# k-gram Based Software Birthmarks

Ginger Myles Christian Collberg
Department of Computer Science
University of Arizona
Tucson, AZ 85721
{mylesg,collberg}@cs.arizona.edu

## ABSTRACT

Software birthmarking relies on unique characteristics that are inherent to a program to identify the program in the event of suspected theft. In this paper we present and empirically evaluate a novel birthmarking technique which uniquely identifies a program through instruction sequences. To evaluate the strength of the birthmarking technique we examine two properties: credibility and resilience to semantics-preserving transformations. We show that the technique provides both high credibility and resilience. Additionally, it complements previously proposed static birthmarking techniques.

## Categories and Subject Descriptors

K.5.1 [**Legal Aspects of Computing**]: Hardware/Software Protection—*Proprietary rights*

## General Terms

Legal Aspects

## Keywords

Software theft detection, Software birthmarking

## 1. INTRODUCTION

The theft of software occurs on a regular basis and in a variety of different forms. People make copies of programs to give to friends and relatives so they do not have to buy a copy; software crackers illegally resell programs, often to unsuspecting consumers; unscrupulous programmers steal a portion of a program and incorporate it into another program in order to decrease product cost and development time. Each of these scenarios constitutes software theft and has an economic impact.

Currently, there are a variety of techniques in use to try to prevent, discourage, and detect theft. In this paper we present and evaluate a specific technique, *software birthmarking*, for the detection of software theft. A software birthmark relies on a unique characteristic, or set of characteristics, that is inherent to a program to uniquely identify it. For an effective birthmarking technique it is highly likely that two programs, or program parts, $p$ and $q$, are copies if they both have the same birthmark. In this paper we propose the use of opcode-level $k$-grams as a software birthmarking technique. This technique computes the set of unique opcode sequences of length $k$ for a set of modules. $k$-grams have been previously used to detect similarity between documents [5, 6, 11, 13] and programs at the source code level [15], but not at the opcode-level. We demonstrate that $k$-gram birthmarks exhibit a high degree of credibility and resiliency and that they nicely complement previously proposed birthmarking techniques.

This paper makes the following contributions:

1. We propose and evaluate a novel static birthmarking technique based on instruction sequences.
2. We provide an independent evaluation of the static birthmarking technique by Tamada, et al. [16, 17].
3. We provide a comparative analysis between our birthmarking technique and Tamada's.

## 2. RELATED WORK

Splitting a file into $k$-grams or chunks is a common technique used to detect similarities between documents and programs. A variety of techniques have been proposed based on this general idea (e.g. [5, 6, 11, 13]). One such example is Moss [1] which is an automated tool used to detect similarities between programs at the source code level. The technique used to identify similarities is called winnowing [15] which divides the file into $k$-grams. A hash of each $k$-gram is then computed and a subset of hashes is selected as the document fingerprint. This technique has proven to be quite successful at detecting plagiarism within student programs. However, one of the drawbacks of systems like Moss is that similarity is computed at the source code level. Often source code is unavailable. In addition, these systems do not consider semantics-preserving transformations and the effects of decompilation on the formatting of the source code. For example, it was shown by Collberg, et al. [10] that given the source code of a Java application, simply compiling then decompiling will cause Moss to indicate 0% similarity between the original and the decompiled source code.

*Code clones* is another technique which is used to identify similarities in code. A clone is a program fragment that is identical to another fragment. Clones appear in software

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*SAC'05*, March 13-17, 2005, Santa Fe, New Mexico, USA.
Copyright 2005 ACM 1-58113-964-0/05/0003 ...$5.00.

for a variety of reasons such as code reuse by copying pre-existing sections of code and then making minor changes. It is desirable to identify and remove the duplicated code to improve software maintenance costs. A variety of techniques have been suggested for the identification of clones such as using abstract syntax trees [4] and comparing sections of code while taking into consideration transformations such as variable and function name changes [2]. Again, the drawback to such techniques is that they are applied at the source code level.

