HAL
archives-ouvertes.fr

# Dynamic Software Birthmark for Java Based on Heap Memory Analysis

Patrick Chan, Lucas Hui, S. Yiu

▶ To cite this version:

Patrick Chan, Lucas Hui, S. Yiu. Dynamic Software Birthmark for Java Based on Heap Memory Analysis. 12th Communications and Multimedia Security (CMS), Oct 2011, Ghent, Belgium. pp.94-107, 10.1007/978-3-642-24712-5_8 . hal-01596200

HAL Id: hal-01596200
https://hal.inria.fr/hal-01596200

Submitted on 27 Sep 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

# Dynamic Software Birthmark for Java based on Heap Memory Analysis

Patrick P.F. Chan, Lucas C.K. Hui, and S.M. Yiu

Department of Computer Science
The University of Hong Kong, Pokfulam, Hong Kong
{pfchan,hui,smyiu}@cs.hku.hk

**Abstract.** Code theft has been a serious threat to the survival of the software industry. A dynamic software birthmark can help detect code theft by comparing the intrinsic characteristics of two programs extracted during their execution. We propose a dynamic birthmark system for Java based on the object reference graph. To the best of our knowledge, it is the first dynamic software birthmark making use of the heap memory. We evaluated our birthmark using 25 large-scale programs with most of them of tens of megabytes in size. Our results show that it is effective in detecting partial code theft. No false positive or false negative were found. More importantly, the birthmark remained intact even after the testing programs were obfuscated by the state-of-the-art Allatori obfuscator. These promising results reflect that our birthmark is ready for practical use.

**Keywords:** software birthmark; software protection; code theft detection; Java

## 1 Introduction

Over the years, code theft has been an issue that keeps threatening the software industry. From time to time, there are cases brought to the court about software license violation. For example, a former Goldman Sachs programmer was found guilty of code theft recently [19]. The software being stolen was for making fast trades to exploit tiny discrepancies in price. Such trading was the core source of revenue of that firm.

Various software protection techniques have been proposed in the literature. Watermarking is one of the well-known and earliest approaches to detect software piracy in which a watermark is incorporated into a program by the owner to prove the ownership of it [9, 7]. Although it cannot prevent software theft, it provides proof when legal action against the thief is needed. However, it is believed that "a sufficiently determined attacker will eventually be able to defeat any watermark" [8]. Watermarking also requires the owner to take extra action (embed the watermark into the software) prior to releasing the software. Thus, some existing Java developers do not use watermarking, but try to obfuscate their source code before publishing. Code obfuscation is a semantics-preserving

transformation of the source code that makes it more difficult to understand and reverse engineer [10]. However, code obfuscation only prevents others from learning the logic of the source code but does not hinder direct copying of them. On the other hand, the thief may further obfuscate the source code rendering code theft detection difficult. Thus, code obfuscation may not be a good mean to prevent software copying.

A relatively new but less popular software theft detection technique is software birthmark. Software birthmark does not require any code being added to the software. It depends solely on the intrinsic characteristics of two programs to determine the similarity between them [23, 17, 20, 15, 22, 14, 10, 18]. It was shown in [17] that a birthmark could be used to identify software theft even the embedded watermark had been destroyed by code transformation. According to Wang et al. [23], a birthmark is a unique characteristic a program possesses that can be used to identify the program. There are two categories of software birthmarks, static birthmarks and dynamic birthmarks. Static birthmarks are extracted from the syntactic structure of programs [22, 18, 13]. Dynamic birthmarks are extracted from the dynamic behavior of programs at run-time [23, 17, 20, 15, 14]. The usual method to destroy the birthmark and prevent discovery of code theft is by obfuscating the program. Since semantics-preserving transformations like code obfuscation only modify the syntactic structure of a program but not the dynamic behavior of it, dynamic birthmarks are more robust against them.

Existing dynamic birthmarks make use of the complete control flow trace or API call trace obtained during the execution of a program [23, 17, 20, 15, 14]. Birthmarks based on control flow trace are still vulnerable to obfuscation attack such as loop transformation. The ones based on API call trace may suffer from not having enough API calls to make the birthmark unique. In this paper, we propose a novel dynamic birthmark, which we call object reference graph (ORG) birthmark, based on the unique characteristics of a program extracted from the heap memory at run-time. The heap memory is a location in the memory in which dynamically created objects are stored.

