International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347 - 6435 (Online)) Volume 3, Issue 2, August 2014)

Software Birthmark Based Theft/Similarity Comparisons of JavaScript Programs

Swati J. Patel[1], Tareek M. Pattewar[2]

[1]*PG Student, Department of Computer Engineering, SES's R. C. Patel Institute of Technology, Shirpur, Maharashtra, India*
[2]*Assistant Professor, Department of Information Technology*, SES's R. C. *Patel Institute of Technology, Shirpur, Maharashtra, India*

***Abstract*—Almost every browser provides an easy way to obtain the source code of JavaScript Programs. Hence it is very important to prevent the copy of the websites so as to protect intellectual property rights of JavaScript developers. Software Birthmark is used to detect the theft/similarity of JavaScript programs. A birthmark is a set of characteristic possessed by a program that uniquely recognizes a program. Birthmark of the software is based on Heap Graph. It is generated by using Google Chrome Developer Tools when the program is in execution. Software's behavioural structure is demonstrated in the heap graph. It describes how the objects are related to each other to deliver the desired functionality of the website. Our aim is to develop and evaluate a system that can find theft/similarity between websites by using Agglomerative Clustering and Improved Frequent Subgraph Mining. To identify if a website is using the original program's code or its module, birthmark of the original program is explored in the suspected program's heap graph. The software is 100 % accurate and finds the theft/similarity between websites. Moreover, it is possible to detect theft/similarity even if the website is obfuscated.**

***Keywords*—Heap graphs, theft identification, dynamic birthmark, agglomerative clustering, improved frequent subgraph mining.**

I. INTRODUCTION

Software industry is severely suffering from piracy. It is our foremost responsibility to stop or at least reduce the rate of piracy. Software theft, also referred as software piracy is an unlicensed copy as well as use of computer programs [1, 2]. Mostly piracy is done by private individuals who copy programs from the workplace to their computers at home. Since data is not so difficult to copy, and the practice of unlawful software is very tough to discover, it is challenging to end software piracy [3].

JavaScript is becoming very popular and hence JavaScript programs are valuable belongings to several companies. JavaScript is an interpreted computer programming language also called as interpreted language because the code of JavaScript program is compiled into machine readable code when it is run by the interpreter.

In order to make the client-side scripts interaction with users, JavaScript was implemented as a part of web browser. This led the user control the browser, communicate and alter the website content that was displayed.

Due to occurrence of Web 2.0 and the fact that excellent platform for developing windows 8 apps are HTML 5 and JavaScript. So it is obvious that JavaScript is the best popular programming language for developing websites. In a survey conducted by Evans Data it was observed that 60% web developers use JavaScript. Use of JavaScript has surpassed all the scripting languages and 3GL [4]. However the source code of JavaScript programs can be easily obtained since it is an interpreted language and most browsers provide very easy method to obtain the source code of web pages and hence it is a threat to the industry to protect the intellectual property rights of the JavaScript designers. Software protection is a significant area for computer experts. Several techniques have been introduced for avoiding software stealing, out of them utmost generally used are watermarking and code obfuscation. Code Obfuscation makes the source code of a program difficult to understand by the humans and watermarking proves the ownership of the program.

Software watermark is the earliest and well-known 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, 6].Watermarking embeds the secret message into the cover image. But watermark can easily be defaced by the strong-minded invader. The owner of the program has to take some extra actions earlier to release the program. Therefore JavaScript developers obfuscate their code before releasing their software.

Code obfuscation is the practice of making code unintelligible and difficult to understand. In code obfuscation the code is converted, that fluctuates the physical look of the program, without changing the black-box provisions of the code of a program.

Hence code obfuscation is also recognized as the semantic-preserving technique to convert the code to change the constructions of the program alters while it's meaning and the functionality doesn't change [7]. 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 behaviour of a program to determine the similarity between programs. A software birthmark could be used to recognize software theft even after finishing the watermark by code transformation.

According to Wang et al. [8], a birthmark is a unique feature a program can have. It can be used to identify the program. To detect software theft,

1) The birthmark of the genuine program (the plaintiff program) is extracted first.
2) The birthmark extracted from genuine program is explored in the heap graph of suspected program.
3) If the birthmark of the program is found in the suspected program, then it can be demanded that theft/similarity is detected.

II. Related Work

Myles et al. established the first dynamic birthmark system. To identify the program, they explored the complete control flow trace of a program implementation. It was proved in their experiments that their technique can struggle 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 [9, 10].

