A Lightweight Approach to Detect Memory Leaks in JavaScript

Ju Qian
College of Computer Science and Technology,
Nanjing University of Aeronautics and Astronautics
Nanjing 210016, China
jqian@nuaa.edu.cn

Long Wang, Xiaoyu Zhou
School of Computer Science and Engineering,
Southeast University
Nanjing 211189, China
zhouxy@seu.edu.cn

Abstract—**Although with garbage collection support, many JavaScript programs still suffer from memory leaks. These leaks can affect application performance and even cause crashes, especially for single page websites. The existing work on JavaScript memory leak mainly focuses on the static detection of leaks toward certain leak patterns. The application scope of such approaches are limited. Previous techniques used for detecting memory leaks in Java-like languages might be extended to JavaScript. However, how to apply these techniques in JavaScript is still a problem. In this paper, we firstly present many common leak detection heuristics used in garbage-collected languages and investigate their effectiveness on JavaScript. According to the investigation results, we then propose a lightweight multi-snapshots based dynamic leak detection method for JavaScript. The initial experimental results show that the proposed approach is effective.**

Keywords- memory leak; JavaScript; dynamic analysis

I. Introduction

JavaScript is a popular language mainly used to develop dynamic web pages. Although with automatic memory management, JavaScript still suffers from the memory leak problem. If a useless object in JavaScript is unintentionally referenced by some long-term living references, the object cannot be reclaimed by a garbage collector. It will hence be leaked and occupy unnecessary memory.

For traditional web applications, the memory leaks in JavaScript may not cause serious problems, since the web pages of an application are usually frequently switched and the leaked memory can be reclaimed when a browser discards old pages. However, modern web applications are often single-paged. In those applications, the web pages of an application are no longer frequently switched. A web page may be alive for hours or even days using Ajax technologies to update contents without completely refreshing the page. Many rich client web applications, such as Google Gmail and Microsoft Office 365, follow this style. In such cases, the leaked memory can be largely accumulated, which can degrade the performance of web applications or even cause the applications to crash due to out of memory error.

To address the issue, early research detects circular references to catch memory leaks in the old browsers. In some old browsers like IE6, the DOM objects are garbage-collected with reference counting, while the other JavaScript objects are garbage-collected using some kinds of mark and sweep algorithms. Useless objects in DOM-related reference cycles may not be effectively reclaimed. To help find the leaked objects, the SIEve/Drip tool [1] tracks all DOM nodes to find the ones involved in reference cycles. It then lets the users to determine which DOM nodes are leaked. Microsoft developed another similar tool named JavaScript Leaks Detector [2]. It reports circular reference caused leaks by simulating IE6 and IE7's garbage collection (GC) mechanisms. In recent browsers, the GC algorithms have already been improved. The unnecessary memory caused by DOM-related reference circles now can be automatically freed by a browser. Therefore, such memory leaks do not need additional efforts to detect and fix any more.

For the memory leaks caused by reachable useless objects, some static techniques have been proposed to detect certain leak patterns in web frameworks [3, 4]. A typical tool in this category is Leak Finder [3]. It detects memory leaks in Google’s Closure library. The tool finds goog.Disposable objects in a heap snapshot and inspects these objects to check whether certain easily to leak objects are freed. Its memory detection capability is limited to the Closure library. Pienaar and Hundt proposed another more advanced tool JSWhiz [4]. The tool summarizes several leak patterns in the Closure library. It can statically detects many kinds of memory leaks based on the abstract syntax tree and type system of JavaScript source code. Even though, the application scope of such work is limited, since the used leak patterns are often bound to certain types of applications. Jensen et al. [5] and Rudafshani and Ward [6] also proposed approaches to detect memory leaks in JavaScript. Their approaches need to track the allocations and accesses of objects and hence can be very costly for large programs. More general and lightweight JavaScript memory detection tools are still in demand.

One insight for developing such general JavaScript memory leak detection techniques is to extend the existing techniques for Java-like languages [7] to JavaScript. Java suffers from memory leak problems similar to JavaScript’s. To detect memory leaks in Java, one kind of approach locates leaks according to the growth trends of heap structures, e.g., [8-11]. Another kind of technique detects memory leaks according to the structural information within the heap, such as the ownership relation [12], the data structure similarity and reoccurrence [13], etc. Besides, some other approaches also find leaks using the object lifetime information, such as the age of objects [7] and the staleness of objects (how long the object have not be used) [14].

