Online Memory Leak Detection in the Cloud-based Infrastructures

Anshul Jindal$^{1[0000-0002-7773-5342]}$, Paul Staab^{2}, Jorge Cardoso$^{2[0000-0001-8992-3466]}$, Michael Gerndt$^{1[0000-0002-3210-5048]}$, and Vladimir Podolskiy$^{1[0000-0002-2775-3630]}$

[1] Chair of Computer Architecture and Parallel Systems, Technical University of Munich, Garching, Germany
`anshul.jindal@tum.de, gerndt@in.tum.de, v.podolskiy@tum.de`

[2] Huawei Munich Research Center, Huawei Technologies Munich,Germany
{`paul.staab, jorge.cardoso`}`@huawei.com`

Abstract. A memory leak in an application deployed on the cloud can affect the availability and reliability of the application. Therefore, to identify and ultimately resolve it quickly is highly important. However, in the production environment running on the cloud, memory leak detection is a challenge without the knowledge of the application or its internal object allocation details.
This paper addresses this *challenge of online detection of memory leaks in cloud-based infrastructure without having any internal application knowledge* by introducing a novel machine learning based algorithm Precog. This algorithm solely uses one metric i.e the system's memory utilization on which the application is deployed for the detection of a memory leak. The developed algorithm's accuracy was tested on 60 virtual machines manually labeled memory utilization data provided by our industry partner Huawei Munich Research Center and it was found that the proposed algorithm achieves the accuracy score of 85% with less than half a second prediction time per virtual machine.

Keywords: memory leak · online memory leak detection · memory leak patterns · cloud· linear regression

1 Introduction

Cloud computing is widely used in the industries for its capability to provide cheap and on-demand access to compute and storage resources. Physical servers resources located at different data centers are split among the virtual machines (VMs) hosted on it and distributed to the users [5]. Users can then deploy their applications on these VMs with only the required resources. This allows the efficient usage of the physical hardware and reducing the overall cost. However, with all the advantages of cloud computing there exists the drawback of detecting a fault or an error in an application or in a VM efficiently due to the layered virtualisation stack [1,4]. A small fault somewhere in the system can impact the performance of the application.

An application when deployed on a VM usually requires different system resources such as memory, CPU and network for the completion of a task. If an application is mostly using the memory for the processing of the tasks then this application is called a memory-intensive application [8]. It is the responsibility of the application to release the system resources when they are no longer needed. When such an application fails to release the memory resources, a **memory leak** occurs in the application [14]. Memory leak issues in the application can cause continuous blocking of the VM's resources which may in turn result in slower response times or application failure. In software industry, memory leaks are treated with utmost seriousness and priority as the impact of a memory leak could be catastrophic to the whole system. In the development environment, these issues are rather easily detectable with the help of static source code analysis tools or by analyzing the heap dumps. But in the production environment running on the cloud, memory leak detection is a challenge and it only gets detected when there is an abnormality in the run time, abnormal usage of the system resources, crash of the application or restart of the VM. Then the resolution of such an issue is done at the cost of compromising the availability and reliability of the application. Therefore it is necessary to monitor every application for memory leak and have an automatic detection mechanism for memory leak before it actually occurs. However, it is a challenge to detect memory leak of an application running on a VM in the cloud without the knowledge of the programming language of the application, nor the knowledge of source code nor the low level details such as allocation times of objects, object staleness, or the object references [10]. Due to the low down time requirements for the applications running on the cloud, detection of issues and their resolutions is to be done as quickly as possible. Therefore, this challenge is addressed in this paper *by solely using the VM's memory utilization as the main metric and devising a novel algorithm called* ***Precog*** *to detect memory leak.*

The main contribution of this paper are as follows:

- **Algorithm**: We propose an online novel machine learning based algorithm **Precog** for accurate and efficient detection of memory leaks by solely using the VM's memory utilization as the main metric.
- **Effectiveness**: Our proposed algorithm achieves the accuracy score of 85% on the evaluated dataset provided by our industry partner and accuracy score of above 90% on the synthetic data generated by us.
- **Scalability**: Precog's predict functionality is linearly scalable with the number of values and takes less than a second for predicting in a timeseries with 100,000 values.

Reproducibility: our code and synthetic generated data are publicly available at: https://github.com/ansjin/memory_leak_detection.