The core idea of the proposed ORG birthmark is that the referencing structure represented by the object reference graph reflects the unique behavior of a program. An object reference graph is a directed graph. The nodes represent objects and the edges represent the referencing between the objects. They are independent of the syntactic structure of the program code and hence, are not to be changed by semantics-preserving code transformations. Although it is likely that software developed for the same purpose have similar dynamic behaviors, they may not have the same objects referencing structure. For example, a programmer may decide to put the file I/O instructions in a separate class for easier maintenance while others may not.

We implemented a library theft detection system exploiting the ORG birthmark to justify this idea. The goal of the system is to detect if a library L is used by a software S. The first phase of the system dumps out the heap during the execution of the software S. The second phase of the system builds ORGs out of

the heap dumps. Finally, the system searches the ORGs to see if the ORG birthmark for library L, $\text{ORGB}_L$, can be found by exploiting a subgraph isomorphism algorithm. Note that to extract the birthmark for library L, we need a software that is known to be using library L. From it, we build the object reference graph with respect to L by focusing only on those objects defined in that library. As classes from the same library often have the same prefix in their names, we can identify them by a prefix match of their names.

We evaluated our ORG birthmark system using 25 large-scale Java programs with most of them of tens of megabytes in size. During the evaluation, our birthmark system successfully detected 2 libraries in the testing programs. This shows that our system is effective in identifying library theft and is able to distinguish programs developed for the same purpose. To test the robustness of the system against semantics-preserving code transformation, we obfuscated the programs with the state-of-the-art Allatori obfuscator. After that, the system could still successfully detect the 2 libraries in the obfuscated programs. This shows that our system is robust against semantics-preserving code transformation.

The rest of the paper is structured as follows. In section 2, we explore the existing works in the literature. The definitions are given in section 3. In section 4, we formulate the threat model in which our system is designed. We provide the design details and evaluation results in section 5 and 6. Further discussion is covered in section 7 and section 8 concludes.

## 2 Related Work

Software birthmark is different from software watermarking in twofold. First, it is solely the characteristics of a program but not an identifier purposely embedded into the program. Therefore, even though the author of a program was not aware of software piracy when he released the program, a software birthmark can still be extracted from the program to help identify the copying of his program. Second, a birthmark cannot prove the authorship of a program. It can only suggest that a program is a copy of another program. In practice, it is used to collect some initial evidences before taking further investigations. Software birthmarks are further divided into static birthmarks and dynamic birthmarks. Static birthmarks (e.g. [22, 18, 13]) are extracted from the syntactic structure of a program and can be destroyed by semantics-preserving transformations. The trend of software birthmark research is going towards the direction of dynamic birthmarks. The rest of this section will discuss a few pieces of latest work on dynamic birthmarks.

The first dynamic birthmark was proposed by G. Myles and C. Collberg [17]. They exploited the complete control flow trace of a program execution to identify the program. They showed that their technique was more resilient to attacks by semantics-preserving transformations than published static techniques. However, their work is still susceptible to various loop transformations. Moreover, the whole program path traces are large and make the technique not scalable.

Tamada et al. proposed two kinds of dynamic software birthmarks based on API calls [15]. Their approach was based on the insights that it was difficult for adversaries to replace the API calls with other equivalent ones and that the compilers did not optimize the APIs themselves. Through analyzing the execution order and the frequency distribution of the API calls, they extracted dynamic birthmarks that could distinguish individually developed same-purpose applications and were resilient to different compiler options.

Schuler et al. proposed a dynamic birthmark for Java that relies on how a program uses objects provided by the Java Standard API [20]. They observed short sequences of method calls received by individual objects from the Java Platform Standard API. By chopping up the call trace into a set of short call sequences received by API objects, it was easier to compare the more compact call sequences. Evaluation performed by the authors showed that their dynamic birthmark solution could accurately identify programs that were identical to each other and differentiate distinct programs. Their API birthmark is more scalable and more resilient than the WPP Birthmark proposed by Myles and Collberg [17].