Although effective for Java, it is still unclear whether these techniques are still suitable for JavaScript, since JavaScript has

DOI reference number: 10.18293/SEKE2018-151

many individual characteristics that are different from Java, such as the dynamic type system and the prototype-based inheritance. To this end, this paper firstly studies the effectiveness of different memory leak detection heuristics which are borrowed from Java on JavaScript and then presents a dynamic approach to detect JavaScript memory leaks on the basis of JavaScript's own characteristics and the existing leak detection heuristics. The approach collects heap snapshots for web applications, and uses a lightweight statistical method combining many heuristics to recommend suspicious leaking objects. The initial experimental results show that the proposed approach can effectively detect memory leaks and hence can be helpful for the users.

II. MEMORY LEAK DETECTION HEURISTICS

The general memory leaks caused by unbroken reference in garbage-collected languages are hard to be precisely detected by static analyses. For such leaks, dynamic analyses are often preferred. Even though there is a rich literature on the dynamic analysis techniques for memory leaks, most of the existing techniques are based on a few core detection heuristics.

A. *Leak detection heuristics*

Table 1 shows many leak detection heuristics (or their core metrics) used by previous research. Before introducing their details, some basic concepts are firstly explained.

GC Roots: Root objects or references where a garbage collector starts its analysis. Typical GC roots include stack variables, static fields, class objects, etc.

Ownership: If in an object reference graph, every path from GC roots to a node *n* going through a node *d*, we say *d* owns *n*.

Leak Root: Root objects or references which directly or indirectly reference the whole leaked data structure. A leak root can represent a collection of leaked objects.

Fringe: Fringe [8] refers to the boundaries between the old objects and the newly created objects in the object reference graph of a heap snapshot.

In the introduction of leak detection heuristics, we suppose the heap change history forms a sequence of heap snapshots $\mathcal{H} = (H_1, H_2, ..., H_n)$, where H_i is the *i*-th snapshot in the heap change history.

DCR: For an object type T, let $D(T)$ and $C(T)$ be the numbers of T's instances destructed and constructed in a heap snapshot, respectively. $DCR(T) = D(T) / C(T)$ is said to be the destruction/construction rate of T. If $DCR(T)$ is continuously low in a heap snapshot sequence, T can be considered as a probably leaked object type [9].

TIV: Given two heap snapshots H_i and H_{i+1}, assume the numbers of objects of a type T in H_i and H_{i+1} is V_i and V_{i+1}, respectively. Then, from H_i to H_{i+1}, the increase volume of type T is $TIV(T) = (V_{i+1} - V_i)$. The types with high TIV values are more likely to be the ones with instances leaked.

TPFI: Assume the numbers of references between two object types T_1 and T_2 are R_i and R_{i+1} in two sequential heap snapshots H_i and H_{i+1}, respectively. Then, the type point-from relationship increment between T_1 and T_2 from snapshot H_i to snapshot H_{i+1} is $TPFI = (R_{i+1} - R_i)$. The larger TPFI, the more likely that the involved types are with objects leaked [10].

LN: Leaf nodes in an object reference graph are not likely to be the root causes of memory leaks. Therefore, it is better to not directly report them as the leak diagnosis results.

TABLE 1. Leak detection heuristics (or their core metrics)

abbreviation	heuristics or their metric
DCR	Destruction/Construction Rate
TIV	Type Increase Volume
TPFI	Type Point-from Increase
LN	Leaf Nodes
IMN	Immutable Nodes
INN	Internal Nodes
NON	Non-owner Nodes
NAI	No Age Intersection
NF	No Fringe
OSR	On-stack Reachability
FOC	Fringe Ownership Count
NOC	New Ownership Count
SOC	Similar Object Count
LS	Life Span

IMN: Immutable objects with sizes not changed in different heap snapshots are unlikely to be leak roots.

INN: Internal objects maintained by the language runtime (e.g., JavaScript VM) are unlikely to be leaked.

NON: An object not owning any other objects is said to be a non-owner object. The non-owner objects are often close to GC roots, and their referenced objects are shared by other references. These non-owner objects are less likely to be leak roots.

NAI: The age of an object describes how long ago it has been created. In a heap snapshot, the objects created in the current heap snapshot and not holding references to the objects created in old snapshots, or the objects created in old snapshots and not holding references to the objects created in the current snapshot are said to be no age intersection objects. NAI objects are unlikely to be leak roots. If a new object does not reference old objects, it is likely to be a temporal object. If no new object is attached to an old data structure, then the old data structure is likely to be stable.