2 Related Work

Memory leak detection has been studied over the years and several solutions have been proposed. Sor et al. reviewed different memory leak detection approaches based on their implementation complexity, measured metrics, and intrusiveness and a classification taxonomy was proposed [11]. The classification taxonomy broadly divided the detection algorithms into *(1) Online detection, (2) Offline detection and (3) Hybrid detection.* The *online detection* category uses either staleness measure of the allocated objects or their growth analysis. *Offline detection* category includes the algorithms that make use of captured states i.e heap dumps or use a visualization mechanism to manually detect memory leaks or use static source code analysis. *Hybrid detection* category methods combine the features offered by online and offline methods to detect memory leaks. Our work falls in the category of online detection therefore, we now restrict our discussion to the approaches related to the online detection category only.

Based on the staleness measure of allocated objects, Rudaf et al. proposed "LeakSpot" for detecting memory leaks in web applications [9]. It locates JavaScript allocation and reference sites that produce and retain increasing numbers of objects over time and uses staleness as a heuristic to identify memory leaks. Vladimir Šor et al. proposes a statistical metric called *genCount* for memory leak detection in Java applications [12]. It uses the number of different generations of the objects grouped by their allocation sites, to abstract the object staleness - an important attribute indicating a memory leak. Vilk et al. proposed a browser leak debugger for automatically debugging memory leaks in web applications called as "BLeak" [13]. It collects heap snapshots and analyzes these snapshots over time to identify and rank leaks. BLeak targets application source code when detecting memory leaks.

Based on the growth analysis objects, Jump et al. proposes "Cork" which finds the growth of heap data structure via a directed graph *Type Points-From Graph* - TPFG, a data structure which describes an object and its outgoing reference [6]. To find memory leaks, TPFG's growth is analyzed over time in terms of growing types such as a list. FindLeaks proposed by Chen et al. tracks object creation and destruction and if more objects are created than destroyed per class then the memory leak is found [2]. Nick Mitchell and Gary Sevitsky proposed "LeakBot", which looks for heap size growth patterns in the heap graphs of Java applications to find memory leaks [7]. "LEAKPOINT" proposed by Clause et al. uses dynamic tainting to track heap memory pointers and further analyze it to detect memory leaks [3].

Most of the online detection algorithms that are proposed focus either on the programming language of the running application or on garbage collection strategies or the internals of the application based on the object's allocation, references, and deallocation. To the best of our knowledge, there is no previous work that solely focuses on the detection of memory leaks using just the system's memory utilization data on which application is deployed. The work in this paper, therefore, focuses on the detection of a memory leak pattern irrespective of the programming language of the application or the knowledge of application's source code or the low-level details such as allocation times of objects, object staleness, or the object references.

Table 1: Symbols and definitions.

Symbol	**Interpretation**
t	a timestamp
x_t	the percentage utilization of a resource (for example memory or disk usage) of a virtual machine at time t
N	Number of data points
$x = \{x_1, x_2, ..., x_N\}$	a VM's memory utilization observations from the Cloud
T	time series window length
$x_{t-T:t}$	a sequence of observations $\{x_{t-T}, x_{t-T+1}, ..., x_t\}$ from time $t - T$ to t
U	percentage memory utilization threshold equal to 100.
C	critical time

3 Methodology for Memory Leak Detection

In this section, we present the problem statement of memory leak detection and describes our proposed algorithm's workflow for solving it.

3.1 Problem Statement

Table 1 shows the symbols used in this paper.

We are given $x = \{x_1, x_2, ..., x_N\}$, an $N1$ dataset representing the memory utilization observations of the VM and an observation $x_t \in R$ is the percentage memory utilization of a virtual machine at time t. The objective of this work is to determine whether or not there is a memory leak on a VM such that an observation x_t at time t reaches the threshold U memory utilization following a trend in the defined critical time C. Formally:

Problem 1. (Memory Leak Detection)

- **Given**: a univariate dataset of N time ticks, $x = \{x_1, x_2, ..., x_N\}$, representing the memory utilization observations of the VM.
- **Output**: an anomalous window for a VM consisting of a sequence of observations $x_{t-T:t}$ such that these observations after following a certain trend will reach the threshold U memory utilization at time $t + M$ where $M \leq C$.

Definition 1. *(Critical Time) It is the maximum time considered relevant for reporting a memory leak in which if the trend line of memory utilization of VM is projected, it will reach the threshold U.*

3.2 Illustrative Example

Fig. 1 shows the example memory utilization of a memory leaking VM with the marked anomalous window between t_k and t_n. It shows that the memory utilization of the VM will reach the defined threshold ($U = 100\%$) within the

