2016 IEEE 27th International Symposium on Software Reliability Engineering Workshops

Software Aging Detection Based on Differential Analysis: An Experimental Study

Rivalino Matias Jr[*], Guilherme O. de Sena[*], Artur Andrzejak[†], Kishor Trivedi[‡]
[*]Federal University of Uberlandia
[†]Heidelberg University, Heidelberg, Germany
[‡]Duke University, Durham, USA
rivalino@ufu.br, guilhermesena@mestrado.ufu.br, artur@uni-hd.de, ktrivedi@duke.edu

***Abstract*—In this study we evaluate the applicability of the differential software analysis approach to detect memory leaks under a real workload. For this purpose, we used three different versions of a widely used software application, where one version was used as baseline (memory leak free) and the other two subsequent versions were our research subjects; one of them is confirmed to suffer from memory leaks and the other is memory leak free. The latter two versions were used to evaluate the accuracy of the proposed approach to detect aging, with respect to the number of true and false alarms. The results confirmed the previous findings obtained with synthetic workloads. The heap usage was a better metric than the resident set size, which has been extensively used for detecting software aging related to memory leakage. Also, the best data processing techniques to combine with the heap usage metric were Cumulative sum control chart and exponentially weighted moving average.**

Keywords—software aging; memory leak; detection; differential analysis

I. Introduction

In the literature of software aging and rejuvenation (SAR) research, erroneous memory consumption is one of the most investigated software aging effects so far [1]. Aging-related memory consumption is mainly caused by memory leakage and memory fragmentation [2], where the former is predominant in the SAR literature. This large interest to investigate memory leaks is justified by the importance and ubiquity of this problem. Different rejuvenation approaches [3] have been proposed to mitigate the effects of memory leaks, since the complete removal of software faults that cause memory leaks is difficult or unfeasible to accomplish during the systems development life cycle.

Despite of advancement of testing-based and other approaches to improve software reliability, aging-related defects remain a challenge. Especially defects related to memory leaks can be difficult to diagnose as their effects frequently manifest after a prolonged execution time and under particular workload conditions. Consequently, short testing time is typically not enough to expose the potential aging effects caused by memory leaks. Indeed, it has been observed that relatively short test durations increase the rate of false alarms of aging when using resource depletion metrics along with detection techniques based on trend analysis approaches, frequently adopted in the software aging literature [4]. Thus, we consider a major challenge to reliably detect the existence of software aging in a short period of test time, especially when dealing with memory leaks.

To address this problem, the study [5] proposed a systematic approach to detect software aging in a shorter period of time and with higher accuracy. The authors followed a *differential software analysis* approach [6]. They contrast the behavior of a new software version under test and its previous stable version that is considered a robust (aging-free) version of the software. Based on the concept of *divergence chart*, the authors compare a baseline signal, computed from the stable software version monitored data, against a target signal computed from monitored data of the software version under aging detection test, and quantify the deviation of both signals. If this deviation is significant statistically, then aging detection is declared. Their experimental study focused on memory leaks. The findings discussed in [5] are encouraging, given that they present preliminary positive evidences on the feasibility of the proposed approach for fast and accurate memory leak detection. The authors evaluated their approach in a variety of usage scenarios thru controlled experiments and emulating real workloads synthetically. Once passed the tests with synthetic workloads, the next step towards the external validity [7] of the proposed method is to evaluate it under real workloads.

Hence, in this paper we evaluate the applicability and effectiveness of the *differential software analysis* discussed in [5] to detect memory-related aging in real-world applications. For this study we selected three versions of a widely used application, where one of them was used as the robust version (memory leak free) baseline, and the other subsequent two versions as the research subjects under analysis; one is confirmed to suffer from memory leaks and the other is confirmed to be memory leak free. The latter two versions were used to evaluate the accuracy of the proposed approach to detect aging, based on the number of true and false alarms. We conducted the same analyses discussed in [5] in order to compare both studies with respect to the results obtained with synthetic and real workloads.

The remainder of the paper is organized as follows. Section II revisits the *differential software analysis* approach discussed in [5]. Section III presents the experimental plan and the real application selected for this study. Section IV discusses the main results obtained and Section V addresses their threats to validity. Finally, Section VI summarizes our conclusion.

This research work was partially supported by CNPq, CAPES, and FAPEMIG.

II. Evaluated Approach

The *differential software analysis* approach evaluated in this study is illustrated in Fig. 1. It consists of three steps: (1) measurements from a target software version under test (SVUT), (2) processing of the collected data for statistical analysis, and (3) detecting unexpected resource usage patterns. The third step is based on the proper comparison of the collected data from the first step, against baseline data obtained from a previous robust version of the software under test. In this work the robust version means a memory leak free version.

In the first step (*Measurement*), the collected data can include multiple system metric values, i.e., aging indicators[1]. In [5], the authors initially investigated 66 aging indicators for their memory leak experimental study, however, they found that only the resident set size (RSS) and the heap usage (HUS) indicators were effective for their purpose, where the latter showed the best results in general. Since our focus was also on memory leaks, in order to compare both studies, we also considered these two aging indicators. Next, the data collected are organized as time series for each aging indicator monitored.

