<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>Heap graph based software theft detection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Chan, PCK; Hui, CK; Yiu, SM</td>
</tr>
<tr>
<td><strong>Citation</strong></td>
<td>IEEE Transactions on Information Forensics and Security, 2013, v. 8 n. 1, p.101-110</td>
</tr>
<tr>
<td><strong>Issued Date</strong></td>
<td>2013</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://hdl.handle.net/10722/190316">http://hdl.handle.net/10722/190316</a></td>
</tr>
<tr>
<td><strong>Rights</strong></td>
<td>Creative Commons: Attribution 3.0 Hong Kong License</td>
</tr>
</tbody>
</table>
Heap Graph Based Software Theft Detection

Patrick P. F. Chan, Student Member, IEEE, Lucas C. K. Hui, Senior Member, IEEE, and S. M. Yiu, Member, IEEE

Abstract—As JavaScript is becoming more and more popular, JavaScript programs are valuable assets to many companies. However, the source code of JavaScript programs can be easily obtained and plagiarized. Java is a popular programming language. According to Wang et al. [6], a birthmark is a unique characteristic a program possesses that can be used to identify the program. To detect software theft, the birthmark of the program under protection (the plaintiff program) is first extracted. The suspected program is then searched against the birthmark. If the birthmark is found, it is highly likely that the suspected program (or part of it) is a copy of the plaintiff program. There are two categories of software birthmarks, static birthmarks and dynamic birthmarks. Static birthmarks are extracted from the syntactic structure of a program [10], [12], [13]. Dynamic birthmarks are extracted from the dynamic behavior of a program at run-time [6]–[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. Previously, dynamic birthmarks make use of the complete control flow trace, the API call trace, or the system call trace obtained during the execution of a program [6]–[9], [11]. Birthmarks based on the control flow trace may still be vulnerable to obfuscation attack such as loop transformation. The ones based on the API (system) call trace may suffer the problem of not having enough API (system) calls to make the birthmark unique. Recently, birthmarks based on the run-time heap have been proposed [14], [15]. However, the evaluation of these birthmarks are based on a small number of tiny programs. Moreover, the birthmark comparison algorithm proposed in [14] does not scale up well and limits the size of the birthmark. The graph isomorphism used in [15] makes the birthmark vulnerable to what we call reference injection attack. This paper proposes a redesigned heap graph based birthmark for JavaScript to make it a scalable and robust solution for detecting software theft. The proposed birthmark is formed by extracting objects from the heap and building a heap graph out of

I. INTRODUCTION

Due to the prevalence of Web 2.0 and the fact that HTML5 and JavaScript will become the first class platform for the development of Windows 8 apps, it is not surprising that JavaScript has become and will continue to be the world’s most popular programming language. According to a survey from Evans Data in 2008, over 60% of developers use JavaScript and that usage has outstripped all 3GL and scripting language use, including Java [1]. However, the source code of JavaScript programs can be readily obtained as it is an interpreted language and major browsers provide very handy methods to get the source code of the web pages. As a result, it is very hard to protect the intellectual property right of JavaScript developers.

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, 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 [6]–[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. To detect software theft, the birthmark of the program under protection (the plaintiff program) is first extracted. The suspected program is then searched against the birthmark. If the birthmark is found, it is highly likely that the suspected program (or part of it) is a copy of the plaintiff program. There are two categories of software birthmarks, static birthmarks and dynamic birthmarks. Static birthmarks are extracted from the syntactic structure of a program [10], [12], [13]. Dynamic birthmarks are extracted from the dynamic behavior of a program at run-time [6]–[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.

