.126	0.132	0.132	0.128	0.118	0.034	0.004	0	0
(sxiv, opensslRMD160)	0.606	0.205	0.099	0.084	0.084	0.082	0.072	0.019	0.004	0	0
(sxiv, opensslSHA)	0.52	0.227	0.128	0.116	0.118	0.123	0.113	0.044	0.006	0	0
(sxiv, zip)	0.568	0.074	0.022	0.021	0.02	0.02	0.018	0.018	0.019	0.024	0.03
Average Score of Software Listed Above	0.552	0.189	0.075	0.069	0.065	0.064	0.059	0.023	0.005	0.003	0.004
Average Score of All Software in Different Categories Tested	0.523	0.183	0.089	0.081	0.078	0.078	0.074	0.051	0.037	0.039	0.024
(bzip2, gzip)	0.702	0	0	0	0	0	0	0	0	0	0
(bzip2, zip)	0.568	0.001	0	0	0	0	0	0	0	0	0
(md5sum, opensslRMD160)	0.919	0.602	0.312	0.144	0.059	0.023	0	0	0	0	0
(md5sum, opensslSHA)	0.895	0.558	0.295	0.116	0.041	0.005	0	0	0	0	0
(md5sum, opensslSHA1)	0.709	0.209	0.019	0.009	0	0	0	0	0	0	0
(md5sum, opensslMD4)	0.879	0.499	0.2	0.107	0.054	0.029	0.007	0	0	0	0
(pho,feh)	0.628	0.336	0.297	0.296	0.298	0.304	0.311	0.386	0.453	0.475	0.342
(qiv,feh)	0.54	0.274	0.256	0.257	0.259	0.266	0.276	0.364	0.452	0.493	0.462
Average Score of Software Listed Above	0.730	0.310	0.172	0.116	0.089	0.078	0.074	0.094	0.113	0.121	0.101
Average Score of All Software in the Same Categories Tested	0.450	0.214	0.158	0.139	0.130	0.126	0.125	0.137	0.136	0.104	0.088
bzip2(gcc_dbg_o1, llvm_release_o3)	0.973	0.968	0.957	0.952	0.947	0.921	0.864	0.771	0.703	0.634	0.493
bzip2(gcc_dbg_o1, gcc_dbg_o2)	0.984	0.981	0.976	0.976	0.976	0.976	0.976	0.904	0.832	0.758	0.613
bzip2(gcc_dbg_o1, llvm_release_o3)	0.973	0.968	0.957	0.952	0.947	0.921	0.864	0.771	0.703	0.634	0.493
bzip2(llvm_dbg_o1, gcc_release_o3)	0.968	0.904	0.873	0.838	0.824	0.785	0.713	0.687	0.619	0.623	0.627
bzip2(llvm_dbg_o1, gcc_dbg_o1)	0.951	0.896	0.866	0.838	0.827	0.794	0.733	0.642	0.574	0.511	0.378
bzip2(llvm_dbg_o1, llvm_dbg_o2)	0.972	0.906	0.877	0.858	0.843	0.826	0.816	0.819	0.794	0.765	0.781
Average Score of Software Listed Above	0.970	0.937	0.918	0.902	0.894	0.871	0.828	0.766	0.704	0.654	0.564
Average Score of All Plagiarized Software Tested	0.983	0.962	0.949	0.938	0.931	0.911	0.874	0.840	0.790	0.763	0.722

respectively. There is almost no benefit to have a k value beyond 4. Surprisingly, there are same number of false positives for both $k = 4$ and $k = 500$, and the maximal similarity score is 0.296 when $k = 4$ versus 0.462 when $k = 500$ for `qiv` and `feh`. This is because their key instructions are quite similar to some extent (we will explain this in Section 4.6), and meanwhile the birthmark similarity calculation considers both k-grams and their frequency. With the increase of k, the impact of frequency is weakened and the type of k-grams starts to play a leading role. Finally, the grey row gives the average similarity scores of all the 26 pairs of comparisons we have conducted. As the data indicate, a small k value such as 4 is almost as good as a large k value such as 500.

4.1.4 *Similarity scores of plagiarized software*

We treat binaries compiled from the same source code but with different compilers or optimization levels as copies of each other. When we compare such binaries we expect large similarity scores to be above $1 - \epsilon$ to indicate plagiarism. In the experiment setup we choose a program such as `bzip2` and compile it with two compilers, one is `llvm` with different flags such as `dbg` and `O1`, the other is `gcc` with various flags. From the 12 versions of binaries we compare their DYKIS birthmarks against each other. Table 3 lists comparison data of randomly chosen pairs. As expected, the similarity scores decrease as k increase. There are false negatives starting from $k = 10$, and the average scores indicate false negatives starting from $k = 300$. Small k values have a clear advantage over large values.

Based on the above observations, we believe a k value between 3-10 is appropriate due to the following reasons

- The algorithm based on small k values is more efficient than that based on large k values.
- For independently developed software, a lower k value, i.e. 1 or 2, incurs significant risk of false positives.
- A larger k value leads to lower similarity score for independently developed software, but the difference is minimal.
- Large k values, i.e. greater than 10, reports much more number of false negatives for plagiarized software.

Therefore, in the rest of the paper, we set the default k value at 4 for all the experiments.

4.2 Resilience to Weak Code Obfuscation

Plagiarized software presented in binary format is often compiled with a different compiler or compiler optimization levels. Therefore, it is essential for binary based birthmarking techniques to handle such relatively weak semantics-preserving code transformation. In our experiments, we choose two open-source compression software `gzip-1.2.4` and `bzip2-1.0.6` as experimental subjects. We also se-

TABLE 4
Statistical difference between `bzip2` versions generated with different compilers and optimization levels

	Size(Kb)	#Functions	#Instructions	#Blocks	#Calls
Max.	204	241	22968	3028	681
Min.	76	185	12440	2341	472
Avg.	133.7	210.4	16127	2725.1	562.1
Stdev.	44	20.1	2874.1	237.5	67.8

lect two compilers `llvm3.2` and `gcc4.6.3` to compile the two programs with multiple optimization levels (`-O1`, `-O2` and `-O3`) and the debug option (`-g`) switched on or off. Such setup leads to 12 different executables for each program. Table 4 gives statistical differences on the size, number of functions, number of instructions, number of basic blocks and number of functional calls of the 12 executables of `bzip2`. The table for `gzip` is similar so we omit it here. The data indicate that even weak code transformation can make significant differences among the produced binaries.

Table 5 summarizes the pair-wise DYKIS birthmark similarity scores between different versions of `bzip2`. It can be observed that the similarity scores are all quite high, with an average value of 0.922. In addition, there are no false positives. The lowest score 0.796 happens between two versions compiled by `llvm` and `gcc`, both with `O1` and debug flag switched on. Similar results are observed for `gzip` as illustrated in Table 6. Not surprisingly the type of compiler has greater impact than optimization flags. The experimental results indicate that DYKIS birthmarks exhibit strong resilience against weak code obfuscations.

4.3 Resilience to Strong Obfuscations

Sophisticated code obfuscation techniques may be used to purposely defeat plagiarism detection tools. In this group of experiments we study the resilience of DYKIS against strong code obfuscations. In particular, we use Java bytecode obfuscation tool `SandMark` [13] to generate a group of obfuscated versions, which are then converted to x86 executables by `GCJ` [32], the GNU ahead-of-time compiler for Java. `SandMark` implements a series of advanced semantics-preserving transformation techniques, including 15 application obfuscations, 7 class obfuscations and 17 method obfuscations. It can defeat almost all birthmark-based plagiarism detection methods targeting a specific language [6], [7], [11], [22], [33].

Our experimental subjects include three Java programs widely used in the birthmark-based plagiarism detection literature, including `JLex`, a lexical analyzer generator, `JavaCUP`, a LALR parser generator for Java, and a `Calculator` implemented using the `JavaCUP` specification. In addition, there are four larger programs, `Avrora`, `Luindex`, `Lusearch` and `Antlr`, from the Dacapo [34] benchmark suite[3]. We design similar experiments as those conducted in [18] to measure the resiliency of DYKIS against single obfuscations, where only one obfuscation technique is applied at a time, and multiple obfuscations, where multiple obfuscation techniques are applied to one program at the same time.

4.3.1 *Resilience to single obfuscation*

We apply the 39 obfuscation techniques implemented in `SandMark` on the experimental subjects one at a time and generate a series of obfuscated versions. In order to ensure correctness of these transformations, all obfuscated versions are tested with a set of inputs and the versions with wrong outputs are eliminated.

