© 2016 IJSRSET | Volume 2 | Issue 2 | Print ISSN : 2395-1990 | Online ISSN : 2394-4099
Themed Section: Engineering and Technology

Improving Performance of Graph Selector in Heap Graph Based Software Theft Detection

Prachi M. Tamgadge, Prof. S. A. Murab
Department of Computer Engineering JCOET, Yavatmal, Maharashtra, India

ABSTRACT

Various applications for web users are designed using client side scripting language such as JavaScript. This source code of JavaScript can be easily copied through browser. JavaScript was implemented as a part of web browsers, so that the user to can control the browser and alter the displayed website contents. Hence the informal cribbing of JavaScript code has become the serious threat. Software watermarking and code obfuscation are two approaches to detect software piracy. But watermarks can be defaced and code obfuscation cannot prevent the code from being copied. Hence software birthmark is introduced in the program to detect the code theft of JavaScript programs. The largest object graph is chosen to become the birthmark of a program. The software birthmark is extracted using the run-time heap. The behavioral structure of the software is extracted into heap graph which shows how the objects are linked together. The aim is the improvement of the graph selector to choose the graph extracted from the program.

Keywords: Watermarking, Obfuscation, Software Birthmark, Heap Graph, Theft Detection

I. INTRODUCTION

Amongst the various platforms for programming JavaScript has become the popular platform for the development of various applications. It offers various features to the developer for the ease of programming. Nowadays the source code of JavaScript programs can be readily available as it is an interpreted language. Hence for protecting the code theft methods like watermarking are used. Watermarking is one of the well-known approach to detect software piracy in which a watermark is added into a program to prove the ownership of it [1]. However, it is believed that "a sufficiently determined attacker will eventually be able to defeat any watermark." [2]. Watermarking [3] also requires the owner to take extra action such as embedding the watermark into the code prior to releasing the software. Thus, some existing JavaScript 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. However, it only prevents others from learning the logic of the source code but does not protect them from being copied. 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 a program to determine the similarity between two programs. A birthmark could be used to identify software theft even after destroying the watermark by code transformation. There are two categories of software birthmarks, static birthmarks and dynamic birthmarks. Static birthmarks are extracted from the syntactic structure of a program. Dynamic birthmarks are extracted from the dynamic behavior of a program at run-time. 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. Identifying same or similar code fragments among different programs or in the same program is very important in some applications. For example, duplicated codes found in the same program may degrade efficiency in both development and execution phase. Code identification techniques such as clone detection can be used to discover and refactor the identical code fragments to improve the program. For another example, same or similar code found in different programs may lead to even more serious issues. If those programs have been individually developed by different programmers, and if they do not embed any public domain code in common, duplicated code can be an indication of software plagiarism or code theft. In code theft cases, determining the similarity of two code fragments becomes much more difficult since plagiarizers can use various code transformation techniques including code obfuscation techniques to hide stolen code from detection. In order to handle such cases, code characterization and identification techniques must be able to detect the identical

code without being easily circumvented by code transformation techniques

A redesigned heap graph based birthmark for JavaScript to make it a scalable and robust solution for detecting software theft can be used. The proposed birthmark is formed by extracting objects from the heap and building a heap graph out of them. A heap graph is a simple directed graph in which the nodes represent the objects and the edges represent the references between them. Since not all the objects and references stem from the software itself, further filtering on them is performed to let us focus on objects and references that truly represent the behavior of the software. The first kind of nodes filtered out are those that are created by the browser. They include, among the others, the objects that are created for the DOM tree and closures of JavaScript builtin functions. The second kind of nodes filtered out are those that are not accessible from the JavaScript program. For references, only the references created for context variables are filtered as they are not accessible from the JavaScript program. The filtered graph forms the birthmark of the program.

II. METHODS AND MATERIAL

A. Literature Review

The term birthmark was first used by Grover [4] where the term was used to mean the unique characteristics exhibited by the program which can be useful to identify the program. The term "birthmark" differs from the term "fingerprint" in that the characteristics used to embed the fingerprint are intentionally placed in the code. The general idea of a software birthmark is similar to that of a computer virus signature.

The first dynamic birthmark was proposed by Myles et al.[5], 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. Whole Program Paths (WPP) is a technique presented to represent a program's dynamic control flow. The WPP is constructed by collecting a trace of the path executed by the program. The trace is then transformed into a more compact form by identifying its regularity, which is repeated code. To collect the trace the edges of the program's control flow graph are instrumented, by uniquely labelling each edge. As the program executes the edges are recorded, producing a trace. 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.

