A Survey on Software Birthmark based Theft Detection of JavaScript Programs using Agglomerative Clustering and Frequent Subgraph Mining

Swati J. Patel
PG Student
Department of Computer Engineering
SES's R. C. Patel Institute of Technology, Shirpur

Tareek M. Pattewar
Assistant Professor
Department of Information Technology
SES's R. C. Patel Institute of Technology, Shirpur

ABSTRACT
JavaScript programs are always under the threat of being copied. Most browsers provide the way to access the code of JavaScript program so it is easily obtainable. Hence it is mandatory to protect the software. Watermarking and code obfuscation are the techniques used to safeguard the software. A Watermark cannot completely protect the code by getting stolen because a potential attacker can easily remove it. Code obfuscation cannot avoid code from being stolen; it only prevents others by understanding the logic of the program. A birthmark of the JavaScript program is the unique characteristics that it possesses. Heap Graph is used to depict the behaviour of a program as how it calls other objects so as to fulfil the desired functionality. It requires efficient merging of heap graphs generated at various points of time. For that agglomerative clustering can be used. Frequent Subgraph Mining is used to find the subgraph that represents the unique behaviour of the program. At the end, the subgraph of genuine program is searched in the graph of the suspected program. Our aim is to survey about the system that can protect the JavaScript programs from being stolen.

General Terms
Heap graphs, JavaScript programs

Keywords
Dynamic birthmark, agglomerative clustering, frequent subgraph mining, theft identification

1. INTRODUCTION
According to Ninth Annual BSA Global Software 2011 Piracy Study, 57% computer users admit that they use pirated software. The global software piracy rate hovered up to 42% in the year2011. The source code of JavaScript programs can be easily obtained since most browser provide very easy method to obtain the source code of web pages and hence it is very necessary for the industry to safeguard the intellectual property rights of the JavaScript developers. Software safeguard is an important topic for computer scientists. There are several techniques for preventing software theft but out of them most widely used are watermarking and code obfuscation that makes the source code of a program difficult to understand by the humans and proves the ownership of the program. Software watermark is the approach to detect software piracy, in which an extra code known as watermark is included as a part of a program to prove the ownership of the program [5], [10].Watermarking embeds the secret message into the cover image. But watermark can easily be defaced by the determined attacker. It requires the owner of the program to take extra action prior to release the software. Hence most JavaScript developers use code obfuscation before releasing their software. Code obfuscation is the practice of making code unintelligible and hard to understand. Code obfuscation is the process of application of transformations to the code, in such a way that the physical appearance of the code changes, while black-box specifications of the program are preserved. Hence code obfuscation is known as the semantic-preserving process of transformation of code in such a manner that the structure of the program changes while it's meaning and the functionality doesn't change [4]. Code obfuscation only prevents others to understand the logic of the source code but does not protect them from being copied.

As both code obfuscation and watermarking are good but not enough techniques to prevent theft of programs a relatively new and less popular technique is introduced and that is software birthmark. Software birthmark does not require any code being added to the software. It depends only on the internal behavior of a program to determine the similarity between two programs. According to Wang et al. [3], a birthmark is a unique characteristic a program possesses that can be used to identify the program. To detect software theft,

(1) The birthmark of the program under protection (the plaintiff program) extracted.

(2) The suspected program is searched against the birthmark.

(3) If the birthmark of plaintiff program is found in the code of suspected one, then it can be claimed as the suspected program or part of it is a copy of the plaintiff program.

1.1 Taxonomy of Software Birthmarks

There are two categories of software birthmarks,

—Static birthmarks: These are extracted from the syntactic structure of a program [1].

Definition 1: (Static Birthmarks) [11]

Let p, q be two components of a program or program itself.

Let f be method for extracting the set of characteristics from a program. Then f(p) is a static birthmark of p if:

1. f(p) is obtainable from p itself.

2. q is copy of p) f(p) = f(q):

—Dynamic birthmarks: These are extracted from the dynamic behavior of a program at run-time [3]. It is an abstraction of run-time behavior of the program.

Definition 2: (Dynamic Birthmarks) [12]

Let p,q be two components of a program or program itself. Let I be the input to p and q. Let f(p,I) the set of

characteristics extracted from a program p with input I. Then f(p,I) is a dynamic birthmark of p if:

1. f(p,I) is obtainable from p itself when executing p with input I.

2. q is copy of p) f(p,I) = f(q,I):

As semantics-preserving transformations like code obfuscation only modify the syntactic structure of a program but not its dynamic behavior, dynamic birthmarks are more robust against them.