NF: No fringe objects refers to the objects owning no objects on the fringe. Memory leaked objects are often connected with fringe objects. No fringe objects are unlikely to be leaked, while the objects referencing to both fringe and no fringe objects have more possibility to be the leak causes [8].

OSR: The objects directly accessible from stack variables are more likely to be temporary objects instead of leak roots.

FOC: If an object owns more objects on the fringe, it is more likely to be a leak root object.

NOC: Objects owning a lot of newly created objects are more likely to be leak roots.

SOC: In a heap snapshot, if an object has more similar objects, then there will be high possibility that such type of objects are leaked.

LS: If an object firstly appears in a snapshot H_i and finally disappears since snapshot H_j, we say its life span is $LS = (j - i)$. The longer life span, the larger possibility that the object is leaked.

B. *Effectiveness of leak detection heuristics on JavaScript*

We conducted experiments on some JavaScript programs to analyze the effectiveness of the above heuristics. The results are discussed as following.

(1) The type memory growth based heuristics (TG)

The DCR and TIV heuristics detect memory leaks according to the memory growth trends of each type. These heuristics are effective for JavaScript. However, the leak sources detected by them are mostly basic types like Object, Array, HTIMDivElement, etc.

This is because JavaScript uses a prototype-based inheritance mechanism, and the types dynamically extended from a root type like *Object* by attaching or removing properties at runtime are difficult to be distinguished from the root type. A basic type can have too many sub-types dynamically extending from it. Only knowing that objects of some basic types are leaked is not very helpful for leak diagnosis. Heuristics DCR and TIV must be combined with techniques that can classify objects with finer granularity to effectively help locating leak sources.

(2) The reference growth based heuristic (RG)

The TPFI heuristic detects memory leaks according to the growth of reference relationships between types. It can rank the leak causing reference relationships in high position. However, most of the reported results are relationships between basic object types. Such relationships are too rough for leak diagnosis. The reasons are similar to that of the DCR/TIV heuristics, which are also due to the very flexible nature of the JavaScript type system.

(3) The heap structure based heuristics (HS)

Heuristics LN, IMN, INN, NON, NAI, NF, OSR, FOC, and NOC mainly detect leaks by analyzing the structural attributes of objects on a single or two sequentially obtained heap snapshots. Our experiments show that applying heuristics LN, IMN, INN, NON, NAI, NF, and OSR can filter out a large number of objects that are unlikely to be leak roots and heuristics FOC and NOC are effective for suspicious leak root ranking. However, determining ownership relationships can be costly for large heap snapshots.

(4) The data structure similarity based heuristic (DSS)

Heuristic SOC can be used to partition objects in to similarity groups and then analyze the properties of these groups to identify leak sources. In Java, we can at least use the type information to distinguish similar objects. However, in the prototype-based JavaScript language, many objects are created by dynamically extending the root *Object* type and the actual type information is hard to determine. Therefore, there need some other techniques to help determine the similarity between objects. Besides, we found the SOC heuristic should better be used together with TG heuristics to get more valuable results. The similarity groups can be viewed as finer-grained resolution of object types or data structures. Such grouping also can benefit many different methods which depend on type or data structure information.

(5) The object lifetime based heuristic (OL)

Heuristic LS detects memory leaks according to the object lifetime information. With this heuristic, we may detect a large number of individual leaked objects instead of a few object types or data structures. Because in JavaScript, objects are not with their types distinguished with fine granularity, such results do not provide clear clue for further leak diagnosis. Besides, when roughly tracking the lifetime of objects according to their occurrences in heap snapshots, without monitoring the uses (reads or writes) of objects, heuristic LS can easily lead to false alarms.

III. A Lightweight Leak Detection Method for Javascript

According to the above findings, we believe an effective and lightweight way for JavaScript memory leak detection is to combine the TG, DSS, and some HS heuristics for leak object identification. We may follow the DSS heuristics to get a better resolution of types or data structures. The TG heuristics can be used to rank suspicious objects, and we can use the HS heuristics to filter out unlikely leaked objects. Under such idea, this section presents a lightweight multi-snapshots based leak detection method (the MS method). The method takes the characteristics of JavaScript language into account and can be more helpful for JavaScript memory leak diagnosis.