The similarity scores are calculated between the original program and its obfuscated versions. Table 7 summarizes the results by listing the maximal, minimal and average similarity scores between the original program and its transformed versions. Column $\mathrm{ACC}_{0.75}$ gives the percentage of comparisons that report a similarity score above 1-0.25. Note that only the data obtained from the successfully transformed versions are considered in the first three columns. Columns $\mathrm{S/F}_{P1}$ and $\mathrm{S/F}_{P2}$ give the number of successful and failed obfuscations using `SandMark` and `GCJ`, respectively. The failures refer to those transforms that cannot be compiled or the transforms that give wrong outputs. The data indicate strong resilience against single obfuscation: the average similarity scores are well above the threshold except one incorrect classification where the similarity score between the original program `Antlr` and its sandmark-obfuscated version `Antlr_StaticMethodBodies` is 0.738.

To give a more intuitive presentation, the comparison results of `JavaCUP` to its 29 obfuscated versions are depicted in Figure 4, where the x-axis lists the obfuscation techniques and the y-axis gives the corresponding similarity scores.

4.3.2 *Resilience to multiple obfuscations*

A plagiarist may attempt to apply multiple obfuscation techniques to a single program in order to generate deeply obfuscated versions. However, deriving a semantics-equivalent disguised copy is not always easy. Transformations with even a single obfuscation are not always successful, as indicated by the columns $\mathrm{S/F}_{P1}$ and $\mathrm{S/F}_{P2}$ in Table 7. Applying multiple obfuscations simultaneously can significantly increase the failure rate. In order to facilitate our experiments we adopt the method used in [12], [18] where the obfuscators in `SandMark` are classified into two categories: data obfuscators and control obfuscators. Only the obfuscators in the same category are applied sequentially to the same experimental object. That is, there

3. We failed to compile other programs in Dacapo with GCJ.

TABLE 5
Similarity scores between `bzip2` binaries generated with different compilers and optimization levels

COMPILER			GCC 4.6.3						LLVM 3.2					
			Debug			Release			Debug			Release		
		OPTLevel	o1	o2	o3	o1	o2	o3	o1	o2	o3	o1	o2	o3
GCC 4.6.3	Debug	o1	1.000	0.823	0.823	1.000	0.823	0.823	0.796	0.798	0.798	0.796	0.798	0.798
		o2	-	1.000	1.000	0.823	1.000	1.000	0.941	0.964	0.964	0.941	0.964	0.964
		o3	-	-	1.000	0.823	1.000	1.000	0.941	0.964	0.964	0.941	0.964	0.964
	Release	o1	-	-	-	1.000	0.823	0.823	0.796	0.798	0.798	0.796	0.798	0.798
		o2	-	-	-	-	1.000	1.000	0.941	0.964	0.964	0.941	0.964	0.964
		o3	-	-	-	-	-	1.000	0.941	0.964	0.964	0.941	0.964	0.964
LLVM 3.2	Debug	o1	-	-	-	-	-	-	1.000	0.971	0.971	0.971	0.971	0.971
		o2	-	-	-	-	-	-	-	1.000	1.000	0.971	1.000	1.000
		o3	-	-	-	-	-	-	-	-	1.000	0.971	1.000	1.000
	Release	o1	-	-	-	-	-	-	-	-	-	1.000	0.971	0.971
		o2	-	-	-	-	-	-	-	-	-	-	1.000	1.000
		o3	-	-	-	-	-	-	-	-	-	-	-	1.000
Statistical Values			Max.	1.000		Min.	0.796		Avg.	0.922		$\text{Acc}_{0.75}$	100%	

TABLE 6
Similarity scores between `gzip` binaries generated with different compilers and optimization levels

COMPILER			GCC 4.6.3						LLVM 3.2					
			Debug			Release			Debug			Release		
		OPTLevel	o1	o2	o3	o1	o2	o3	o1	o2	o3	o1	o2	o3
GCC 4.6.3	Debug	o1	1.000	1.000	1.000	1.000	1.000	1.000	0.812	0.812	0.812	0.812	0.812	0.812
		o2	-	1.000	1.000	1.000	1.000	1.000	0.812	0.812	0.812	0.812	0.812	0.812
		o3	-	-	1.000	1.000	1.000	1.000	0.812	0.812	0.812	0.812	0.812	0.812
	Release	o1	-	-	-	1.000	1.000	1.000	0.812	0.812	0.812	0.812	0.812	0.812
		o2	-	-	-	-	1.000	1.000	0.812	0.812	0.812	0.812	0.812	0.812
		o3	-	-	-	-	-	1.000	0.812	0.812	0.812	0.812	0.812	0.812
LLVM 3.2	Debug	o1	-	-	-	-	-	-	1.000	1.000	1.000	1.000	1.000	1.000
		o2	-	-	-	-	-	-	-	1.000	1.000	1.000	1.000	1.000
		o3	-	-	-	-	-	-	-	-	1.000	1.000	1.000	1.000
	Release	o1	-	-	-	-	-	-	-	-	-	1.000	1.000	1.000
		o2	-	-	-	-	-	-	-	-	-	-	1.000	1.000
		o3	-	-	-	-	-	-	-	-	-	-	-	1.000
Statistical Values			Max.	1.000		Min.	0.812		Avg.	0.897		$\text{Acc}_{0.75}$	100%	

TABLE 7
Resiliency of DYKIS against single obfuscation

	Max.	Min.	Avg.	$\text{ACC}_{0.75}$	S/F_{P1}	S/F_{P2}
JLex	1.000	0.977	0.998	100%	34/5	34/0
JavaCUP	1.000	0.960	0.995	100%	37/2	29/8
Calculator	1.000	0.823	0.994	100%	37/2	31/6
Avrora	0.984	0.878	0.933	100%	11/28	9/2
Luindex	1.000	0.805	0.977	100%	26/13	18/8
Lusearch	1.000	0.998	0.999	100%	26/13	17/9
Antlr	0.981	0.738	0.910	94.4%	24/15	18/6

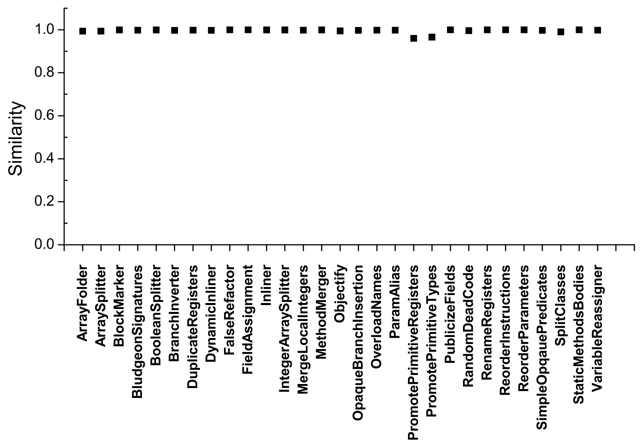

Fig. 4. Similarity scores between `JavaCUP` and its `SandMark`-obfuscated versions

are no transformations that utilize obfuscators from both categories. The specific `SandMark` obfuscators that we use in this group of experiments are listed in Table 8. The order we apply the obfuscators in each category is random.

Besides the `SandMark` obfuscators, six commercial and open source obfuscation tools, including `Zelix KlassMaster`[4], `Allatori`[5], `DashO`[6],

4. http://www.zelix.com/klassmaster
5. http://www.allatori.com

TABLE 8
Sandmark Obfuscators that are used to generate deeply obfuscated versions

	Obfuscators	JLex	JavaCUP	Calculator	Avrora	Luindex	Lusearch	Antlr
Control Obfuscation	Transparent Branch Insertion	√	√	√	√	√	√	√
	Dynamic Inliner	√	√	√				
	Method Merger	√	√	√		√	√	√
	Reorder Instructions	√	√	√		√	√	√
	False Refactor	√	√	√	√	√		
	Branch Inverter	√	√	√	√		√	√
	Simple Opaque Predicates	√	√	√		√		
Data Obfuscation	Array Folder	√	√	√				
	Integer Array Splitter	√	√	√				
	Promote Primitive Registers	√	√	√		√	√	√
	Variable Reassigner	√	√	√	√	√		
	Duplicate Registers	√	√	√				
	Merge Local Integers	√	√	√			√	√
	Boolean Splitter	√	√	√	√	√	√	√

`JShrink`[7], `ProGuard`[8] and `RetroGuard`[9], that support renaming, encryption and control flow obfuscations are also selected. Under the requirement of semantic equivalence between the original and the transformed programs, we turn on as many obfuscators as possible for each tool. Semantic equivalence is confirmed via empirical study rather than theoretical proof. We consider a transformed program is equivalent to the original one if they produce the same outputs in our experiments. Together with the two categories of `SandMark`, such experimental setup can produce up to eight deeply obfuscated versions for each program. However, not all transformations are successful. For example, the `RetroGuard` fails to obfuscate the four Java programs from the Dacapo benchmark, while `GCJ` fails to compile the `Allatori`-obfuscated versions of the four programs into executables.