In the second step (*Data processing*), each aging indicator based time series is used to obtain baseline and target signals by means of different data processing techniques. In [5], the authors evaluated seven techniques: Rolling linear regression (LR), moving average (MA), moving median (MM), Hodrick-Prescott filter (HP), Shewhart control chart (SH), Cumulative sum control chart (CS), and exponentially weighted moving average (EW). All these techniques and their respective parameterization were used in this work to allow the comparison of the studies.

Rolling linear regression is the moving-average equivalent to the well-known linear regression technique [8], where the line fit is made over the time series data in a sliding window. Moving average [9] is applied to the time series data in order to smooth out short-term fluctuations and highlights longer-term trends or cycles. If the time series data contain outliers or surges, moving median [9] gives a more robust estimate of the trend. When the data is assumed to consist of trend and cycles, the Hodrick-Prescott filter [10] is useful to extract the trend from time series containing cycle components. The effectiveness of HP filter for software aging detection was previously investigated in [11].

The remaining three techniques are statistical process control (SPC) charts [12]. They are appropriate to capture shifts in the mean of the process under control (e.g., memory consumption), where the Shewhart charts are suitable to capture large shifts ($\geq 1.5\sigma$), CuSum charts are applicable to detect small shifts ($< 1.5\sigma$), and exponentially weighted moving average charts are able to detect both small and medium to large shifts [12].

Regarding the parameter values for the three SPC techniques, we adopted the same parameterization used in [5]. For CS and EW, we set the parameters to have comparable 3-sigma Shewhart's (SH) control limits. The SH chart with 3-sigma control limits, $SH_{3\sigma}$, has a very low probability (approx. 0.27%) of making a type I error (false-positive), i.e., a value falling outside of the control limits under normal behavior. In this case, false alarms are expected to occur in average once every 370.37 (1÷0.0027) observations.

[1] Aging indicators are explanatory variables that suggest whether or not the system is healthy; aging effects such as memory leakage are detected by monitoring proper aging indicators [13].

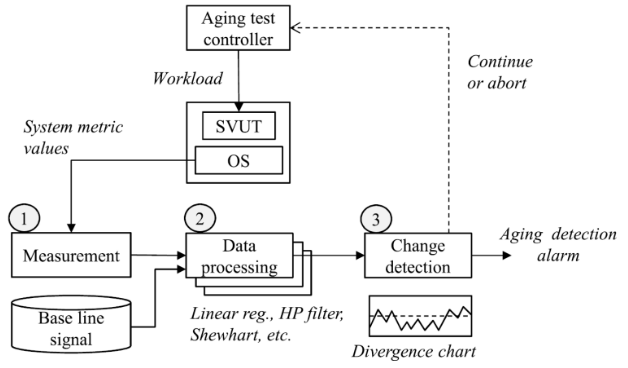

Fig. 1. Overview of the evaluated approach [5].

Based on the algorithms described in [12] and [14], we found the following parametrization relationship for the three SPC techniques, which was used in our experimental study: $SH_{3\sigma}$ (d_2 = 1.128), $CS_{3\sigma}$ (k = 0.5; h = 4.77), and $EW_{3\sigma}$ (L = 2.701; λ = 0.1). A more detailed explanation of these techniques is out of scope of this work and the reader is encouraged to consult their corresponding cited references.

In the third step (*Change detection*), the baseline and target signals computed in the previous step are compared against each other for detecting abnormal divergence between the two signals. For this purpose, firstly it is necessary to compute a series of divergence values, $\{f_i\}$, where each value relates to a normalized difference between the target and baseline signals, at the same sampling rate, from the beginning of the test time. To obtain the divergence values it is necessary to use three time series: target signal $\{f_t\}$, lower $\{L_t\}$ and upper $\{U_t\}$ bounds of a control limit interval. These two bounds series are derived from the baseline signal and indicate the range of the filtered target signal. The computation of series $\{f_t\}$, $\{L_t\}$ and $\{U_t\}$ depends on the data processing technique and a complete explanation of this procedure can be found in [5]. Hence, given the three series, for a fixed time t (i.e., corresponding elements of the three series), the divergence value f_i is computed by

$$\left(\frac{f_t - U_t}{U_t - L_t}\right)^+, \quad (1)$$

where $X^+ \equiv \max(X, 0)$. Thus, if the f_t is within the limit interval (or below), we get zero, otherwise the distance of f_t from the upper bound in units being the width of the control interval. The latter property is the normalization allowing uniform thresholds for the different data processing techniques.