Previously, dynamic birthmarks make use of the complete control flow trace, the API call trace, or the system call trace obtained during the execution of a program [6]–[9], [11]. Birthmarks based on the control flow trace may still be vulnerable to obfuscation attack such as loop transformation. The ones based on the API (system) call trace may suffer the problem of not having enough API (system) calls to make the birthmark unique. Recently, birthmarks based on the run-time heap have been proposed [14], [15]. However, the evaluation of these birthmarks are based on a small number of tiny programs. Moreover, the birthmark comparison algorithm proposed in [14] does not scale up well and limits the size of the birthmark. The graph isomorphism used in [15] makes the birthmark vulnerable to what we call reference injection attack.

This paper proposes a redesigned heap graph based birthmark for JavaScript to make it a scalable and robust solution for detecting software theft. 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 built-in 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.

To detect whether a website is using the plaintiff program, we search for the birthmark of it, which is the filtered heap graph, in the heap graph of the suspected website. We make use of subgraph monomorphism algorithm to do the searching. Subgraph monomorphism is different from subgraph isomorphism that 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. The advantage of this is that even if the thief, in an attempt to escape from detection, deliberately adds some garbage references between the objects, the birthmark of the plaintiff program can still be identified.

We tested our solution using two commodity JavaScript frameworks, Prototype [16] and MooTools [17], and successfully detected uses of them in 200 websites without any false positives or false negatives. We also applied semantics-preserving obfuscation on the two frameworks and compared the birthmark of the original version and the birthmark of the obfuscated version. We found that there is no difference between them. This shows that our birthmark is robust against obfuscation attack which is the commonly considered attack in the literature.

The primary contributions of this paper are:

- **Algorithm.** We redesign the heap graph birthmark for JavaScript to improve its scalability and its robustness against reference injection attack. We propose a pruning mechanism on the nodes and edges based on its nature to filter out those that are not representing the unique characteristics of the program concerned. For the detection step, we make use of graph monomorphism to avoid false negatives resulting from additional references injected into the suspected program.

- **Evaluation.** We implement a prototype of the proposed approach and use the tool to scan 200 large-scale websites from the wild. We demonstrate that our prototype system successfully detected the uses of two JavaScript frameworks in those websites with zero false positive rate and zero false negative rate. We remark this is the first large scale evaluation of a birthmark algorithm. This shows that our birthmark algorithm is ready for practical use.

- **A Scalable Implementation.** We fully automate every step of the prototype system to make it scalable. It took less than an hour to finish checking all the 200 websites in our testing set. Moreover, we provide an analysis of the situations in which the birthmark detection step will take substantially longer.

The rest of this paper is organized as follows: Section II provides some background information of software birthmark; Section III formulates the threat model in which our system is designed; Section IV provides the details of the design of our system; Section V reports the evaluation results; Section VI discusses the limitations and future work. Section VII surveys related work and compares it with our work; Section VIII concludes the paper.

### II. Preliminaries

This section first provides the definition of dynamic birthmarks to ease further discussion. We borrow part of the definition from Tamada et al. [9]. It is the first formal definition appearing in the literature and has been restated in subsequent papers related to dynamic software birthmark. After that, the definitions of subgraph monomorphism and \( \gamma \)-monomorphism are given. Finally, the formal definition of an HG birthmark is introduced.

#### A. Software Birthmarks

A software birthmark is a group of unique characteristics extracted from a program that can uniquely identify the program. There are two categories of software birthmarks: static birthmarks and dynamic birthmarks. We focus on dynamic birthmarks in this research.

1) **Dynamic Birthmarks:** A dynamic birthmark is one that is extracted when the program is executing. In other words, it is an abstraction of the run-time behavior of the program. Therefore, semantics-preserving transformations of the code like obfuscation cannot defeat dynamic birthmarks. It is a generally accepted fact that dynamic birthmarks are more robust compared with static birthmarks.

**Definition 1:** (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 \( I \).

Next, we will discuss about subgraph monomorphism which is the technique we use to perform detection of a birthmark in a program.