Similarity scores are calculated between each original program and its successfully obfuscated versions. Table 9 shows the experimental results, where column headings are abbreviations for `SandMark_Control`, `SandMark_Data`, `KlassMaster`, `Allatori`, `DashO`, `JShrink`, `ProGurad`, and `RetroGuard`, respectively. Table cells marked with "-" indicate failed transformations. It can be observed that all but two similarity scores are above 0.89, which indicates that DYKIS is resilient even to rather complex obfuscations.

4.4 Resilience to cross-platform plagiarisms and binary obfuscation techniques

In this section, we evaluate the resilience of DYKIS against another two possible plagiarism scenarios. In the first scenario a plagiarist steals a program in a platform (e.g. Linux) and then compiles and distributes it in another platform (e.g. Windows). In the second scenario a plagiarist generates copies using binary obfuscation techniques [14], [35]–[37] implemented in specialized binary obfuscators and packing tools.

TABLE 9
Resilience against multiple obfuscations

	SC	SD	KM	AT	DO	JS	PG	RG
JLex	1.0	0.98	1.0	0.977	1.0	1.0	1.0	1.0
JavaCUP	0.996	0.975	0.988	0.987	0.988	0.978	1.0	1.0
Calculator	1.0	1.0	1.0	1.0	**0.592**	1.0	1.0	1.0
Avrora	0.947	0.999	0.936	-	0.898	0.921	0.92	-
Luindex	0.998	1.0	0.957	-	0.986	1.0	0.997	-
Lusearch	1.0	1.0	0.998	-	0.999	1.0	1.0	-
Antlr	**0.599**	0.939	0.965	-	0.975	-	0.903	-

4.4.1 Resilience to cross-platform plagiarisms

Cross-platform plagiarism is common because it does not require too much efforts and can effectively evade platform-dependant plagiarism detection techniques. For example, system call based birthmarking SCSSB [12] is ineffective in detecting cross-platform plagiarisms because different system calls are used depending on whether the program is executed in Windows or Linux. This problem may be alleviated by maintaining a mapping between system calls in different platforms. However, creating such mapping is not easy, since it is labor intensive and the mapping may be incorrect due to various reasons. For example, documentations about system calls in Windows is incomplete [38].

In order to evaluate the resilience of DYKIS against cross-platform plagiarisms, we use experimental subjects that have both Linux and Windows versions. They include the two compression programs `bzip2-1.0.2` and `gzip-1.2.4`, the five `openssl` programs and the `UPX`[10] packing tool. These programs are compiled with `gcc` under Linux, and with the Microsoft C/C++ compiler `cl` under Windows. The

6. https://www.preemptive.com/products/dasho
7. http://www.e-t.com/jshrink.html
8. http://proguard.sourceforge.net
9. http://java-source.net/open-source/obfuscators/retroguard
10. http://upx.sourceforge.net/

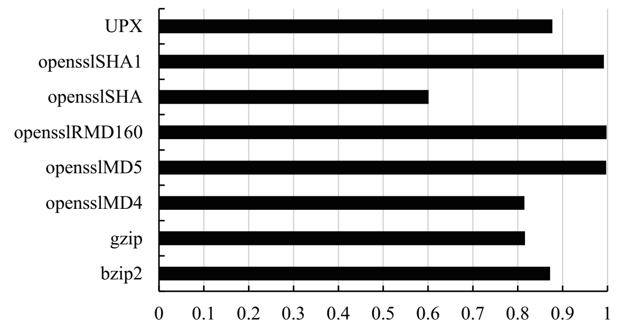

Fig. 5. Similarity scores between the Linux and Windows versions

experimental results are illustrated in Figure 5, where the similarity scores are calculated between the Linux and Windows version of each program. It can be observed that most scores are greater than the threshold 0.75 (1-0.25). On the other hand, similarity varies for different programs depending on the degree of dependency on platforms. For example, the similarity score between the two versions of `opensslMD5` is 0.997, while the score between the two versions of `opensslSHA` is just 0.601. We believe DYKIS can be used to detect cross-platform plagiarism as long as the programs do not have too many platform-specific operations such as system calls. Otherwise we need to modify the dynamic key instructions to exclude platform-specific operations.

4.4.2 Resilience to binary obfuscation techniques

Binary obfuscation techniques [14], [35]–[37] are widely used to hide the maliciousness of malware or to prevent illegal modification of software. The same techniques can be used to hinder plagiarism detection. Despite many binary obfuscation techniques are proposed, few binary code obfuscators are publicly available. The one that we have successfully downloaded is `binobf`[11]. On the other hand, many binary obfuscation techniques have been incorporated into packing tools [14], [39], [40]. These packers incorporate other techniques such as compression and encryption. By significantly modifying the copies, these packers can defeat most static software birthmarks.

In this group of experiments, we firstly evaluate DYKIS against the the binary obfuscator `binobf` that implements strong obfuscation techniques [35], [36]. We have managed to generate obfuscated versions for three programs: `bzip2`, `gzip` and `md5sum`[12]. After obfuscation, the size of the three programs increases about 18, 25 and 94 times, respectively. Despite such significant changes, the similarity scores between

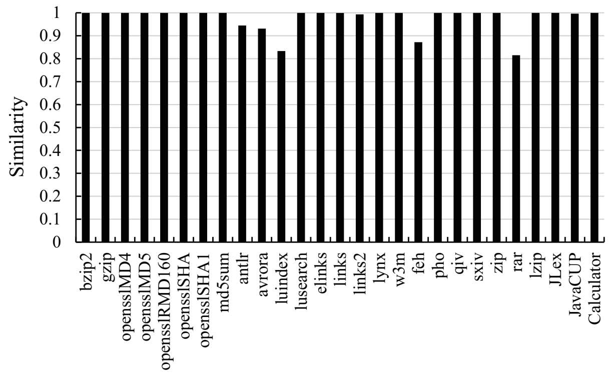

Fig. 6. Similarity scores between a program and its UPX-packed version

`bzip2`, `gzip` and `md5sum` and their obfuscated versions are 0.778, 1.0 and 0.999, respectively. The experiments confirm that DYKIS correctly recognizes the copies obfuscated by `binobf`.

Next we evaluate DYKIS against various packing tools that also implement binary obfuscations. The only publicly available packing tool that we know for the ELF-format, executable file format under Linux, is `UPX`. On the other hand, packing tools for the PE-format, executable file format under Windows, are abundant. Thus under Linux, we utilize `UPX` to pack all the ELF binaries in our benchmarks as listed in Table 2. Figure 6 depicts the similarity scores between each program and its corresponding version processed by `UPX`. It can be observed that all similarity scores are above the threshold 0.75. As for Windows, we select six widely used PE-format packers that include `ASProtect`[13], `Fsg`[14], `Nspack`[15], `PECompact`[16], `WinUpack`[17], and `UPX` (Windows version). These packers are applied to obfuscate the PE-format binaries of `bzip2`, `gzip` and five `openssl` programs. Not all transformations are successful. For example `ASProtect` and `WinUpack` are not able to process `bzip2`. For those binaries that can be successfully transformed, we compute the similarity scores between the original programs and their packed versions. All the similarity scores are 1.0, which indicates strong resilience of DYKIS against packing tools.

4.5 Similarity between successive releases of the same program

The successive releases of a software is likely to be similar to each other. Comparing similarity between consecutive releases can be a good robustness indicator of DYKIS birthmarks. We choose five releases

11. http://www.cs.arizona.edu/ debray/binary-obfuscation/
12. There are several limitations to use `binobf`. For example, it cannot handle binaries produced by `gcc` newer than version 3, and it requires the input binaries to be statically linked but still relocatable.
13. http://www.aspack.com/asprotect64.html
14. http://fsg.soft112.com
15. http://nspack.download-230-13103.programsbase.com
16. http://bitsum.com/pecompact
17. http://de.wikipedia.org/wiki/Upack

TABLE 10
Similarity scores between different releases of `gzip`

	gzip1.2.4a	gzip1.2.4	gzip1.3.13	gzip1.4	gzip1.5
gzip1.2.4a	1.000	1.000	0.995	0.992	0.997
gzip1.2.4	-	1.000	0.995	0.992	0.997
gzip1.3.13	-	-	1.000	0.997	0.992
gzip1.4	-	-	-	1.000	0.995
gzip1.5	-	-	-	-	1.000

TABLE 11
Similarity scores between different releases of `feh`

	feh1.4	feh1.12	feh2.3	feh2.9	feh2.10
feh1.4	1.000	1.000	0.887	0.872	0.735
feh1.12	-	1.000	0.887	0.872	0.735
feh2.3	-	-	1.000	0.759	0.847
feh2.9	-	-	-	1.000	0.863
feh2.10	-	-	-	-	1.000

of `gzip` and compile all of them with `gcc 4.6` and optimization level `O2` as our experimental subjects.