Wang et al. proposed system call dependence graph (SCDG) based software birthmark called SCDG birthmark [23]. An SCDG is a graph representation of the dynamic behavior of a program, where system calls are represented by vertices, and data and control dependences between system calls are represented by edges. The SCDG birthmark is a subgraph of the SCDG that can identify the whole program. They implemented a prototype of SCDG birthmark based software theft detection system. Evaluation of their system showed that it was robust against attacks based on different compiler options, different compilers and different obfuscation techniques.

## 3 Problem Definitions

The section first provides the definition of dynamic birthmarks to ease further discussion. We borrow part of the definition from Tamada et al [15]. They are the first formal definition appearing in the literature and have been restated in subsequent papers related to dynamic software birthmark. After that, the formal definition of an ORG birthmark is introduced.

### 3.1 Software Birthmarks

A software birthmark is a group of unique characteristics extracted from a program that can uniquely identify the program. There are two categories of software birthmarks: static birthmarks and dynamic birthmarks. We focus on dynamic birthmarks in this research.

**Dynamic Birthmarks** A dynamic birthmark is one that is extracted when the program is executing. It relies on the run-time behavior of the program. Therefore, semantics-preserving transformations of the code like obfuscation cannot

defeat dynamic birthmarks. Dynamic birthmarks are more robust compared with static birthmarks.

*Definition 1.* (Dynamic Birthmark) Let $p$, $q$ be two programs or program components. Let $I$ be an input to $p$ and $q$. Let $f(p, I)$ be a set of characteristics extracted from $p$ when executing $p$ with input $I$. $f(p, I)$ is a dynamic birthmark of $p$ only if both of the following criteria are satisfied:

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

This definition is basically the same as that of static birthmarks except that the birthmark is extracted with respect to a particular input $I$.

### 3.2 ORG Birthmark

Before we give the definition of ORG birthmark, we need to define what is an object reference graph (ORG). An ORG is a directed graph representation of the structure formed between objects through object referencing. A node represents an object while an edge represents a field of one object referring to another. Objects instantiating the same class are grouped together and are denoted by one node. All the referencing made by this group of objects is represented by the out-going edges from that node. Multiple referencing to the same class of objects by this group of objects is represented by a single edge. We ignore any self-referencing as that can be exploited by an attacker to defeat the birthmark easily. We now give the formal definition of ORG.

*Definition 2.* (ORG: Object Reference Graph) The object reference graph of a program run is a 2-tuple graph $ORG = (N, E)$, where

- $N$ is a set of nodes, and a node $n \in N$ corresponds to a class with non-zero number of instantiations
- $E \in N \times N$ is the set of references between objects, and each edge $n_1 \rightarrow n_2 \in E$ corresponds to one or more references from any field of any objects instantiating the class represented by node $n_1$ to any other objects instantiating the class represented by $n_2$. There is no duplicated edge between two nodes.

Figure 1 shows an example ORG for 4 objects instantiating 3 classes. In Figure 1 (a), there are 4 objects, namely Tom, Jack, Peter, and John, with Tom and Jack instantiating the same class *Cat*. In Figure 1 (b), it shows an ORG with three nodes corresponding to the 3 classes in Figure 1 (a). Note that in a real ORG, the class name is not denoted by the node name. The node name in this figure is for illustration purpose only. Although Tom and Jack are referencing each other through the field *Brother*, this is not captured in the ORG as they belong to the same class *Cat*. Both of them reference the object Peter which belongs to the class *Dog* via the field *Friend*. This is represented by one edge in the ORG from node *Cat* to node *Dog*. The reference from Peter to John

