ISSN: 2348 - 2117

International Journal of Engineering Technology and Computer Research (IJETCR)
Available Online at www.ijetcr.org
Volume 2 Issue 6 Page No. 50-54

Dynamic birthmark of JavaScript programs using Heap Graph: A Review

Mayuri A. Pandit[1], Archana C. Lomte[2]

[1]JSPM's Bhivarabai Sawant Institute of Technology and Research, Wagholi Pune 412207, India

mayurip90@gmail.com

[2]JSPM's Bhivarabai Sawant Institute of Technology and Research, Wagholi Pune 412207, India

archanalomte@gmail.com

ABSTRACT

JavaScript is the most widely used language nowadays; also most of the browsers readily provide the source code of JavaScript program, because of this the plagiarism of JavaScript programs became a serious threat. Such programs are always under threat of being stolen and to avoid that Software watermarking and Code Obfuscation techniques have been used to prove the ownership and to make the code complicated to understand. But these techniques have no control on copying the program. So there is a new technique introduced to detect the software theft which called "Software Birthmark". In which the unique characteristics of a specific program (birthmark) is used to prove the ownership. Heap Graph is generated when any program is under execution and this is nothing but a Birthmark of that software. To detect the theft, birthmark of a suspected website is searched in the heap graph. The heap graph describes how the objects are linked to achieve the functionality and the behavioral structure of the software.

Key Words: Heap graph, Code Obfuscation, software birthmark, Code theft detection

INTRODUCTION:

JavaScript is most widely used language and it became platform for the development of Windows app .According to the survey 60% of developers use JavaScript language .However JavaScript programs are easily available. Many browses provide very easy methods to get the source code of web pages. So it is very difficult to protect intellectual property rights of JavaScript program. Software programs are always having a threat to be stolen by someone else. Hence to prevent the intellectual property right of software and to prove the ownership of that software the techniques such as Software Watermarking and Code Obfuscation are used.

Software theft is also referred as a software piracy. It is an unauthorized use or copy of computer software. To prevent our code being copied Watermarking is used. To detect software piracy Software Watermarking is used, where an additional code (watermark) is included to the program to prove the ownership. But now it is very easy to remove watermark from program, so any one can copy the program by removing watermark. Hence most of the developers use Code Obfuscation technique. Making the code unintelligible and hard to understand the Code Obfuscation method is used. This technique only changes the physical appearance of program but the black box specification remains same. So it is also called as semantic-preserving process. But using Code Obfuscation we can only prevent others from understanding our program but not from copying the program. So both the techniques are inefficient to prove the intellectual property of program. Because of this Software birthmark is used in order to detect the theft of JavaScript programs. Hence Software birthmark is used to detect the theft of the JavaScript programs.

A. Software Birthmark: - As both code obfuscation and software watermarking are not enough to prevent program from theft a relatively less popular technique is introduced that is software birthmark. According to the Wang et al, a birthmark is a unique characteristic that the program shows, which is used to identify the program. To detect software theft,

1. The plaintiff program is extracted that is the birthmark of the program under protection.
2. The suspected program is search for the birthmark.
3. If the birthmark of the plaintiff program is found in the suspected code then, it can claim that it is copied program.

B. Categories of Software Birthmark: - There are the two categories of software birthmark,

a. Static Birthmark: These are extracted from the syntactic structure of a program

Definition 1: (Static Birthmarks)

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).

b. Dynamic Birthmark: These are extracted from the dynamic behavior of a program at run-time. It is an abstraction of run-time behavior of the program.

Definition 2: (Dynamic Birthmarks)

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).

C. Heap Graph Based Birthmark: - This birthmark technique is form by extracting objective from the heap and building a heap graph and building a heap graph from them.

Heap: Heap is a tree like data structure which satisfies the heap property.

Heap Graph: Heap Graph is simple directed graph in which nodes represent objects and edges represent the references between the object.

1. THE THREAT MODEL:

This section gives the complete problem statement of the paper.