As illustrated in Table 10, the average similarity scores between different releases of `gzip` are all above 0.99. The high similarity is due to the fact that `gzip` is a relatively mature software with focused functionality. We further compare two different adjacent versions of two image processing software: `feh2.9.1` versus `feh2.9.2`, and `qiv2.2.3` versus `qiv2.2.4`. The similarity scores are 0.994 and 0.98 respectively. By examining the upgrade reports of `qiv2.2.4` and `feh2.9.2`, we find that the newer releases just fix several bugs submitted in the previous releases, indicating few code changes. For `feh`, similarity scores between 5 major releases published from year 2010 to 2014 are further calculated and summarized in Table 11. The scores become relatively lower since many slight changes are made for each yearly published version.

4.6 Credibility

In this section, we compare independently developed programs to evaluate the credibility of DYKIS. Three types of software are selected, including four image processing programs, five compression and decompression programs, and six encryption and decryption programs. To give an overall view, Figure 7 depicts the basic distribution of the similarity scores between pairs of different programs. The x-axis represents the similarity ranges between DYKIS birthmarks, and the y-axis represents the percentage of birthmarks pairs that belong to each range. For example, range 10-19 means a similarity score between 0.10 and 0.19. It can be observed that about 90 percent similarity scores are below the threshold 0.25, indicating very few incorrect classifications.

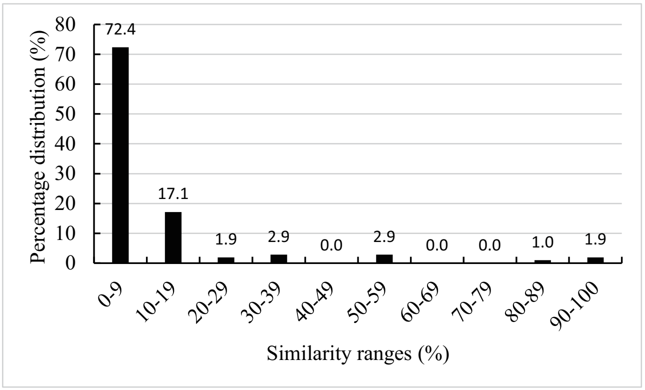

Fig. 7. Distribution of similarity scores between independently developed programs

4.6.1 Similarity between independent programs in the same category

The comparison data are presented in green, yellow and pink colors in Table 12. The data give mixed results, with similarity scores ranging from 0 to 0.93. Following is our explanation.

- Except for the `gzip-zip` pair, the similarity scores of the compression programs are close to 0. The low similarity scores exhibited by comparisons between `gzip`, `bzip`, `rar` and `lzip` are due to the fact that each program adopts a different compression algorithm. On the other hand, `gzip` and `zip` are both based on the same compression algorithm *deflate* that is implemented in the `zLib` library. The high similarity score 0.811 is because `gzip` contains code from `zLib` and `zip` is dynamically linked to system-wide `zLib` [18].
- The similarity scores between the encryption/decryption programs are slightly higher but most are still below 0.25. This is because these programs share the same front-end, even though their kernel modules are implemented differently to realize different types of encryption and decryption. The similarity score between `md5sum` and `opensslMD5` is around 0.9, since they both simply implement the `MD5` algorithm.
- The relatively high similarity scores between the image processing software can be explained by many shared image processing libraries. Image processing libraries constitute a major component of these programs because few instructions remain once we remove instructions from the libraries. The sharing of image processing libraries is confirmed by checking the dependencies of each program with the `apt-cache depends` command. The high similarity achieved by the `pho-qiv` pair is because the dependencies of `pho` are totally included in those of `qiv`. The relatively low similarity of the `qiv-feh` pair is because `qiv` is implemented mainly on top of `imlib2` and `gtk2`, while `feh` is based on `imlib2` and

TABLE 12
Similarity scores between independently developed programs

Category		Compression and Decompression					Image Processing				Encryption and Decryption					
		gzip	bzip2	zip	rar	lzip	qiv	feh	pho	sxiv	md5 sum	OL-MD4	OL-MD5	OL-RMD160	OL-SHA	OL-SHA1
Compression and Decompression	gzip	1.000	0.000	**0.811**	0.058	0.000	0.000	0.000	0.000	0.000	0.000	0.003	0.003	0.002	0.003	0.004
	bzip2	-	1.000	0.000	0.000	0.000	0.036	0.033	0.041	0.118	0.000	0.000	0.000	0.000	0.000	0.000
	zip	-	-	1.000	0.077	0.000	0.011	0.025	0.012	0.015	0.001	0.000	0.000	0.000	0.000	0.000
	rar	-	-	-	1.000	0.000	0.084	0.129	0.099	0.072	0.000	0.000	0.006	0.013	0.009	0.000
	lzip	-	-	-	-	1.000	0.007	0.002	0.023	0.002	0.000	0.000	0.000	0.000	0.000	0.000
Image Processing	Qiv	-	-	-	-	-	1.000	**0.302**	**0.930**	**0.462**	0.034	0.145	0.128	0.076	0.107	0.185
	feh	-	-	-	-	-	-	1.000	**0.301**	**0.478**	0.000	0.000	0.003	0.006	0.004	0.000
	pho	-	-	-	-	-	-	-	1.000	**0.430**	0.015	0.131	0.107	0.068	0.096	0.173
	sxiv	-	-	-	-	-	-	-	-	1.000	0.013	0.121	0.099	0.064	0.089	0.156
Encryption and Decryption	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
Statistical Values		**Max.**			**0.930**	**Min.**		**0.000**		**Avg.**		**0.093**		**ACC$_{0.25}$.**		**90.5%**

several other libraries such as `imagemagick` and `giblib1`. With only five out of eighteen shared dynamic libraries, their similarity score is 0.302.

Although initially puzzled at the wide range of similarity scores, we find that the scores match the documentation and our source code analysis. Incorrect classifications, as indicated by the bold fonts in Table 12, occur mainly because the corresponding program pairs share parts of the program code or implement exactly the same algorithms. The validity of this group of experiments can be further verified by manual adjustment based on our findings. For example, we are able to reduce the similarity scores between encryption programs by detaching `PIN` from their front-end. However, we believe such manual efforts are not desirable in practice so we choose not to present the data.

4.6.2 *Similarity between independent programs in different categories*

The blue area in Table 12 gives the comparison results between the software in different categories. The data show that all the similarity scores are below the threshold 0.25, indicating no incorrect classifications.

4.7 Comparison With SCSSB Birthmarks

This section compares DYKIS against SCSSB [12], an existing birthmark extracted from dynamic system call sequence, with respect to three performance metrics `URC`, `F-Measure` and `MCC`. `URC` measures resilience and credibility, while the other two are more comprehensive metrics introduced for amending the problem of `URC` that focuses on the rate of correct classifications. All the pairs of programs from Section 4.2 to Section 4.6 are taken as the experimental subjects, except those in Section 4.4.1 because SCSSB cannot detect cross-platform plagiarisms.

4.7.1 *Performance evaluation with respect to URC*

As discussed earlier resilience and credibility reflect different qualities of a birthmark. `URC` (Union of Resilience and Credibility) [25], defined below, is a metric proposed to evaluate birthmarks that considers both aspects.

$$\texttt{URC} = 2 \times \frac{R \times C}{R + C}, \tag{3}$$

where R represents the ratio of correctly classified pairs where plagiarism exists and C represents the ratio of correctly classified pairs that plagiarism does not exist. The value of `URC` ranges from 0 to 1, with higher value indicating a better birthmark. Let EP be the set of pairs of programs such that $\forall(p,q) \in EP$, p is a copy of q, and JP be the set of pairs such that $\forall(p,q) \in JP$, a plagiarism detection tool *claims* that p is a copy of q. Similarly, let EI be the set of pairs such that $\forall(p,q) \in EI$, p and q are independent, and JI be the set of pairs that are *claimed* independent by a plagiarism detection tool. R and C are formally defined as:

$$R = \frac{|EP \cap JP|}{|EP|} \quad \text{and} \quad C = \frac{|EI \cap JI|}{|EI|}.$$

Figure 8 depicts the experimental results, where the red and blue lines denote the data for DYKIS and SCSSB, respectively. Obviously the threshold value ε has an impact on plagiarism detection. In the extreme case of 0, no conclusion can be made because a similarity score has to be greater than 1 to claim plagiarism and has be to smaller than 0 to claim the opposite. Therefore in our experiments we vary the value of ε from 0 to 0.5, as shown in the x-axis. Note that ε cannot be greater than 0.5, otherwise plagiarism can be claimed to exist and non-exist at the same time. As illustrated in Figure 8, DYKIS performs better than SCSSB.