Based on the series of divergence values, a *divergence chart* (e.g., Fig. 2) is created and used to detect significant changes in the SVUT resource usage pattern; in this study it means substantial and consistent memory usage increase in comparison with the reference baseline. The *divergence chart* is used according to the following protocol. Assume that the current divergence value f_i is below a given threshold value, α. When a sequence of five consecutive divergence values, f_{i+1} to f_{i+5}, are above or equals the threshold α, then it is considered that a *divergence event* has occurred. This event is associated to the sampling time of f_{i+5}. The choice of five consecutive values came from SPC theory, based on the probability of false alarms for a given control limit interval. In [5], preliminary

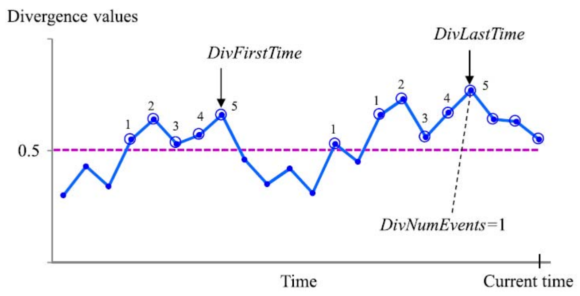

Fig. 2. Divergence chart and associated metrics [5].

analyses showed that for higher values (e.g., 10) the quality of the results did not change significantly. Like in [5], in this study we set the threshold α to 0.5, which was evaluated experimentally and showed good balance between detection latency and robustness. Due to normalization of divergence values, one can use a common threshold for all combinations of metrics and techniques.

Since the *divergence chart* computation is independent of specific data processing techniques and aging indicators, the results of change detections obtained applying different techniques and aging indicators are comparable to each other. Hence, in this study we applied all techniques and aging indicators evaluated in [5], in order to be able to compare the results of both studies. To evaluate and compare different technique and aging indicator combinations, the following concepts are used. The time of the first *divergence event* is called *DivFirstTime*. This is the earliest warning of detected aging effects and its value is dependent on the latency of a combination (technique and aging indicator). Note that the corresponding event may be a false alarm, and more divergence events could occur afterward. So, the robustness of a technique and indicator combination is estimated by *DivNumEvents*, which is the number of divergence events observed after *DivFirstTime*. *DivNumEvents* quantifies the number of false alarms. Since software aging progresses over time, the occurrences of false alarms are expected to disappear after a certain amount of test time. This gives rise to another metric: *DivLastTime*, which is the time of the last divergence event.

Thus, an ideal technique and indicator combination would have *DivNumEvents* = 0 and consequently *DivFirstTime* = *DivLastTime*. The higher is the spread between *DivFirstTime vs*. *DivLastTime* and/or larger *DivNumEvents*, the less robust is the given combination. Note that values of *DivLastTime* and *DivNumEvents* are available only after a sufficiently long minimal test execution time. This minimal test time must be fixed in advance depending on the system under test. Terminating test execution earlier and adaptively (e.g., after first occurrence of *DivFirstTime* or a certain number of *DivNumEvents*) can potentially further shorten the detection time. Adaptive test termination is innately less robust and was not studied in [5], so we also did not consider it in this work. Fig. 2 illustrates the above described concepts. Note that both events labeled with "5" are divergence events. There is just one such event after time *DivFirstTime* (namely at *DivLastTime*), therefore *DivNumEvents* = 1; which means one false positive alarm observed for the evaluated combination.

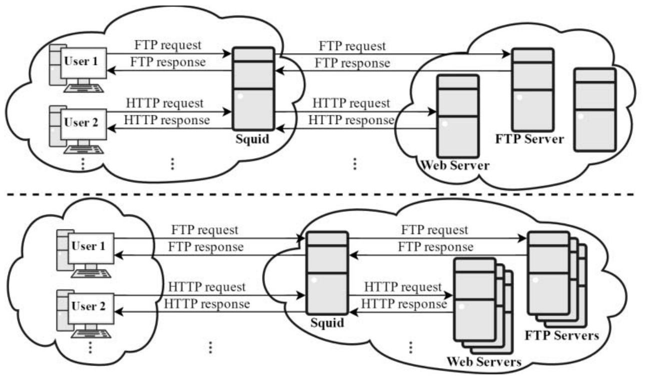

Fig. 3. Typical Squid usage scenarios: client-side proxy/cache (top), and server-side or reverse proxy/cache (bottom).

III. Experimental Study

A. Investigated Application

The software analyzed in this study was Squid [15]. It is a caching proxy for the web, which supports protocols such as HTTP/S, FTP, DNS, and others. Squid is a mature software project with almost 24 years of active existence; its current version has more than 810,000 lines of code. Squid has been used in thousands of Internet service providers and web sites around the world, where Wikipedia.org is probably the best-known large organization using it [16]. It has also been delivered embedded in many network appliance products [17]. Fig. 3 shows how Squid is typically used. As can be seen, it sits between clients and servers, playing a middle tier role for different protocols. The benefits of a proxy/cache system extends to areas such as security (e.g., content filtering), performance (e.g., content caching), and availability (e.g., load balancing), among others. As a central controller its failures might have high impact on the system's dependability.

For this study we investigated Squid's issue tracking system and found that version 3.2.1 was diagnosed with memory leak. As explained in Section II, in addition to the version under test, the evaluated approach requires a robust version to be used as baseline. We analyzed the previous versions of Squid looking for the closest appropriate one to be used as baseline and selected version 3.1.23; supplementary tests confirmed that this version is memory leak free. Additionally, we tested the approach against Squid version 3.5.6, which was released next to version 3.2.1 and fixed the memory-leak problem. The experiments that used these three versions of Squid are described in the next subsection.