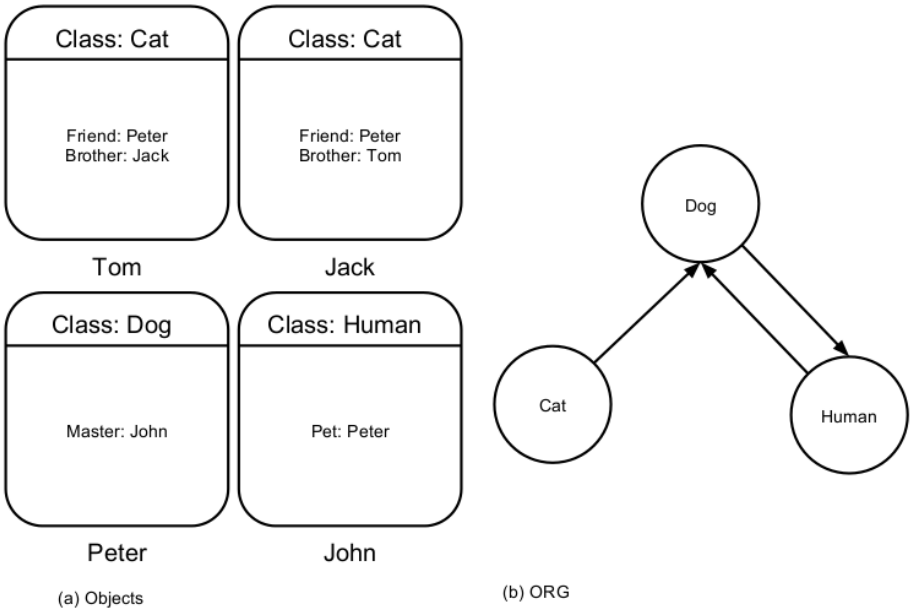


**Fig. 1.** An Example ORG

via the field *Master* and the reference from John to Peter via the field *Pet* are represented by the edge from node *Dog* to node *Human* and the edge from node *Human* to node *Dog* on the ORG respectively.

Next, we state the definition for $\gamma$-isomorphism [12] which serves the purpose of comparing ORG birthmarks.

*Definition 3.* (Graph Isomorphism) A graph isomorphism from a graph $G = (N, E)$ to a graph $G' = (N', E')$ is a bijective function $f\colon \mathbb{N} \to \mathbb{N}'$ such that $(u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$.

*Definition 4.* (Subgraph Isomorphism) A subgraph isomorphism from a graph $G = (N, E)$ to a graph $G' = (N', E')$ is a bijective function $f\colon \mathbb{N} \to \mathbb{N}'$ such that $f$ is a graph isomorphism from $G$ to a subgraph $S \subset G'$.

*Definition 5.* ($\gamma$-Isomorphism) A graph $G$ is $\gamma$-isomorphic to $G'$ if there exists a subgraph $S \subseteq G$ such that $S$ is subgraph isomorphic to $G'$, and $|S| \geq \gamma|G|, \gamma \in (0, 1]$.

Based on the $\gamma$-isomorphism definition, the OBG birthmark can be defined.

*Definition 6.* (ORGB: Object Reference Graph Birthmark) Let $p$, $q$ be two programs or program components. Let $I$ be an input to $p$ and $q$, and $ORG_p$, $ORG_q$ be object reference graphs of the program runs with input $I$ for $p$, $q$ respectively. A subgraph of the graph $ORG_p$ is ORG birthmark of $p$, $ORGB_p$, if both of the following criteria are satisfied:

- program or program component $q$ is in a copy relation with $p \Rightarrow ORGB_p$ is subgraph isomorphic to $ORG_q$.
- program or program component $q$ is not in a copy relation with $p \Rightarrow ORGB_p$ is not subgraph isomorphic to $ORG_q$.

Although our experiment showed that ORGB is robust to state-of-the-art obfuscation techniques, we relax subgraph isomorphism to $\gamma$-isomorphism in our detection for robustness to unobserved and unexpected attacks. Hence, a program $p$ is regarded as a copy of another program $q$ if the ORGB of $p$ is $\gamma$-isomorphic to ORGB of $q$. We set $\gamma = 0.9$ in experiments since we believe that overhauling 10% of an ORGB is almost equivalent to changing the overall architecture of a program component.

## 4 Threat Model

In the attack scenario, Bob is the owner of a program $P$. The core part of it is a library $L$ which is also developed by him. Alice wants to write another program Q which has similar functionalities as $P$. Obtaining a copy of program $P$, Alice reverse engineers it and gets the source code. She extracts the library $L$ from program $P$ and uses it in her own program Q. In order to escape from code theft detection, she obfuscates the source code before compilation.