2. LITERATURE SURVEY

The first dynamic birthmark was proposed by Myles et al. To identify the program, they explored the complete control flow trace of a program execution. They proved that their technique can resist to any kind of attacks by code obfuscation. There is a drawback that their work is sensitive to various loop transformations. Besides, the whole program path traces are large and hence it is not feasible to scale this technique further [7].

Tamada et al. proposed two kinds of dynamic software birthmarks based on API calls. Their approach was based on the capacity to understand the hidden truths that it was difficult for opponent to alter the API calls with other equivalent ones and that the compiler did not make the effective use of the APIs themselves. Run time information of API calls was used as a strong signature of the program. The dynamic birthmark was extracted by looking at the execution order and the frequency distribution of API calls. These extracted dynamic birthmarks could differentiate personally developed same-purpose applications and could resist to different compiler options. This promising result led to subsequent researches on dynamic birthmarks based on API calls [8].

Schuler et al. proposed a dynamic birthmark for Java that where a program uses objects provided by the Java Standard API. The short sequences of method calls received by distinct objects from Java Platform Standard API were observed. Then the call traces were decomposed into a set of short call sequences received by API objects. The proposed dynamic birthmark system could accurately identify programs that were similar to each other and distinguish separate programs. In addition, they showed that all birthmarks of obfuscated programs were identical to that of the original program [13].

Wang et al. put forward SCGG birthmark which is a software birthmark based on dependence graph. An SCDG is a graph representation of the dynamic behaviour of a program, where each vertex represents system call and edges denote data and control dependences between system calls. The evaluation of their system showed that it was robust against attacks based on different compiler options, different compilers and different obfuscation techniques. It is the first system that can detect software component theft where only small piece of code is stolen [13].

Chan et al. proposed the first dynamic birthmark based on the runtime heap for JavaScript programs. It is in the form of an object reference tree. A tree comparison algorithm was used to compare two birthmarks and gave a similarity score between two birthmarks. However, due to efficiency problem of the tree comparison algorithm, the depth of the tree was limited to 3 in order to make the running time of the algorithm practical. On the other hand, new birthmark is an object graph and graph monomorphism was used to search for the birthmark in the heap graph of the suspected program. Although they limited the size of the heap graphs in the system, the limitation is less restrictive. It is because the root node of the heap graph is actually at level 2 of the whole object reference graph with reference to the virtual node. Even though the size of the heap graph was limited, the current birthmark captured far more information than the previous system [9].

Later, they proposed another heap based birthmark system. This time, the birthmark system was for Java programs. They used a different algorithm named as graph isomorphism, for birthmark detection. As graph isomorphism is too restrictive and makes the birthmark system vulnerable to reference injection attack. On the contrary, the current birthmark system uses graph monomorphism for birthmark detection which makes this system robust against such attack [1].

3. METHODOLOGY

Fig. 1 shows the overview of birthmark system. It outlines the processes that the plaintiff program and the suspected program undergo[12].

3.1JavaScript Heap Profiler

Being an interpreted language, JavaScript allows for the creation of objects at any time. On the other head, one of the design elements of the V8 JavaScript engine is efficient garbage collection. Hence the JavaScript heap keeps changing due to object creations and garbage collections. To make entire use of the behaviour exhibited by the objects in the heap, each and every object is captured that appears in the heap. In order to achieve this, the objects that disappear from the heap due to garbage collection must be ignored. Therefore, the JavaScript heap profiler takes multiple dumps of the heap and merges them together later on. After kicking off the JavaScript program, the browser keeps dumping the JavaScript heap in every2 seconds. Since taking a snapshot will actually trigger a garbage collection, the heap of the browser is made larger to delay garbage collection and dump the heap more frequently hoping that every object is captured before it becomes garbage.

3.2 Graph Generator and Filter

Since Chromium browser is used to dump out the JavaScript heap in prototype system, the JavaScript engine that powers the Chromium browser is V8 JavaScript Engine. The heap dumps generated by the modified Google Chromium browser are in the form of object reference trees. It is similar to the object reference graph in which the nodes represent the objects while the edges represent the references between them. The only difference is that objects are duplicated to remove cycles in the graph. Although this will increase the size of the data structure, a tree structure allows us to control the number of objects to be included for comparison as we can easily do so by limiting the depth of the tree to be explored [2].

For each snapshot taken using the Chromium browser, a death first search traversal is performed and the heap graph is printed out with nodes and edges that pass a filter. A filter is described in details as follows. Objects in the V8 JavaScript heap are divided into six categories, INTERNAL, ARRAY, STRING, OBJECT, CODE, CLOSURE. Objects that belong to INTERNAL, ARRAY, STRING, and CODE categories are not included in heap graphs.