B. Experimental Design

The memory leak problem detected in Squid 3.2.1 and reproduced in this study is related to the FTP protocol. It occurs when a FTP directory listing is parsed. In detail, a client request for listing an FTP server's directory triggers Squid to parse the results received from an FTP server, and this parsing creates a memory leak. Our experimental plan considered reproducing this memory leak problem using two types of workload: constant and varying.

In the *constant* workload, every 30 seconds the FTP client requested to Squid one operation of FTP server directory listing. In the *varying* workload, every 30 seconds a random number (between 0 and 10 inclusive) of FTP server directory listing operations were performed. All random numbers used in this study were obtained from the RANDOM.ORG web site, which offers true randomness.

We automated all tests to avoid any possible uncontrolled influence caused by human intervention. For this purpose, we selected 100 FTP URLs that pointed to different real FTP server directories located at the Internet. The same list of URLs was used in all experiments, in a round-robin fashion during the whole experiment runtime. In the *varying* workload experiments, the cases where the random number was greater than one involved multiple operations of directory listing, which used one different FTP URL per listing operation.

Each experiment run lasted for 4 hours, from which the first 3 h 30 min Squid processed the workload and in the last 30 minutes the system was at rest (without workload). This last 30-minute period was used to observe whether the amount of memory used by Squid was temporarily allocated as a consequence of the workload processing or not. For each test scenario, we replicated the experiment execution 10 times in order to reduce the influence of experimental errors. Thus, the results presented in Section IV were based on the averaged values of the ten runs.

C. *Testbed and Instrumentation*

Our testbed was composed of one computer hosting two virtual machines (VM1 and VM2) whose settings follow:

- Host machine: Intel Core i7-4510U CPU @ 2.60GHz, 8GB RAM, 500 GB disk, Windows 10.
- VM1: 2 CPU Cores, 1.0 GB RAM, 50 GB disk, Linux 4.4.0-28-generic x86_64.
- VM2: 2 CPU Cores, 1.0 GB RAM, 50 GB disk, Linux 2.6.32-431.el6.x86_64 x86_64.

We used the VMWare Workstation 11.1.0 as virtual machine monitor. VM1 was used to run the workload generator and VM2 to run Squid. Fig. 4 illustrates this setup. The workload generator used was a program written in JavaScript for this specific purpose. This program performed the *constant* and *varying* workloads, according to the procedures described in Section III.B. In order to execute our JavaScript based workload generator, we ran it inside the web browser Mozilla Firefox 47.0, which had its proxy settings configured for the Squid software running in VM2. Hence, all requests from the workload generator program running in Firefox passed through the Squid.

In terms of data collecting instrumentation, we monitored the *resident set size* (RSS) and the *heap usage* (HUS) and used them as aging indicators with respect to the Squid process running in VM2. For the RSS monitoring, we created a shell script to collect the RSS values from the Linux /proc file system. For the HUS, we intercepted all dynamic allocation and deallocation operations performed by Squid during runtime, and registered their parameter values that were used to measure the amount of memory allocated and deallocated from the Squid's heap during the experiments. For this purpose, we used a dynamic memory allocator wrapper called *DebugMalloc*, which was introduced in [18]. Given that the RSS was monitored every five seconds, the data collected for HUS were adjusted to be presented in the same time interval.

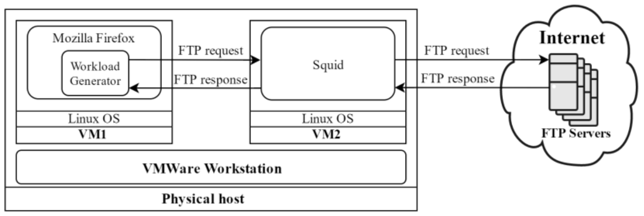

Fig. 4. Test-bed setup.

IV. Results

A. *Aging Detection Correctness*

Given that the Squid 3.2.1 is known to suffer from memory leak under the workload exercised in this study, we computed the divergence charts for all combinations of techniques and aging indicators assessed in [5], in order to evaluate their aging detection correctness under real workload. Fig. 8 (Appendix) presents the *divergence charts* computed. As described in Section II, these charts are used for changing detection, and they display the *divergence values* calculated thru the normalized difference between the baseline and the target signals. In Fig. 8, the baseline and target signals were obtained by monitoring Squid 3.1.23 and Squid 3.2.1, respectively. The series of *divergence values*, $\{f_i\}$, is represented by the black solid line and the threshold, α=0.5, is indicated by the horizontal red dashed line. The time (hh:min) is shown in the *x*-axis.

