International Journal of Advance Foundation and Research in Computer (IJAFRC)
Volume 1, Issue 2, Feb 2014. ISSN 2348 - 4853

Software Birthmark for Theft Detection of JavaScript Programs: A Survey

Swati J. Patel*, Tareek M. Pattewar
Department of Computer Engineering*, Department of Information Technology,
SES's R. C. Patel Institute of Technology, Shirpur, MS, India
swatipatel108@gmail.com , tareekpattewar@gmail.com

A B S T R A C T

JavaScript is a widely used language; along with this the source code of JavaScript program is readily available, as provided by most browsers. And hence we must agree that the plagiarism of JavaScript programs is a serious threat. Software Programs are always under the threat of being stolen and so that techniques such as Software Watermarking and Code Obfuscation are used in order to prove the ownership of the program as well as to make the code difficult to understand by the humans. But these techniques cannot avoid programs being copied and the watermark can be defaced. Hence Software Birthmark is used in order to detect the theft of JavaScript programs. Our aim is to survey a relatively new technique Software Birthmark. A birthmark is a characteristic possessed by a program that uniquely identifies that particular program. Birthmark of the software is based on Heap Graph. It is generated when the program is in execution. Software's behavioural structure modelled in the heap graph as how the objects are linked together to provide the desired functionality. To detect whether a website is using the program's code or its component, birthmark of the program is searched in the heap graph of the suspected website.

Index Terms: Heap Graph, Software Birthmark, Software Safeguard, Theft Detection.

I. INTRODUCTION

According to Ninth Annual BSA Global Software 2011 Piracy Study, 57% computer users admit that they use pirated software. The global software piracy rate hovered up to 42% in the year 2011[1].

A. *Software Theft*

Software theft, also referred as software piracy is an unlicensed copy as well as use of computer programs [2]. Mostly piracy is done by private individuals who copy programs from the workplace to their computers at home. Since data is very easy to copy, and the use of illegal software is very hard to detect, it is difficult to stop software piracy [3].

B. *JavaScript Programs and their Theft*

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. Hence it is obvious that JavaScript is the most popular programming language. 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 developers. Software safeguard is an important topic for computer scientists. Several techniques have been introduced for preventing software theft, out of them most widely 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.

1) Software Watermark: 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 determined attacker. It requires the owner of the program to take extra action prior to release the software. Hence most JavaScript developers use code obfuscation before releasing their software.

2) Code Obfuscation: Code obfuscation is the practice of making code unintelligible and difficult to understand. In code obfuscation the code is transformed, which changes the physical appearance of the code, without altering the black-box specifications of the program. Therefore code obfuscation is also known as the semantic-preserving procedure to transform the code in such a way that the structures of the program changes 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.

C. Software Birthmark

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 two programs. A birthmark could be used to identify software theft even after destroying 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 suspected program is searched against the birthmark.

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

1) *Static Birthmark*: These are extracted from the syntactic structure of a program [9].

Definition 1: (Static Birthmarks) [10]

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

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

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

2) *q* is copy of *p* => *f(p)* = *f(q)*

Dynamic Birthmark: These are extracted from the dynamic behaviour of a program at run-time [8]. It is an abstraction of run-time behaviour of the program.

Definition 2: (Dynamic Birthmarks) [11]

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

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

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

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

As semantic preserving transformations like code obfuscation modifies only the syntactic structure of a program but not its runtime behaviour, dynamic birthmarks are more robust against them.

D. Heap Graph based Birthmark

Birthmarks are found to be based on run-time heap. A birthmark is formed by extracting objects from the heap snapshot and building a heap graph out of the heap snapshot.

1) Heap: A heap is a tree-like data structure that satisfies the heap property: If A is parent of B then *key(A)* is ordered with respect to *key(B)* with same ordering applied across the tree. Either the keys of parent node are always greater than or equal to that of child node and the highest key is the root node which is known as max heap, or the keys of parent node are always less than or equal to those of child node and lowest key is the root node which is known as min heap [12].

2) Heap Graph: A heap graph is a simple directed graph in which nodes represent the objects and edges represents the references between them. Heap Graph is a directed graph representation of "points-to" relation between JavaScript objects in the JavaScript heap. The Heap Graph of a program is a 2-tuple graph, *HG = (N, E)* where, *N* represents a set of nodes and *E* represents the reference between objects. All objects and references never represent the behaviour of the system. For this reason, the heap graph is filtered to concentrate on the objects and references that purely depict the behavior of the software. Graphs are printed with depth first search of object in JavaScript heap following reference between them. Three attributes are there in each node, namely, [11]

a. Node Name - Name of the object.

b. Node Type - Each edge is marked by its type.

c. Node ID - Unique ID assigned to the object.