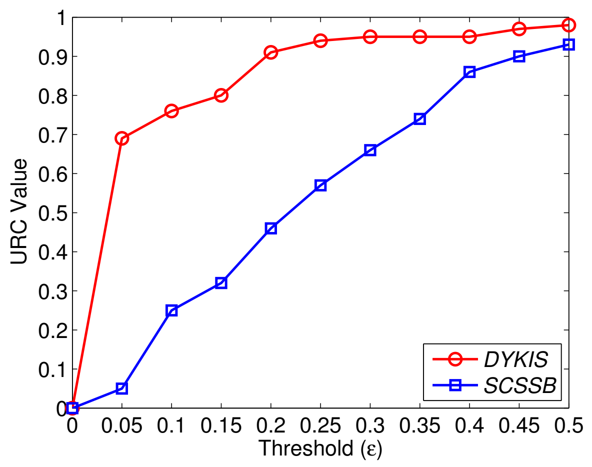

Fig. 8. DYKIS vs. SCSSB with respect to URC.

4.7.2 *Performance evaluation with F-Measure and MCC metrics*

Equation 1 indicates that the birthmark-based plagiarism detection methods give three-value results. That is, if the similarity score between two birthmarks is between ϵ and $1-\epsilon$, there is no definite answer whether plagiarism exists or not. The inconclusiveness reflects the nature of birthmark-based techniques, which are mostly used to collect evidence rather than prove or disprove the existence of plagiarism. Such three-value outcome explains the reason why in Figure 8 URC gives better results with higher value of ϵ. This is because URC mainly measures the rate of correct classifications, while inconclusiveness is considered an incorrect classification. As the value of ϵ increases, the chance of inconclusiveness becomes smaller, leading to less number of incorrect classifications. In order to study the impact of the threshold on incorrect classification, we formally define the inconclusive ratio (InconRatio) and false classification ratio (FCRatio) as:

$$InconRatio = \frac{|EP| + |EI| - |JP| - |JI|}{|EP| + |EI|}$$

$$FCRatio = \frac{|EP| + |EI| - |EP \cap JP| - |EI \cap JI|}{|EP| + |EI|}$$

Figure 9 depicts how the ratios of inconclusiveness and false classification change by varying the value of ϵ from 0 to 0.5. It can be observed that DYKIS has less inconclusiveness and false classification than SCSSB. At the same time, both ratios of DYKIS and SCSSB decrease as ϵ increases. This clearly indicates that URC favors large ϵ values. On the other hand, as pointed out by Schuler [11], a smaller ϵ is desired in practice.

In order to address the problem, we further compare DYKIS and SCSSB with two other metrics, F-Measure and MCC (Matthews Correlation Coefficient) [41], that are widely used in the areas of information retrieval and machine learning. However, the two metrics cannot be directly applied as they can measure binary classifications only. In the following we revise the definition of sim by removing inconclusiveness:

$$sim\left(p_{\mathcal{B}}, q_{\mathcal{B}}\right) = \begin{cases} \geq \varepsilon & p \text{ is a copy of } q \\ < \varepsilon & p \text{ is not a copy of } q \end{cases} \tag{4}$$

F-Measure is based on the weighted harmonic mean of $Precision$ and $Recall$:

$$\texttt{F-Measure} = \frac{2 \times Precision \times Recall}{Precision + Recall}, \tag{5}$$

where $Precision$ and $Recall$ are defined as:

$$Precision = \frac{|EP \cap JP|}{|JP|} \quad \text{and} \quad Recall = \frac{|EP \cap JP|}{|EP|}$$

MCC, defined below, is regarded as one of the best metrics that evaluate true and false positives and negatives by a single value.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{D}} \quad and \tag{6}$$

$$D = (TP + FP)(TP + FN)(TN + FP)(TN + FN), \tag{7}$$

where TP, TN, FP and FN are the number of true positives, true negatives, false positives and false negatives, respectively. They can be computed using the following formulas:

$$TP = |EP \cap JP|; \qquad FN = |EP \cap JI|$$
$$FP = |EI \cap JP|; \qquad TN = |EI \cap JI|$$

The left and right sub-figures in Figure 10 depict the experimental results with respect to F-Measure and MCC, respectively. The data for DYKIS are shown in red while the data for SCSSB are shown in blue. It can be observed that DYKIS always performs better than SCSSB except when $\epsilon = 0.85$.

4.8 Impact of Inputs

As discussed in Section 3.3, plagiarism detection between two programs is based on the average of similarity scores calculated under multiple inputs. Since in most cases an exhaustive testing of all inputs is not possible, how to choose inputs to improve the efficiency and validity of plagiarism detection becomes a valid research question. While this question is out of scope of this paper, in this section we evaluate the impact of different types of inputs on plagiarism detection. For example, many image processing software can take inputs of various file formats such as jpeg, png, and bmp because different parse routines are used to process different types of inputs. If our plagiarism detection gives different results under different types of inputs, we say DYKIS is sensitive to inputs; otherwise it is not. Note that during comparison, each pair of programs is still taking the same inputs. However, we run the comparison multiple times with different types of inputs.

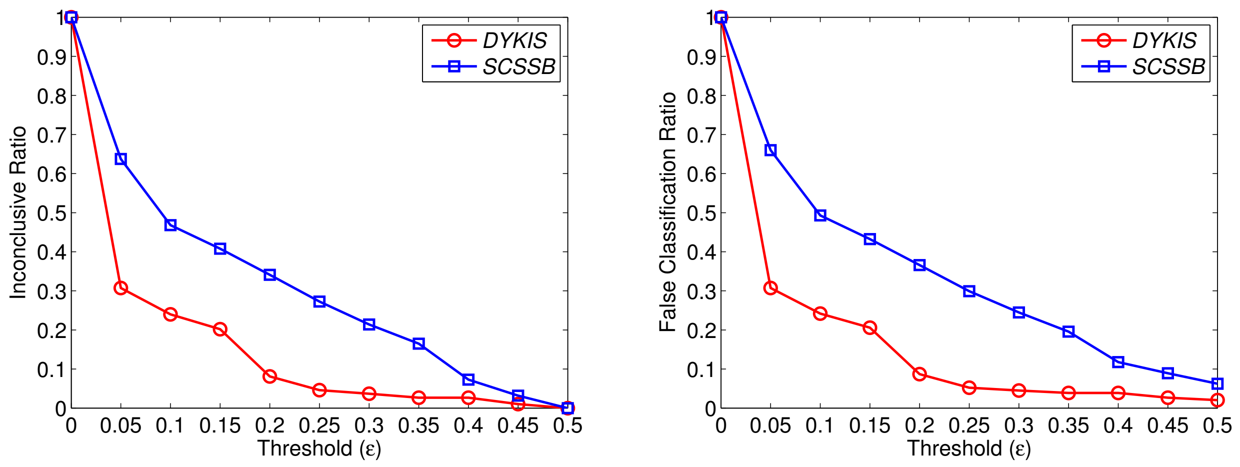

Fig. 9. The inconclusiveness and false classification ratios of DYKIS and SCSSB.

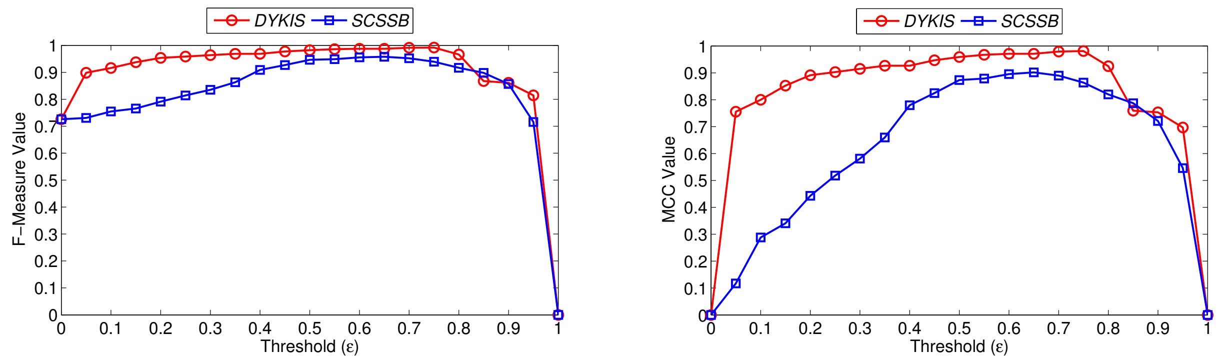

Fig. 10. DYKIS vs. SCSSB with respect to `F-Measure` and `MCC`.

