CM30171: Exam

January 20, 2022

1 BLeak: Automatically Debugging Memory Leaks in Web Applications

(b) As the name suggests, the paper tries to address the problem of memory leaks in web applications. A memory leak is where applications use memory without deallocating it after use. In web applications this takes the form of 'unneeded event listeners, repeatedly injecting iframes and CSS files, and failing to call cleanup routines in third- party libraries.' amongst other things. Memory leakage is a performance issue, slowly degrading the performance of a system over time as memory is continually consumed, and eventually spilled over into virtual memory which potentially could cause thrashing and complete system meltdown. They provide a proof-of-concept system called BLeak (which is active and in use[1], although development appears to be inactive) which is able to identify memory leaks by running round trips on website routes and examining excess memory usage, using novel techniques. The motivation for the paper stems from the fact that current memory leak detection tools are ineffective for web applications. 'leaks in web applications are fundamentally different from leaks in traditional C, C++, and Java programs', and this is because whereas here we can used staleness-based approaches of garbage collection[2] where heap objects are tracked and allocated to certain areas of memory based on their 'hotness' or 'staleness' (and in the case of the Hound system, stale objects are virtually compacted, compacting multiple virtual memory pages into one physical page without performing moves in virtual address space), this approach won't work for web apps as leaked objects are frequently interacted with via event listeners. Similarly for 'growth-based techniques'[3], which track the head of a data structure in heap space (also known as a 'leak root') that is 'exhibiting unbounded growth', these are not applicable to web environments as they assume singular owners of memory locations, whereas in web apps there can be many owners of an object. Static-type information methods do not work on Javascript for obvious reasons, and finally, existing browser-based leak detection tools are not developer-friendly, as for example the Chrome Memory tab in developer tools only provides heap snapshots which developers would have to manually compare, that also do not provide any mechanism for identifying the location in the source code from which these leaks originate.

(c) The BLeak system is used to detect, rank by severity and diagnose memory leaks, using a configuration script as input. The configuration script dictates a user flow through a website, and would typically consist of a round trip, whereby a user navigates from some main page away to a different one and then back again. The analysis is done by comparing between the initial visit to the main page and the second visit. BLeak tracks *paths* of objects as opposed to directly tracking object instances, allowing for more accurate tracking between transitions. The example given is `history=history.concat(newItems)`, where history is being overwritten with a new larger array. With direct object tracking, this case would not be recognized, however with path tracking it is. BLeak takes the round-trip defined by the configuration file and runs through it multiple times, taking a heap snapshot at each initial state, and tracing paths from the garbage collection roots that are continually growing through these iterations. A GC root is the marked starting point of a garbage collector, from which it traverses the graph of the webpage, and marks any unvisited nodes to be garbage collected. In the BLeak system, if objects continue to grow through multiple iterations of the round-trip, they are marked as leak roots.

    By using the final snapshot of the final iteration, and the list of leak roots, BLeak ranks memory leaks using their custom *LeakShare* metric, where the highest ranked memory leak corresponds to the leak for which the fix would free the most memory with the least amount of effort on the part of the developer. The LeakShare metric divides credit for a given shared leak object between its leak roots, thus a high score of a given object by the LeakShare metric would be an object that leaks a high amount of memory with a low number of leak roots (and therefore less locations in the source code which the developer has to modify). A low score would be a low amount of memory leaked with a high number of leak roots. Correspondingly, a developer could inspect the LeakShare metric and only eliminate the root which leaks the most memory for a given shared object. Whilst this would not eliminate the leak it would reduce it the most for the amount of effort they wish to put in.

To identify the sources of the leaks within the source code, BLeak uses its MITMProxy (Man-In-The-Middle) that is inserted when the web application is loaded from the script to 'transparently rewrite' the corresponding JavaScript for each webpage. An algorithm Propogate-Growth tracks the growth of nodes through heap snapshots. After the final snapshot, an algorithm 'FindLeakPaths' finds paths (where paths are a single route through the heap graph, which consists of nodes that are objects in the heap, and edges that are references between objects) through the heap to leaking nodes. JavaScript Reflection is used to insert diagnostic hooks at all of the paths reported by FindLeakPaths. The round-trip is ran one more time to collect stack traces, and finally the LeakShare metric is calculated. Both of these reports are collated in a file for the developer.

