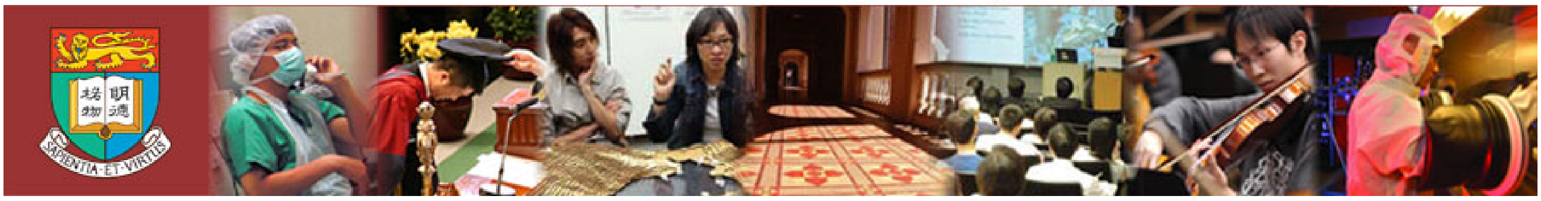

2011 35th IEEE Annual Computer Software and Applications Conference

JSBiRTH: Dynamic JavaScript Birthmark Based on the Run-time Heap

Patrick P.F. Chan, Lucas C.K. Hui, S.M. Yiu
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
{*pfchan*|*hui*|*smyiu*}@*cs.hku.hk*

***Abstract*—JavaScript is currently the dominating client-side scripting language in the web community. However, the source code of JavaScript can be easily copied through a browser. The intellectual property right of the developers lacks protection. In this paper, we consider using dynamic software birthmark for JavaScript. Instead of using control flow trace (which can be corrupted by code obfuscation) and API (which may not work if the software does not have many API calls), we exploit the run-time heap, which reflects substantially the dynamic behavior of a program, to extract birthmarks. We introduce JSBiRTH, a novel software birthmark system for JavaScript based on the comparison of run-time heaps. We evaluated our system using 20 JavaScript programs with most of them being large-scale. Our system gave no false positive or false negative. Moreover, it is robust against code obfuscation attack. We also show that our system is effective in detecting partial code theft.**

***Keywords*-birthmark; software protection; code theft detection; JavaScript**

I. Introduction

The rise of Web 2.0, in particular the Ajax technology, rendered JavaScript the dominating client-side scripting language in the web community. According to a survey from Evans Data, over 60% of developers use JavaScript and that usage has outstripped all 3GL and scripting language use, including Java [1]. However, JavaScript source code can be readily obtained through the view-source function of a browser. There is no doubt that the intellectual property right of web application developers is at risk.

Software protection continues to be an important topic for computer scientists. Watermarking is one of the well-known and earliest approaches to detect software piracy in which a watermark is incorporated into a program to prove the ownership of it [2], [3]. However, it is believed that "a sufficiently determined attacker will eventually be able to defeat any watermark." [4]. Watermarking also requires the owner to take extra action (embed the watermark into the software) 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 [5]. However, code obfuscation only prevents others from learning the logic of the source code but does not hinder direct copying of them.

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 [6], [7], [8], [9], [10], [11], [12], [13]. It was shown in [7] that a birthmark could be used to identify software theft even after destroying the watermark by code transformation. According to Wang et al. [6], 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 birthmarks extracted from the syntactic structure of a program [10], [12], [13]. Dynamic birthmarks are birthmarks extracted from the dynamic behavior of a program at run-time [6], [7], [8], [9], [11]. 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. We remark that there does not exist any proposed software birthmark, static or dynamic, for JavaScript.

Existing dynamic birthmarks make use of the API, system call, or complete control flow trace obtained during the execution of a program [6], [7], [8], [9], [11]. Birthmarks based on control flow may still be vulnerable to obfuscation attack such as loop transformation. The ones based on API (or system call) may suffer the problem of not having enough API calls to make the birthmark unique. In this paper, we provide an alternative for extracting the dynamic birthmark based on the unique characteristics of a program extracted from the run-time heap. The run-time heap is a location in the main memory which stores all dynamically created objects. The structures of the objects as well as the connection between them reflect the unique behavior of a program. They are independent of the syntactic structure of the program code and hence, are not to be changed by semantics-preserving code transformations.

We developed a system, called JSBiRTH, to make use of the run-time heap to detect software piracy for JavaScript. The core idea of the system is to extract the object reference tree for each object in the run-time heap and compare the similarity between them. The system prototype for evaluation consists of two modules. The first module performs birthmarks extraction. We modified the Google Chromium browser to dump the heap on a regular interval when JavaScript programs are running [14]. The second module is responsible for birthmarks comparison. We designed and implemented a birthmark comparison algorithm to compare the birthmarks given by the first module.