#### B. Subgraph Monomorphism

**Definition 2:** (Graph Monomorphism) A graph monomorphism from a graph \( G = (N, E) \) to a graph \( G' = (N', E') \) is a bijective function \( f : N \rightarrow N' \) such that \( (u, v) \in E \Rightarrow (f(u), f(v)) \in E' \).

For comparison purpose, we also give the definition of graph isomorphism as follows.
Fig. 1. Example heap graph obtained from the JavaScript heap of www.gmail.com. (Limited to 100 nodes.)

Fig. 2. Example heap graph obtained from the JavaScript heap of www.apple.com. (Limited to 100 nodes.)

**Definition 3:** (Graph Isomorphism) A graph isomorphism from a graph \( G = (N, E) \) to a graph \( G' = (N', E') \) is a bijective function \( f: N \rightarrow N' \) such that \( (u, v) \in E \iff (f(u), f(v)) \in E' \).

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. We use graph monomorphism instead of graph isomorphism because we want to avoid false negatives when there exist 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, what we call 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 exist such references (edges) in the heap graph of the suspected program, reference injection attack will not hinder the detection of software theft.

**Definition 4:** (Subgraph monomorphism) A subgraph monomorphism from a graph \( G = (N, E) \) to a graph \( G' = (N', E') \) is a bijective function \( f: N \rightarrow N' \) such that \( f \) is a graph monomorphism from \( G \) to a subgraph \( S \subseteq G' \).

**Definition 5:** (\( \gamma \)-monomorphism) A graph \( G \) is \( \gamma \)-monomorphic to \( G' \) if there exists a subgraph \( S \subseteq G \) such that \( S \) is subgraph monomorphic to \( G' \), and \( S \geq \gamma |G|, \gamma \in (0, 1] \).

**C. Heap Graph Based Birthmark**

Before we give the definition of our heap graph based birthmark (HGB), we need to define what a heap graph (HG) is. An HG is a directed graph representation of the “points-to” relation between JavaScript objects in the JavaScript heap. The formal definition of a heap graph is given as follows.

**Definition 6:** (HG: Heap Graph) The heap graph of a program run is a 2-tuple graph \( HG = (N, E) \), where
- \( N \) is a set of nodes, and a node \( n \in N \) corresponds to an object in the JavaScript heap.
- \( E \in N \times N \) is the set of edges, and each edge \( n_1 \rightarrow n_2 \in E \) corresponds to a reference between the object represented by node \( n_1 \) and the object represented by node \( n_2 \). There is no duplicated edge between two nodes.

Figs. 1 and 2 shows two examples of heap graphs extracted from www.gmail.com and www.apple.com respectively. The graphs are printed out in the course of depth first search traversals of the objects in the JavaScript heap following the references between them. We limit the number of nodes to 100 to make the size of the graphs suitable for presentation. There are three attributes in each node: node name, node type, and node ID. Node name is the name of the object and node ID. Node name is the name of the object and node ID is a unique ID assigned to that object. Each edge is also marked by its type. We will explain further about the node type and edge type in Section IV when we talk about the design of our
birthmark. Basically, a heap graph starts with a root node with its child nodes representing the DOM windows of the web pages embedded in it.

Based on the $\gamma$-monomorphism definition and the definition of a heap graph we just provided, the HG birthmark can be defined.

**Definition 7:** (HGB: Heap Graph based Birthmark) Let $p$, $q$ be two programs or program components. Let $I$ be an input to $p$ and $q$, and $HG_p$, $HG_q$ 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:

- program or program component $q$ is in a copy relation with $p \Rightarrow HGB_p$ is subgraph monomorphic to $HG_q$.
- program or program component $q$ is not in a copy relation with $p \Rightarrow HGB_p$ is not subgraph monomorphic to $HG_q$.