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. Runtime information of API calls was used as a strong signature of the program. The dynamic birthmark was mined by observing the execution order and the frequency distribution of API calls. These mined dynamic birthmarks can distinguish individually established identical purpose applications and could resist to different compilers.

This promising result motivated the researches to work on dynamic birthmarks based on API calls [11].

Schuler et al. proposed a dynamic birthmark for Java that perceives how a program uses objects provided by the Java Standard API. The small orders of method calls received by distinct objects from Java Platform Standard API were detected. The call traces then were divided into a group 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. Moreover, they presented that all the birthmarks of obfuscated programs were identical to that of the original program [12]. API birthmark was more scalable and more resilient than the Whole Program Path Birthmark by Myles and Collberg [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 behavior of a program. Every vertex of the graph is a system call and edges of the graph represent the data and control dependences between system calls. Their software theft detection system was on the basis of SCDG birthmark. Evaluation of their system showed that it was vigorous against attacks based on obfuscation techniques and dissimilar compilers. They developed the first system that is capable of finding software component theft where only some part of code is stolen [8].

Chan et al. proposed the first dynamic birthmark based on the run-time heap for JavaScript programs. It is in the form of an object reference tree. They used tree comparison algorithm so as to compare two birthmarks and gave a similarity score between the birthmarks. Due to problem of efficiency for 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. Even though 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. However, the size of the heap graph was limited; the current birthmark captured far more information than the earlier system. Furthermore, the assessment of the birthmark system was of much larger scale where 200 websites compared with 20 JavaScript programs in their work and the results were promising [14, 15].

International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347 - 6435 (Online)) Volume 3, Issue 2, August 2014)

Later, they proposed another heap based birthmark system. The birthmark system was for detecting theft in Java programs. For birthmark detection, graph isomorphism algorithm was used. 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 [16].

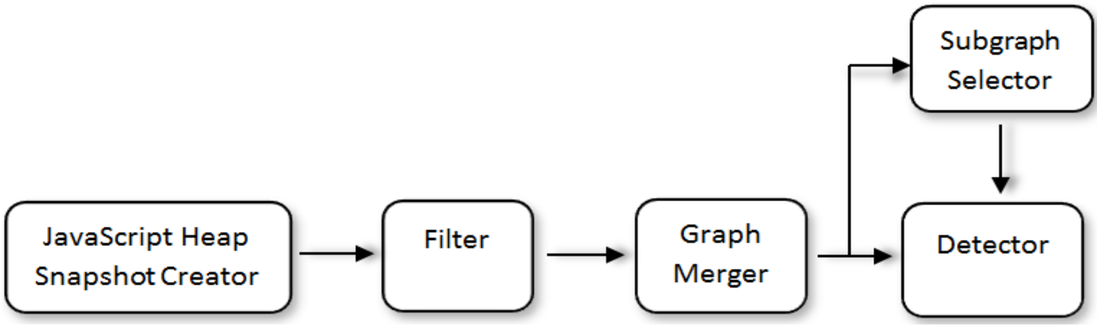

Figure I System Overview [17]

III. Methodology

Figure I show the overview of birthmark system. It summaries the procedures followed by the plaintiff program and the suspected program [17].

JavaScript Heap Snapshot Creator - Runs a JavaScript program and takes multiple heap snapshots in the course of its execution.

Filter - Traverses the objects in the heap snapshots and filters out objects.

Graph Merger - Merges the filtered heap graphs together to form one single graph.

Subgraph Selector - Selects a subgraph from the heap graph to form the birthmark of the plaintiff program. This step is not needed for the suspected program.

Detector - Searches for the birthmark of the plaintiff program in the heap graph of the suspected program.

A. JavaScript Heap Snapshot Creator

Being an interpreted language, JavaScript allows for the creation of objects at any time. On the other hand, one of the design elements of the V8 JavaScript engine is efficient garbage collection. As a result, the JavaScript heap keeps changing due to object creations and garbage collections.

To make full use of the behavior exhibited by the objects in the heap, each object is seized which 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, in every 2 seconds, the browser keeps discarding the JavaScript heap.

B. Filter

Google Chromium browser generates the heap dumps in the form of object reference trees. It is similar to the object reference graph where nodes represent the objects and edges represent the references between them. The only difference is that objects are duplicated to remove cycles in the graph. 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.

