Detection and Root Cause Analysis of Memory-Related Software Aging Defects by Automated Tests

Felix Langner
Institute of Computer Science
Heidelberg University, Germany
felix.langner@uni-heidelberg.de

Artur Andrzejak
Institute of Computer Science
Heidelberg University, Germany
artur@uni-hd.de

***Abstract*—Memory-related software defects manifest after a long incubation time and are usually discovered in a production scenario. As a consequence, this frequently encountered class of so-called software aging problems incur severe follow-up costs, including performance and reliability degradation, need for workarounds (usually controlled restarts) and effort for localizing the causes. While many excellent tools for identifying memory leaks exist, they are inappropriate for *automated* leak detection or isolation as they require developer involvement or slow down execution considerably. In this work we propose a lightweight approach which allows for *automated* leak detection during the standardized unit or integration tests. The core idea is to compare at the byte-code level the memory allocation behavior of related development versions of the same software. We evaluate our approach by injecting memory leaks into the YARN component of the popular HADOOP framework. The results show that the approach can detect and isolate such defects with high precision, even if multiple leaks are injected at once.**

***Index Terms*—Software Aging, Memory Leaks, Automated Testing, Automated Debugging**

I. INTRODUCTION

Latent software defects such as memory leaks become observable only after a prolonged execution time of the process image - a time ranging from hours to weeks. This property is responsible for the costly consequence that such defects are likely to escape the traditional testing process and are discovered for the first time in a production scenario. Another problem is that even if noticed, the root causes of such defects are difficult to isolate. As a consequence, such defects create very high follow-up costs. These include operational costs due to system performance and reliability degradation, programming and configuration effort for workarounds (usually scheduled restarts), and debugging work for localizing the root causes.

Memory leaks are only a special (albeit quite frequent) case of *software aging* defects. The software aging phenomenon is described as a progressive performance or functionality degradation in a software system [1]. Typical causes are unreleased resources (such as heap memory or file descriptors), accumulation of numerical errors, and file system degradation.

A recent study of bug reports in large open-source projects [2] estimates that about 1% of all discovered malfunctions belong into this category. However, due to the factors described above the impact of such defects is among the most detrimental on the overall cost-of-ownership of a software system. Aging issues occur in any type of software that is sufficiently complex, but it is particularly troublesome in long-running applications. Examples include telecommunication systems, web-servers, web-service middleware [3], or cloud computing infrastructure [4].

Until today, the primary technique to combat software aging are controlled restarts known as *software rejuvenation* [5]. The major handicap of this solution is its focus on the elimination of aging *symptoms* after the defective software system has been already deployed. A more effective way of dealing with aging would be to detect, isolate and remove the defects already at the software development stage, preferably during the integration tests. However, the latent nature of these defects make this undertaking very difficult without considerable additional resources in terms of time and hardware. As aging is usually noticeable only after a prolonged execution, currently scenario-tailored stress or performance tests are used to identify this phenomenon. Such tests require a dedicated infrastructure (ideally similar to the deployment environment) and need to run for many hours. Especially the latter requirement can be hardly fulfilled during integration tests, and so testing against software aging remains a very rare exception.

In case of memory-related defects, the tool support is quite good [6]. The primary focus of these tools is to help in isolating the root cause of aging or to detect usage of non-allocated or already deallocated memory (see e.g. Electric Fence). Only few of them attempt to discover the presence of aging in the first place. Another disadvantage of these leak-detection tools is that they can be hardly used for *automated* detection of memory leaks. This is caused by the necessary developer involvement to verify the potential leaks (e.g. in case of LeakBot) or a huge execution slowdown (Valgrind).

This work in progress proposes an approach for automated detection of memory leaks with low runtime and setup overhead. It targets specifically the development scenario (or the software update process) and can be used for leak detection during unit or integration tests. It exploits differences in the behavior of memory allocation of related (usually subsequent) versions of software under development. Specifically, we compare (between older and newer software versions) the amount of allocated heap memory for each individual allocation site in code (e.g. each usage of `new` in Java).