TABLE 13
Standard deviation of similarity scores for each program pair executed with multiple inputs

name	bzip2	gzip	zip	lzip	md5sum	OL-MD4	OL-MD5	OL-RMD160	OL-SHA	OL-SHA1
bzip2	0	0	0	0.0106	0	0	0	0	0	0
gzip	-	0	0.0343	0	0	0	0	0	0	
zip	-	-	0	0	0.0002	0	0	0	0	0
lzip	-	-	-	0	0	0	0	0	0	0
md5sum	-	-	-	-	0	0.0102	0.0130	0.0009	0.0142	0.0005
OL-MD4	-	-	-	-	-	0	0.0355	0.0236	0.0298	0.1093
OL-MD5	-	-	-	-	-	-	0	0.0120	0.0184	0.0682
OL-RMD160	-	-	-	-	-	-	-	0	0.0101	0.0328
OL-SHA	-	-	-	-	-	-	-	-	0	0.0483
OL-SHA1	-	-	-	-	-	-	-	-	-	0

In this group of experiments we select 10 programs, including `bzip2`, `gzip`, `zip`, `lzip`, `md5sum`, `openssl`, `MD4`, `MD5`, `RMD160`, `SHA` and `SHA1`, that can handle different types of input files. We calculate the standard deviation σ of the similarity scores for different inputs. Table 13 gives the experimental results, from which we observe that the standard deviation for each pair of programs under comparison is very small. It indicates that the similarity scores between two programs do not vary significantly under different inputs.

5 Threats to Validity

Two obvious attacks to DYKIS are noise injection and instruction rearrangement. By injecting different noise instructions during executions, DYKIS may compare these instructions and report false negatives. We argue that such strategy will not succeed because the noise instructions are not key instructions, otherwise the program logic will be changed. As a result, the noise instructions are removed from the dynamic key instruction sequence. In fact, the obfuscators `binbof` and `Sandmark` implement noise injection, and our experimental results confirm that DYKIS is resilient to this type of attack. In the second attack, a plagiarist may attempt to change the dynamic key instruction sequence by adjusting the order of executed instructions. However, due to the complex control and data dependencies among instructions, evading plagiarism by such approach is almost impossible. Light-weight rearrangement will not cause observable impact to the similarity scores, while heavy-weight rearrangement will affect the semantics of the original program. The obfuscators used in our experimental evaluation section, as well as the compiler optimization levels, perform instruction reordering to some extent. As indicated by the experimental results, DYKIS is robust against them.

DYKIS relies on dynamic taint analysis to extract key instructions. As a result, a plagiarist may exploit the weakness of current taint analysis techniques [42], to evade detection. Consider the statement `x=input`, where `x` is data-dependent on `input`. It can be transformed into the following code snippet:

```
if (input==0)
   x=0;
else if (input == 1)
   x=1;
...
else if (input == N)
   x=N;
```

The compound `if` statement also implements `x=input`. However, by using the `if` statement `x` is no longer data-dependent on `input`. Since most existing taint analysis techniques do not consider control dependency, `x` is not tainted, i.e. not related to program inputs. In such case the assignments to `x` are not considered key instructions in our approach, which leads to false negatives. This problem will be alleviated by the advances in taint analysis techniques. For example, several recent approaches [43], [44] have been proposed to consider control dependencies in taint propagation.

DYKIS is suitable for whole program plagiarism detection, where a complete program is copied and then disguised through various automatic transformations or manual modifications without changing the semantics of the original programs. An example of whole program plagiarism is cheating on homework where a student copies another student's program and then make slight changes such as renaming variables and reordering program statements [8], [45]. In recent years, whole program plagiarism on mobile markets start to rise [46], where plagiarists tag their own names or embed advertisements in stolen mobile applications (apps). Many of the stolen apps have been processed with code obfuscations to evade plagiarism detection. According to a recent study [47], about 5% to 13% of apps in the third-party app markets are copied and redistributed from the official Android market.

Besides whole program plagiarism, there exist many cases that only part of a program is copied. Unfortunately DYKIS, same as other dynamic birthmarks, cannot be directly applied to such partial program plagiarism detection. Because it relies on the comparison of dynamic key instruction sequences, DYKIS will give very low similarity scores if only partial sequences are the same between two programs. That is, if there is a small portion of a code being copied, DKYIS is not able to detect such plagiarism. A straightforward solution is to extract only those key instructions from the suspected components. But this requires manual efforts and domain knowledge.

6 Related Work

6.1 Software Watermarking

Software watermarking [3] is one of the earliest and most well-known approaches to software plagiarism detection. Watermarks are classified into four main types according to their functionality [48]: authorship mark (watermarks for identifying authors), fingerprinting mark (watermarks for identifying the serial number or purchaser), validation mark (watermarks for verifying that the software is still the same as it has been authored) and licensing mark (watermarks for controlling how the software can be used). Watermarks are highly susceptible [7], [49] to semantics-preserving obfuscations [13]. It is believed that a sufficiently determined attacker will eventually be able to defeat any watermark [4].

6.2 Static Source Code Based Birthmarks

Four types of static birthmarks were proposed by Tamada [5] that include constant values in field variables, sequence of method calls, inheritance structure and used classes. The average similarity scores of the four birthmarks are used to determine plagiarism. These birthmarks are vulnerable to obfuscations and are only applicable to Java programs. Birthmarks proposed by Prechelt and Ji [8], [9] were computed with token sequences generated by parsing source code. Such approaches are weak to junk code insertion and statement reordering. Plagiarism was determined by mining program dependency graphs (PDGs) in Gplag [10] and similarity between PDGs was calculated by graph isomorphism algorithms. By taking control and data flow relations into account, as we also do in this paper, the method showed better robustness against semantics-preserving code transformations. However, source code is required and similarity calculation based on graph isomorphism is costly.

6.3 Static Binary Code Based Birthmarks

Myles and Collberg [7] proposed k-gram based static birthmarks, each of which was a set of Java bytecode sequences of length k. The similarity between two birthmarks was calculated through set operations that ignore the frequency of elements in the set. Although being more robust than birthmarks proposed by Tamada [5], the birthmarks were still vulnerable to code transformation attacks. Weighted k-gram based static birthmarks [25] improved upon Myles and Collberg's [7] by taking the frequency of each k-length operation code sequence into consideration. However, the improvement in detection ability seems minor while introducing extra cost in computing change rate of k-gram frequencies. A static birthmark-based on windows API calls disassembled from executables was proposed by Choi [16] to detect plagiarism of windows applications. The requirement for deobfuscating binaries before applying their method is too restrictive and thus reduces its availability. Lim [33] used control flow information that reflected runtime behaviors to supplement static approaches. Recently he proposed to analyze stack flows obtained by simulating operand stack movements to detect copies [30]. Yet they are only available to Java programs. Hemel et. al. [50] suggested three methods to find potential cloned binaries within a program repository by simply treating binaries as normal files. Specifically, similarity between two binaries were evaluated by calculating the ratio of shared string literals, by calculating the compression ratio, and by computing binary deltas. Since no syntactic or semantic attributes of binary executables are considered, low detection accuracy is expected. An obfuscation-resilient method based on longest common subsequence of semantically equivalent basic blocks was proposed by Luo et. al. [51]. They utilized symbolic execution to extract from basic blocks symbolic formulas, whose pairwise equivalence were checked via a theorem prover. Yet this method has difficulty in handling indirect branches. In addition, symbolic execution combined with theorem proving is not scalable.

6.4 Dynamic Software Birthmarks

Myles and Collberg [6] suggested to use the complete dynamic control graph of an execution as a birthmark. Even with compression our study shows that such method does not scale. Schuler [11] treated Java standard API call sequences at object level as birthmarks for Java programs. The same principle was applied in Tamada's works [15], [52] where API call sequences of windows executables were used to derive birthmarks. Apparently API based birthmarks are all language dependent. To address the problem Wang et. al. [12] proposed two dynamic birthmarks based on system calls: System Call Short Sequence Birthmark (SCSSB) and Input Dependent System Call Subsequence Birthmark (IDSCSB). SCSSB treated the sets of k-length system call sequences as birthmarks. IDSCSB was introduced to avoid system call insertion attack. However both birthmarks have limited applicability to software that has few system calls, such as scientific computing programs.

In Lu's work [17], a complete dynamic instruction trace was recorded during program execution, from which a dynamic birthmark was extracted by applying the k-gram algorithm. However such birthmark could not even identify two versions generated from the same program with different compiler optimization levels. By treating the slice rather than the whole instruction sequence as program characterizations, Bai [26] proposed a dynamic birthmark for Java based on MSIL instructions rather than assembly instructions. Besides being language and operating system dependent, their approach is also weak against code obfuscations.