To test the effectiveness of the system, 20 existing JavaScript programs, with most of them being large-scale, were downloaded from the Internet. Our system showed zero similarity between different programs and high similarity, more than 96% , between birthmarks from the same program. For robustness against code obfuscation attack, the system showed more than 95% of similarity between the original and the obfuscated copy. Furthermore, we also show that the system is able to detect partial code theft by comparing a set of programs which use a common JavaScript library.

The rest of the paper is structured as follows. Section II discusses some related work. The problem definitions are given in Section III. Section IV provides the overview of our proposed approach. The details about the implementation of our birthmark system are given in Section V. The evaluation results of the system are discussed in section VI. Further discussion is covered in Section VII and Section VIII concludes.

II. Related Work

The first dynamic birthmark was proposed by G. Myles and C. Collberg [7]. 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 [9]. Their approach was based on the insights that it was difficult for adversaries to alter the API calls with other equivalent ones and that the compiler did not optimize the APIs themselves. They extensively used runtime information of API calls as a strong signature of the program. 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 resilience to different compiler options. This promising result led to subsequent researches on dynamic birthmarks based on API calls.

Schuler et al. proposed a dynamic birthmark for Java that observes how a program uses objects provided by the Java Standard API [8]. The proposed API birthmark observes 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 is 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. Moreover, all birthmarks of obfuscated programs were identical to that of the original program. Most importantly, their API birthmark was more scalable and more resilient than the WPP Birthmark by Myles and Collberg [7].

Wang et al. proposed dependence graph based software birthmark called SCDG birthmark [6]. 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. It is the first system that is able to detect software component theft where only partial code is stolen.

III. Problem Definitions

This section provides the static birthmarks and dynamic birthmarks to ease further discussion. We borrow part of the definitions from Tamada et al [9]. These are the first formal definitions appearing in the literature and have been restated in subsequent papers related to software birthmark.

A software birthmark is a group of unique characteristics extracted from a program that can uniquely identify the program. The purpose of software birthmarks is to detect whether two programs are from the same origin or not.

A. Static Birthmarks

A static birthmark is one that can be extracted solely from the program source code.

Definition 1. (Static Birthmark) Let p, q be two programs or program components. Let $f(p)$ be a set of program characteristics extracted from the source code of p. $f(p)$ is a static birthmark of p only if both of the following criteria are satisfied:

1) $f(p)$ is obtained only from p itself
2) program q is a copy of $p \Rightarrow f(p) = f(q)$

B. Dynamic Birthmarks

A dynamic birthmark is one that is extracted when the program is executing.

Definition 2. (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.

Static birthmarks are vulnerable to code obfuscation which changes the structure of a program without affecting the dynamic behavior of the program. As a result, a code thief can destroy a static birthmark by applying obfuscation to the stolen code and escape from theft detection. On the other hand, dynamic birthmarks are more robust to such kind of attack as they rely on the run-time behavior of the program.

IV. Approach Overview

This section provides an overview of the birthmark system. The system comprises two modules. The first module generates dump files of the run-time heap. It is done by the modified Google Chromium browser. The second module compares 2 dump files and calculates their similarity.

A. Tree Representation of the Heap

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. More importantly, individual objects can be accessed quickly through the virtual node which we will explain in a second.

B. Tree Merging

During execution, the Chromium browser keeps dumping out the heap at 5-second intervals until it gets 20 dumps. Before we can compare the heap dumps captured from the execution of two programs, the 20 dumps of each of the two programs must be merged into one single data structure.

Figure 1 illustrates the merging of two dumps that are captured during the execution of a JavaScript program. The two trees at the left are from two consecutive dumps. They are merged and the tree at the right is formed. Let the node with ID 12 be node A and the node with ID 10 be node B. Comparing the two trees, it is not difficult to see that node A is missing in the right tree. That means it has been removed after the first dump is captured. That is possible in real situation as objects could be removed by the garbage collector. Another change is that a new node, node B, has been created and added under the node with ID 14. That represents an object creation.