In this way we are able to detect memory leaks with negligible runtime overhead. Moreover, we can also isolate the root causes with high precision. Combined with the very modest effort of setup and code instrumentation, the approach can be realistically applied in traditional testing processes.

This paper is structured as follows. Section II describes the problem context and presents the details of our approach. Section III contains the description of evaluation environment and experiments and shows the results of the evaluation. In Section IV we discuss the related work and state our conclusions in Section V.

II. Description of the Approach

This section describes the key concepts of our approach.

A. The concept of version comparison

Our approach assumes the existence of two versions v_0 ("base"), v_1 ("target") of a software artifact which differ by some changes introduced in the development process. A typical example are versions between one or more code commits in a development of a (non-trivial) project. Due to code changes, the newer version v_1 might have some (additional) memory leak defects. The key idea is to compare the memory allocation behavior of both versions under the *same* workload. If any (significant) differences or anomalies are observed, they are likely to indicate new aging defects.

In our previous work [7] we used this idea to compare the *cumulative* heap memory usage (after a run of unit or integration test) via operating system metrics, specifically Resident Set Size (RSS). However, this approach is less sensitive (new leaks with low memory usage can be "covered up" by other allocation sources) and does not support defect isolation.

In this paper we refine this approach by investigating heap memory *allocation sites*, i.e. code locations which use `new` in Java or `malloc` in C. We compare behavior of each pair of corresponding allocation sites between versions. Increased memory allocation (in the newer version v_1) for a specific pair of allocation sites is likely to indicate a new aging defect.

Of course, version v_1 might contain some newly introduced allocation sites (not matched in v_0). If any such site has high final memory allocation after a test run, we list them as a potential defect source to be examined by the developer.

B. Technical implementation

The preliminary step consists of code instrumentation to record amount of heap memory allocated at each allocation site. To this end we implemented a Java agent named Live Object Monitor (LOM). It is based on the Java-allocation-instrumenter [8] and monitors object allocations and their destruction by the garbage collector. As a negative side-effect, it delays the instantiation and destruction of objects due to the monitoring process.

Given a target software version (i.e. v_1), we execute a unit or integration test (taken from a regular test suite for the particular software) under LOM instrumentation. During execution, our tool records a list of all allocation sites that have been active during execution. In addition, it also creates a list $Heap(v_1)$ of all Java objects (and their memory size) which are alive after the test finished (but the JVM is not yet shut down) - called *residual objects*. Each such object is annotated by an id of its allocation site. To simplify further analysis, we group all residual objects with the same allocation site into an *allocation family*.

The subsequent filtering step compares this data with the analogous data collected previously for v_0 (i.e. $Heap(v_0)$; note that for each development version, collected data can be used twice: first for the target v_1, later for the base v_0). To this end we remove from $Heap(v_1)$ *one* object of instance family f for *each* object of the same instance family found in $Heap(v_0)$. If all objects of a specific instance family f could be removed from $Heap(v_1)$, we mark the corresponding allocation site as "clean". Consequently, the remaining allocation sites have more object allocations in v_1 than in v_0. They are subject to further investigation.

C. Analyzing suspicious allocation sites

If the process described above reports any new suspicious allocation sites, we rank them primarily by the defect type (below) and secondarily by the amount of leaked bytes. The defect types are defined as follows:

a. The allocation site S is new in the sense that it is present in v_1 but not in v_0. Such site is considered as a potential leak and the amount of leaked bytes is defined as the total memory consumption all of the residual objects for S for the execution of v_1.

b. The allocation site S exists in both v_0 and in v_1. In this case the amount of leaked bytes is defined as the total memory of all residual objects for S in v_1 minus the memory of the residual objects for S in v_0 (i.e. extra memory allocated during execution of v_1).

In a development scenario, the resulting ranked list would be transferred to the developers for further investigation. As the description of each potential defect also states the allocation site, the developers can quite easy debug whether any such list item is a false alarm or a leak.

III. Experimental Evaluation

We evaluate our approach via experiments described in Section III-A. The obtained results are summarized and discussed in Section III-B.