Although our experiment showed that HGB is robust to state-of-the-art obfuscation techniques, we relax subgraph monomorphism to $\gamma$-monomorphism in our detection for robustness to unobserved and unexpected attacks. Hence, a program $p$ is regarded as a copy of another program $q$ if the HGB of $p$ is $\gamma$-monomorphic to HGB of $q$. We set $\gamma = 0.9$ in experiments since we believe that overhauling 10% of an HGB is almost equivalent to changing the overall architecture of a program component.

III. THREAT MODEL

We are focusing on library theft for large scale programs. Our threat model is similar to the one stated in [15].

In the attack scenario, Bob is the owner of a program $P$. The core part of it is a library $L$, which is also developed by him. Alice wants to write another program $Q$ which has similar functionalities as $P$ does. Obtaining a copy of program $P$, Alice reverse engineers it and gets the source code. She extracts the library $L$ from program $P$ and uses it in her own program $Q$. In order to escape from code theft detection, she obfuscates the source code before compilation.

Later, Bob discovers that the program $Q$ developed by Alice functions similarly to his own program $P$. He wants to find out if program $Q$ uses the library $L$ developed by him. Since the source code of program $Q$ is obfuscated and illegible, he cannot justify it by reverse engineering program $Q$ and looking at the source code. He then gets help from our dynamic birthmark system. He executes program $P$ and gets the birthmark with respect to library $L$. After that, he executes program $Q$ and gets the birthmark of the whole program $Q$. Obtaining the birthmark with respect to library $L$, $HGB_L$, and the heap graph of the whole program $Q$, $HG_Q$, he then finds out whether $HGB_L$ is $\gamma$-subgraph monomorphic to $HG_Q$ or not to identify code theft of library $L$.

IV. SYSTEM DESIGN

Fig. 3 shows the overview of our birthmark system. It outlines the processes that the plaintiff program and the suspected program undergo. 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 according to our design decisions that we will discuss later in this section. 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. In this section, we will discuss each of these processes and state the rationales behind our design.

A. Javascript Heap Profiler

Being an interpreted language, JavaScript allows for the creation of objects at anytime. 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. Fig. 4 shows a heap profile of the initialization phase (first 10 sec) of GMail. We observe that the number of objects is increasing in the early stage. Later on, there are some drops and it eventually stabilizes after a while.

To make full use of the behavior exhibited by the objects in the heap, we try to capture every object that appears in the heap. In order to achieve this, we need to avoid missing those objects that disappear from the heap due to garbage collection. Therefore, the JavaScript heap profiler in our design 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, we make the heap of the browser 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 we make use of the Chromium browser [18] to dump out the JavaScript heap in our prototype system, the following discussion is in the context of V8 JavaScript Engine [19], which is the JavaScript engine that powers the Chromium browser.

For each snapshot taken using the Chromium browser, we perform a death first search traversal of it and print out the heap graph with nodes and edges that pass a filter. We describe such a filter in details as follows.

Objects in the V8 JavaScript heap are divided into six categories: INTERNAL, ARRAY, STRING, OBJECT, CODE, and CLOSURE. We do not include in our heap graph objects that belong to INTERNAL, ARRAY, STRING, and CODE categories. The reasons behind this design decision are as follows: INTERNAL objects are virtual objects for housekeeping purpose and are not accessible from the program code. For Array objects, they represent an array of elements objects. However, our observation shows that 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, we only include in our heap graph OBJECT and CLOSURE objects. 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, and INTERNAL. We do not include in our heap graph references that belong to CONTEXT_VARIABLE and INTERNAL categories. The reasons behind this design decision are as follows: CONTEXT_VARIABLE is a variable in a function context, accessible by its name from inside a function closure. Therefore, it is not accessible 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, we only include in our heap graph ELEMENT and PROPERTY references. ELEMENT references are regular properties with numeric indices, accessed via [] (brackets) notation and PROPERTY references are regular properties with names, accessed via the . (dot) operator, or via [] (brackets) notation.