The evaluation methodology for BLeak is conducted via experiments on production web applications (i.e. creating configuration files for commercial applications such as Airbnb, Google Maps etc. and running the diagnostic tool with these configurations). Evaluation of the system is performed on 6 key metrics:

(1) Precision
(2) Accuracy of Diagnoses: Does BLeak spot the correct location in the source code for a given leak?
(3) Overhead
(4) Impact of Discovered Leaks
(5) Utility of Ranking

(6) Staleness vs. Growth: Comparison of system to a staleness-based leak detector

Clearly these are reasonable assessment metrics. The system is only valuable if it is precise with its memory leak detections, and if it is accurate in its placement of leaks within source code then this is an additional benefit over other detection tools as there are not any satisfactory solutions for this. Additionally overhead has implications for production use; whether teams will choose to forgo the detection process because the tradeoff of time to run BLeak versus the impact on end users (usually users don't care about memory leaks and rarely inspect their System Monitor/Task Manager) is too high. The metrics 'Impact of Discovered Leaks' and 'Utility of Ranking' are both evaluation metrics for the LeakShare metric, which is essential if the system should be successful in helping developers allocate their time efficiently. Finally the last metric is a comparison to existing systems; the system is not worth using if it performs worse than incumbent solutions.

The system was tested on 5 different web applications (arbitrary, as no benchmark for memory leak detection performance exists) by running a single round-trip configuration 8 times on each website, on a Macbook Pro with Core i5 and 16GB RAM. For each application, reported leaks were analysed, fixes were applied where appropriate and the impact of the fixes were measured. Finally, LeakShare was compared against other ranking metrics.

(d) The result of their findings is that the system idenifies leaks with 96.8% precision, where precision is defined by the ratio of true leaks roots to total leak roots found (i.e. disregarding false positives), and identifies the code responsible for the leak in all but one case. The number of false positives can apparently be reduced with an increased number of iterations (only 8 are used for the testing). The runtime for locating, diagnosing and ranking leaks is 7 minutes on average for each application, of which the majority is receiving and parsing heap snapshots. This is a reasonable overhead, and could be reduced even more if Chrome/Firefox developers improved the efficiency of these heap snapshots. As for the impact of the memory fixes suggested by BLeak, between the initial runs of the website and the runs once fixes were implemented, heap growth was reduced by 93%. Finally, LeakShare is compared against the ranking metrics 'retained size' and 'transitive closure size'. Retained size is defined as 'the amount of memory the garbage collector would reclaim if the leak root were removed from the heap graph', and transitive closure size of a leak root is defined as ' the size of all objects reachable from the leak root'. The reasoning for comparing against these two metrics is that retained size is the metric used by most built-in (Chrome,Firefox) heap snapshot viewers, And that related work (Xu et al.[4]) uses transitive closure size to measure the performance of their own growth-based technique. Ranking metrics are compared by fixing memory leaks in the order suggested by the given metric, in quartiles. They find that LeakShare outperforms or matches these two other metrics. Finally, they manually analyzed the leaks using a staleness-based technique, and found that at least 77% of the leaks found by BLeak would not be found using this conventional technique.

(e) Overall, this system appears to be a breakthrough in web applications memory leak detection, and although it isn't perfect, it appears to vastly outperform all existing techniques. As noted in the background section and related work, web applications is a domain of memory leak detection in which there has been little improvement, so this system is a huge success in this regard. There are a few notable limitations however of the study.