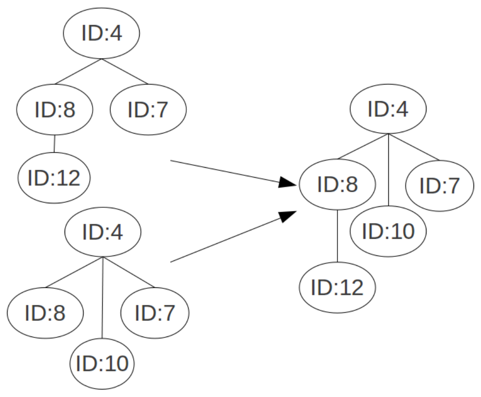

Figure 1. An example of merging two trees

After the merging, both node A and node B are retained in the new tree. Therefore, the new tree contains all the information captured in the two trees. Retaining all of the information is the main goal of this algorithm as every node and every edge is representing the characteristic of the program.

C. Object pruning

We ignore objects that apparently do not represent the behavior characteristic of a program.

1) Standard Objects: There are a lot of standard objects residing in the heap along with the custom objects in the JavaScript programs. Examples of such are HTMLDocument, JSON, DOMWindow, to name a few. Since these standard objects do not represent the characteristic of a program, we remove them from the object reference tree.

2) Auxiliary Objects: In JavaScript, even a function is an object in the heap. Consequently, there are a lot of ”secretly created” objects in the heap by the JavaScript engine. We ignore these objects as they do not represent the behavior of the program. Moreover, they can be misleading because their structure are very similar and independent of the behavior of the program. Considerable amount of such objects will make two unrelated programs look similar.

D. Node Comparison

After getting the two merged trees from two heap dump files, we are ready to compare them. Before going into the comparison process, we first discuss about the way we compare two nodes. Since the name of the node can be changed by code obfuscation, we focus on the structure of a node. Essentially, the structure of a node can be represented by the number of edges from that node as well as the types of those edges.

Let E_A , E_B be the set of edges coming out from node A and node B respectively denoted by their types. The similarity $NSim(A, B)$ of the two nodes is calculated as follows:

$$NSim(A,B) = \frac{|E_A \cap E_B|}{|E_A|}$$

A software plagiarizer can deliberately create useless references between objects. Therefore, we do not take into account the unmatched edges in node B, which is from the suspected program, when calculating the similarity between two nodes.

E. Object Reference Tree Comparison

After applying the above two pruning techniques, objects are compared in the following way. For each object Obj_{i1} in the plaintiff program p, an object Obj_{i2} of the same type with size difference less than a percentage m in terms of no. of node in the subtree is chosen from the pool of objects of the suspected program q such that $OSim(Obj_{i1}, Obj_{i2})$ is the highest among all unpaired objects in q. This process is repeated until all Obj_{i1} is paired up with an object in the suspected program or all objects in the suspected program has been exhausted. The similarity of two programs, $Sim(p, q)$ is given by:

$$Sim(p,q) = \frac{\sum OSim(Obj_{i1}, Obj_{i2}) \times \text{Size of } Obj_{i1}}{\text{Total Heap Size of } p}$$

The algorithm $OSim()$ for two objects recursively calculates the similarity between the nodes in their object reference trees. It first compares the root nodes based on their structure as mentioned before. It then groups the children of the root nodes based on their node types. Within each pair of groups of nodes having the same type, the tree comparison algorithm is called recursively to find out the similarity between pairs of subtrees. The subtrees are then paired up such that similar subtrees are in one pair. The final similarity score is calculated based on the similarity of the root node as well as the similarity of their subtrees weighted by their size.

Since, in practice, the object reference graph maybe of many levels, we limit the depth of the tree to d and stop exploring the tree once that level is reached. In the system prototype, we set d to 3 and m to 50%.

V. Implementation

We implemented a prototype of the run-time heap based birthmark system. The entire system consists of about 500 lines of C++ code. We leveraged the open source Chromium browser from Google of version 7.0.520.0 to build the birthmark extraction module [14]. The Google V8 JavaScript engine is the JavaScript engine embedded in the Chromium browser [15]. For the purpose of our research, we modified the V8 JavaScript engine such that it dumps out the heap whenever it comes across a $dumpHeap()$ function call when executing a JavaScript program. To achieve this, we leveraged a feature called function templates of the V8 JavaScript engine. The dump heap function is an API function provided by the V8 JavaScript engine. It dumps the run-time heap and returns a tree structure that contains the objects in the heap. In order to trigger the $dumpHeap()$ function during the execution of a JavaScript program, we instrument the candidate JavaScript programs by adding a code snippet at the beginning of them to call the $dumpHeap()$ function on every 5 seconds for 20 times.

VI. Evaluation

To evaluate our implementation of the birthmark extraction system, four groups of JavaScript programs were chosen for testing purpose. The first group consisted of 4 text editors. The second group consisted of 4 graphic libraries. The third group consisted of 5 games/game engines. The fourth group consisted of 7 programs which used the jQuery library [16].