As can be seen in Fig. 8, for constant workload and using RSS, all techniques, except SH, failed to detect the aging effects within the 4-hour test time. On the other hand, for the same constant workload but using HUS, all SPC techniques detected aging correctly. The same analysis was conducted for the varying workload. We can see that using RSS only the CS detected the aging effects and all other techniques failed (false-negatives). On the other hand, all techniques in combination with HUS were able to detect the aging effects within the test time. Contrasting these results with the corresponding findings in [5], which were based on synthetic workloads, we observe that HUS outperformed RSS in both studies. However, differently from [5], in our experiments HUS was largely better. Similarly, in general, the SPC techniques showed the best detection accuracy in both studies.

B. *Aging Detection Time*

In order to compare the performance of the combinations of techniques and aging indicators that correctly detected the memory leakage in Squid 3.2.1, we analyzed their detection times, mainly based on *DivFirstTime* and *DivLastTime*. If competing combinations show the same *DivLastTime*, then *DivNumEvents* is used as a second selection criterion, where the lowest value is better; it indicates lesser false alarms. This analysis can be better observed in Fig. 5, where the time (in hour) is shown in the *x*-axis.

For constant workload, all SPC techniques accurately (i.e., *DivFirstTime*=*DivLastTime*) detected the aging effects. In combination with HUS, their detection times were: SH (7 min 40 s), EW (9 min 20 s), and CS (9 min 35 s). Along with RSS, the SH detection time was (33 min 40 s). For varying workload, all techniques combined with HUS detected the

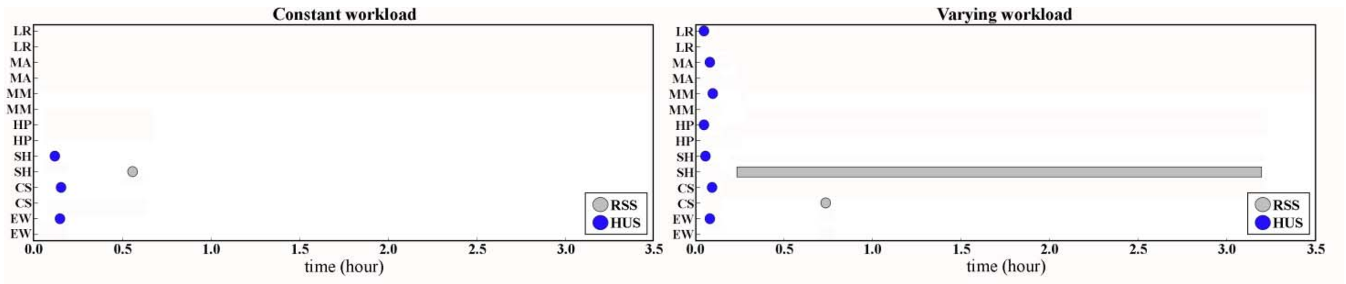

Fig. 5. First (*DivFirstTime*) and last (*DivLastTime*) detection times per technique and aging indicator combination; for constant (left) and varying (right) workloads.

aging effects as follows: LR and HP (3 min 00 s), SH (3 min 35 s), MA and EW (5 min 05 s), CS (5 min 45 s), and MM (5 min 55 s). For CS combined with RSS the detection time was 43 min 40 s. Note that all the above-mentioned aging detections had no false alarms (*DivFirstTime* = *DivLastTime*). Differently, SH with RSS produced 11 false-positive alarms (i.e., *DivNumEvents* > 0), from 15 min 45 s (*DivFirstTime*) to 3 h 7 min 45 s (*DivLastTime*), which is represented in Fig. 5 as a gray horizontal bar.

Comparing these findings with the corresponding detection times in [5], once more the SPC techniques presented the best performance in both studies. On the other hand, in this work most of the detection times for varying workload were shorter than for constant workload, while [5] reported the opposite. Surprisingly, aging detection in varying workload scenarios is expected to be harder to detect than in constant workload scenarios, given the higher entropy. In this work, based on a real workload (Squid), the best detection times were 3 min and 7 min 40 s, for varying and constant workloads respectively. Fig. 6 summarizes the results for the detection time analyses. It is possible to note significant favorable differences for varying workload, which corroborate the visual analysis of the *divergence charts*, where the best techniques identified show higher efficiency for this workload scenario. These findings are encouraging when compared to results in the literature. For instance, searching for studies reporting memory leak detection times in real-world applications, similar to our case study, we found in [19] a minimal test time of 2.5 hours to detect memory leaks in a middleware for critical applications. Also, we found in [20] two comparable case studies: Apache MQ broker (36 minutes) and an eHealth web app (2 h 30 min).

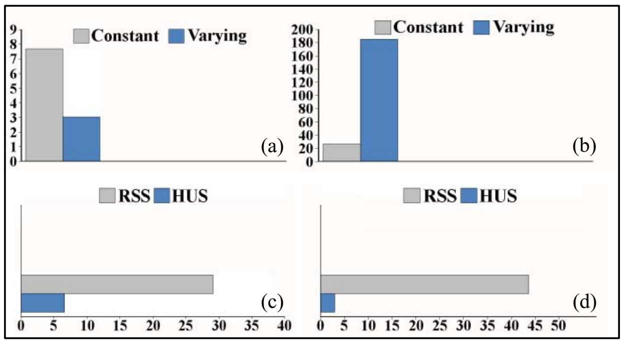

Fig. 6. Summary of detection times (values in *x*-axis and *y*-axis in minutes): Best detection times (a); Difference of best and worst detection times (b); Best detection time for constant workload (c) and varying workload (d).