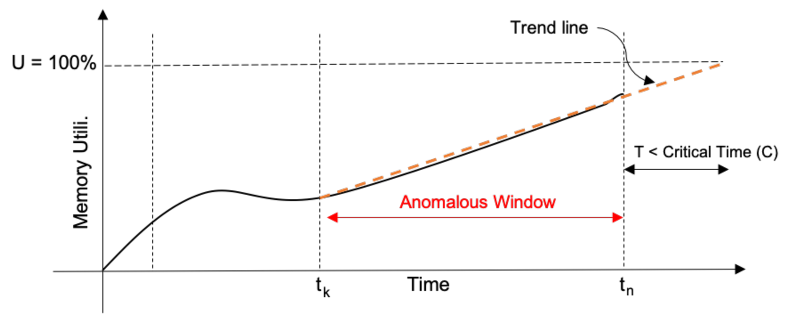


Fig. 1: Example memory utilization of a memory leaking VM with the marked anomalous window.

defined critical time C by following a linearly increasing trend (shown by the trend line) from the observations in the anomalous window. Therefore, this VM is regarded as a memory leaking VM.

Our developed approach can be applied for multiple VMs as well. We also have conducted an experiment to understand the memory usage patterns of memory leak applications. We found that, if an application has a memory leak, usually the memory usage of the VM on which it is running increases steadily. It continues to do so until all the available memory of the system is exhausted. This usually causes the application attempting to allocate the memory to terminate itself. Thus, usually a memory leak behaviour exhibits a linearly increasing or ”sawtooth” memory utilization pattern.

3.3 Memory Leak Detection Algorithm: Precog

The Precog algorithm consists of two phases: offline training and online detection. Fig. 2 shows the overall workflow of the Precog algorithm.

Offline training: The procedure starts by collecting the memory utilization data of a VM and passing it to *Data Pre-processing* module, where the dataset is first transformed by resampling the number of observations to one every defined resampling time resolution and then the time series data is median smoothed over the specified smoothing window. In *Trend Lines Fitting* module, firstly, on the whole dataset, the change points $P = \{P_1, P_2, ..., P_k\}$, where $k \leq n-1$, are detected. By default, two change points one at the beginning and other at the end of time series data are added. If the change points are not detected, then the algorithm will have to go though each data point and it will be compute intensive, therefore these points allows the algorithm to directly jump from one change point to another and selecting all the points in between the two change points. *Trend Lines Fitting* module selects a sequence of observations $x_{t-L:t}$ between the two change points: one fixed P_1 and other variable P_r where $r \leq k$ and a line is fitted on them using the linear regression. The R-squared score, size of the window called as *duration*, time to reach threshold called *exit time* and

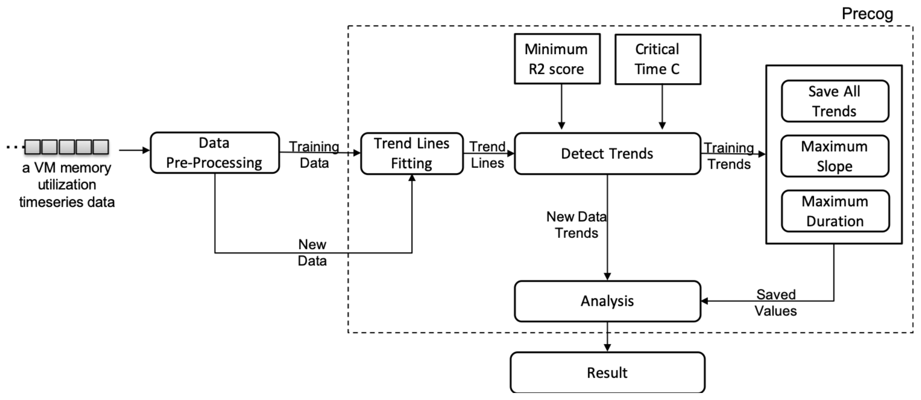


Fig. 2: Overall workflow of Precog algorithm.

slope of line are calculated. This procedure is repeated with keeping the fixed change point the same and varying the other for all other change points. Out of all the fitted lines, the best-fitted line based on the largest duration and highest slope is selected for the fixed change point. If this best-fitted lines' time to reach threshold falls below the critical time then its slope and duration are saved as historic trends.

This above procedure is again repeated by changing the fixed change point to all the other change points. At the end of this whole procedure, we get for each change point, a best-fitted trend if it exists. Amongst the captured trends, maximum duration and the maximum slope of the trends are also calculated and saved. This training procedure can be conducted routinely, e.g., once per day or week. The method's pseudocode is shown in the algorithm's 1 *Train function.*

Online detection: In the Online Detection phase, for a new set of observations $\{x_k, x_k+1, x_k+2, ..., x_k+t-1x_k+t\}$ from time k to t where $t-k \geq P_{min}$ belonging to a VM after pre-processing is fed into the *Trend Lines Fitting* module. In *Trend Lines Fitting* module, the change points are detected. A sequence of observations $x_{t-L:t}$ between the last two change points starting from the end of the time series are selected and a line is fitted on them using the linear regression. The R-squared score, slope, duration and exit time to reach threshold of the fitted line is calculated. If its slope and duration are greater than the saved maximum counter parts then that window is marked anomalous. Otherwise, the values are compared against all the found training trends and if fitted-line's slope and duration are found to be greater than any of the saved trend then, again that window will be marked as anomalous. This procedure is further repeated by analyzing the observations between the last change point P_k and the previous next change point until all the change points are used. This is done for the cases where the new data has a similar trend as the historic data but now with a higher slope and longer duration. The algorithm's pseudo code showing the training and test method are shown in the algorithm 1.