    Firstly, issues with the testing methodology. For each web application tested, there is only a single path taken through the webpage. It is possible and likely that there are other leaks throughout other routes of the webpage, and they do not factor this into their analysis. Fortunately, due to the array of web pages tested, the lack of multiple paths on a single web page is made up for by the variety of web pages they test. Secondly, there is no control variable in their testing; there is no value provided for the 'true' number of memory leaks on a webpage, and as such they have nothing to compare their reported number of leaks to. Thus it is impossible to tell whether BLeak finds all the leaks for a given path. Again, this is somewhat remedies by the 'Leak Impact' metric of heap growth reduction of 93%, which suggests that for the most part, BLeak finds the most significant memory leaks.

Along the lines of the testing methodology, there is also a general issue with the design of the system. A developer has to input a certain path through the webpage, and the system will only iterate through this single round-trip. Again this means that we are not solving memory leaks in all views of the webpage but only a certain route. There is an implication here for the developer that he must first know of a problematic route through the webpage to test, as the typical flow through the webpage may not have significant memory leakage compared to other routes. However, we must consider this in context, as the typical flow constitutes the majority of user traffic, and as such, memory leaks solved here will impact and benefit the majority of users, but it is not a perfect solution. It is not known whether the analysis and solution of memory leaks of a single path contributes to the majority of leaks throughout the webpage; this would depend on the design and possible flows of that particular web application. If a web application has a lot of different flows, the solution of a single route may not have much impact on the overall memory leakage.

Another issue is the resolution of the memory leaks. Whilst this may not have been the intention of the author, it would be useful to have BLeak automatically resolve memory leak issues. Having to manually implement fixes is another source of error from the developer, as they may resolve these memory leaks incorrectly (although this appears to be a trivial task if it is the case that we simply need to close unused event listeners, call library cleanup functions etc.). Again it is not clear that this would be desirable to the developer; it is detracting from the autonomy of the developer in that the system may not implement fixes in the way they desire (although it is most likely the case that there is an optimal way to resolve memory leaks, however it is not known how tractable it is for this to be done automatically).

Finally, whilst this may not be an issue of the system itself, looking at the GitHub for this project, it does not appear there has been a huge uptake of this technology by developers, with 377 stars at the time of writing. This is not necessarily indicative of the value of the system, as it may be the case that this has not had much recognition outside of the academic realm due to lack of promotion. However, it does raise questions about the practicality of its use in production, although there is no evidence in the paper to support this, and it does appear to be a simple, easy to use system. Again as previously mentioned, this could be due to the impact on users; unless memory leakage of a webpage are completely unacceptable and causing thrashing/system failure, it is rare that a user would pick up on this, as systems nowadays typically have plenty of RAM. As a result, it is likely as well that developers see memory leak fixes as a waste of their time; this is supported in the paper by a bug fix posted for Google Maps where the developer closed the issue as 'Infeasible' because 'We're not really sure in how many places we're leaking :-('[5](Although it does appear this issue has since been reopened and partially resolved).

References

[1] John Vilk and Emery D. Berger. *Plasma-umass/bleak: Bleak: Automatically debugging memory leaks in web applications.* URL: `https://github.com/plasma-umass/BLeak`.

[2] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. "Efficiently and Precisely Locating Memory Leaks and Bloat". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 397–407. ISBN: 9781605583921. DOI: `10.1145/1542476.1542521`. URL: `https://doi-org.ezproxy1.bath.ac.uk/10.1145/1542476.1542521`.

[3] Nick Mitchell and Gary Sevitsky. "LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications". In: *ECOOP 2003 – Object-Oriented Programming.* Ed. by Luca Cardelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 351–377. ISBN: 978-3-540-45070-2.

[4] Guoqing Xu and Atanas Rountev. "Precise Memory Leak Detection for Java Software Using Container Profiling". In: *ACM Trans. Softw. Eng. Methodol.* 22.3 (July 2013). ISSN: 1049-331X. DOI: `10.1145/2491509.2491511`. URL: `https://doi.org/10.1145/2491509.2491511`.

[5] *Bug: Destroying Google Map Instance Never Frees Memory.* URL: `https://issuetracker.google.com/issues/35821412?pli=1`.