By introducing data flow and control flow dependency analysis, Wang et. al. [23] proposed a system call dependency graph based birthmark, and graph isomorphism is utilized for calculating similarity between birthmarks. Patrick et al. [22] proposed a heap graph birthmark for JavaScript utilizing heap memory analysis, and graph monomorphism algorithm was applied for similarity computation. But to be effective, these graph based birthmarks require that the programs under protection to have prominent referencing structures. Also, since graph isomorphism and monomorphism algorithms are NP-complete in general, several thousand nodes will make the methods impractical to use.

6.5 Related Work in Other Domains

One of the most relevant research fields to software plagiarism detection is clone detection, which aims to find duplicate code within a single program to improve software maintenance, program comprehension, and software quality. With such purpose most clone detection algorithms operate on source code only. There exist several mature systems [53]–[56] that are able to detect clones accurately on large scale software. Similar to plagiarism detection, clone detection identifies cloned fragments by abstracting the program into a set of characteristics. The abstraction can be categorized into the following types: String-based [57], [58], Token-based [53], [54], AST-based [56], [59]–[61], PDG-based [62]–[64], Behavior-based [65], and Memory-State-based [55]. Sebjornsen et. al. [66] proposed a clone detection algorithm for binary executables by converting assembly instructions to a higher level representation. Clone detection techniques may be used for plagiarism detection. However, existing clone detection approaches are fragile against code obfuscations [12]. String-based schemes are weak to even simple identifier renaming. AST-based schemes are weak against reordering and control replacement. Token-based approaches are easily defeated by junk code insertion and the others are vulnerable to transformations such as function inlining and outlining.

Malware detection aims to determine unknown attributes of a suspicious program against a set of programs by extracting features and performing classification or clustering. The techniques can generally be classified into two types according to program features applied: signature-based [67]–[70] and behavior-based [71]–[73]. However, due to the uniqueness of malware samples, these techniques can not be applied directly to software plagiarism detection. Researches in code based search engine [20], [74]–[77] also involve program characterization. However these techniques mainly target small source code snippets and merely consider obfuscations, since their goal is to retrieve certain piece of code to assist development.

7 Conclusion and Future Work

In this paper we have proposed a dynamic birthmark called DYKIS, which we believe is resilient to weak and strong code obfuscations. Based on its definition we have implemented algorithms to extract such birthmarks from binary executables and to compare their similarities using cosine distance. Our intensive experiments on 342 versions of 28 different programs indicate our approach is efficient and effective. The benchmarks are available online at
`http://labs.xjtudlc.com/labs/wlaq/benchmark.html`
To the best of our knowledge this is the first publicly available benchmark suite for software plagiarism detection.

Our implementation leads to a software plagiarism detection tool DYKIS-PD that has been successfully demonstrated at a conference [27]. DYKIS-PD is publicly available for download at
`http://labs.xjtudlc.com/labs/wlaq/dbpd/site/`

In recent years, whole program plagiarism of mobile apps has become a serious problem. About 5% to 13% of apps in the third-party app markets are copied and redistributed from the official Android market. We plan to conduct case studies and optimize DYKIS for this domain. In addition, DKYIS is suitable for whole program plagiarism detection. We will explore whether DKYIS can be adapted to detect partial program plagiarisms.

Acknowledgement

This work extends our previous conference paper published at IEEE HPCC. The research was supported in part by National Science Foundation of China under grants (91118005, 91218301, 61221063, 61428206, 61203174, 91418205, 61472318), Ministry of Education Innovation Research Team (IRT13035), Key Projects in the National Science and Technology Pillar Program of China (2013BAK09B01), the Fundamental Research Funds for the Central Universities, and the National Science Foundation (NSF) under grant CCF-1500365. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] "http://sourceauditor.com/blog/tag/lawsuits-on-open-source/."

[2] "http://www.martinsuter.net/blog/2009/08/skype-joltid-licensing-dispute-epic-ma-screwup.html."

[3] C. Collberg and C. Thomborson, "Software watermarking: Models and dynamic embeddings," in *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '99)*. ACM, 1999, pp. 311–324.

[4] C. S. Collberg, E. Carter, S. K. Debray, A. Huntwork, J. D. Kececioglu, C. Linn, and M. Stepp, "Dynamic path-based software watermarking," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '04)*, 2004, pp. 107–118.

[5] H. Tamada, M. Nakamura, and A. Monden, "Design and evaluation of birthmarks for detecting theft of Java programs," in *IASTED Conf. on Software Engineering (IASTEDSE '04)*, 2004, pp. 569–574.

[6] G. Myles and C. S. Collberg, "Detecting software theft via whole program path birthmarks," in *Proc. Int. Conf. Information Security (ISC '04)*, 2004, pp. 404–415.

[7] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proc. ACM Symp. Applied Computing (SAC '05)*, 2005, pp. 314–318.

[8] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with jplag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.

[9] J.-H. Ji, G. Woo, and H.-G. Cho, "A source code linearization technique for detecting plagiarized programs," in *Proc. Annual SIGCSE Conf. Innovation and Technology in Computer Science Education (ITiCSE '07)*, 2007, pp. 73–77.

[10] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD '06)*, 2006, pp. 872–881.

[11] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for Java," in *Proc. IEEE/ACM Int. Conf. Automated Software Engineering (ASE '07)*, 2007, pp. 274–283.

[12] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Detecting software theft via system call based birthmarks," in *Annual Computer Security Applications Conference (ACSAC '09)*, 2009, pp. 149–158.

[13] C. Collberg, G. Myles, and A. Huntwork, "Sandmark-a tool for software protection research," *IEEE Security and Privacy*, vol. 1, no. 4, pp. 40–49, 2003.

[14] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys*, vol. 46, no. 4, 2013.

[15] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. Ichi Matsumoto, "Design and evaluation of dynamic software birthmarks based on api calls," *Info. Science Technical Report NAIST-IS-TR2007011*, pp. 0919–9527, 2007.

[16] S. Choi, H. Park, H. il Lim, and T. Han, "A static api birthmark for windows binary executables," *Journal of Systems and Software*, vol. 82, no. 5, pp. 862–873, 2009.

[17] B. Lu, F. Liu, X. Ge, B. Liu, and X. Luo, "A software birthmark based on dynamic opcode n-gram," in *Int. Conf. Semantic Computing (ICSC '07)*, 2007, pp. 37–44.

[18] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *Proc. Int. Conf. Softw. Eng. (ICSE '11)*, 2011, pp. 756–765.

[19] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *ISOC Symp. Network and Distributed System Security (NDSS '05)*. Internet Society, 2005.

[20] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Proc. Int. Conf. Softw. Eng. (ICSE '12)*, 2012, pp. 364–374.

[21] Z. Tian, Q. Zheng, T. Liu, and M. Fan, "DKISB: Dynamic key instruction sequence birthmark for software plagiarism detection," in *IEEE Int. Conf. High Performance Computing and Communications. (HPCC '13)*, 2013, pp. 619–627.

[22] P. P. F. Chan, L. C. K. Hui, and S.-M. Yiu, "Heap graph based software theft detection," *IEEE Trans. Information Forensics and Security*, vol. 8, no. 1, pp. 101–110, 2013.

[23] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proc. ACM Conf. Computer and Communications Security (CCS '09)*. ACM, 2009, pp. 280–290.

[24] K. Fukuda and H. Tamada, "A dynamic birthmark from analyzing operand stack runtime behavior to detect copied software," in *ACIS Int. Conf. Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD '13)*, 2013, pp. 505–510.

[25] X. Xie, F. Liu, B. Lu, and L. Chen, "A software birthmark based on weighted k-gram," in *IEEE Int. Conf. Intelligent Computing and Intelligent Systems (ICIS '10)*, vol. 1, 2010, pp. 400–405.

[26] Y. Bai, X. Sun, G. Sun, X. Deng, and X. Zhou, "Dynamic k-gram based software birthmark," in *Proc. Austr. Softw. Eng. Conf. (ASWEC '08)*, 2008, pp. 644–649.

[27] Z. Tian, Q. Zheng, M. Fan, E. Zhuang, H. Wang, and T. Liu, "DBPD: A dynamic birthmark-based software plagiarism detection tool," pp. 740–741, 2014.

[28] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '05)*, 2005, pp. 190–200.

[29] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *Proc. ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environments (VEE '12)*, 2012, pp. 121–132.

[30] H. il Lim and T. Han, "Analyzing stack flows to compare Java programs," *IEICE Trans. Information and Systems*, vol. 95-D, no. 2, pp. 565–576, 2012.

[31] H. Park, H. il Lim, S. Choi, and T. Han, "Detecting common modules in Java packages based on static object trace birthmark," *Computer Journal*, vol. 54, no. 1, pp. 108–124, 2011.