Later, Bob discovers that the program $Q$ developed by Alice functions similarly to his own program $P$. He wants to find out if program $Q$ uses the library $L$ developed by him. Since the source code of program $Q$ is obfuscated and illegible, he cannot justify it by reverse engineering program $Q$ and looking at the source code. He then gets help from our dynamic birthmark system. He executes program $P$ and gets the birthmark with respect to library $L$. After that, he executes program $Q$ and gets the birthmark of the whole program $Q$. Obtaining the birthmark with respect to library $L$, $ORGB_L$, and the birthmark of the whole program $Q$, $ORG_Q$, he then finds out whether $ORGB_L$ is $\gamma$-isomorphic to $ORG_Q$ or not to identify code theft of library $L$.

## 5 System Design

In this section, we will give details of the design of our dynamic birthmark system. Figure 2 shows the overview of our system. The plaintiff program is the original program owned by the program owner. The defendant program is a program developed by someone else that is suspected to some partial code from the plaintiff program. The processes that the plaintiff and the defendant program undergo are the same except that there is an extra process, the classes refiner, for the plaintiff program. In this section, these processes will be introduced one by one.

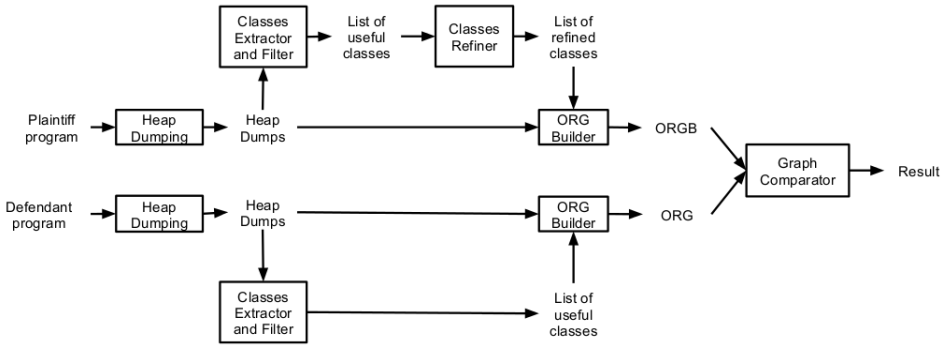


**Fig. 2.** System Overview

### 5.1 Heap Dumping

The heap is dumped using *jmap* [5] from J2SE SDK at an interval of 2 seconds and the dumps will later be merged. This is to avoid any information loss due to garbage collection. In our experiment, we kept dumping for 1 minute.

### 5.2 Classes Extractor and Filter

We make use of the *jhat* library from JDK [4] to parse the dump files generated by *jmap* [5]. The comprehensive list of classes appearing in the dumps is first extracted. However, not all classes represent the unique behavior of the program. Hence, we perform further filtering on this list of classes.

The first group of classes to be pruned out are classes that are provided by Java or Sun since they do not represent the unique characteristics of the program. Their names start with *java*, *javax*, and *sun*. Thus, all classes with these prefixes in their names are removed from the class list. Attackers may try to escape from detection by changing the class names into names with such prefixes. However, we can avoid that by further checking the addresses or hash values of the classes that are actually referenced.

Next, we need to filter out classes that have no instantiation at all. It is because such classes will become standalone nodes with no outgoing and incoming edge in the resulting ORG. They do not represent any unique characteristic of the program.

### 5.3 Classes Refiner

The next two refining steps are done only when extracting birthmark for a library. That is, it is done by Bob when extracting the birthmark for library $L$. (Refer to the threat model discussed in section 4). In order to extract the birthmark for a specific library only, we have to filter out classes which do not

belong to that library. To achieve this, we have to know the package name of the library which must be available to Bob as he is the developer of the library. The names of those classes in the library all start with the package name. Therefore, classes with names not starting with that prefix are filtered out from the class list as they do not belong to that library.

The second refining step is to filter out classes that are usage dependent. A library may create different objects for different use cases. We need to avoid such discrepancy by observing different applications that are known to have used that library. By comparing the heap dumps of these applications, we can learn what are the classes that commonly appear in them with the same object reference structure.