3) Heap Graph based Birthmark: Consider *p* and *q* is two programs or program components. Let *I* be an input to *p* and *q*. Let *HG(p),HG(q)* be heap graphs of the program runs with input *I* for *p, q* respectively. A subgraph of the graph *HG(p)* is *HG* birthmark of *p*, *HGB(p)* if both of the following criteria are satisfied:

1) program or program component is in a copy relation with *p* which implies that *HGB(p)* is Subgraph monomorphic to *HG(q).*

2) program or program component is not in a copy relation with *p* which implies that *HGB(p)* is not Subgraph monomorphic to *HG(q).*

To detect whether a website is using the plaintiff program, first its birthmark is searched, which is the filtered heap graph, in the heap graph i.e. birthmark of the suspected website.

Subgraph monomorphism algorithm is used to do the searching [11].

E. Subgraph Monomorphism

A graph is made up of vertices and lines called nodes and edges. It may be undirected and may contain loop. It is a representation of a set of objects. Some pairs of the objects are connected by links.

1) Graph Isomorphism: A graph isomorphism from graph *G = (N, E)* to a graph *H = (N', E')* is a bijective function

f: N □ *N'* such that (u, v) □ E, (f(u), f(v)) □ E'

2) Graph Monomorphism: A graph monomorphism from a graph G = (N, E) to a graph H = (N', E') is a bijective function *f: N → N'* such that *(u, v) ∈ E <=> (f(u), f(v)) ∈ E'.*

3) Subgraph Monomorphism: A subgraph monomorphism from a graph G = (N, E) to a graph H= (N', E') is a bijective function *f: N → N'* such that *f* is a graph monomorphism from *G* to a subgraph *S* such that *S* is subset of H.

The only difference between graph isomorphism and graph monomorphism is that for graph monomorphism, the mapping needs not to be surjective. That means a pattern graph is mapped to a subgraph in the base graph even if there exist some edges in that subgraph that do not appear in the pattern graph. Graph monomorphism is used instead of graph isomorphism to avoid false negatives when there are some edges in the base graph (the heap graph of the suspected program) that do not appear in the pattern graph (the heap graph of the plaintiff program). This technique is reference injection attack, can be easily exploited by the software thief in an attempt to escape from being detected. Since a graph monomorphism can be found even if there are such references (edges) in the heap graph of the suspected program, reference injection attack will not hinder the detection of software theft [11].

γ-monomorphism: If there exists a subgraph S is subset of H such that S is subgraph monomorphic to H, and

$| S | \geq \gamma | G |, \gamma \in (0, 1]$.

II. RELATED WORK

The first dynamic birthmark was proposed by Myles et al. To identify the program, they explored the complete control flow trace of a program execution. They proved that their technique can resist to any kind of attacks by code obfuscation. There is a drawback that their work is sensitive to various loop transformations. Besides, the whole program path traces are large and hence it is not feasible to scale this technique further [13].

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 extracted by looking at the execution order and the frequency distribution of API calls. These extracted dynamic birthmarks could differentiate personally developed same purpose applications and could resist to different compilers. This promising result motivated the researches to work on dynamic birthmarks based on API calls [14].

Schuler et al. proposed a dynamic birthmark for Java that perceives how a program uses objects provided by the Java Standard API. The short sequences of method calls received by distinct objects from Java Platform Standard API were observed. Then the call traces were decomposed into a set of short call sequences received by API objects. The proposed dynamic birthmark system could accurately identify programs that were similar to each other and distinguish separate programs. In addition, they showed that all birthmarks of obfuscated programs were identical to that of the original program [15]. API birthmark was more scalable and more resilient than the Whole Program Path Birthmark by Myles et al. [13].

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. 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. Moreover, the evaluation of the proposed birthmark system is of much larger scale where 200 websites compared with 20 JavaScript programs in their work and the results were promising [12].

Later, they proposed another heap based birthmark system. 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. On the contrary, the current birthmark system uses graph monomorphism for birthmark detection which makes this system robust against such attack [16].

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. Each vertex is a system call and edges represent data and control dependences between system calls. 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 [17].

III. THE THREAT MODEL

This section focuses on library theft for large scale programs [11].

1) Bob is the owner of the program P also he written his own library L for his program P.

2) Alice wants to develop a program Q which has similar functionalities as P. Alice obtains a copy of P and reverse engineer it to get into the source code.

3) Alice extracts library L and uses it in her program Q. In order to escape from theft, she obfuscates the source code before compilation.

4) Later Bob knew that program Q developed by Alice works same as P. He is curious to know whether his Library L is used to develop Q or not. So he decides to reverse engineer the code. But while reverse engineering he failed because it was impossible for him to reverse engineer since the code of Q was obfuscated.

5) Software Birthmark can help Bob:

- He runs program P and gets birthmark with respect to Library L.
- Then he executes program Q and gets the birthmark of the entire program.
- Heap graph based birthmark with respect to library L, HGB(L), and heap graph of program Q, HG(Q).