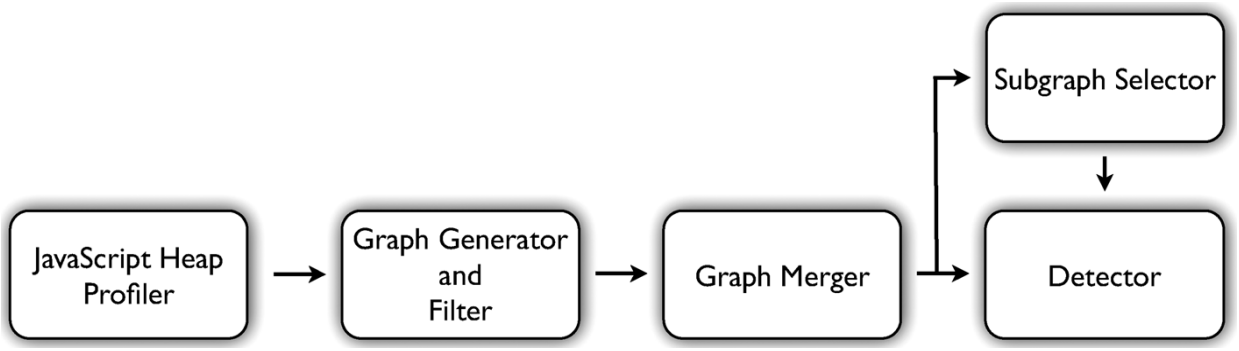

Fig. 1. System Overview

The reasons behind this design decision are as follows: INTERNAL objects are virtual objects for housekeeping purpose and are not accessible from the program code. For ARRAY objects, they represent an array of elements objects. However, arrays are actually represented by an object of the type OBJECT with name Array and the references from the array are coming out from that object. Therefore, ARRAY objects are not included. For STRING and CODE objects, there is no reference coming out from them. Therefore, they are not included as well. To sum up only OBJECT and CLOSURE objects are used in heap graph. They are JavaScript objects and function closures respectively.

References between objects in the V8 JavaScript heap are divided into 4 categories, CONTEXT VARIABLE, ELEMENT, and PROPERTY, INTERNAL. References that belong to CONTEXT_VARIABLE and INTERNAL categories are not included in the heap graph. The reasons behind this design decision are as follows: CONTEXT VARIABLE is a variable in a function context, accessible by its name from inside a function closure. Therefore, it is not accessible by objects outside that function and it is automatically created by V8 for housekeeping purpose. INTERNAL references are properties added by the JavaScript virtual machine. They are not accessible from JavaScript code. Therefore, only ELEMENT and PROPERTY references are included in the heap graph. ELEMENT references are regular properties with numeric indices, accessed via [ ] (brackets) notation and PROPERTY references are regular properties with names, accessed via the '.' (dot) operator, or via [ ] (brackets) notation.

There are some objects created by the JavaScript engine that exist not just for one program. For example, the HTML Document object can be found in the heap graphs of all the JavaScript programs. Therefore, it is needed to filter such objects out as they compromise the uniqueness of the heap graph. Basically, the filtered objects include objects created to represent the DOM tree and function closure objects for JavaScript built-in functions. The output of the graph generator and filter is a set of filtered heap graphs captured at different points of time.

3.3 Graph Merger

There is a unique ID assigned to every object in the JavaScript heap by the V8 JavaScript engine. Moreover, the ID of an object does not change across multiple dumps and therefore, can be used to identify the object. The Graph Generator and Filter also annotates each node in the heap graph with its object ID. Therefore, it is easy to identify whether or not two nodes in two heap graphs refer to the same object. The graph merger takes multiple heap graphs as input and outputs a superimposition of them (one single graph) that includes all the nodes and edges appearing in the input heap graphs.

3.3.1 Agglomerative Clustering

Agglomerative Clustering is a subtype of Hierarchical Clustering. Agglomerative hierarchical clustering begins with every case being a cluster unto itself. At successive steps, similar clusters are merged. The algorithm ends with everybody in one cluster. In agglomerative clustering, once a cluster is formed, it cannot be split; it can only be combined with other clusters. Agglomerative clustering does not let cases separate from clusters that they have joined. An approach that roots in the clustering of point sets is to begin grouping the vertices into clusters by forming a two-vertex cluster from the two most similar vertices. The intuition is that at least the two closest points should be placed in the same cluster with each other. Such merging then continues until only a desired number of clusters remains or another stopping condition is met. At each iteration, one picks the two clusters (singletons or larger) that have the highest similarity value to be merged. This approach is generally known as the pair wise nearest neighbours method [6].

3.4 Subgraph Selector

After going through the above steps, the resulting heap graph is one that contains custom objects only and can be used to identify the JavaScript program. However, it is impossible to use the entire graph as the birthmark of the program since the graph is too large for the subgraph monomorphism tool such as VF Lib. In fact, the subgraph monomorphism problem itself is known to be NP complete. The graph, which can comprise hundreds of nodes, is too large for the algorithm and may lead to very long execution time.