We installed these programs on our testing server and extracted birthmarks from them using our modified Chromium browser. We then compared the birthmarks using our birthmark comparison program. The birthmark extraction and comparison were performed on a computer with Intel Core2Duo 2.66Ghz, 3GB Ram, running Ubuntu 10.10.

During the experiment, we launched the browser with a command line argument to specify the address of the testing program. This avoided loading of any default page which might lead to loading of extra objects in the heap. After launching the browser with the target page, we randomly played around with the software and triggered some functions of them. The modified Chromium browser dumped out the heap while we were playing around with the software. A message showed up after twenty heap dumps were captured. We then closed the browser and obtained a text file containing the twenty heap dumps captured. We repeated the above for each of the testing program.

A. Effectiveness

This part of the evaluation tested whether the system could effectively detect the similarity of two programs. Most importantly, a good birthmark system should not produce any false positive. The 3 groups of programs were independently developed similar purpose programs and served the objective of our evaluation well.

Among the four text editors, only NicEdit and jWYSIWYG had custom objects. There were some pairs of programs that could not be compared using our system because there was no custom object in either of them. This shows that our system is not applicable to all programs. For the graphic libraries and games game engines, since they were more large scale, all of them had custom objects.

Throughout the evaluation, the similarity between the same programs was always one (with two exceptions which are 0.985 and 0.967). The similarity between two distinct programs was always zero. This shows that our birthmark extraction and comparison method can determine similarity between two programs accurately. More importantly, there was no false positive meaning that it did not give high similarity score for two programs which were not copies of each other.

B. Robustness

Next, we tested the robustness of our system. That is, the ability for it to detect similarity between a program before and after obfuscation. Attackers of birthmarks usually exploit obfuscation to alter the birthmark of the infringing program copy. Therefore, robustness against such attack is of crucial significance.

We obfuscated 8 of our testing programs with the Jasob 3 obfuscator [17]. Jasob 3 is a state-of-the-art obfuscation tool for JavaScript used by hundreds of companies and individuals to protect and optimize their web content. We extracted birthmarks for each of the obfuscated program. For each of the program, we compared the birthmark extracted from the original copy and the birthmark extracted from the obfuscated copy. All of the pairs gave similarity higher than or equal to 0.955 and 5 of them gave similarity 1. This shows that our methodology is robust against obfuscation.

C. Ability to Detect Partial Code Theft

Finally, we tested whether our system could detect partial code theft. To simulate partial code theft, we leveraged programs using the same JavaScript library. We downloaded 8 programs which all used the jQuery library [16]. We updated the 8 programs to use the latest version of jQuery to make the experiment more effective.

We considered a sample jQuery object and tested its presence in the 8 jQuery programs. The presence of the jQuery library object was detected in all of the 8 programs. Furthermore, all the matched jQuery object in the 8 program demonstrated high similarity to the sample jQuery object.

This part of experiment showed that our system could identify library theft accurately.

VII. Discussion

A. Counterattacks

There are two ways to alter the structure of the object trees. The first way is to inject dummy objects to dilute the birthmark. The second way is to restructure the existing objects.

1) Objects Injection Attack: One possible attack to our scheme is to create a lot of dummy objects to dilute the birthmark. For each dummy object, there are two possible attack scenarios. The first scenario is that it is placed right below the virtual node (the node connecting all the objects). We assume this dummy object is not similar to any of the original object or at least not to the extent that it is chosen to be paired up with those objects in p. Otherwise, it will only contribute to the similarity score and will actually aid the detection. In the normal case, this dummy object is not similar to the original objects and is not paired up. As shown by the definition of $Sim(p, q)$ in section IV subsection E, the unpaired object in the suspected program q is not taken into account. Therefore, they have no effect on the similarity calculation. The second attack scenario is that the dummy object appears in the internal node of an object reference tree. Similar to the argument for the first attack scenario, the dummy object will be unpaired and will actually be ignored.

2) Object restructuring: Another possible attack to our scheme is to restructure an object. There are four possible attack scenarios. The first attack scenario is to remove some references from an object. This will most probably badly affect the behavior of the program and is not practical. The second attack scenario is to add some useless references to other objects. As shown in our definition of node similarity in section IV subsection D, the extra references to other object for a node in the suspected program will not affect the calculation of the node similarity. However, another effect is that the object will become more bulky and may be not be paired with the objects in the plaintiff program even though it is originally an object in the plaintiff program. This is controlled by the parameter m. The parameter m controls the balance between the false positive and false negative rate. On one hand, a small m will lower the false positive rate as it avoids an object in the plaintiff program being wrongly considered part of a huge unrelated object in the suspected program. On the other hand, a larger m will allow us to survive under this attack scenario and achieve a lower false negative rate. This shows that our system bares the same limitation of intrusion detection systems where there exists a fundamental trade-off between false positives and false negatives.