After the above 4 filtering and refining steps, a list of classes that can represent the unique behavior of the application or library is obtained. We can then proceed on to build the ORG based on this list.

### 5.4 Building the ORG/ORGB

For each of the dumps, we build the ORG as follows. Nodes are first created to represent the classes on the class list. After that, for each class on the class list, we transverse all the objects in the heap that instantiate that class. For each of such objects, we check the objects referenced by it one by one. For referenced objects which are also on the class list, we add an edge between the 2 nodes corresponding to the 2 classes to which the 2 objects (the referenced object and the referrer object) belong on the ORG if there is no such an edge yet.

After this process, an ORG is built with nodes representing classes on the class list and edges representing referencing between objects instantiating the classes represented by the nodes. Note that there is only one edge even there are more than one reference between objects from the same pair of classes. Also, self-referencing or referencing between objects in the same class are ignored and not captured on the ORG. Finally, the ORGs from the dumps are merged together to form a graph that embraces all the nodes appearing in the ORGs.

The process of building the ORGB is the same.

### 5.5 Birthmark Comparison

We make use of a library of the VF graph isomorphism algorithm [16, 11] called VFLib [6]. To test if a library $L$ is used in a program $P$, we extract the ORG of the whole program $P$, $ORG_P$, and the birthmark of library $L$, $ORGB_L$, as mentioned earlier in this section. Note that the same input must be used, particularly when the library $L$ is input dependent. It is because the structure of the heap may be input dependent in that case. We then check if $ORGB_L$ is $\gamma$-isomorphic to $ORG_P$. If yes, we conclude that library $L$ is used in program $P$. Otherwise, we conclude that library $L$ is not used in program $P$.

## 6 Evaluation

In this section, we will report the evaluation results on the effectiveness, the ability to distinguish same purpose programs, and the robustness of the prototype of our system.

### 6.1 Experiment setup

We evaluated our birthmark system using 25 large-scale programs with most of them of tens of megabytes in size. The 25 programs were divided into 4 groups. The first group consisted of 6 programs with all of them using the JAudiotagger library. JAudiotagger is a third-party Java library for reading the ID3 tags in MP3 files [2]. The second group consisted of 5 programs with all of them using the JCommon library. JCommon is a third-party Java library containing miscellaneous classes that are commonly used in many Java applications [3]. The third group consisted of 11 programs with all of them using neither the JAudiotagger library nor the JCommon library. The fourth group consisted of 3 programs which also read MP3 tags but without using the JAudiotagger library.

### 6.2 Effectiveness

The birthmarks for JAudioTagger and JCommon library were first extracted. To extract the birthmark of a library, two programs where used to extract the common birthmark as mentioned in section 5.3. In our experiment, Jaikoz and Rapid Evolution 3 were used to extract the birthmark for JAudioTagger, $ORGB_{JAT}$, while iSNS and JStock were used to extract the birthmark for JCommon, $ORGB_{JC}$. For the 6 programs using the JAudiotagger library, a common MP3 file was used as the input file. For the programs using the JCommon library, it was impossible to control the input to the library without looking at the source code to get the idea of how the library was used them. However, the final filtering step mentioned in section 5.3 helped filter out the classes that were usage dependent. During the experiment, the applications were launched and a few actions were performed on them before the heaps were dumped.

We tested the presence of $ORGB_{JAT}$ in the ORGs of Simpletag, Filerename, Jajuk, and MusicBox. All tests gave positive results. The $ORGB_{JAT}$ was not found in ORGs of any other programs in our set of testing programs. For the JCommon library, $ORGB_{JC}$ was found in the ORGs of Paralog, SportsTracker, and Zeptoscope. Again, it was not found in ORGs of any other programs in our set of testing programs.

This part of the evaluation shows that the birthmark is effective in detecting library theft. During the experiment, no false positive or false negative was found.

### 6.3 Distinguishing Same Purpose Programs

In this part, we try to find out if programs developed for the same purpose can be distinguished by our system. During the experiment, the same MP3 file

used for extracting the birthmark of JAudiotagger library was used as input to the fourth group of testing programs. We tested if the library birthmark from JAudiotagger, $ORGB_{JAT}$, could be found in their ORGs. Our experiment results showed that the $ORGB_{JAT}$ was not found in all 3 of their ORGs. We conclude our system can distinguish same purpose programs.