Objects in the V8 JavaScript heap are divided into twelve categories, HIDDEN, ARRAY, STRING, OBJECT, CODE, CLOSURE, REGEXP, NUMBER, NATIVE, SYNTHETIC, CONCATENATED STRING, and SLICED STRING. Objects that belong to INTERNAL, ARRAY, STRING, and CODE categories are not included in heap graphs. Hence all the objects other than these four are included in the heap graph. References between objects in the V8 JavaScript heap are divided into six categories CONTEXT VARIABLE, ELEMENT, PROPERTY, INTERNAL, HIDDEN, SHORTCUT and WEAK. References that belong to CONTEXT VARIABLE and INTERNAL categories are not included in the heap graph. Therefore, only ELEMENT, PROPERTY, INTERNAL, HIDDEN, SHORTCUT and WEAK objects are included in the heap graph.

C. Graph Merger

JavaScript engine assign a unique ID to every object in the JavaScript heap. Moreover, the ID of an object remains the same across multiple dumps and so it can be used to identify the object. The Filter also annotates each node in the heap graph with its object ID. Thus, it is easy to detect if two nodes in two heap graphs denote the same object. In this system, the graph merger takes two heap graphs as input and outputs a merged single graph that includes all the nodes and edges appearing in the input heap graphs. To accomplish this we have used Agglomerative Clustering to merge the graphs. Agglomerative Clustering is a Hierarchical Clustering approach for merging graphs. Agglomerative Clustering begins with two different heap graphs as an input and outputs the final single merged heap graph. Agglomerative Clustering is applied on both the heap snapshots of genuine as well as suspected website.

It takes the filtered heap graph in the form of heap snapshot from the second module Filter.

D. Subgraph Selector

After going through the above steps, the subsequent heap graph contains objects that are related to the functionality of that program only and can be used to identify the JavaScript program. However, it is difficult to use the whole graph as the birthmark of the program since the graph is too large for the detection step. The graph, which can comprise hundreds of nodes, is too large for the algorithm and may lead to very long execution time. Improved Frequent Subgraph Mining is used to select the subgraph of program. Improved Frequent Subgraph M is essential in order to get the frequently occurring nodes. Frequent nodes are those that occur several times in the snapshot. To be precise, if a node is frequent then it is said that the node is called many-a-times while the website was being executed. Using FSM, we get all those nodes which contribute only to the functionality of the program. Summarizing all, subgraph selector selects the small graph from the whole graph of plaintiff program in such a way that it can be formed a birthmark of the plaintiff program.

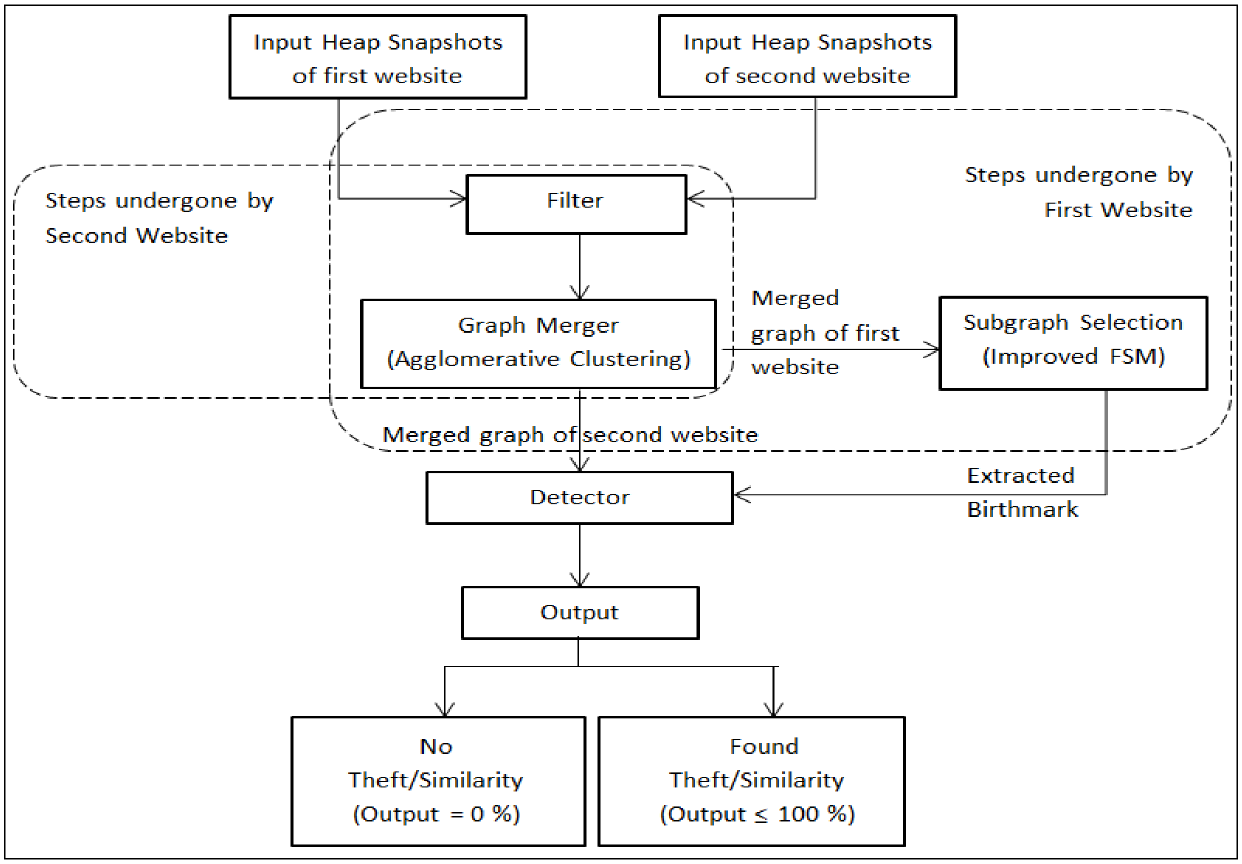
Input Heap Snapshots of first website
Input Heap Snapshots of second website
Steps undergone by Second Website
Steps undergone by First Website
Filter
Graph Merger (Agglomerative Clustering)
Merged graph of first website
Subgraph Selection (Improved FSM)
Merged graph of second website
Detector
Extracted Birthmark
Output
No Theft/Similarity (Output = 0 %)
Found Theft/Similarity (Output ≤ 100 %)