Haruaki Tamada et. al, proposed birthmarks based on Dynamic Software Birthmarks to Detect the Theft of Windows Applications[6]. Applications running on the operating system can use many features called API function calls, provided by the Operating System. The typical API function calls are file input/output, synchronized objects such as semaphore, mutual exclusion and critical section, user interface and graphics. Since the most of API function calls cannot be replaced by other instructions without affecting the program behaviour, history of their executions can be used as robust birthmarks. For example, the high level OS does not allow direct operations to the file system from user applications, and it only allows file input/output via API function calls. Also, the operations to GUIs are allowed only via API function calls. It indicates that the birthmark using API function calls has good tolerance against program transformation attacks.

Haruaki Tamada et. al, proposed design and evaluation of birthmarks for detecting theft of java programs[7]. They presented four types of birthmarks to provide a reasonable evidence of theft of Java class files. The results showed that the proposed birthmarks could successfully distinguish (non-copied) class files in practical Java packages except some tiny classes, and that they achieved relatively good tolerance to program obfuscation. Compared to watermarking, the advantage is that the birthmarks are easily used without any extra code. Limitation is that birthmarks might be a bit weaker evidence than watermarks. However, watermarking and birthmarking are not exclusive techniques. Hence, they suggested combined use of watermarking and birthmarking would cover the limitation of each other.

Ginger Myles and Christian Collberg introduced the k-gram based software birthmarks[8]. A k-gram is a contiguous substring of length k which can be comprised of letters, words, or opcodes. The k-gram birthmark is based on static analysis of the executable program. For each method in a module the set of unique k-grams by sliding a window of length k over the static instruction sequence as it is laid out in the executable is computed. The birthmark for the module is the union of the birthmarks of each method in the module. The order of the k-grams within the set is unimportant as is the frequency of occurrence of each k-gram. By using the unique k-grams without their associated frequency the birthmark is less susceptible to semantics-preserving transformations. For example, an obfuscation which duplicates basic blocks will increase the frequency of those k-grams in the block. Additionally, because the birthmark is independent of the order of the methods in the module or the modules within the program, the technique can be used at the module or program level. In order to use k-grams to uniquely identify a program it must be true that a specific set of k-grams is unique to a program.

Tamada et. al, proposed birthmarks based on design and evaluation of dynamic software birthmarks based on API Calls[9]. Applications running on an operating system (OS) can use various build-in features of the OS by calling APIs. The file input/output, synchronized objects such as semaphore, mutex and critical section and graphic user interface (GUI) are the typical API function calls. For API calls involved in a program p (in a binary form), they focused on the following properties.

- ✓ It is hardly possible to replace the API calls with other instructions without changing the behaviour of p.
- ✓ In general, a compiler does not optimize the APIs themselves.

As for the first property, they assumed a relatively complex program (application) operating on the recent sophisticated operating systems such as MS Windows, where every access to system resources is strictly managed via APIs. In such OS, any operation to the file system, for instance, must be done via file I/O APIs. The operations to GUIs (widgets) must be also performed by API calls. Hence, it is almost impossible to alter these API calls with other user-made instructions.

David Schuler et al. [10] proposed a dynamic birthmark for Java that perceives how a program uses objects provided by the Java Standard API. To extract a birthmark from a program, they statically instrument the byte code of the program as well as the byte code of the Java API classes and then run the program. The instrumentation detects for each API object the methods invoked from the program. From this information the birthmark is computed at runtime in memory (for efficiency) and written to a file when the program terminates. The key idea is to replace each API call site in the user program with a call to a proxy method that was added to the API class, which requires instrumentation of both the API and the program itself. Using method interposition, they captured all method calls from the user program to the API, whereas API-to-API calls remain unaltered. 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. API birthmark was more scalable and more resilient than the Whole Program Path Birthmark by Myles and Collberg.