[32] "GCJ, https://gcc.gnu.org/java/."

[33] H. il Lim, H. Park, S. Choi, and T. Han, "A method for detecting the theft of Java programs through analysis of the control flow information," *Information and Software Technology*, vol. 51, no. 9, pp. 1338–1350, 2009.

[34] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proc. Annual ACM SIGPLAN Conf. Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA '06)*, 2006, pp. 169–190.

[35] C. Linn and S. K. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. ACM Conf. Computer and Communications Security (CCS '03)*, 2003, pp. 290–299.

[36] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *Proc. USENIX Security Symposium (USENIX Security '07)*, 2007.

[37] M. Madou, L. V. Put, and K. D. Bosschere, "LOCO: an interactive code (de)obfuscation tool," in *Proc. ACM SIGPLAN Symp. Partial Evaluation and Semantics-based Program Manipulation (PEPM '06)*, 2006, pp. 140–144.

[38] D. Gao, M. K. Reiter, and D. X. Song, "Behavioral distance for intrusion detection," in *Int. Symp. Recent Advances in Intrusion Detection (RAID '05)*, 2005, pp. 63–81.

[39] F. Guo, P. Ferrie, and T. cker Chiueh, "A study of the packer problem and its solutions," in *Int. Symp. Recent Advances in Intrusion Detection (RAID '08)*, 2008, pp. 98–115.

[40] M.-J. Kim, J.-Y. Lee, H. Chang, S. Cho, Y. Park, M. Park, and P. A. Wilsey, "Design and performance evaluation of binary code packing for protecting embedded software against reverse engineering," in *IEEE Int. Symp. Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC '10)*, 2010, pp. 80–86.

[41] B. W. Mathews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica Et Biophysica Acta (bba) - Protein Structure*, vol. 405, pp. 442–451, 1975.

[42] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in *Proc. Int. Conf. Detection of Instrusions and Malware and Vulnerability Assessment (DIMVA '08)*, 2008, pp. 143–163.

[43] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: dynamic taint analysis with targeted control-flow propagation," in *ISOC Symp. Network and Distributed System Security (NDSS '11)*, 2011.

[44] B. Kang, T. Kim, B. Kang, E. G. Im, and M. Ryu, "TASEL: dynamic taint analysis with selective control dependency," in *Proc. Conf. Research in Adaptive and Convergent Systems (racs '14)*. ACM, 2014, pp. 272–277.

[45] G. Cosma and M. Joy, "An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis," *IEEE Trans. Computers*, vol. 61, pp. 379–394, 2012.

[46] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. ACM Conf. Security and Privacy in Wireless and Mobile Networks (WiSec '14)*, 2014, pp. 25–36.

[47] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. ACM Conf. Data and Application Security and Privacy (CODASPY '12)*, 2012, pp. 317–326.

[48] J. Nagra, C. D. Thomborson, and C. S. Collberg, "A functional taxonomy for software watermarking," in *Proc. Austr. Conf. Computer Science (ACSC '02)*, 2002, pp. 177–186.

[49] C. Collberg and C. Thomborson, "On the limits of software watermarking," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1998.

[50] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proc. Working Conf. Mining Software Repositories (MSR '11)*, 2011, pp. 63–72.

[51] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng. (FSE '14)*, 2014, pp. 389–400.

[52] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Dynamic software birthmarks to detect the theft of windows applications," in *Int. Symp. Future Software Technology (ISFST '04)*, 2004.

[53] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Software Engineering*, vol. 32, no. 3, pp. 176 – 192, 2006.

[54] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[55] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: memory comparison-based clone detector," in *Proc. Int. Conf. Softw. Eng. (ICSE '11)*, 2011, pp. 301–310.

[56] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc. Int. Conf. Softw. Eng. (ICSE '07)*, 2007, pp. 96–105.

[57] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. IEEE Int. Conf. Software Maintenance (ICSM '99)*, 1999, pp. 109–118.

[58] B. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. Working Conf. Reverse Engineering (WCRE '95)*, 1995, pp. 86–95.

[59] H. A. Basit and S. Jarzabek, "Detecting higher-level similarity patterns in programs," in *Proc. 5th Joint Metting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. (ESEC/FSE '05)*, 2005, pp. 156–165.

[60] I. Baxter, C. Pidgeon, and M. Mehlich, "DMS®: program transformations for practical scalable software evolution," in *Proc. Int. Conf. Softw. Eng. (ICSE '04)*, 2004, pp. 625–634.

[61] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Proc. Working Conf. Reverse Engineering (WCRE '06)*, 2006, pp. 253–262.

[62] J. Krinke, "Identifying similar code with program dependence graphs," in *Proc. Working Conf. Reverse Engineering (WCRE '01)*, 2001, pp. 301–309.

[63] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Int. Symp. Static Analysis (SAS '01)*, 2001, pp. 40–56.

[64] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proc. Int. Conf. Softw. Eng. (ICSE '08)*, 2008, pp. 321–330.

[65] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proc. Int. Symp. Software Testing and Analysis (ISSTA '09)*, 2009, pp. 81–92.
[66] A. Sæbjørnsen, J. Willcock, T. Panas, D. J. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proc. Int. Symp. Software Testing and Analysis (ISSTA '09)*, 2009, pp. 117–128.
[67] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *Journal of Machine Learning Research*, vol. 6, pp. 2721–2744, 2006.
[68] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Proc. Int. Conf. Detection of Instrusions and Malware and Vulnerability Assessment (DIMVA '06)*, 2006, pp. 129–143.
[69] X. Hu, T. cker Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proc. ACM Conf. Computer and Communications Security (CCS '09)*, 2009, pp. 611–620.
[70] S. Chaki, C. Cohen, and A. Gurfinkel, "Supervised learning for provenance-similarity of binaries," in *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD '11)*, 2011, pp. 15–23.
[71] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Proc. Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '08)*, 2008, pp. 108–125.
[72] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *ISOC Symp. Network and Distributed System Security (NDSS '09)*, vol. 9, 2009, pp. 8–11.
[73] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proc. ACM Conf. Computer and Communications Security (CCS '07)*, 2007, pp. 116–127.
[74] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding api components and examples," in *Proc. IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC '06)*, 2006, pp. 195–202.
[75] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "Mudablue: An automatic categorization system for open source repositories," *Journal of Systems and Software*, vol. 79, no. 7, pp. 939–953, 2006.
[76] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng. (FSE '10)*. ACM, 2010, pp. 157–166.
[77] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proc. Int. Conf. Softw. Eng. (ICSE '11)*, 2011, pp. 111–120.

Zhenzhou Tian received the B.S. degree in computer science and technology from Xi'an Jiaotong University, China, in 2010. He is currently working toward the Ph.D. degree in the Department of Computer Science and Technology at Xi'an Jiaotong University, China. His research interests include trustworthy software, software plagiarism detection, and software behavior analysis.

Qinghua Zheng received the B.S. degree in computer software in 1990, the M.S. degree in computer organization and architecture in 1993, and the Ph.D. degree in system engineering in 1997 from Xi'an Jiaotong University, China. He was a postdoctoral researcher at Harvard University in 2002. He is currently a professor in Xi'an Jiaotong University, and the dean of the Department of Computer Science. His research areas include computer network security, intelligent e-learning theory and algorithm, multimedia e-learning, and trustworthy software.

Ting Liu received his B.S. degree in information engineering and Ph.D. degree in system engineering from School of Electronic and Information, Xi'an Jiaotong University, Xi'an, China, in 2003 and 2010, respectively. Currently, he is an associate professor of the Systems Engineering Institute, Xi'an Jiaotong University. His research interests include Smart Grid, network security and trustworthy software.

Ming Fan received the B.S. degree in computer science and technology from Xi'an Jiaotong University, China, in 2013. He is currently working toward the Ph.D. degree in the Department of Computer Science and Technology at Xi'an Jiaotong University, China. His research interests include trustworthy software and malware detection of Android Apps.

Eryue Zhuang received the B.S. degree in software and microelectronics from Northwestern Polytechnical University, China, in 2014. She is currently working toward the M.S. degree in the Department of Computer Science and Technology at Xi'an Jiaotong University, China. Her research interests include trustworthy software and user behavior analysis.

Zijiang Yang is an associate professor in computer science at Western Michigan University. He holds a Ph.D. from the University of Pennsylvania, an M.S. from Rice University and a B.S. from the University of Science and Technology of China. Before joining WMU he was an associate research staff member at NEC Labs America. He was also a visiting professor at the University of Michigan from 2009 to 2013. His research interests are in the area of software engineering with the primary focus on the testing, debugging and verification of software systems. He is a senior member of IEEE.