Figure II Computational Steps [18]

E. 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. It takes subgraphs of the objects found from heap graph of genuine program and checks whether the subgraph of the plaintiff program can be found in heap graph of suspected program. Once there is a match found, the detector raises an alert and reports where the match is found.

Figure II shows the computational steps undergone by the system. Computation starts with taking heap snapshots of first and second websites. Snapshots are passed to filter where extra objects are removed out.

Now filtered nodes and edges are given to Graph Merger. Since now only nodes and edges are remained we call it as heap graph. Graph Merger merges these filtered nodes from two different snapshots of same website. Agglomerative Clustering is used to merge graphs. Now merged graph of second website is given as input to Subgraph Selector. It selects the birthmark using Improved FSM. At last, Detector takes merged heap graph of second website and birthmark extracted by Subgraph Selector and searches the birthmark against the heap graph of second website. It produces the output in percentage. The output is categorized into 2 types on the basis of percentage of theft/similarity detected.

IV. EXPERIMENTAL ANALYSIS

As Chan et al. [14] worked on 200 websites; we also used the same set of websites to check the accuracy of our system. We conducted same experiments on the implemented system. We observed that like previous system, our system also detects same number of websites developed by using both the JavaScript frameworks viz. Prototype and Mootools. Hence we can say that the system gives accurate results. The results are as shown in Table I. Fig. 3 depicts the bar chart representation of experimental results carried out in Table I.

TABLE I
EXPERIMENTAL RESULTS

JavaScript Framework	System Detection Results	Manual Checking Results	Accuracy
Prototype	21 hits	21 hits	100 %
Mootools	25 hits	25 hits	100 %

The system is developed in three stages:

1. System without using Agglomerative Clustering.
2. Using Agglomerative Clustering and FSM.
3. Using Agglomerative Clustering and Improved FSM.

The systems were checked for resistance against obfuscation and we obtained results shown in Table II.