Wang et al. [11] put forward behaviour based software theft detection. A system call dependence graph (SCDG) is a graphical representation of the behaviours of a program, is a good candidate for behaviour based birthmarks. In a SCDG, system calls are represented by vertices, and data and control dependences between system calls by edges. A SCDG shows the interaction between a program and its operating system and the interaction is an essential behaviour characteristic of the program. Although a code stealer may apply compiler optimization techniques or sophisticated semantic preserving transformation on a program to disguise original code, these techniques usually do not change the SCDGs. It is also dicult to avoid system calls, because a system call is the only way for a user mode program to request kernel services in modern operating systems. For example, in operating systems such as Unix/Linux, there is no way to go through the file access control enforcement other than invoking open()/read()/write() system calls. Although an exceptionally sedulous and creative plagiarist may correctly overhaul the SCDGs, the cost is probably higher than rewriting his own code, which conflicts with the intention of software theft. After all, software theft aims at code reuse with disguises, which requires much less eff ort than writing one's own code. To extract SCDG birthmarks, automated dynamic analysis is performed on both plaintiff and suspect programs to record system call traces and dependence relation between system calls. Since system calls are low level implementation of interactions between a program and an OS, it is possible that two dierent system call traces represent the same behaviour. Thus, they filtered out noises, which cause the diff erence, from system call traces. Then, SCDGs are constructed and both plaintiff and suspect SCDG birthmarks are extracted from the SCDGs. Evaluation of their system showed that it was vigorous against attacks based on obfuscation techniques and different compilers. It is the first system that is able to find software component theft where only some part of code is stolen.

Chan et al. [12] proposed the first dynamic birthmark based on the run-time 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.

Later, they proposed another heap based birthmark system [13]. This time, the birthmark system was for detecting theft

in Java programs. Graph isomorphism algorithm was used for birthmark detection. As graph isomorphism is too restrictive and makes the birthmark system vulnerable to reference injection attack. They used the largest heap graph from the program to be the birthmark of the program.

But most of the previous techniques cannot handle advanced obfuscation techniques. The methods based on source code analysis are not practical since the source code of suspicious programs typically cannot be obtained until strong evidences have been collected.

B. The Structure of A Heap Graph

A heap graph [13] is a simple directed graph in which the nodes represent the objects and edges represent the references between them. The structure of the heap graph is as shown in the following figure.

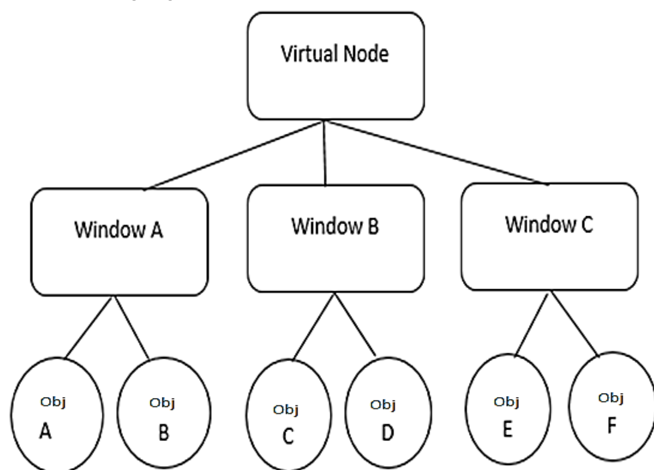

Figure 1. Structure of Heap graph

A heap graph starts with the virtual node which is the entry point to all the nodes in the heap. The virtual node points to one or more window objects which represent the different DOM windows residing on the web page. A window object in turn points to the various objects in its DOM windows. The objects under the window nodes are compared on the basis of their sizes in terms of the number of nodes and number of edges reachable from the nodes of them.

C. The Heap Graph Based Theft Detection System

Following figure shows the overview of software birthmark system [13]. It outlines the processes that the plaintiff program and the suspected program undergo. The objects of heap graph are considered as the nodes and the references are treated as the edges.

The JavaScript heap profiler is used to take the snapshots. The snapshots are in the form of heap graphs which are accessible through the virtual nodes of the heap graph generated. The heap profiler first triggers the garbage collections so that the weakly reachable objects are ensured to be reachable from the root nodes. The heap contents are iterated to count the entries and references. The references are filled between the entries. The dominators of the entries are set. Further the retained size of each entry is calculated. Thus in this way a snapshot is taken and converted into the text file.

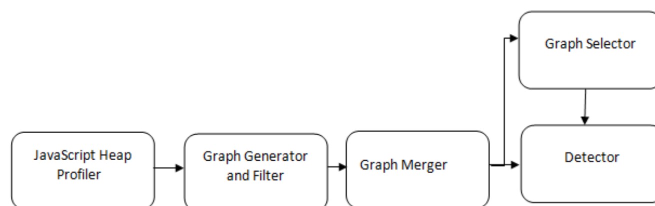

Figure 2. Heap graph based software theft detection

The graph generator takes the output of the heap profiler as an input. For each snapshot a depth first search traversal is performed. Then it is passed to the filter. The filter traverses the objects in the heap snapshots and builds heap graphs out of them. All objects and references never represent the behavior of the system. For this reason the filtering is important. So that the objects and references which purely depicts the behavior of the software are concentrated more. The output of the graph generator and filter is a set of heap graphs captured at different points of time.