A. Experimental setup

We prepared a test environment by introduced artificial and "parametrizable" memory-leaks defects into several development versions of the open source project Apache Hadoop

[9]. These defects are inserted into to YARN Resource Manager component of HADOOP.

1) System under study: All experiments were executed on a virtual machine with four dedicated cores (i7 CPU with 2.4 GHz) and 3GB of main memory running UBUNTU 12.04 64-bit.

We installed four different development versions of APACHE HADOOP retrieved from the source code repository. The selection of these versions is based on the number of changes affecting the YARN component, which provides the resource management functionality for HADOOP.

The latest unstable release 2.0.3-alpha servers us as the initial ($h0$) version for the approach. We use three succeeding development versions, numbered $h1$ to $h3$.

2) Integration test scenario: We select k-means clustering from the MAHOUT machine learning library (snapshot version 0.8) [10] as the map-reduce algorithm to be executed by the different HADOOP versions. As input data we use *synthetic_control.data* provided by the MAHOUT examples. We vary the number of map-reduce jobs by setting a very high convergence goal yet limiting the number of iterations i to $i \in \{1, 5, 10\}$. Since the k-means algorithm creates a new map-reduce job for each iteration, we can thereby control the length of the test.

All runs are executed 10 times to obtain higher confidence in results.

3) Artificial memory leak defects: For programming languages with managed memory like Java, software aging is frequently caused from repeatedly inserting objects into collections (e.g. lists) without proper removal of unused/expired objects. We also follow this schema. Our artificial leaks are configurable in their *strength* (i.e. the amount of memory being leaked with each activation) denoted by parameter s. The used leak objects are byte arrays with a randomized length drawn from a continuous uniform distribution $unif(1, s)$. This randomness makes results more “unpredictable” and therefore more natural. For simplicity the parameter s is one $\{1, 10, 100, 1000\}$.

The artificial defect are identified by numbers from 1 to 5 (where defect 5 is a real memory leak defect discovered in the HADOOP development; its strength is neither configurable nor randomized).

B. Evaluation

Following [11] we use the absolute rank of a potential defect to measure the efficiency of the defect detection. We assume that only ranks 1 to 20 are significant, assuming that a developer would investigate only at most the first 20 entries of a defect report. The distribution of the ranking results is visualized by box plots which use data from all 10 test run repetitions. The top ranked (i.e. most significant) defect has rank 1. We inverted the scaling of the rank axis in order to make it easier to compare the different plots with each other.

Figure 1 shows the ranks of the artificial defects of type *a*., i.e. for allocation sites that occur for the first time. In the base software version v_0 all artificial defects are deactivated.