Baker and Manber [3] adapt three tools, previously designed to identify similarity in source code and text, to identify similarities in Java bytecode. Siff is applied to a large collection of files to identify pairs which contain a large number of common blocks. Dup can be applied to sets of files to identify similar segments despite renaming transformations. The third tool is the UNIX tool diff. Of the three tools, siff and diff cannot be directly applied to the disassembled bytecode but instead requires the bytecode to be processed into a "normal form." Additionally, the authors acknowledge that their techniques will not withstand obfuscation.

Tamada, et al. [16, 17] have proposed a birthmarking technique specific to Java classfiles which is a combination of four individual birthmarks: constant values in field variables (CVFV), sequence of method calls (SMC), inheritance structure (IS), and used classes (UC). These four birthmarks could be used individually but the combination yields a more believable and reliable technique.

The Tamada technique relies on characteristics that are statically available and targets class level theft. A dynamic birthmark technique has also been developed which uniquely identifies a program based on a complete control flow trace of its execution [14]. This technique targets whole program theft.

## 3. SOFTWARE BIRTHMARKS

The definition of a birthmark is tied to the meaning of *copy*. What does it mean for a program module $q$ to be a *copy* of another module $p$? The most obvious definition is that $q$ is an exact duplicate of $p$. However, this does not take into consideration that a software cracker is likely to apply semantics-preserving transformations to $q$ to disguise any copying. For example, instructions could be reordered or bogus code could be added that is never actually executed. In the event of a program transformation we would still like to be able to say that $q$ is a copy of $p$. Additionally, it is important to note that two programs can exhibit the same external behavior and not be copies, but they must exhibit the same external behavior if they are copies. For example, iterative and recursive versions of the same function will exhibit the same external behavior but are not copies of each other.

The following definition of a birthmark is a restatement of the definition given by Tamada, et. al. [16, 17].

*Definition 1.* (Static Birthmark) Let $p, q$ be sets of modules. Let $f$ be a method for extracting a set of characteristics from a module. Then $f(p)$ is a birthmark of $p$ if:

1. $f(p)$ is obtained only from $p$ itself (without any extra information), and
2. $q$ is a copy of $p \Rightarrow f(p) = f(q)$.

| Window Size | Unique $k$-grams | Total $k$-grams |
|---|---|---|
| 1 | 197 | 5,416,799 |
| 2 | 8727 | 5,281,483 |
| 3 | 86,345 | 5,109,318 |
| 4 | 310,659 | 4,914,995 |
| 5 | 634,551 | 4,765,130 |
| 6 | 937,360 | 4,631,019 |
| 7 | 1,170570 | 4,518,672 |

Table 1: Number of unique $k$-grams and total number of $k$-grams for $1 \leq k \leq 7$.

The strength of software birthmarking lies in its ability to detect software theft given a potentially hostile adversary even when the source code is unavailable. This is crucial since most programs are distributed without source.

## 4. K-GRAM BASED BIRTHMARKS

A $k$-gram is a contiguous substring of length $k$ which can be comprised of letters, words, or in our case opcodes. The $k$-gram birthmark is based on static analysis of the executable program. For each method in a module we compute the set of unique $k$-grams by sliding a window of length $k$ over the static instruction sequence as it is laid out in the executable.

The birthmark for the module is the union of the birthmarks of each method in the module. The order of the $k$-grams within the set is unimportant as is the frequency of occurrence of each $k$-gram. By using the unique $k$-grams without their associated frequency the birthmark is less susceptible to semantics-preserving transformations. For example, an obfuscation which duplicates basic blocks will increase the frequency of those $k$-grams in the block. Additionally, because the birthmark is independent of the order of the methods in the module or the modules within the program, the technique can be used at the module or program level.