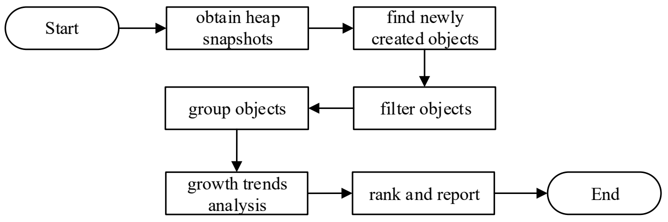

Figure 1. The workflow of the MS method

A. The detection method

The proposed method is a dynamic approach which detects memory leaks based on two or more heap snapshots obtained from a memory leaking program. The workflow of the method is shown in Figure 1. It takes 6 main steps.

In the first step, we obtain a sequence of heap snapshots from the execution of the target program. Then, the heap snapshots will be parsed and we traverse the object reference graphs embedded in the heap snapshots and compare every two adjacent snapshots to locate the newly created objects in a snapshot. The objects occurred in the current snapshot but do not occur in a previous one are considered as the newly created objects.

After that, the objects unlikely to cause memory leaks will be filtered out according to heuristics IMN, INN, NAI, and OSR.

Next, we classify the newly created objects into partitions according to the connections between objects. For two newly created objects, if they are connected, then the two objects will be put into the same partition. Each partition can be regarded as an individual data structure. We then follow the idea of heuristic SOC to categorize these data structures into similarity groups. Instead of doing object level similarity analysis, we find common parent objects of the objects in the above partitions on the object reference graph. A common parent object can be viewed as the representing node of one or more partitions. The data structures referenced by a common parent object are usually similar. Finding common parent objects works as a kind of data structure level similarity grouping. This can result in fewer groups compared to object level similarity grouping. Each common parent object can be regarded as a candidate leak root. By inspecting these leak roots, it will be easier to diagnose the root causes of memory leaks.

For the calculated candidate leak roots, we will further analyze their memory growth in the heap snapshots. Unlike doing memory growth analysis at type-level, the previous grouping can make the analysis results easier for further inspection. The total size of the objects in each object group represented by a common parent object is used as an approximation of the occupied memory of a leak root. If the occupied memory continues to grow, then the candidate is considered suspicious; otherwise, it will be excluded from the leak detection results.

After all the heap snapshots have been analyzed, we rank the candidate leak roots according to their totally occupied memory and the number of objects in the categorized object groups. The top ranked candidates will be reported as leak detection results.

To better show which objects are leaked, we use a chain of object property names that used to reach a leak root in the object reference graph as the identification of the leak root.

TABLE 2. The experimental subjects

Name	Library	Source
JQueryWeb	JQuery	http://javascript.info/tutorial/memory-leaks
ExtWeb	ExtJS	http://www.sencha.com/forum/showthread.php?263439-ExtJS-Memory-Leak
YuiWeb	YUI	http://yuilibrary.com/trac-archive /tickets/2530415.html
DojoWeb	Dojo	http://www.ibm.com/developerworks/cn/web /wa-sieve/
MeteorWeb	Meteor	https://github.com/meteor/meter/issues/1157
BackWeb	Backbone	http://plnkr.co/edit/xfJWIF?p=info
AngularWeb	Angular	https://github.com/angular/angular.js/issues/4864

B. *Experimental analysis*

We conducted an initial experimental study on 7 JavaScript programs using popular libraries JQuery, ExtJS, etc. to validate the effectiveness of the proposed approach. The subjects are listed in TABLE 2. In the experiment, we use the Chrome browser to run the subject programs for a while and then use Chrome DevTools to obtain snapshots at different time points. The snapshots are exported to local files for further analysis. Each obtained heap snapshot can be viewed as an object reference graph. These snapshots are parsed and analyzed with Java language.

TABLE 3 shows the effects of different analysis steps in our lightweight leak detection method. The table only lists the experimental data when analyzing two adjacent snapshots. In the table, column *#new* shows the number of identified newly created objects. Column *#filter* shows the number of remaining objects after doing object filtering. Column #partition lists the number of categorized newly created object connection partitions, and column *#parent* shows the number of calculated common parent nodes for the object partitions. From the table, we can see that object filtering can greatly reduce the number of the objects to need be analyzed, and the object partitioning and common parent grouping can effective categorize objects into suspicious object groups.