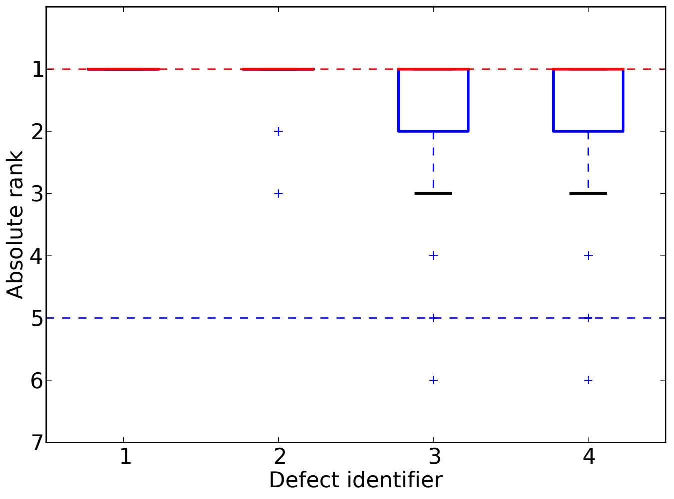

Figure 1. The absolute rank of defects of type *a*

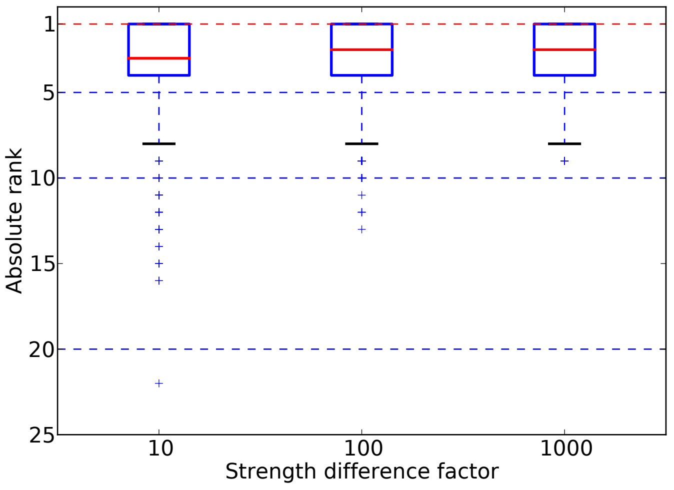

Figure 2. The absolute rank of defects of type *b* for comparison partners having the same defect enabled but with values of the strength parameter (larger strength parameter on the x-axis)

The applied workload was an integration test with number of iterations $i = 1$. We obtain the target software version v_1 by enabling one of the four artificial defects 1 to 4.

Figure 1 indicates that our approach works well for the defects of type *a*. The medians have rank 1 (i.e. in at least half of the considered cases the artificial defect is correctly ranked as first). The outliers are not worse than 6. These results are independent of the strength parameter s and the HADOOP version of the defective comparison partner.

In Figure 2 we plot the impact of the strength parameter s on the rank for the artificial defects of type *b*. The target software version v_1 is compared against the version v_0 with the same detect id enabled. However, they have different leak strength factors (v_1 has larger leak strength factors). All target versions v_1 are derived from all HADOOP versions $h1$ to $h3$.

The quality of ranking increases with the strength factor s (there are less outliers with higher ranks). This can be explained by the fact that the amount of leaked bytes is the

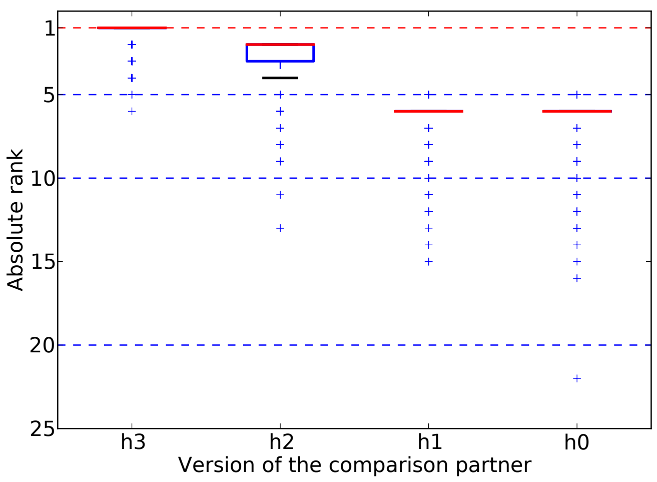

Figure 3. The absolute rank of defects of type b for different amount of code changes between target version v_1 (first left) and base version v_0

secondary ranking criterion. A leak with higher strength (more leaked bytes) is thus ranked higher.

Figure 3 illustrates how the amount of code changes between comparison partners (v_0 and v_1) influence the quality of results. The amount of code changes is proportional to the "distance" between software versions. The defective target version v_1 is based on the newest HADOOP $h3$, while v_0 is based on the HADOOP version shown on the x-axis (the amount of changes increases from left to right). For $h3$, the base version v_0 differs from v_1 only by the defect configuration. As expected, greater version distance translates to lower quality of the ranking. This can be explained by the fact that larger code differences imply stronger deviations of (non-defective) memory allocation patterns. This creates noise and makes harder to rank leaks correctly.

In summary the artificial defects have a reasonable rank (i.e. below rank 20) for all test scenarios in almost all experimental results. Defects of type a. can be detected more easily, as their ranks are in majority of cases between 1 and 5. Type b defects could not be detected as easily and they have ranks ranging between 1 and 10.