There are some objects created by the JavaScript engine that exist not just for one program. For example, the HTMLDocument object can be found in the heap graphs of all the JavaScript programs we studied. Therefore, we need to filter such objects out as they dilute 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. In the next step, we are going to merge these graphs together to form a single graph which embraces all the information gathered in the heap graphs.

C. Graph Merger

There is a unique ID assigned to every object in the JavaScript heap by the V8 JavaScript engine. Moreover, the ID of an object does not change across multiple dumps and therefore, 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, we can tell 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. The algorithm of graph merger is shown in Algorithm 1.

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

Require: $\forall$ connected graphs $g_i = (N_i, E_i) \in G$, $\exists$ labelling function $f_i : n \rightarrow i$ where $n \in N_i$ and $i$ is a positive integer

Ensure: $M = (N, E)$ is connected and is a superimposition of graphs in $G$ with labelling function $f : n \rightarrow i$ where $n \in N$ and $i$ is a positive integer

$N \leftarrow N_i$
$E \leftarrow E_i$
$f \leftarrow f_i$

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

$N \leftarrow N \cup N_i$
$E \leftarrow E \cup E_i$

Combine mapping $f$ and $f_i$

end if

end for

In a nutshell, it superimposes all the graphs one by one by taking the union set of the nodes and edges of the two graphs being merged. In order to make the resulting superimposition graph also connected, we need to ensure that there is at least one object in common (with the same object ID) in two graphs before superimposing them.

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, we cannot use the entire graph as the birthmark of the program since the graph is too large for the subgraph monomorphism tool, VFLib, we use [20]. 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 we use to select the subgraph to be used as the birthmark, we need to first study about the structure of the heap graph. 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. Fig. 5 shows the structure of a heap graph.

We look at all the objects under the Window nodes and compare their sizes in terms of the number of nodes and number of edges reachable from the nodes of them. We select the largest object and the subgraph reachable from it to be the birthmark as that captures the most information of the heap.

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. Evaluation

In order to evaluate our method, we built a prototype of our birthmark system and used it to detect the uses of two JavaScript frameworks in 200 websites. In this section, we first discuss about the implementation of our prototype system. Following that, we present the evaluation results.

A. Implementation

Our prototype system consists of two modules. The first module is a modified chromium browser coupled with a browser extension. It plays the role of the JavaScript Heap Profiler and the Graph Generator and Filter. The second module consists of some C++ programs that play the role of the Graph Merger, the Subgraph Selector, and the Detector. The entire prototype runs under Mac OS X 10.7.3 and comprises 750 LOC.

1) First Module: We make use of the HeapProfiler API provided by the V8 JavaScript engine to take snapshots of the JavaScript heap. The snapshots provided are in the form of heap graphs accessible via the virtual nodes. We modified the Chromium browser such that it calls HeapProfiler to take a snapshot of the heap every time a dumpHeap function is called from the JavaScript program. HeapProfiler takes 5 steps to take a snapshot. The first step is to trigger two garbage collections in order to ensure that all objects, including weakly reachable ones, are reachable from the root. The second step is to iterate heap contents to count entries and references. The third step is to fill references between the entries. The forth step is to set the dominators of the entries. The dominator of an object A is an object that exists in every simple path from the root to the object A. The final step is to calculate the retained size of each entry (the total size of the entries reachable from that entry, including itself). The snapshot taken is then printed out to a text file.

To print out a heap graph, we traverse it starting from the virtual node. Since we need to filter out nodes and edges that do not represent the unique behavior of the program as mentioned in Section IV, we selectively skip the nodes and edges that need to be filtered out during the traversal.

Since the heap snapshot taking is triggered by the dumpHeap function call from the JavaScript program, we need to insert it into the JavaScript program being investigated transparently. To do this, we develop a simple browser extension that inserts a code snippet into web pages the modified browser visits. The extension is built using a technique called Content Scripts [21]. Content scripts are JavaScript files that run in the context of web pages. They can read details of the web pages the browser visits, or make changes to them. We set the script to be run at document_idle such that the browser chooses a time to inject scripts between the time when the DOM is complete and immediately after the window.onload event fires. This allows us to ensure that the snapshot taking does not start before the document is fully loaded and the objects are created. The code snippet inserted calls the dumpHeap function in every 2 seconds.