Then he finds out whether, HGB(L) is monomorphic to HG(Q) or not, to identify code theft of Library L.

IV. METHODOLOGY

Figure 1 shows the overview of birthmark system. It outlines the processes that the plaintiff program and the suspected program undergo.

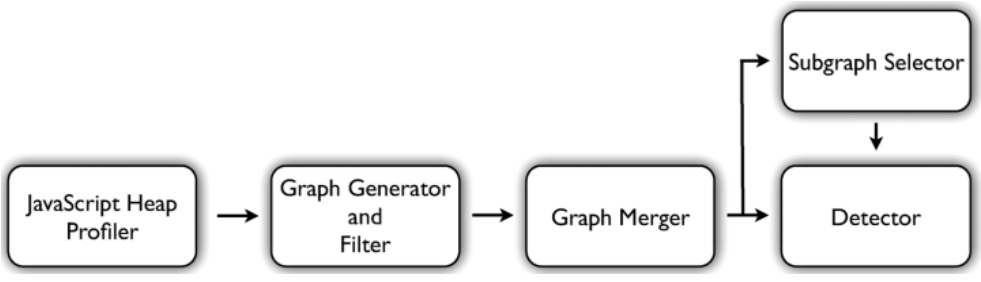

Figure 1. **System Overview**

The **JavaScript heap profiler** runs a JavaScript program and takes multiple heap snapshots in the course of its execution.

The **graph generator and filter** traverses the objects in the heap snapshots and builds heap graphs out of them. It also filters out objects.

The **graph merger** merges the filtered heap graphs together to form one single graph.

The **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.

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

A. JavaScript Heap Profiler

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.

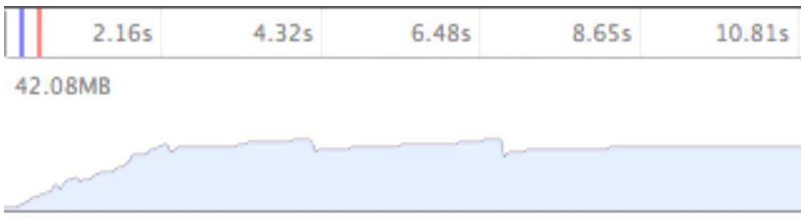

Figure 2. Heap profile of Gmail initialization phase

Figure 2 shows a heap profile of the initialization phase (first 10 sec) of Gmail. The number of objects is increasing in the early stage. Later on, there are some drops and it eventually it becomes stable after some time. To make full use of the behavior exhibited by the objects in the heap, every object is captured that appears in the heap. In order to achieve this, the objects that disappear from the heap due to garbage collection must be ignored. Therefore, the JavaScript heap profiler takes multiple dumps of the heap and merges them together later on. After kicking off the JavaScript program, the browser keeps dumping the JavaScript heap in every 2 seconds. Since taking a snapshot will actually trigger a garbage collection, the heap of the browser is made larger to delay garbage collection and dump the heap more frequently hoping that every object is captured before it becomes garbage.

B. Graph Generator and Filter

Since Chromium browser is used to dump out the JavaScript heap in prototype system, the JavaScript engine that powers the Chromium browser is V8 JavaScript Engine.

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. Although this will increase the size of the data structure, a tree structure enables us to limit the number of objects to be considered for comparison [12]. 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 six categories:

- INTERNAL
- ARRAY
- STRING
- OBJECT
- CODE
- CLOSURE

Objects that belong to INTERNAL, ARRAY, STRING, and CODE categories are not included in heap graphs. The reasons behind this design decision are as follows: INTERNAL objects are virtual objects and they are not accessible from the program code.

For ARRAY objects, they represent an array of elements objects. However, arrays are actually represented by an object of the type OBJECT with name Array and the references from the array are coming out from that object. Therefore, ARRAY objects are not included.

For STRING and CODE objects, there is no reference coming out from them. Therefore, they are not included as well. To sum up only OBJECT and CLOSURE objects are used in heap graph. They are JavaScript objects and function closures respectively. References between objects in the V8 JavaScript heap are divided into 4 categories:

- CONTEXT VARIABLE
- ELEMENT
- PROPERTY
- INTERNAL

References that belong to CONTEXT VARIABLE and INTERNAL categories are not included in the heap graph. The reasons behind this design decision are as follows:

CONTEXT VARIABLE is a variable in a function context. It can be accessed by its name from inside a function closure. Therefore, it cannot be accessed by objects outside that function and it is automatically created by V8 for housekeeping purpose. INTERNAL references are properties added by the JavaScript virtual machine. They are not accessible from JavaScript code. Therefore, only ELEMENT and PROPERTY references are included in the heap graph. ELEMENT references are regular properties with numeric indices, accessed via [ ] (brackets) notation. PROPERTY references are regular properties with names, accessed via the '.' (dot) operator, or via '[ ]' (brackets) notation. JavaScript engine creates some objects that exist not just for one program.