The accuracy of our approach depends to a large degree on the amount of code changes between the two comparison partners. The closer the software versions, the more exact the ranking of the artificial defects. Also the strength of a defect is an important factor, as it exposes the leak more clearly.

IV. Related Work

Over the past decades software aging research was mainly focused on software rejuvenation [5]. Thereby controlled restarts are utilized to mitigate the symptoms of software aging. These restarts are either scheduled periodically or adaptively [12]. Both variants require modeling of the software aging that affects the system [13], [14]. The most intensively investigated scheduling paradigm is the adaptive scheduling also referred as proactive rejuvenation.

Proactive rejuvenation is either based on analytical methods or measurements. The survey in [15] provides an outline of existing work following the former approach and the later approach is covered by [3], [13], [16], [12], [17]. Since the primary goal is to minimize rejuvenation cost and maximize system availability branches like recovery oriented computing also study how to optimize the actual restart process [18], [19]. Also virtualization and replication are deployed to provide uninterrupted service for software that suffers from software aging despite the rejuvenation procedures [20], [16], [21].

All rejuvenation-based approaches only reduce the symptoms of software aging, but they do not help to remove the aging-related defects. An alternative active research area concerns the memory debuggers [6]. Their utility is restricted to memory leaks, a special case of software aging. These tools help to isolate memory-related defects in the source code of a program and can be categorized by the their appliance technique (i.e. static analysis, compile/link time-based or runtime-based).

Most related to our approach are LeakBot [22] and LeakChaser [23], since they both are runtime-based and developed for the Java Virtual Machine. However, they are not able to detect aging during unit or integration tests as they require human involvement to interpret results.

The version comparison approach was introduced in our previous work [7], where we used operating system metrics to compare the memory consumption behavior of two versions of the same software. In this we extended and improved this approach by using using monitoring of memory allocation on the level of individual allocation sites. This increased the sensitivity to leaks and allowed defect isolation (root cause analysis). Due to low runtime overhead of our method (investigated but not shown in this work) our method can enhance unit or integration tests with capability of detecting memory leaks.

In another recent approach, accelerated tests based on aging factors are used to reduce the time to model the software aging of a system [24], [25]. These method could facilitate using leak detection during unit or integration tests. However, currently it does not reduces the detection time significantly.

V. Conclusion

In this work we proposed and evaluated a new approach to early detect software aging related memory leak defects during the software development using automated tests. The evaluation results confirm that this approach can pinpoint the root causes of memory leak defects in a realistic usage scenario. Therefore, we introduced artificial memory leak defects to a component of the popular big data computing framework HADOOP.

Moreover, the artificial defects stand out from statistical noise due to our ranking strategy. What we consider noise during our investigation actually are suspicions for which we did not determine whether they point to true memory leak defects or not. The absolute ranks of the known artificial defects are below rank 20 in the majority of the cases throughout all the experiments.

In future work explorative studies need to verify our results in real world software development processes.

We also plan to generalize the main idea of our approach to any kind of leaking resource involved in software aging (e.g. memory, open files, semaphores, sockets, etc.). For not managed resources, static probing can be a valid equivalent to our heap memory instrumentation of the java virtual machine. The probing therefore should target any kind of allocate/open and free/close operation within the source code.

To improve the performance of our approach instrumentation as well as the testing itself can be done change centric. Since the instrumentation overhead depends on the amount of objects monitored, we can decrease this overhead by monitoring objects only if they are relevant to the source code change.

VI. Acknowledgments

This work is supported in part by the grant AN 405/2-1 entitled *Automated, minimal-invasive Identification and Elimination of Defects in Complex Software Systems* financed by the Deutsche Forschungsgemeinschaft (DFG).

References

[1] D. L. Parnas, "Software aging," in *Proceedings 16th International Conference on Software Engineering (ICSE '94)*, pp. 279–287, May 1994.

[2] F. Machida, J. Xiang, K. Tadano, and Y. Maeno, "Aging-related bugs in cloud computing software," in *ISSRE Workshops*, pp. 287–292, 2012.