There is one set of text files, storing the multiple snapshots taken, generated for each object under the Window nodes. Each text file represents the subgraph that contains all the nodes and references accessible from the object.

2) Second Module: The second module is a bunch of C++ programs and shellscripts that do the various tasks. The graph merger is based on the graph superimposition algorithm mentioned in Section IV. It takes text files representing the same object and merges them together. The output of it is a single text file that contains the graph of the superimposition of the object subgraphs.

The subgraph selector sorts the text files storing the object subgraphs by size and selects the largest file to become the birthmark. It is because the object subgraph stored in the largest file has the largest total number of nodes and edges.

For the detector, it takes the largest object graph from the plaintiff program and tries to search for it in the object graphs of the suspected program. We make use of the VFLib library [20] which provides the subgraph monomorphism algorithm to do the searching. Once there is a match found, the detector reports the object subgraph of the suspected program in which the birthmark is found.

B. Evaluation Results

1) Effectiveness: We chose two subject JavaScript libraries for experiments: Prototype [16] and MooTools [17]. They are JavaScript frameworks that aim to ease development of dynamic web applications. Their birthmarks are shown in Figs. 6
and 7 respectively. Our testing program set consists of 200 websites divided into two groups. The first group consists of 81 websites and the second group consists of 119 websites. The two groups of websites were reported having used the Prototype framework and the MooTools framework respectively.

To extract the birthmarks of the two JavaScript frameworks, we created simple web pages with the JavaScript files of different versions of the two frameworks embedded in them. For prototype, we focused on versions starting from 1.6.0. For MooTools, we focused on versions starting from 1.3.0. We browsed the web pages using our modified Chromium browser coupled with the extension that takes heap snapshots. We then obtained the birthmarks of the various testing libraries. Next, we ran our prototype system to try to find the birthmarks of Prototype library in the first group of testing websites and find the birthmarks of MooTools library in the second group of testing websites. We discovered that some websites do not have the birthmarks in them. For the first group of websites, our system detected the birthmarks of Prototype library in 21 of them only. For the second group of websites, our system detected the birthmarks of MooTools library in 25 of them only.

To justify the results, we manually checked the existence of the two libraries in their corresponding groups of websites. We checked the existence of the Prototype library by searching for the phrase “var Prototype” in the code base of the websites as we found that this phrase appears in the source code of all versions of it. Similarly, we checked the existence of the MooTools library by searching for the phrase “MooTools={version” in the code base of the websites. Our manual checking results are the same as the results given by the detector. We conclude that the accuracy of our birthmark system is 100% for the two libraries. Table I summarizes our evaluation results of this section.

2) Robustness: Defacing the birthmark so as to escape from detection is the major goal of software birthmarks attackers. The commonly used technique is code obfuscation which is a semantics-preserving transformation of the source code. In order to test the robustness of our birthmark system, we applied code obfuscation to the source code of the different versions of the two JavaScript frameworks we used and saw the difference between the birthmarks of the original frameworks and the birthmarks of the obfuscated frameworks. The obfuscator we used is Jasob 3 [22]. It is a state-of-the-art obfuscation tool for JavaScript used by hundreds of companies and individuals to protect and optimize their web content. It replaces descriptive variable and function names with meaningless names. We found that there is no difference between the two sets of birthmarks. Therefore, our birthmark system is robust against this kind of code obfuscation attack. In fact, our system is robust against any obfuscations that do not change the referencing structure of heap objects. In Section VI, we will discuss about potential attacks to our system.