In order to use $k$-grams to uniquely identify a program it must be true that a specific set of $k$-grams is unique to a program. To investigate this idea we examined the frequency of $k$-grams , where $1 \leq k \leq 7$, in 222 Java jar-files obtained from the Internet. The programs range in size from 2 to 11,329 methods and 1 to 586 classes. We are assuming that since these programs were obtained from a variety of sources that they represent a reasonable random sampling of Java programs. We discovered that even the top 10 most frequently occurring k-grams have a very low frequency. For $k \geq 3$ the top 10 frequencies were less than 2%. This leads to a hypothesis that two independent programs will have very few $k$-grams in common. Table 1 shows the total number of unique $k$-grams over all 222 programs and the total number of $k$-grams. From this data we see that as the value of $k$ increases the ratio of unique $k$-grams to total $k$-grams decreases which also indicates that two programs will have few $k$-grams in common for larger values of $k$.

### 4.1 Similarity of k-gram Birthmarks

The $k$-gram birthmark is the set of unique opcode sequences of length $k$. Let $f(p) = \{p_1, ..., p_n\}$ and $f(q) = \{q_1, ..., q_m\}$ be birthmarks of the sets of modules $p$ and $q$ respectively. We say that two sets of modules are the same if and only if $f(p) = f(q)$, i.e. if $|f(p)| = |f(q)| = |f(p) \cap f(q)|$. The proliferation of code obfuscation and optimization tools

has made it far more likely that a software cracker will apply a semantics-preserving transformation to defeat the detection of software theft. Even in the event of this type of attack we would like to be able to conclude that $q$ is a copy of $p$.

Broder [6] explores a parallel idea for comparing documents. He defines the two mathematical notions of *resemblance* and *containment* in order to quantitatively measure the similarity of two documents. The resemblance of two documents, $p$ and $q$, is defined to be:

$$r(p,q) = \frac{|f(p) \cap f(q)|}{|f(p) \cup f(q)|}$$

while the containment of $p$ within $q$ is defined as:

$$c(p,q) = \frac{|f(p) \cap f(q)|}{|f(p)|}$$

To clarify which measurement we should choose to define the similarity of two sets of modules consider the following scenarios.

1. Alice creates a program and sells it to Bob. Bob makes copies and re-sells the program under a new name.
2. Alice creates a program and sells it to Bob. Bob applies a series of semantics-preserving code transformations to the program, makes copies, and re-sells the program under a new name.
3. Alice creates a program and sells it to Bob. Bob removes a module to use in his own similar program to make his program better. He then sells the program at a cheaper price.

In each of these scenarios at least part of Alice's program is contained in Bob's. If we can identify a large percentage of Alice's program in Bob's then we are able to show that Bob copied Alice's, even if he made some changes or additions. Since resemblance will also consider the additions made by Bob, it is not the correct measure of similarity for detecting theft. The containment measure $c(p,q)$ is therefore the correct quantity for us.

*Definition 2.* (Similarity) Let $f(p) = \{p_1, ..., p_n\}$ and $f(q) = \{q_1, ..., q_m\}$ be $k$-gram birthmarks extracted from the sets of modules $p$ and $q$. The similarity between $f(p)$ and $f(q)$ is defined by:

$$s(p,q) = \frac{|f(p) \cap f(q)|}{|f(p)|} \times 100.$$

The above definition of similarity assumes that $p$ is the original and $q$ is the stolen copy. It is often the case that such an assumption is valid. However, in the event that it is unknown which is the original, but it is still desirable to compute the similarity, such as detecting plagiarism in two student programs, then the similarity would be defined as $max(s(p,q), s(q,p))$.

# 5. EVALUATION

The strength of a birthmark technique is evaluated based on the following two properties:

*Property 1.* (Credibility) Let $p$ and $q$ be independently written sets of modules which may accomplish the same task. We say $f$ is a *credible* measure if $f(p) \neq f(q)$.

*Property 2.* (Resilience) Let $p'$ be a set of modules obtained from $p$ by applying semantics-preserving transformations $\mathcal{T}$. We say that $f$ is *resilient* to $\mathcal{T}$ if $f(p) = f(p')$.

The likelihood of the birthmark producing a false positive is evaluated through the use of Property 1. It is undesirable for the birthmark technique to indicate that two independently implemented sets of modules are copies even if they accomplish the same task. Since it is unlikely that the two independent sets of modules contain all of the same details the birthmark should capture those details which are likely to differ.