1. Bob is an owner of the program P and he writes his own library L for the program p.
2. Suppose Alice wants to develop program Q which is similar in to the program P. Alice copies the program P and then reverse engineered it to obtain the source code of program P.
3. Alice extracts the library L of the program P and uses that library for her program. And to escape from the theft she obfuscates the code.
4. Later Bob understand that the program Q is functioning same as program P. he is curious to know about whether his library L is used for developing Q or not. So he decides to reverse engineer it. But due to the code obfuscation he is failed to do that.
5. Software birthmark will be helpful to the Bob:

a. He runs the program P with respect to library L and obtains the birthmark.

b. Then he obtains the birthmark of program Q by executing it.

c. For the library L, heap graph based birthmark is HGB (L), and heap graph of program Q, HG (Q).

d. Then Bob find out is HGB (L) is monomorphism to the HG (Q) or not, to identify the code theft of the library L.

2. RELATED WORK:

G. Myles and C. Collberg [5] were proposed the first dynamic birthmark by exploiting the complete control flow trace of program execution. They did the semantic-preserving transformations and proved that this technique is more durable than published static techniques.

Tamada et al. worked on the dynamic software birthmarks. They proposed two kinds of dynamic birthmark based on the API Calls [7]. With the insights that it was difficult for anyone to alter the API calls with other similar API calls they used the runtime information of the API calls as the signature of the program. From this API calls they extracted the dynamic birthmark which distinguish one program from another and were rigid to several compiler options.

Schuler *et al.* observed the Java program for how it uses objects provided by the API [6] and proposed the dynamic birthmark for JAVA. As it is easy to compare the compact call sequence they chop up the call trace into the short call sequence received by API. This API birthmark technique is more scalable than the one by Myles and Collberg [5].

Wang *et al.* [4] proposed the "SCDG birthmark". It represents the dynamic behavior of a program using graph is nothing but SCDG. The Vertices of the graph is system calls while the data and control dependencies between system calls are the edges. The SCDG identifies complete program and its prototype is based on the software theft detection. It is the most robust techniques against several compliers. Also it is the first system which is able to detect partial code theft.

P. Chan, L. Hui and S. Yiu [8] proposed the first dynamic birthmark based on the run-time heap. This is also the first dynamic birthmark for JavaScript program. It is in the form of object reference tree, they used tree comparison algorithm to compare two birthmarks and give the similarity score. But this is only working for the trees with depth 3. Then the authors introduce new technique as Object Graph and they use the graph monomorphism to search birthmark in heap graph.Clustering of point sets involves the forming of two-vertex cluster. The idea behind this is to keep two closest points in same cluster and then one picks the two clusters with highest similarity and merge them.

2. In the sub graph selection phase the birthmark selected of a program is huge and because of this the theft detection time increases. Here a technique to select the smallest possible heap graph and use it as a birthmark can be used which called as "Frequent sub graph Mining"

Frequent sub graph Mining (FSM): It is the soul of graph mining. It is used to make birthmark more representative by getting the sub graph those are frequently appearing in the heap graph extracted from program. However on large graph the FSM is slow.The objective of FSM is to extract all the frequent sub graphs.

4. REVIEW PROCESS:

Figure shows the overview of the entire birthmark system. It shows the process that the plaintiff program and suspected program undergoes.

The **JavaScript heap profiler** is a module that runs the JavaScript program and takes the heap snap shot.

The **graph generator and filter** traverse the object in the heap snap shot and then build the heap graph from them. It is also use to filter the objects.

The **heap graph merger** is used to merge the filtered heap graph together and form the single graph.

The **subgraph selector** is used to select the subgraph from the heap graph and then used to form birthmark of the plaintiff program. This step is only for the plaintiff program not needed for the suspected program.

Finally**, the detector** searches the heap graph of the suspected program for the birthmark of the plaintiff program.