C. *Testing Against a Control Group*

In addition to the tests previously discussed, which evaluated the *differential software analysis* approach against the Squid version 3.2.1 that suffers from memory leakage, we also tested this approach against the Squid version 3.5.6; this is the subsequent Squid version with the memory leak problem fixed (our control group). With respect to the aging detection correctness, Fig. 9 (Appendix) presents the *divergence charts* computed for this analysis. Due to the limited number of pages available, we plotted only the combinations that presented divergence events.

As can be seen, for constant workload and using RSS, the SH failed to recognize the leak-free version, and erroneously confirmed memory leakage. For the same constant workload but using HUS, again SH was the only technique that failed in recognizing the memory leak-free version. Fig. 7 shows the erroneous positive detection times of memory leak. Among all techniques investigated, SH is the only one that keeps the original data point with no data smoothing. Since RSS values presented higher variability than HUS values, specifically in the beginning of the series, which could be caused by the expected higher page-fault rate during the software initialization, we hypothesize that the initial transient fluctuations in the data could explain these SH's false alarms.

On the other hand, for the varying workload, we can see that all techniques combined with both RSS and HUS were able to correctly recognize the absence of memory leaks. Note that for the combination SH and RSS, although the thirteen occurrences of false alarm, from time 2 h 6 min 10 s onwards all divergence values fell below the threshold, thus confirming the memory leak-free version within the test time. Contrasting these findings with the ones discussed in Sections IV.A and IV.B, we mainly observe that HUS outperformed RSS to detect both presence and absence of memory leaks, and the best results were obtained for the varying workload scenario.

V. Threats to Validity

Like any empirical research, this study has limitations that must be considered when interpreting its results. In this section, we highlight some internal and external validity threats [4].

A. *Internal Validity*

This validity addresses the quality of the evidences adopted to support our conclusion. Experimental errors are present in any experimental study, so, in order to mitigate this threat our results were based on the average of multiple replications of each experiment. Therefore, our numerical results must be

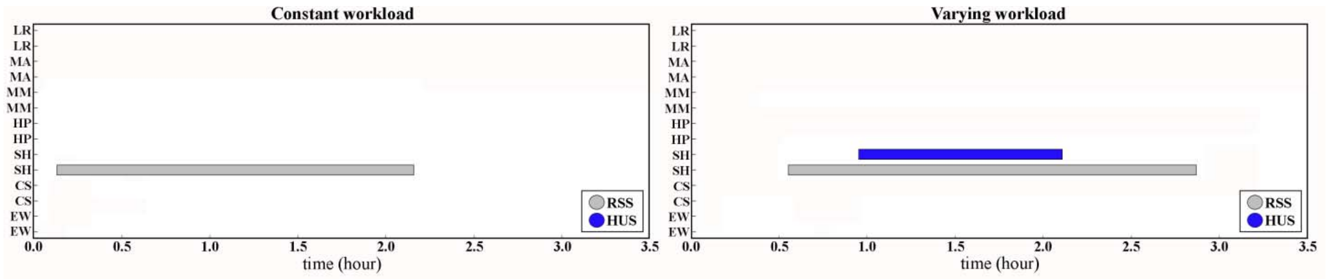

Fig. 7. Erroneous positive detection times of memory leak per technique and aging indicator combination; for constant (left) and varying (right) workloads.

interpreted as averages. Moreover, the instrumentation used to collect HUS data considered nanosecond precision for the time stamp registered for each allocation or deallocation request, but the real accuracy is limited to the quality of the hardware timer. We observe a transient period in the beginning of all the time series analyzed, where the data variability is considerably higher than in the rest of the series. This higher variability in the beginning of the series may have influenced the results for the SH technique, which does not perform data smoothing and thus would work better without this initial transient data points.

B. External Validity

This validity addresses the extent to which our results can be generalized. As described in Section I, the *differential software analysis* approach investigated in this work was firstly evaluated under a varied of synthetic workloads in [5]. Complementarily, in this work we assessed the approach based on a real application. Contrasting the findings obtained in both studies, it is possible to conclude that they corroborate with each other in most of the main results. In terms of representativeness of the selected application, we consider Squid a good research subject for representing the category of web-oriented server applications, which are typically affected by software aging. Hence, although our findings are specific to Squid, we expect that they may be compatible with similar server applications like web servers, ftp servers, file servers, etc. A possible threat to the external validity of our results is related to the FTP request rate used. For scenarios where the Squid is placed on the client side (Fig. 3, top), a 30-second FTP request rate is unusual for most organizations. However, when placed at the server side (Fig. 3, bottom), mainly in large service providers, this rate might be suitable. A lower request rate would reduce the amount of aging effects accumulated and consequently change the detection times. Our experimental study is not complete enough to assure that the obtained results related to accuracy and performance of the techniques and aging indicators combinations can be generalized to a large variety of software systems. On the other hand, the *differential software analysis* approach evaluated is general enough to support various techniques and aging indicators, as well as be applicable to different software categories.