The graph merger takes the multiple labelled connected heap graphs from generator and filter as an input. The superimposition of all the graphs is done one by one. The union set of nodes and edges of the two graphs is considered.

The graph selector selects a graph from the heap graph to form the birthmark of the plaintiff program. The largest object graph reachable from the node is chosen as the birthmark because it captures the most information of the heap. This step is not needed for the suspected program.

The detector takes the graph from the original program and entire heap graph of the suspected program as an input. Finally, the detector searches for the birthmark of the plaintiff program in the heap graph of the suspected program. Once there is a match found the detector raises an alert and reports where the match is found.

III. RESULTS AND DISCUSSION

Proposed Approach

The performance of graph selector is focused because currently largest object graph is chosen to become the birthmark of the program. But the birthmark should be more representative of the program. The time taken by large graph

for mining is slow. So performance tuning is done on graph selector to make it more robust and practical. We take the two heap snap shot for request searched in the chrome browser. The heap snap shot contains all nodes like object, arrays, string, closure, etc. The snapshot file is processed and object contents are retrieved. Next we get the distinct objects present in the each heap snapshot. The overall size for the each object present in the each heap snap shot is calculated. The two heap snapshots are merged. The distinct value is computed on the basis of comparison algorithm. By using this distinct value software theft will be predicted. The comparison algorithm is as follows:

```
match outputs(List I, List O, split-seq-Node N)
if O is empty then
    return true
end if
o1 head(O)
for all k to N children do
k.matchSet = k.matchSet{o1}
if matchOutputs(I, k.matchSet, k ) then
    if matchOutputs(I, tail(O), N) then
    return true
    end if
end if
```

IV. CONCLUSION

The heap graph extracted from the program is used as a birthmark to identify the program. The similar functioning programs are compared to detect the software theft. This system is reliable because the birthmark cannot be defaced. It provides a novel technique of using heap graph as a birthmark.

V. REFERENCES

[1] C. Collberg and C. Thomborson, "Software watermarking: Models and dynamic embeddings," in Proceeding Symposium Principles of Programming Languages (POPL'99), 1999, pp. 311–324.

[2] A. Monden, H. Iida, K. I. Matsumoto, K. Inoue, and K. Torii, "Watermarking java programs," in Proceeding International Symposium 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 Proceeding ACM SIGPLAN 2004 Conference Programming Language Design and Implementation (PLDI '04), New York, 2004, pp. 107–118, ACM.

[4] Derrick Grover. Program identification. In Derrick Grover, editor, The Protection of Computer Software – Its Technology and Applications, pages 122–154. Cambridge University Press, 1989.

[5] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in Proceeding Inf. Security 7th International Conference (ISC 2004), Palo Alto, CA, Sep. 27–29, 2004, pp. 404–415.

[6] H. Tamada, K. Okamoto, M. Nakamura, and A. Monden, "Dynamic software birthmarks to detect the theft of windows applications," in Proceeding International Symposium Future Software Technology Xian, China, 2004.

[7] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. I. Matsumoto, "Design and Evaluation of birthmarks for detecting theft of java programs", in Proceeding IASTED International Conference Software Engineering, 2004, pp. 569-575.

[8] G. Myles and C. Collberg, "K-gram based software birthmarks," in Proceeding 2005 ACM Symposium Application Computing (SAC '05), New York, 2005, pp. 314–318, ACM.

[9] 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, Rep., 2007.

[10] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for java," in Proceeding 22nd IEEE/ACM International Conference Automated Software Engineering (ASE '07), New York, 2007, pp. 274–283, ACM.

[11] X. Wang, Y. C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in Proceeding 16th ACM Conference Comput. and Commun. Security (CCS '09), New York, 2009, pp. 280–290, ACM

[12] P. Chan, L. Hui, and S. Yiu, "Jsbirth: Dynamic JavaScript birthmark based on the run-time heap," in Proceeding. 2011 IEEE 35th Annual Comput. Software and Applications Conference (COMPSAC), July 2011, pp. 407–412.

[13] P. Chan, L. Hui, and S. Yiu, "Heap graph based software theft detection" 2013 IEEE Transactions on Information Forensics and Security, 2013,v. 8 n. 1, p.101