3) Performance: Since our birthmark system will be used to search for the birthmark of the protected software component among numerous software, execution performance is a critical requirement. In light of this, all the processes of the system are automated. We used our system to analyze 200 websites with most of them being large-scale (e.g., apple.com, cnet.com, nasa.gov, etc). And the analysis of all of them only took less than an hour. That is to say, each website only took less than 18 seconds on average. And the whole analysis was done in a batch.
VI. DISCUSSION

A. Limitations

Since our birthmark system relays on the referencing structure between objects, the program under protection is required to have a significant referencing structure that can be used to uniquely identify it. The target of our system is large scale programs which usually exercise the object oriented programming paradigm. We observed, however, that some large scale programs are simply procedural and our system is not suitable for them. One of such examples is the jQuery library which does not have a prominent referencing structure. We observed that it mainly consists of procedures encapsulated in the jQuery object. However, we do see the trend that programs are becoming more and more object-oriented. For example, earlier versions of MooTools are not as object-oriented as newer versions of MooTools. We observed that the size of the heap graph of MooTools is strictly increasing along with the updates. The same is also true for the Prototype library.

Another limitation of our system is that graph monomorphism we use at the detection step is NP-complete. Although it was designed to handle large graphs (up to 2–3000 nodes), the pruning step fails if the graphs exhibit strong symmetries, for example if they are almost completely connected or they are mostly in tree structures as studied in [23]. In that case, the running time becomes very long and it is not practical to be used. We did not come across this situation throughout our study though.

B. Potential Attacks

There are 3 kinds of potential attacks to our birthmark system. We are going to discuss them one by one in this section.

1) Object/Reference Injection: The attacker can inject objects or references that are of no use into the program. There will be addition nodes and edges on the resulting heap graph. However, the attacker can only control the base graph but not the pattern graph. The base graph is the heap graph extracted from the suspected program and the pattern graph is the heap graph extracted from the plaintiff program. Since subgraph monomorphism is used in our detection system, the extra nodes and edges on the base graph will not affect the matching of the pattern graph. Therefore, our birthmark system is immune to this kind of attack.

2) Object/Reference Removal: The attacker can purposely remove classes or references from the program. The resulting heap graph will have some nodes and edges missing. This will make our detection system fail to detect the birthmark in the suspected program. However, we argue that it is hard to remove a class or reference while preserving the behavior of the program. We can reasonably assume that such an attack is not feasible in practice. If it is feasible, the design of the original program is problematic as it contains some redundant classes or redundant references between classes. As a last resort, we can relax subgraph monomorphism to γ-monomorphism to capture incomplete instances of the birthmark.

3) Class Refactoring: Class refactoring refers to the process of restructuring classes. It is commonly used to improve the design of object oriented programs. The two underlying techniques that can bring an impact to the structure of the heap graph, class coalescing and class splitting, have been studied in a paper on obfuscation of design intent in object-oriented programs [24]. Class coalescing refers to the merging of two classes into one class. On the contrary, class splitting refers to splitting a class into two classes. However, as discussed in the paper, there are preconditions that need to be satisfied before class coalescing and class splitting can be done without changing the behavior of the obfuscated program.

For class coalescing, the two classes to be coalesced cannot not extend different library classes. In their experiments, they showed that only about 30% of classes on average are eligible for coalescing. That result is for Java programs only. Since JavaScript is a prototype-oriented programming language, the challenge is even bigger as each object has a prototype chain which is typically longer than the inheritance chain and combining two classes means combining the prototype chain of the two classes. Otherwise, some references may not be resolved if some classes in the original prototype chain is missing. And even if the two classes can be coalesced, every declaration of a field, local variable, or method argument with type of the two classes has to be replaced with the coalesced class. Finally, their experiments showed that class coalescing results in a significant run-time penalty.