It is also desirable that a birthmark technique is able to detect theft even in the presence of semantics-preserving transformations. There are a variety of code obfuscation and optimization tools publicly available, for example Codeshield [8], SandMark [9], and Smokescreen [12], thus it is highly likely that a software cracker will apply at least one transformation prior to the distribution of illegal copies. Property 2 addresses this issue.

To evaluate the effectiveness of the $k$-gram birthmark we examined its ability to satisfy the above two properties. For Property 1 we performed two experiments. First we examined its ability to distinguish between randomly selected Java applications. The second experiment focused on distinguishing between independently implemented programs which accomplish the same task. To test Property 2 we used three different code obfuscation tools.

## 5.1 Credibility

The first experiment used to evaluate the credibility of the $k$-gram birthmark examined the percentage of similarity of 111 pairs of Java programs. These pairs were obtained by pairing the 222 Java applications obtained from the Internet and used in Section 4. We examined the percentage of similarity between the pairs using $k$-grams where $1 \leq k \leq 8$. We found that as $k$ increased the percentage of similarity between the pairs decreased. However, with $k = 8$ there were still 5 pairs with similarity greater than 60% and one pair with 100% similarity. Because these pairs appeared to be outliers we examined the programs and found that the programs with 100% similarity were in fact identical and that the 4 other pairs consisted of programs where one was a later version of the other. The results indicate that with larger values of $k$ there is a very minimal chance of producing a false positive using $k$-grams.

An important feature of a birthmark technique is that it can distinguish between two sets of modules that are independently implemented even if they accomplish the same task. To demonstrate that the $k$-gram birthmark technique is able to make this distinction we applied it to two problems: calculating the factorial function and generating Fibonacci numbers. For each of these problems there exists an iterative and a recursive solution. We found that even the smaller values of $k$ produced results indicating that it is unlikely one is a copy of the other. For $k = 2$ the similarity was only 30% for both sets and the percentage decreased as $k$ increased.

## 5.2 Resilience

To test the resilience of the $k$-gram birthmark technique we used three code obfuscation/optimization tools: Codeshield, SandMark, and Smokescreen to automatically trans-

form a test program into a semantically equivalent but not identical program. We chose the test program Conzilla [7] from the set of 222 Java programs used in Section 4. Conzilla is a tool designed to aid in organizing and exploring electronically stored components of information. The Conzilla tool has 524 classes and 3092 methods.

For Codeshield and Smokescreen we applied the highest level of obfuscation provided by the tool. Both of these tools include name obfuscation, the elimination of debugging information, and some type of control flow obfuscation. Additionally, Smokescreen supports dead code elimination. The SandMark tool did not include an automatic obfuscation tool but instead allowed us to pick between 33 obfuscations which can be applied individually or in succession. We computed the similarity using $k$-grams and the three obfuscation tools for $2 \leq k \leq 8$.

For Codeshield the similarity was found to be 100% for each $k$ and for Smokescreen the similarity decreased from 93% to 62% as $k$ increased. Of the 33 single obfucations in SandMark, 25 were discovered to have a similarity of at least 80%.

One important characteristic to note about the $k$-gram birthmark is that as $k$ increases there is a decrease in similarity for many of the transformations. This is to be expected and thus the value of $k$ must be chosen wisely. Choosing an appropriate value for $k$ is discussed in the next section.

## 5.3 Credibility vs. Resilience Tradeoff

It was shown in Sections 5.1 and 5.2 that as the value of $k$ increases the credibility increases and the resiliency decreases. Thus it is necessary to identify the value of $k$ which maximizes both the credibility and resiliency. Through the data generated for the evaluation we are able to form a hypothesis as to an appropriate value of $k$. At $k = 4$, 106 of the pairs exhibit a similarity $< 55\%$ and at $k = 5$ those same 106 pairs exhibit a similarity $\leq 40\%$, which also holds true for $6 \leq k \leq 8$. The 5 remaining pairs should exhibit a higher percentage of similarity since they are in fact similar. For $k = 4$ and 5 only 11 of the transformations yield a similarity of $\leq 90\%$, so $k = 4$ and 5 exhibit a high degree of resistance to transformation. Thus our analysis indicates that choosing $k = 4$ or 5 provides an appropriate tradeoff between credibility and resilience for the $k$-gram birthmark technique.