### 6.4 Robustness

In this final part of evaluation, the robustness of the system against semantics-preserving obfuscation is evaluated. Obfuscation means transforming a program $P$ to program $P'$ such that it functions the same as $P$ but its source code becomes difficult to understand mainly to deter reverse engineering [10]. We obfuscated all the 11 programs in the first two groups of programs using the state-of-the-art Allatori Java obfuscator [1]. We tested the presence of the $ORGB_{JAT}$ and the $ORGB_{JC}$ in the birthmarks of them. Our system could still detect the birthmark of the corresponding library in all of them. This shows that our birthmark system is robust against state-of-the-art obfuscation.

## 7 Discussion

In this section, we first discuss the situations in which our birthmark system is not applicable. After that, we discuss possible attacks to deface the birthmark.

### 7.1 Limitations

Since our birthmark extracts information from the heap to identify the program, the heap memory plays a major role in providing enough unique characteristics of the program. There are two main requirements for our birthmark to be effective. First, there must be enough heap objects. For large-scale applications, in which intellectual property right is a critical issue, there are usually many classes that are strongly connected by referencing. In practice, this requirement is satisfied by most libraries or applications. Second, the input to the library or application must be controllable. In some cases, it may be difficult to do that. For instance, it is hard to feed in the same input for a library that reports the current market values of stocks as it is time-critical. In that case, we can only take into account objects that are not input dependent and filter out other objects on the heap when extracting the birthmark.

### 7.2 Attacks

The most feasible attacks are class splitting and class coalescing as suggested in [21] by M. Sosonkin et al. Figure 3 shows how these two techniques can affect our birthmark. In the figure, it illustrates how the birthmark of a program will be altered if class splitting or class coalescing is applied on the class represented by the black node in the middle.

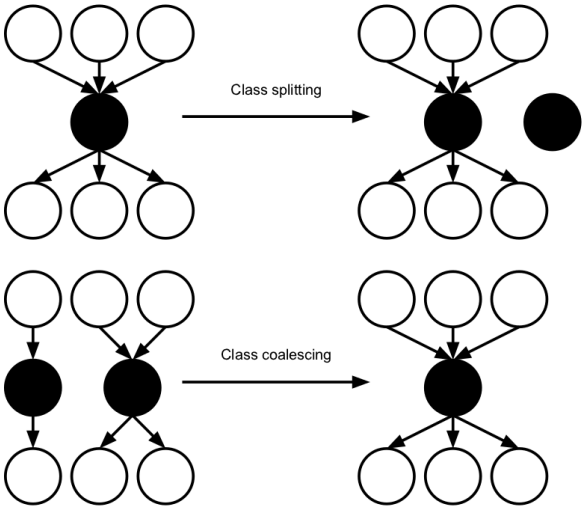


**Fig. 3.** Class splitting and class coalescing

For class splitting, Sosonkin stated in the paper that they believed in practice, splitting a class into two classes not related by inheritance or aggregation is possible only in situations where the original design is flawed and there should have been several different classes. In other words, all references between the original class and the other classes are now going through the inheriting class. Therefore, the change on the heap structure is not influential and the original birthmark can still be found.

For class coalescing, the structure is drastically changed. The original birthmark can no longer be found in the new heap structure. However, evaluation done by Sosonkin et al. showed that, unlike class splitting, class coalescing introduces tremendous amount of overhead proportional to the number of classes coalesced. Therefore, intensive class coalescing is not practical. For small amount of class coalescing, we can loosen our birthmark detection scheme and allow partial matching of the birthmark to be sufficient for a conclusion of a copy relation.

## 8 Conclusion

We have described the design details, implementation, and evaluation of our novel dynamic birthmark system. We implemented and evaluated the birthmark system using 25 testing programs. The evaluation showed that it is reliable and robust against semantics-preserving obfuscation. This research provides a novel dynamic birthmark and supplements the existing dynamic birthmarks. Future work includes combining the heap approach with the system call approach and looking into low-level object-oriented languages like C++.