3.4.1 Frequent Subgraph Mining

Frequent Subgraph Mining (FSM) is the essence of graph mining. Frequent subgraph mining can be used to get the frequent subgraph that appears in all the heap graphs extracted from the program. This can make the birthmark more representative of the program. However, the running time of frequent subgraph mining on large graphs is slow and there should be some performance tuning in order for it to be practical. The objective of FSM is to extract all the frequent subgraphs. The straightforward idea behind FSM is to grow candidate subgraphs, in either a breadth first or depth first manner (candidate generation),and then determine if the identified candidate subgraphs occur frequently enough in the graph data set for them to be considered interesting(support counting). The two main research issues in FSM are thus how to efficiently and effectively.

(1) Generate the candidate frequent subgraphs.

(2) Determine the frequency of occurrence of the generated subgraphs. Effective candidate subgraph generation requires that the generation of duplicate or superfluous candidates is avoided [9].

3.5 Detector

The detector takes the subgraph from the plaintiff program and the entire heap graph of the suspected program as inputs and determines whether the selected subgraph of the plaintiff program can be found in the heap graph of the suspected program. Similar to what is done by the subgraph selector; it takes subgraphs of the objects under the Window objects from the suspected program and uses subgraph monomorphism to check whether the subgraph of the plaintiff program can be found in them. Once there is a match found, the detector raises an alert and reports where the match is found.

4. CONCLUSION

Software Birthmark system generates Heap Graph of the system which is treated as the Birthmark to find similarities between two similarly functioning applications and distinguish distinct applications. This system can resist to reference injection attack due using of subgraph monomorphism while searching the heap graph of plaintiff program in the heap graph of suspected program.

5. ACKNOWLEDGEMENT

We are grateful to those who have helped us in this survey.

6. REFERENCES

[1] P. Chan, L. Hui, and S. Yiu. Dynamic software birthmark for java based on heap memory analysis. In Springer-Verlag, editor, IFIP TC 6/TC 11 Int. Conf. Commun. and Multimedia Security(CMS11), vol. 12, pages 94–106, Berlin, Heidelberg, 2011.

[2] P. Chan, L.Hui, and S.Yiu. Jsbirth: Dynamic JavaScript birthmark based on the run-time heap. In 2011 IEEE 35th Annual Computer Software and Application Conference (COMPSAC),pages 407–412, July 2011.

[3] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu,C. Linn, and M. Stepp. Dynamic path-based software watermarking. In ACM, editor, Programming Language Design and Implementation (PLDI 04), pages 107–118, New York, 2004.

[4] C. Collberg, C. Thomborson, and D. Low. Taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, Auckland, New Zealand, 2003.

[5] Christian Collberg and Clark Thomborson. Software watermarking: models and dynamic embeddings. Technical report, Department of Computer Science, University of Auckland, Private Bag 92091, Auckland, New Zealand, 2003.

[6] Pasi Franti, Olli Virmajoki, and Ville Hautamaki. Fast agglomerative clustering using a k-nearest neighbour graph. In IEEE Transactions on Pattern Analysis AND Machine Intelligence, number 11 in vol. 28, pages 1875–1881, November 2006.

[7] G.Myles and C. Collberg. Detecting software theft via whole program path birthmarks. In Inf. Security 7th Int. Conf. (ISC2004), pages 404–414, Palo Alto, CA, September 2004.

[8] H.Tamada, K.Okamoto, and K.Matsumoto. Design and evaluation of dynamic software birthmarks based on API calls. Technical report, Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5 Takayamacho, Ikoma-shi, Nara, 6300101 Japan, 2007.

[9] Michihiro Kuramochi and George Karypis. An efficient algorithm for discovering frequent subgraphs. In IEEE Transactions on Knowledge and Data Engineering, number 9 in vol. 16,pages 1038–1051, September 2004.

[10] Akito Monden, Hajimu Lida, Ken ichi Matsumoto, Katsuro Inoue, and Koji Torii. Watermarking java programs. In International Symposium of Future Software Technology, Nanjing, China, 1999.

[11] Ginger Myles and Christian Collberg. K-gram based software birthmarks. In ACM, editor, Symposium on Application Computing (SAC 05), pages 314–318, 2005.

[12] P.Chan, L.Hui, and S.Yiu. Heap graph based software theft detection. IEEE Transaction On Information Forensics and Security, pages 101–110, January 2013.

[13] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for java. In IEEE/ACM International Conference of Automated Software Engineering (ASE 07), volume 22, pages 274–283, New York, 2007.