## 5.4 Tamada Birthmark Technique

Tamada, et al. [16, 17] proposed a Java specific birthmarking technique which is composed of four individual birthmarks: constant values in field variables (CVFV), sequence of method calls (SMC), inheritance structure (IS), and used classes (UC). CVFV extracts type and initialization values for the field variables declared in the class. SMC examines the sequence of method calls in the class as they appear in the classfile. Because it is easy to change the names of methods within the application only those method calls which are members of well-known classes (e.g. classes in J2SDK) are considered in the sequence. IS traverses the inheritance structure of the class back to `java.lang.Object` only considering those classes within the set of well-known classes. If a class not in the well-known set is encountered `null` is inserted in the sequence in its place. UC examines an alphabetic ordering of all classes in the set of well-known classes used by the class, i.e. superclass of the given class, return and argument types of methods, types of fields, etc.

| Obfuscation | Similarity | | | | | |
|---|---|---|---|---|---|---|
| | CVFV | SMC | IS | UC | All 4 | $k = 5$ |
| SM Test 1 | 80 | 31 | 89 | 100 | 49 | 84 |
| SM Test 2 | 3 | 0 | 9 | 6 | 1 | 65 |
| Codeshield | 1 | 2 | 23 | 40 | 5 | 100 |
| Smokescreen | 1 | 2 | 25 | 100 | 7 | 74 |

SM Test 1 : Instruction Ordering, Opaque Branch Insertion, Modify If Else
SM Test 2 : Class Splitter, Buggy Code, Primitive Promoter, Add Bogus Fields

Table 2: Similarity using the Tamada et al. birthmark techniques and $k$-grams with $k = 5$ on an original and obfuscated version of Conzilla.jar.

The Tamada birthmark works well given two classfiles, but the strength suffers when given two collections of classfiles. This is because three of the four individual birthmarks are order dependent, i.e. they rely on the order of the characteristics as they are statically laid out in the classfile. For example, consider the SMC birthmark, i.e. the sequence of method calls as they are laid out in the classfile. The order is easy to manipulate by simply rearranging the methods in the class. Performing such a transformation would cause the birthmark to indicate a lower similarity. Only UC addresses this issue by using an alphabetic ordering of the classes. In addition, after an obfuscation has been applied it is not always clear which original class to compare with which obfuscated class. To handle this problem every class in the original must be compared with every class in the obfuscated program, which for large programs may not be a feasible option.

We used the same automated credibility and resilience evaluation on the Tamada birthmark. Using the $k$-grams we identified a single pair of applications which were identical and four pairs where one program was a later version of the other. All four techniques identified the pair of identical programs as being 100% similar, but only UC was able to identify even one of the four additional very similar pairs. In fact, IS and UC identified a pair of very different programs as being remarkably similar even though they were not. Additionally, the four techniques were unable to distinguish between the recursive and iterative versions of factorial and Fibonacci. Only SMC was able to make a single distinction and that was for the factorial programs. Thus, when used at the program level, without manual intervention, the birthmarks produce both false positives and false negatives. If it is known which classfiles to compare or only single classes are compared then the credibility may increase.

We also evaluated the resiliency of the four techniques using the three obfuscation tools on Conzilla.jar. The results are given in Table 2. These results show that the SMC birthmark performs very poorly when applied blindly, CVFV and IS perform adequately for many of the transformations, and UC demonstrates the highest level of resiliency. When the four birthmarks are used in conjunction the overall resiliency is very low.

| Obfuscation | Similarity | |
|---|---|---|
| | Tamada | Tamada and 5-gram |
| SM Test 1 | 49 | 75 |
| SM Test 2 | 1 | 21 |
| Codeshield | 5 | 75 |
| Smokescreen | 7 | 55 |

SM Test 1 : Instruction Ordering, Opaque Branch Insertion, Modify If Else
SM Test 2 : Class Splitter, Buggy Code, Primitive Promoter, Add Bogus Fields

Table 3: By combining the Tamada Birthmark and the $k$-gram birthmark it is possible to provide stronger evidence of software theft than just using the Tamada, et al. technique.