The third and fourth attack scenario are class splitting and class coalescing as suggested in [18] by M. Sosonkin et al. 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, our birthmark system is still robust as it considers the tree as a whole and dose not rely on special nodes that cannot be altered.

B. Comparison with the Current Approach

Current dynamic birthmarks make use of the system call or API call trace to extract characteristics of a program at run-time. However, some programs may not have enough system calls or API calls to make the call trace large enough to form a birthmark that can uniquely identify the program. Similarly, there are some programs that do not define enough custom objects so that our approach cannot extract a birthmark that can uniquely identify them. Rather than stating which approach is better, the two approaches should essentially be complements to each other such that more programs can benefit from software birthmarks to protect their copyright.

C. Future Work

An approach which makes use of both the system or API call traces and the heap dumps will be even more powerful. Conjunction of the two approaches can cover programs which use very few system or API calls but create considerable amount of custom objects or vice versa. Besides, both time and space complexity of the merging and birthmarks comparison algorithm can be further improved.

VIII. Conclusion

We designed the first dynamic birthmark extraction and comparison methodology which exploits the run-time heap. It is also the first birthmark methodology for JavaScript. We implemented our design using the Google Chromium browser and developed a birthmark system prototype. Evaluation of our system shows that it is robust against code obfuscation attack. During the evaluation, it gave no false positive or false positive. It is also capable to detect partial code theft of large-scale programs.

Acknowledgment

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] E. Data, “Javascript dominates emea development,” http://www.evansdata.com/press/viewRelease.php?pressID=127, January 2008.

[2] C. Collberg and C. Thomborson, “Software watermarking: Models and dynamic embeddings,” in *Proceedings of Symposium on Principles of Programming Languages, POPL’99*, 1999, pp. 311–324.

[3] K.-i. M. K. I. Akito Monden, Hajimu Iida and K. Torii, “Watermarking java programs,” in *Proceedings of International Symposium on Future Software Technology*, 1999.

[4] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp, “Dynamic path-based software watermarking,” in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, ser. PLDI ’04. New York, NY, USA: ACM, 2004, pp. 107–118.

[5] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” Tech. Rep. 148, Jul. 1997, http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html.

[6] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, “Behavior based software theft detection,” in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS ’09. New York, NY, USA: ACM, 2009, pp. 280–290.

[7] G. Myles and C. Collberg, “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.

[8] D. Schuler, V. Dallmeier, and C. Lindig, “A dynamic birthmark for java,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE ’07. New York, NY, USA: ACM, 2007, pp. 274–283.

[9] M. N. A. M. Haruaki Tamada, K. Okamoto and K. ichi Matsumoto, “Design and evaluation of dynamic software birthmarks based on api calls,” Nara Institute of Science and Technology, Tech. Rep., 2007.

[10] H. Tamada, M. Nakamura, and A. Monden, “Design and evaluation of birthmarks for detecting theft of java programs,” in *In Proc. IASTED International Conference on Software Engineering*, 2004, pp. 569–575.

[11] M. N. Haruaki Tamada, K. Okamoto and A. Monden, “Dynamic software birthmarks to detect the theft of windows applications,” in *In Proc. International Symposium on Future Software Technology*, 2004.

[12] G. Myles and C. Collberg, “K-gram based software birthmarks,” in *Proceedings of the 2005 ACM symposium on Applied computing*, ser. SAC ’05. New York, NY, USA: ACM, 2005, pp. 314–318.

[13] M. N. A. M. H. Tamada, K. Okamoto and K. ichi Matsumoto, “Detecting the theft of programs using birthmarks,” Graduate School of Information Science, Nara Institute of Science and Technology, Tech. Rep., 2003.

[14] “Google chromium project,” http://code.google.com/chromium/.

[15] “Google v8 javascript engine,” http://code.google.com/p/v8/.

[16] “jquery,” http://jquery.com/.

[17] “Jasob 3,” http://www.jasob.com/.

[18] M. Sosonkin, G. Naumovich, and N. Memon, “Obfuscation of design intent in object-oriented applications,” in *Proceedings of the 3rd ACM workshop on Digital rights management*, ser. DRM ’03. New York, NY, USA: ACM, 2003, pp. 142–153.