For class splitting, dependencies among methods and fields of the original class need to be considered. In general, conservative dependency analysis is required when splitting a class. Furthermore, the authors believe that 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, instead of a single class, there should have been several different classes. When splitting a class C into two classes C1 and C2, C2 is made a subclass of C1. This has minimum effects on the resulting heap graph as the node that represents the class C in the original heap graph is now replaced by a node representing C2. And since C2 is a subclass of C1, all the references connecting class C is now connecting class C2 instead. Therefore, the structure of the heap graph is essentially the same.

C. Future Improvements

In this section, we describe future improvements that we hope to see.

1) Improved Graph Selector: Currently, we choose the largest object subgraph to become the birthmark of a program. We do not know if there exists a better way to do it. One preliminary idea is to use frequent subgraph mining to get the frequent subgraph that appears in all the heap graphs we extracted from the program. This can make the birthmark more representative of the program. However, the running time of frequent subgraph mining on large graphs is slow and there should be some performance tuning in order for it to be practical.

2) Faster Detector: Due to the theoretical running time limit of graph monomorphism algorithm we used in our detector, we need to limit the size of the heap graphs in order to control the running time of the detector. This essentially has two effects on our system. On one hand, this makes the birthmark capture less
information from what is available. It is because there are objects and references that are useful for identifying the program but missed in our heap graphs, they represent information loss. On the other hand, this potentially makes our system miss some alarms. It is because the sizes of the heap graphs of the suspected programs are limited as well, it is possible that it makes the birthmark incompletely captured in the heap graphs and ultimately leads to false negatives. Currently, we limit the size of the pattern graph (the object subgraph of the plaintiff program) to 20 nodes and that of the base graph (the object subgraph of the suspected program) to 40 nodes. We find this gives us a good balance between running time (less than few minutes for each test) and detection accuracy (100% in our evaluation). Although the current result is promising, we do hope to see ways to make the detector even more efficient and allow us to feed the system with larger heap graphs.

VII. 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.

We proposed the first dynamic birthmark based on the run-time heap [14]. It is also the first dynamic birthmark for JavaScript programs. The proposed birthmark is in the form of an object reference tree. We used a tree comparison algorithm to compare two birthmarks and gave a similarity score between two birthmarks. However, due to efficiency problem of the tree comparison algorithm, we needed to limit the depth of the tree to 3 in order to make the running time of the algorithm practical. On the other hand, our new birthmark is an object graph and we use graph monomorphism to search for the birthmark in the heap graph of the suspected program. Although we do limit the size of the heap graphs in our new 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 (which is considered the root node of the whole heap graph in our previous work). Even though we limit the size of the heap graph, the current birthmark captures far more information than the previous system. Moreover, the evaluation of this birthmark system is of much larger scale (200 websites compared with 20 JavaScript programs in their work) and the results are promising.

Later, we proposed another heap based birthmark system [15]. This time, the birthmark system is for Java programs. We also used a different algorithm, graph isomorphism, for birthmark detection. However, as pointed out earlier in this paper, graph isomorphism is too restrictive and makes the birthmark system vulnerable to reference injection attack. On the contrary, our new birthmark system uses graph monomorphism for birthmark detection and this makes our system robust against such attack. Besides, the scale of the evaluation of that system is much smaller (25 programs compared with 200 websites in this work). We remark that the scale of evaluation of our new birthmark system is the largest so far.

VIII. CONCLUSION

In this paper, we proposed a robust heap graph based software birthmark system for JavaScript programs. We made our birthmark robust against reference injection attacks and streamlined every process of the system to make it scalable. We evaluated our birthmark system using 200 large-scale websites and the experiment results are promising with 100% accuracy. We discussed the limitations of the system as well as the various potential attacks. As JavaScript is getting more and more popular nowadays and the source code of JavaScript programs can be readily obtained, our birthmark system brings to the industry a practical solution to protect their intellectual property right. Although software birthmark is a relatively new and less focused research area for the time being, we hope that our work can stir up more discussions in the community and that will eventually lead to even better work in the future.
The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper.

REFERENCES