For example, the HTML Document object can be found in the heap graphs of all the JavaScript programs. Therefore, it is needed to filter such objects out as they compromise the uniqueness of the heap graph. Basically, the filtered objects include objects created to represent the DOM tree and function closure objects for JavaScript built in functions. The output of the graph generator and filter is a set of filtered heap graphs captured at different points of time.

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 Graph Generator and Filter also annotates each node in the heap graph with its object ID. Therefore, it is easy to identify whether or not two nodes in two heap graphs refer to the same object. The graph merger takes multiple heap graphs as input and outputs a superimposition of them (one single graph) that includes all the nodes and edges appearing in the input heap graphs. Algorithm 1 is graph merger algorithm.

Algorithm 1 Calculate superimposition of a set of labelled connected graphs G

Require: For all connected graphs $g_i = (N_i, E_i) \in G$, $\exists$ labelling function $f_i : n \to l$ where $n \in N_i$ and l is a positive integer.
Ensure: M = (N, E) is connected and is a $f : n \to l$ where $n \in N$ and l is a positive integer

$N \leftarrow N_1$

$E \leftarrow E_1$

$f \leftarrow f_1$

for all $g_i \in G$ where $i \in [2,|G|]$ do

if $\exists\ n \in N_i, n' \in N$ where $f_i(n)=f(n')$ then

$N \leftarrow N \cup N_i$

$E \leftarrow N \cup E_i$

Combine mapping f and f_i

end if

end for

D. Subgraph Selector

After going through the above steps, the resulting heap graph is one that contains custom objects only and can be used to identify the JavaScript program. However, it is impossible to use the entire graph as the birthmark of the program since the graph is too large for the subgraph monomorphism tool such as VFLib. In fact, the subgraph monomorphism problem itself is known to be NP-complete. The graph, which can comprise hundreds of nodes, is too large for the algorithm and may lead to very long execution time. To explain the method used to select the subgraph to be used as the birthmark, the study about the structure of the heap graph is a must. A heap graph starts with a 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 represents the different DOM windows residing on the web page. A Window object in turn points to the various objects in its DOM window. Figure 3 shows the structure of a heap graph. The objects under the Window nodes are compared with respect to their sizes in terms of the number of nodes and number of edges reachable from the nodes of them. The largest object and the subgraph reachable from it are selected as the birthmark since that captures the most information of the heap. 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.

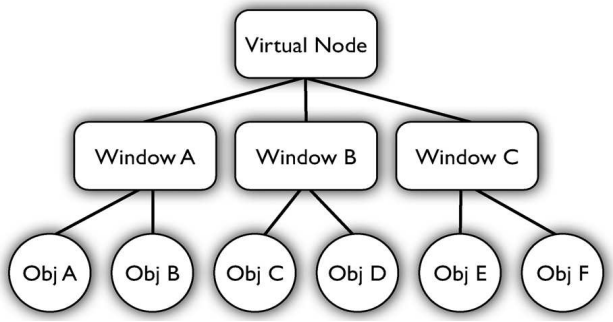

Figure 3. Structure of a heap graph

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

V. CONCLUSIONS

Software Birthmark system generates Heap Graph of the system which is treated as the Birthmark to find similarities between two similarly functioning applications and distinguish distinct applications. This system is reliable and scalable also it can resist to reference injection attack due to the use of subgraph monomorphism while searching the heap graph of plaintiff program in the heap graph of suspected program. We observed that the proposed system is very efficient but also needs some improvements.

VI. ACKNOWLEDGEMENT

We are thankful to all the personalities who helped us throughout this survey.

VII. References

[1] Ninth annual BSA global software 2011 piracy study, 2012, [accessed on April 14, 2013]. [Online]. Available: http://globalstudy.bsa.org/2011/

[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] 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, pp. 569-575, 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] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[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] 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.

AUTHORS PROFILE

Swati J. Patel received her Bachelors degree in Computer Engineering from North Maharashtra University, Jalgaon, MS, India, currently pursuing her Masters degree in Computer Engineering from the same university. Her area of interest include information security, Software Engineering and Soft Computing.

Tareek M. Pattewar is Assistant Professor, Department of Information Technology, R C Patel Institute of Technology at Shirpur, NMU University. His research focuses on Embedded System, Computer Architecture, Data Mining and Real Time Operating System. Tareek M. Pattewar obtained his B.E. in Computer Engineering from RTM Nagpur University, Nagpur, M.E. in Embedded System & Computing from RTM Nagpur University, Nagpur and currently he is working in R C Patel Institute of Technology at Shirpur NMU University.