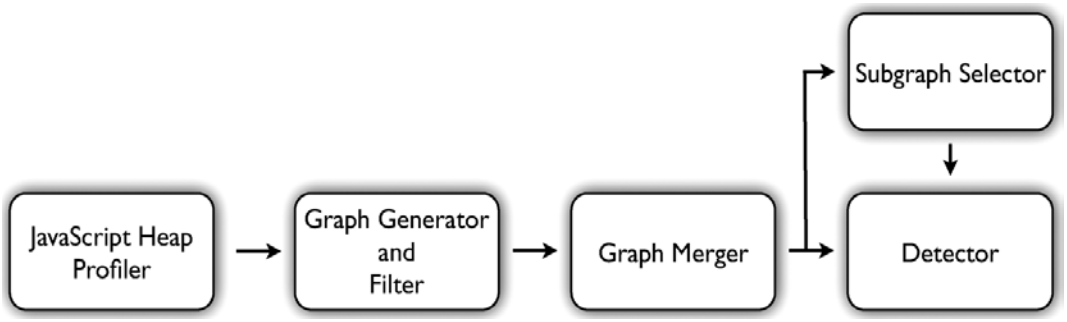

Figure 1: System Overview

A. JavaScript Heap Profiler:

JavaScript allows for the creation of the object at run time because it is an interpreted language. As a result, the JavaScript heap keeps changing because of object creation and garbage collection.

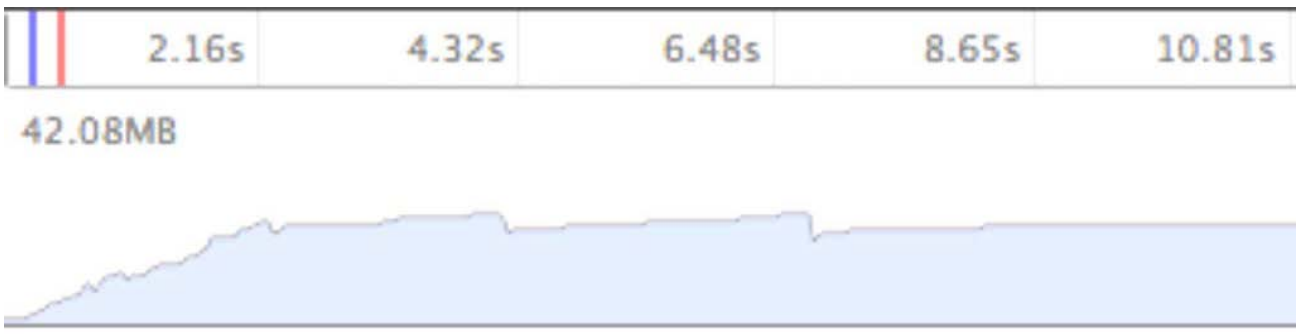

Figure 2: Heap profile of initialization phase

Figure 2 shows Heap profile of initialization phase for Gmail. In the early stage the numbers of objects are increased. After some time there are some drops and then system become stable. The JavaScript heap profiler takes multiple dumps of the heap and then merges them together.

B. Graph Generator and Filter:

Since the chromium browser to dump out the JavaScript Heap. The modified chromium browser generates the

heap dump in the form of object reference tree. It is similar to the object reference graph. The only difference is that to remove cycle objects are duplicated. For each snapshot taken the depth first search is performed and then heap graph is printed and then passed to the filter.

There are six categories of object in the V8 JavaScript heap:

ARRAY
INTERNAL
OBJECT
CODE
CLOSURE
STRING

The objects belongs to the INTERNAL, ARRAY, STRING and CODE categories are not present in the heap graph. The output of the graph generator and filter is a set of heap graph captured at different point.

C. Graph Merger:

The V8 JavaScript engine assigns unique ID to every object in the JavaScript heap. During the multiple dumps the ID of the object does not changes across multiple dumps. It is easy to identify whether the two nodes of heap graph refers the same object or not.

The graph merger takes multiple heap graphs as an input and gives superimposition of them means a single graph. So the superimposition algorithm is used for merging.

D. Subgraph Selector:

As we go through the above steps, we obtain one heap graph which contains only custom objects that can be used to identify the JavaScript program. Entire heap graph cannot be used as a birthmark of a program as the graph is very large for the monomorphism tool, so. The graph with many nodes is very large for the algorithm and it takes very long time for execution. So for the method we have to select subgraph which is then used as a birthmark. A virtual node is an entry point for all the nodes in heap graph. One or more window objects are pointed by the virtual node. Window objects are used to represent Dom windows which are present on the web page.