VI. Conclusion

In this study, we assessed the external validity of the *differential software analysis* approach proposed in [5]. For this purpose, we used the web caching proxy Squid, a widely-adopted server application. The results obtained with Squid indicated that the *heap usage*, HUS, is a better metric (aging indicator) than the *resident set size*, RSS, which has been preferably used for aging detection related to memory leakage. This result considered different data processing techniques used in previous works. In this case, the best results were observed using the SPC techniques Cumulative sum control chart (CS) and exponentially weighted moving average (EW). Importantly, this study confirmed the main findings obtained in [5], however using a real workload that complementarily offers stronger evidence of the robustness of the approach evaluated.

References

[1] D. Cotroneo, R. Natella, R. Pietrantuono and S. Russo, "A Survey of Software Aging and Rejuvenation Studies," ACM J. on Emerging Technologies in Computing Systems, vol. 10, pp. 1-34, January 2014.

[2] A. Macedo, T. B. Ferreira and R. Matias, "The mechanics of memory-related software aging," in *Int. Workshop on Software Aging and Rejuvenation (WoSAR)*, San Jose, CA, 2010, pp.1-5.

[3] J. Alonso, R. Matias, E. Vicente, A. Maria and K. S. Trivedi, "A Comparative Experimental Study of Software Rejuvenation Overhead," Performance Evaluation Int. J., vol. 70, pp. 231-250, September 2012.

[4] F. Machida, A. Andrzejak, R. Matias and E. Vicente, "On The Effectiveness of Mann-Kendall Test for Detection of Software Aging," in *Int. Workshop on Software Aging and Rejuvenation (WoSAR)*, Pasadena, CA, 2013, pp. 269-274.

[5] R. Matias, A. Andrzejak, F. Machida, D. Elias and K. S. Trivedi, "A Systematic Differential Analysis for Fast and Robust Detection of Software Aging," in *Proc. IEEE 33rd Int. Symp. on Reliable Distributed Systems*, Nara, 2014, pp. 311-320.

[6] F. Langner and A. Andrzejak, "Detecting Software Aging in a Cloud Computing Framework by Comparing Development Versions," in *IFIP/IEEE Int. Symp. on Integrated Network Management (IM),* Ghent, 2013, pp. 896-899.

[7] J. Siegmund, N. Siegmund and S. Apel, "Views on internal and external validity in empirical software engineering," in *IEEE/ACM 37th Int. Conf. on Software Engineering (ICSE),* Florence, 2015, pp. 9-19.

[8] J. Fox, Applied Regression Analysis, Linear Models, and Related Methods, 1st ed., SAGE Publications, Inc., 1997.

[9] J. D. Hamilton, Time Series Analysis, Princeton University Press, 1994.

[10] R. J. Hodrick and E. C. Prescott, "Post-war U.S. business cycles: An Empirical investigation," J. of Money, Credit and Banking, vol. 29, pp. 1-16, February 1980.

[11] P. Zheng, Q. Xu and Y. Qi, "An advanced methodology for measuring and characterizing software aging," in *Int. Workshop on Software Aging and Rejuvenation (WoSAR),* Dallas, TX, 2012, pp. 253-258.

[12] D. C. Montgomery, Introduction to statistical quality control, 6th ed., John Wiley & Sons, Inc., 1996.

[13] M. Grotke, R. Matias and K. S. Trivedi, "The Fundamentals of Software Aging," in *Int. Workshop on Software Aging and Rejuvenation (WoSAR)*, Seattle, WA, 2008, pp. 1-6.

[14] J. Yang and V. Makis, "On the performance of classical control charts applied to process residuals," Computers and Industrial Engineering, vol. 33, pp. 121-124, October 1997.

[15] The Squid Software Foundation. (2016, Aug 08). *Squid: Optimising Web Delivery* [Online]. Available: http://www.squid-cache.org/

[16] O. Livneh, Wikimedia Foundation. (2016, Aug 08). *How we made editing Wikipedia twice as fast* [Online]. Available: https://blog.wikimedia.org/2014/12/29/how-we-made-editing-wikipedia-twice-as-fast/

[17] The Squid Software Foundation. (2016, Aug 08). *Squid-based Products* [Online]. Available: http://www.squid-cache.org/Support/products.html

[18] D. Costa and R. Matias, "Characterization of dynamic memory allocations in real-world applications: an experimental study," in *IEEE 23rd Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Atlanta, GA, 2015, pp. 93-101.

[19] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia and S. Russo, "Memory leak analysis of mission-critical middleware," J. of Systems and Software, v.83, pp.1556–1567, September 2010.

[20] V. Šor, S. N. Srirama and N. Salnikov-Tarnovski, "Memory leak detection in Plumbr," J. of Software: Practice and Experience, v.45, pp.1307–1330, October 2015.