## 5.5 k-gram Birthmark vs. Tamada Birthmark

The Tamada birthmarking technique is the only known module level technique which targets program theft at the bytecode level. Because of this we are using it as an evaluation base for the $k$-gram technique.

Table 2 includes the resiliency results for the $k$-gram birthmark with $k = 5$ and the Tamada Birthmark. From this table it can be seen that the $k$-gram birthmark has a significantly higher level of resilience for every transformation tested.

The $k$-gram birthmark provides an advantage over the Tamada techniques in that it can be blindly applied at either the module or program level. It is also a technique which is more likely to be able to stand on its own and would not necessarily have to be used in conjunction with other techniques. The four Tamada techniques really should be used in conjunction especially for smaller classes or programs. The $k$-gram birthmark could be used with other techniques to provide more conclusive evidence of program theft. Table 3 shows that if the Tamada birthmark is combined with the $k$-gram birthmark it is possible to provide stronger evidence of software theft than just using the Tamada technique.

# 6. SUMMARY

In this paper we proposed the use of opcode level $k$-grams for use in identifying software theft. We evaluated the technique with respect to two properties: credibility and resilience. We found that as the value of $k$ increases the credibility increases but the resilience decreases. We conclude that $k = 4$ or 5 is an appropriate value which maximizes both credibility and resilience. Additionally, we provided an independent evaluation of Tamada's birthmark algorithm. We discovered that the $k$-gram technique could be blindly used at both the module and program level, but that the Tamada technique would require human interaction to be used reliably at the program level. We additionally discovered that the $k$-grams produce fewer false positives and false negatives and could reliably be used to identify different versions of programs. Overall, we demonstrated that the $k$-gram birthmark could be used individually to detect program theft or in conjunction with other techniques to provide stronger evidence. It is currently unknown whether the $k$-gram technique is sensitive enough to be used to detect plagiarism in student programs. This is due to the nature of student assignments which often result in very similar programs. However, we feel this would be an interesting future evaluation.

# 7. REFERENCES


[1] A. Aiken. Moss – a system for detecting software plagiarism. `http://www.cs.berkeley.edu/~aiken/moss.html`.

[2] B. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pages 86–95, 1995.

[3] B. Baker and U. Manber. Deducing similarities in java sources from bytecodes. In *Proceedings of the USENIX Annual Technical Conference*, 1998.

[4] I. D. Baxter, A. Yahin, L. M. D. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM: The Internation Conference on Software Maintance*, pages 368–377, 1998.

[5] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *ACM SIGMOD international conference on Management of data*, pages 398–409, 1995.

[6] A. Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES '97)*, pages 21–29, 1998.

[7] Center for User Oriented IT Design. Conzilla the concept browser. `http://www.conzilla.org`.

[8] CodingArt. Codeshield java bytecode obfuscator. `http://www.codingart.com/codeshield.html`.

[9] C. Collberg. Sandmark. `http://www.cs.arizona.edu/sandmark/`.

[10] C. Collberg, G. Myles, and M. Stepp. Cheating cheating detectors. Technical Report TR04-05, University of Arizona, 2004.

[11] N. Heintze. Scalable document fingerprinting. In *Proceedings of USENIX Workshop on Electronic Commerce*, 1996.

[12] Lee Software. Smokescreen java obfuscator. `http://leesw.com`.

[13] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, CA, USA, 17–21 1994.

[14] G. Myles and C. Collberg. Detecting software theft via whole program path birthmarks. In *Information Security Conference*, 2004.

[15] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 SIGMOD Conference*, 2003.

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

[17] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto. Design and evaluation of birthmarks for detecting theft of java programs. In *Proc. IASTED International Conference on Software Engineering (IASTED SE 2004)*, pages 569–575, Feb 2004.