[3] A. Andrzejak and L. Silva, "Deterministic models of software aging and optimal rejuvenation schedules," in *10th IFIP/IEEE Symposium on Integrated Management (IM 2007)*, (Munich, Germany), May 2007.

[4] J. Araujo, R. Matos, P. Maciel, and R. Matias, "Software aging issues on the eucalyptus cloud computing infrastructure," in *Proc. IEEE Int Systems, Man, and Cybernetics (SMC) Conf*, pp. 1411–1416, 2011.

[5] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: Analysis, module and applications," in *Proceedings of Fault-Tolerant Computing Symposium FTCS-25*, June 1995.

[6] Wikipedia, "Memory debugger." http://en.wikipedia.org/wiki/Memory_debugger, 2013.

[7] F. Langner and A. Andrzejak, "Detecting software aging in a cloud computing framework by comparing development versions," 2012.

[8] J. Manson, "java-allocation-instrumenter: A Java agent that rewrites bytecode to instrument allocation sites." https://code.google.com/p/java-allocation-instrumenter/, February 2012.

[9] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[10] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in Action*. Manning Publications Co. 20 Baldwin Road PO Box 261 Shelter Island, NY 11964: Manning Publications Co., first ed., 2011.

[11] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 199–209, ACM, 2011.

[12] V. Castelli, R. Harper, P. Heidelberg, S. Hunter, K. Trivedi, K. Vaidyanathan, and W. Zeggert, "Proactive management of software aging," *IBM Journal Research & Development*, vol. 45, March 2001.

[13] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. Trivedi, "A methodology for detection and estimation of software aging," in *Proceedings of the 9th Int'l Symposium on Software Reliability Engineering*, pp. 282–292, 1998.

[14] K. Vaidyanathan and K. S. Trivedi, "A measurement-based model for estimation of resource exhaustion in operational software systems," in *Proceedings of 10th IEEE Int'l Symposium on Software Reliability Engineering*, pp. 84–93, November 1999.

[15] K. Vaidyanathan and K. S. Trivedi, "A comprehensive model for software rejuvenation," *IEEE Trans. Dependanble and Secure Computing*, vol. 2, pp. 1–14, April-June 2005.

[16] A. Andrzejak, M. Moser, and L. Silva, "Managing performance of aging applications via synchronized replica rejuvenation," in *DSOM 2007*, (Silicon Valley, CA, USA), October 2007.

[17] L. Li, K. Vaidyanathan, and K. Trivedi, "An approach for estimation of software aging in a web-server," in *ISESE'02*, pp. 91–102, 2002.

[18] A. B. Brown and D. A. Patterson, "Embracing failure: A case for recovery-oriented computing," in *High Performance Transaction Processing Symposium*, October 2001.

[19] G. Candea, E. Kiciman, S. Zhang, and A. Fox., "Jagr: An autonomous self-recovering application server," in *Proc. 5th Int Workshop on Active Middleware Services*, June 2003.

[20] L. M. Silva, J. Alonso, P. Silva, J. Torres, and A. Andrzejak, "Using virtualization to improve software rejuvenation," in *IEEE International Symposium on Network Computing and Applications (IEEE-NCA)*, (Cambridge, MA, USA), July 2007.

[21] A. Nagarajan, F. Mueller, C. Engelmann, and S. Scott, "Proactive fault tolerance for HPC with Xen virtualization," in *International Conference on Supercomputing*, June 2007.

[22] N. Mitchell and G. Sevitsky, "LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications," *Lecture Notes in Computer Science*, vol. 2743, pp. 351–377, 2003.

[23] G. Xu, M. D. Bond, F. Qin, and A. Rountev, "Leakchaser: helping programmers narrow down causes of memory leaks," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 270–282, 2011.

[24] R. Matias, P. A. Barbetta, K. S. Trivedi, and P. J. de Freitas Filho, "Accelerated degradation tests applied to software aging experiments," *IEEE Transactions on Reliability*, vol. 59, no. 1, pp. 102–114, 2010.

[25] R. Matias, K. S. Trivedi, and P. R. Maciel, "Using accelerated life tests to estimate time to software aging failure," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pp. 211–219, IEEE, 2010.