**Acknowledgements**

The work described in this paper was partially supported by the General Research Fund from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. RGC GRF HKU 713009E), the NSFC/RGC Joint Research Scheme (Project No. N_HKU 722/09), and HKU Seed Fundings for Basic Research 200811159155 and 200911159149.

## References

1. Allatori. http://www.allatori.com/.
2. Jaudiotagger. http://www.jthink.net/jaudiotagger/.
3. Jcommon. http://www.jfree.org/jcommon/.
4. jhat. http://download.oracle.com/javase/6/docs/technotes/tools/share/jhat.html.
5. jmap. http://download.oracle.com/javase/1.5.0/docs/tooldocs/share/jmap.html.
6. Vflib. http://www-masu.ist.osaka-u.ac.jp/~kakugawa/VFlib/.
7. Akito Monden, Hajimu Iida, K.-i. M. K. I., and Torii, K. Watermarking java programs. In *Proceedings of International Symposium on Future Software Technology* (1999).
8. Collberg, C., Carter, E., Debray, S., Huntwork, A., Kececioglu, J., Linn, C., and Stepp, M. Dynamic path-based software watermarking. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation* (New York, NY, USA, 2004), PLDI '04, ACM, pp. 107–118.
9. Collberg, C., and Thomborson, C. Software watermarking: Models and dynamic embeddings. In *Proceedings of Symposium on Principles of Programming Languages, POPL'99* (1999), pp. 311–324.
10. Collberg, C., Thomborson, C., and Low, D. A taxonomy of obfuscating transformations. Tech. Rep. 148, July 1997. http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html.
11. Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. Performance evaluation of the vf graph matching algorithm. In *Proceedings of the 10th International Conference on Image Analysis and Processing* (Washington, DC, USA, 1999), ICIAP '99, IEEE Computer Society, pp. 1172–.
12. Eppstein, D. Subgraph isomorphism in planar graphs and related problems. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 1995), SODA '95, Society for Industrial and Applied Mathematics, pp. 632–640.
13. H. Tamada, K. Okamoto, M. N. A. M., and ichi Matsumoto, K. Detecting the theft of programs using birthmarks. Tech. rep., Graduate School of Information Science, Nara Institute of Science and Technology, 2003.
14. Haruaki Tamada, K. Okamoto, M. N., and Monden, A. Dynamic software birthmarks to detect the theft of windows applications. In *In Proc. International Symposium on Future Software Technology* (2004).
15. Haruaki Tamada, K. Okamoto, M. N. A. M., and ichi Matsumoto, K. Design and evaluation of dynamic software birthmarks based on api calls. Tech. rep., Nara Institute of Science and Technology, 2007.

16. L.P. Cordella, P. Foggia, C. S. M. V. Subgraph transformations for the inexact matching of attributed relational graphs. *Computing* (1998).
17. Myles, G., and Collberg, C. Detecting software theft via whole program path birthmarks. In *Information Security 7th International Conference, ISC 2004, Palo Alto, CA, USA, September 27-29, 2004. Proceedings* (2004), pp. 404–415.
18. Myles, G., and Collberg, C. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing* (New York, NY, USA, 2005), SAC '05, ACM, pp. 314–318.
19. NYTimes. Former goldman programmer found guilty of code theft. `http://dealbook.nytimes.com/2010/12/10/ex-goldman-programmer-is-convicted/`, December 2010.
20. Schuler, D., Dallmeier, V., and Lindig, C. A dynamic birthmark for java. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2007), ASE '07, ACM, pp. 274–283.
21. Sosonkin, M., Naumovich, G., and Memon, N. Obfuscation of design intent in object-oriented applications. In *Proceedings of the 3rd ACM workshop on Digital rights management* (New York, NY, USA, 2003), DRM '03, ACM, pp. 142–153.
22. Tamada, H., Nakamura, M., and Monden, A. Design and evaluation of birthmarks for detecting theft of java programs. In *In Proc. IASTED International Conference on Software Engineering* (2004), pp. 569–575.
23. Wang, X., Jhi, Y.-C., Zhu, S., and Liu, P. Behavior based software theft detection. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 280–290.