The final results of our lightweight leak detection method are shown in the rightmost two columns of TABLE 3. The results indicate that our proposed method can detect memory leaks in high precision. The reported numbers of suspicious leak roots are small, which can reduce the effort of further leak diagnosis and fixing.

TABLE 4 shows the analysis time consumed by different leak detection methods on the same snapshots. TG, RG, HS, and OL stand for the detection methods with different groups of heuristics applied, respectively. From these data, we can see that our light weight multi-snapshots based method consumes very little time. This is because we only analyze the newly created objects on each snapshot, which greatly reduces the number of objects need to be processed. We use a lightweight method to calculate metrics for suspicious leak root ranking, which also reduces the analysis cost.

TABLE 3. Effects of different analysis steps

Subject	#new	#filter	#partition	#parent	#detected leak roots	#actual leak roots
JQueryWeb	95101	28431	14221	2	1	1
ExtWeb	19334	906	534	34	9	3
YuiWeb	12151	7050	937	14	2	1
DojoWeb	1805	290	84	2	1	1
MeteorWeb	165896	77609	4681	197	5	2
BackWeb	9030	2510	421	36	2	1
AngularWeb	3800	856	599	21	5	3

TABLE 4. Analysis time of different methods (ms)

Subject	TG	RG	HS	OL	MS
JQueryWeb	384	550	16325	465	198
ExtWeb	960	4466	5497	4168	179
YuiWeb	341	557	4436	560	547
DojoWeb	211	1175	1069	1113	135
MeteorWeb	638	1242	91332	1146	395
BackWeb	550	923	4540	827	553
AngularWeb	326	737	1116	691	190

IV. Conclusion

In this paper, we firstly investigate the effectiveness of many common leak detection heuristics on JavaScript programs. Based on the investigation results, we propose a lightweight multi-snapshots based leak detection method for JavaScript. The method combines many effective heuristics and takes the characteristics of JavaScript language into consideration. Our experimental results show that it is both effective and efficient. In the future, we plan to further improve the method and conduct more experiments on more subjects to further validate its effectiveness.

Acknowledgment

This work is supported by the China Defense Industrial Technology Development Program (Grant No. JCKY2016206B001 and JCKY2014206C002), the Science and Technology Planning Project of Jiangsu Province (BY2016003-02), and the National Natural Science Foundation of China (Grant No. 61472175).

References

[1] IE/Sieve. http://home.online.nl/jsrosman/

[2] JavaScript Memory Leak Detector. http://blogs.msdn.com/b/gpde/archive /2009/08/03/javascript-memory-leak-detector-v2.aspx

[3] Leak Finder for Javascript. http://code.google.com/p/leak-finder-for-javascript/

[4] J. A. Pienaar, R. Hundt. JSWhiz: Static analysis for JavaScript memory leaks. IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2013, pp. 1-11.

[5] S. H. Jensen, M. Sridharan, K. Sen, S. Chandra. Meminsight: Platform-independent memory debugging for JavaScript. In Symposium on the Foundations of Software Engineering, 2015.

[6] M. Rudafshani, P. AS Ward, LeakSpot: detection and diagnosis of memory leaks in JavaScript applications, Software: Practice and Experience, 47(1): 97-123, 2017.

[7] V. Šor, S. N. Srirama. Memory leak detection in Java: Taxonomy and classification of approaches. Journal of Systems and Software, 2014.

[8] N. Mitchell, G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In the European Conference on Object-Oriented Programming, 2003.

[9] K. Chen, J. B. Chen. Aspect-based instrumentation for locating memory leaks in Java programs. In Annual International Computer Software and Applications Conference (COMPSAC), 2007.

[10] M. Jump, K. S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. ACM SIGPLAN Notices. 2007, 42(1): 31-38.

[11] J. Qian, D. Zhou, Prioritizing test cases for memory leaks in Android applications, Journal of Computer Science and Technology, 31(5), 2016.

[12] D. Rayside, L. Mendel. Object ownership profiling: a technique for finding and fixing memory leaks. In International Conference on Automated Software Engineering, 2007, pp. 194-203.

[13] E. K. Maxwell, G. Back, N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In SIGKDD International Conference on Knowledge Discovery and Data Mining, 2010.

[14] H. Yu, X. Shi, and W. Feng. LeakTracer: Tracing leaks along the way. In 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2015 pp. 181-190.