TABLE III
OBFUSCATION RESULTS

System	Detection Rate
System without using Agglomerative Clustering	80%
Using Agglomerative Clustering and FSM	50%
Using Agglomerative Clustering and Improved FSM	80%

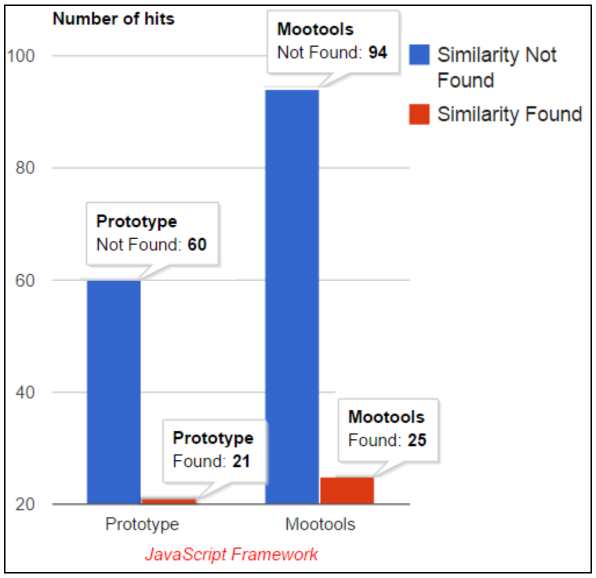

Figure III Experimental Analysis [18]

Acknowledgment

We are thankful to colleagues and reviewers for their suggestions which helped us to proceed further for this research.

REFERENCES

[1] S. Patel and T. Pattewar, Software Birthmark for Theft Detection of JavaScript Programs: A Survey, in Proc. Of International Conference on Recent Trends and Innovations in Engineering and Technology, International Journal of Advance Foundation and Research in Computer (IJAFRC), Volume 1, Issue 2, pp. 29-38, Feb 2014.

[2] Software theft, [accessed on April 14, 2013]. [Online]. Available: http://www.javvin.com/softwareglossary/SoftwareTheft.html

[3] Software piracy, [accessed on April 14, 2013]. [Online]. Available: http://www.fastiis.org/our services/enforcement/software theft/

[4] E. Data, JavaScript dominates EMEA development, Jan 2008, [accessed on April 14, 2013]. [Online]. Available: http://www.evansdata.com/press/viewRelease.php?pressID=127

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

[6] A. Monden, H. Iida, K. I. Matsumoto, K. Inoue, and K. Torii, Watermarking java programs, in International Symposium of Future Software Technology, Nanjing, China, 1999.

[7] C. Collberg, C. Thomborson, and D. Low, A taxonomy of obfuscating transformations, University of Auckland, Auckland, New Zealand, Tech. Rep. 148, 2003.

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

[9] G. Myles and C. Collberg, Detecting software theft via whole program path birthmarks, in Inf. Security 7th Int. Conf. (ISC 2004), Palo Alto, CA, pp. 404-414, September 2004.

[10] G. Myles and C. Collberg, K-gram based software birthmarks, in Symposium on Application Computing (SAC 05), ACM, Ed., pp. 314-318, 2005.

[11] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. I. Matsumoto, Design and evaluation of dynamic software birthmarks based on API calls, Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5 Takayama-cho, Ikoma-shi, Nara, 6300101 Japan, Technical Report, 2007.

[12] D. Schuler, V. Dallmeier, and C. Lindig, A dynamic birthmark for java, in IEEE/ACM International Conference of Automated Software Engineering (ASE 07), no. 22, New York, pp. 274-283, 2007.

[13] H. Tamada, M. Nakamura, and A. Monden, Design and evaluation of birthmarks for detecting theft of java programs, in IASTED International Conference of Software Engineering, 2004, pp. 569-575

[14] P. F. Chan, C. K. Hui and S.M. Yiu, Heap graph based software theft detection, IEEE Transaction on Information Forensics and Security, vol. 8, pp. 101-110, January 2013.

[15] P. F. Chan, C. K. Hui and S.M. Yiu, Jsbirth: Dynamic JavaScript birthmark based on the run-time heap, in 2011 IEEE 35th Annual Computer Software and Application Conference (COMPSAC), pp. 407-412, July 2011.

[16] P. F. Chan, C. K. Hui and S.M. Yiu, Dynamic software birthmark for java based on heap memory analysis, in IFIP TC 6/TC 11 Int. Conf. Communication and Multimedia Security (CMS11), Springer-Verlag, Ed., no. 12, Berlin, Heidelberg, pp. 94-106, 2011.

[17] S. J. Patel and T. M. Pattewar, Software Birthmark Based Theft Detection of JavaScript Programs Using Agglomerative Clustering and Frequest Subgraph Mining, in Proc. Of IEEE International Conference on Embedded Systems (ICES), Coimbatore, Tamil Nadu, 2014.

[18] Swati Patel, Software Birthmark Based Theft/Similarity Comparisons of JavaScript Programs, SES's R. C. Patel institute of Technology, Shirpur, MS, India, Technical Report, 2014.