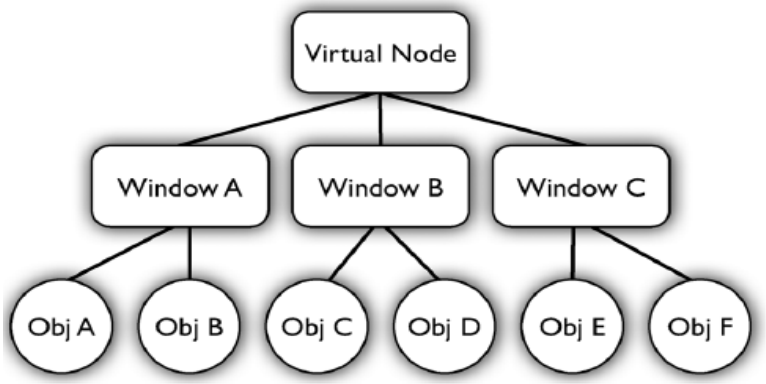

Figure 3: Structure of heap graph

Heap graph structure is shown in figure 3. We have to observe all the objects in a window node and then have to compare their size. We have to select largest object of a subgraph which is reachable from it.

E. Detector:

The subgraph from the plaintiff program and the heap graph of the suspected program is given as an input to the detector. It detects whether the subgraph which is selected as birthmark of plaintiff program is present in the heap graph of suspected program or not.

Once the match is found, it raises the alarm and then report where match is actually found.

5. IMPORTANCE:

To prevent our code from being copied watermarking and code obfuscation techniques are used. But these techniques are not able to prove the ownership of the program. So there is an importance of the birthmark system. Following are the features of the proposed system.

1. It is the first dynamic birthmark which is based on the run time heap.
2. It is the first dynamic birthmark for the JavaScript programs and birthmark uses object reference tree.
3. It uses algorithm which compares the two birthmarks and gives similarity score.
4. This system is able to detect the theft of the obfuscated programs.

6. CONCLUSION:

This Software birthmark system uses heap graph which is used as birthmark to find the similarities between the

applications which functioned similarly. It also distinguishes the different applications. This birthmark system provides solution to prevent intellectual property rights. This system has great resistance against the reference injection attack and it also streamlined each and every process which makes it scalable.

The limitation of this system is that it uses graph monomorphism which is NP complete. The system is designed to handle large graphs. So running time become very long so, practically not able to use. So the future work must be required to overcome these limitations.

7. REFERENCES:

1. E. Data, JavaScript Dominates EMEA Development January 2008
2. A. Monden, H. Iida, K. I. Matsumoto, K. Inoue, and K. Torii, "Watermarking java programs," in *Proc. Int. Symp. Future Software Technology*, Nanjing, China, 1999.
3. C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp, "Dynamic path-based software watermarking," in *Proc.ACM SIGPLAN 2004 Conf. Programming Language Design and Implementation (PLDI '04)*, New York, 2004, pp. 107–118, ACM
4. X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proc. 16th ACM Conf. Comput. and Commun. Security (CCS '09)*, New York, 2009, pp. 280–290, ACM.
5. G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in *Proc. Inf. Security 7th Int. Conf. (ISC 2004)*, Palo Alto, CA, Sep. 27–29, 2004, pp. 404–415.
6. D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for java," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Software Eng.(ASE '07)*, New York, 2007, pp. 274–283, ACM.
7. H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. I. Matsumoto, Design and Evaluation of Dynamic Software Birthmarks based on API Calls, Nara Institute of Science and Technology, Tech.Rep., 2007.
8. P. Chan, L. Hui, and S. Yiu, "Jsbirth: Dynamic JavaScript birthmark based on the run-time heap," in *Proc. 2011 IEEE 35th Annual Comput. Software and Application Conf. (COMPSAC)*, Jul. 2011, pp. 407–412.
9. P. P. F. Chan, L. C. K. Hui, and S.M. Yiu, "Dynamic software birthmark for java based on heap memory analysis," in *Proc. 12th IFIP TC 6/TC 11 Int. Conf. Commun. and Multimedia Security (CMS'11)*, Berlin, Heidelberg, 2011, pp. 94–106, Springer-Verlag