APPENDIX

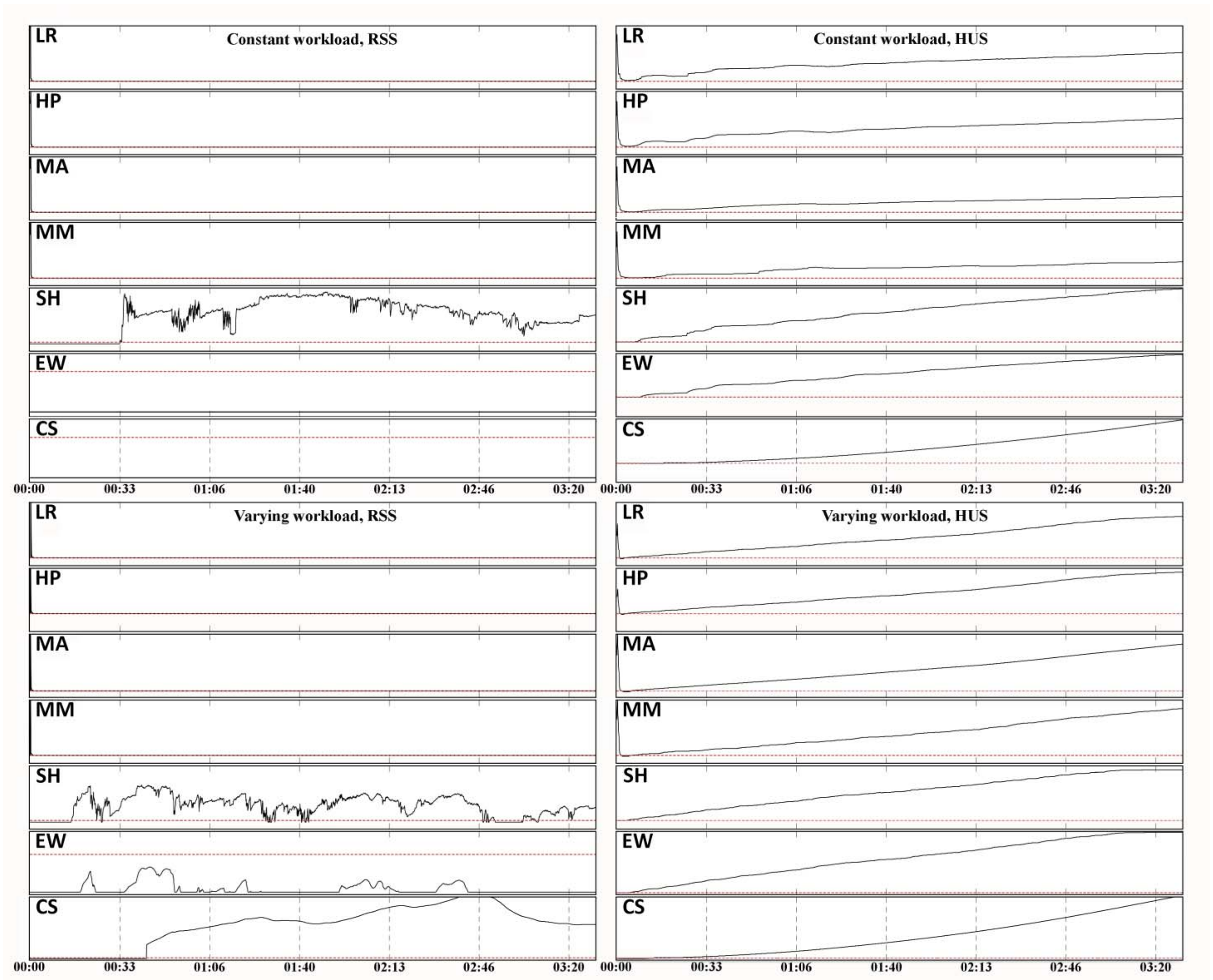

Fig. 8. Divergence charts for RSS (left) and HUS (right) under constant (upper) and varying (lower) workloads, for all data processing techniques evaluated.

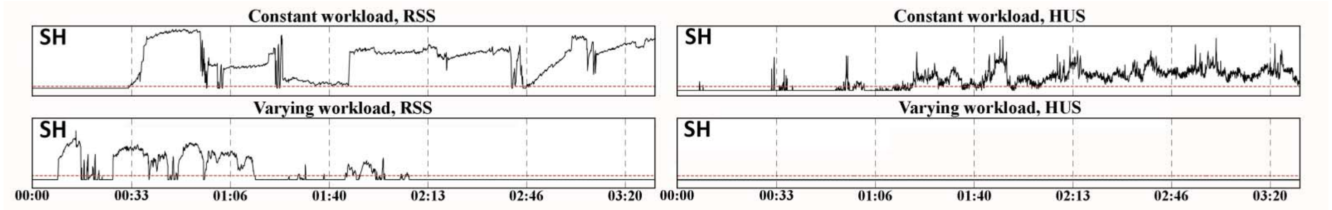

Fig. 9. Divergence charts for RSS (left) and HUS (right) under constant (upper) and varying (lower) workloads, for the Shewhart control chart.