Definition 2. *(Change Points) A set of time ticks which deviate highly from the normal pattern of the data. This is calculated by first taking the first-order difference of the input timeseries. Then, taking their absolute values and calculating their Z-scores. The indexes of observations whose Z-scores are greater than the defined threshold (3 times the standard deviation) represents the change points. The method's pseudocode is shown in the algorithm's 1 CPD function.*

4 Evaluation

We design experiments to answer the questions:

- **Q1. Memory Leak Detection Accuracy**: how accurate is Precog in the detection of memory leaks?
- **Q2. Scalability**: How does the algorithm scale with the increase in the data points?
- **Q3. Parameter Sensitivity**: How sensitive is the algorithm when the parameters values are changed?

We have used F1-Score (denoted as F1) to evaluate the performance of the algorithms. Evaluation tests have been executed on a machine with 4 physical cores (3.6 GHz Intel Core i7-4790 CPU) with hyperthreading enabled and 16 GB of RAM. These conditions are similar to a typical cloud VM. It is to be noted that the algorithm detects the cases where there is an ongoing memory leak and assumes that previously there was no memory leak. For our experiments, hyper-parameters are set as follows. The maximum threshold U is set to 100 and the defined critical time C is set to 7 days. The smoothing window size is 1 hour and re-sampling time resolution was set to 5 minutes. Lastly, the minimum R-squared score $R2_{min}$ for a line to be recognized as a good fit is set to 0.75. 65% of data was used for training and the rest for testing. However, we also show experiments on parameter sensitivity in this section.

4.1 Q1. Memory Leak Detection Accuracy

To demonstrate the effectiveness of the developed algorithm, we initially synthetically generated the timeseries. Table 2 shows the F1 score corresponding to each memory leak pattern and also the overall F1 score. Table 2 shows that Precog is able to reach an overall accuracy of 90%.

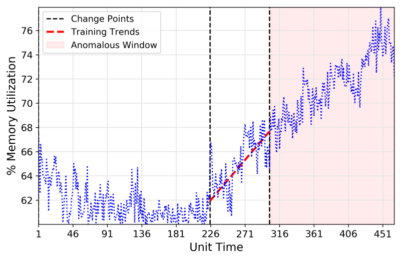
In addition, to demonstrate the effectiveness of the developed algorithm on the real cloud workloads, we evaluated Precog on the real Cloud dataset provided by Huawei Munich which consists of manually labeled memory leak data from 60 VMs spanned over 5 days and each time series consists of an observation every minute. Out of these 60 VMs, 20 VMs had a memory leak. Such high number of VMs having memory leaks is due to the fact that applications with memory leak were deliberately run on the infrastructure. The algorithm achieved the F1-Score of `0.857`, recall equals to `0.75` and precision as `1.0`. Average prediction time per test data containing approximately `500` points is `0.32` seconds.

Algorithm 1: Precog Algorithm

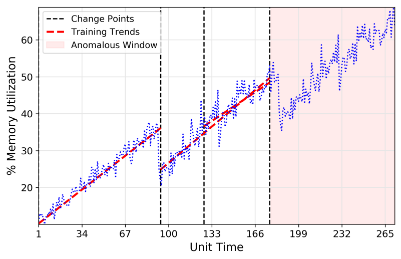
```
Input: input_Train_Ts,R2_score_min, input_Test_Ts, critical_time
Output: anomalous list a
1  Function CPD($x = input\_Ts, threshold = 3$):
2      $absDiffTs$ = first order absolute difference of $x$
3      $zScores$ = calculate z-scores of $absDiffTs$
4      $cpdIndexes$ = indexes of $(zScores > threshold)$
5      return $cpdIndexes$                 // return the change-points indexes
6  Function TRAINING($x = input\_Train\_Ts,\ R2\_score\_min, C = critical\_time$):
       // Train on input_Train_Ts
7      P = CPD(x)                          // get Change-points
8      p1 = 0
9      while $p1 <= length(P)$ do
10         p2 = p1
11         $D_b, S_b, T_b = 0$  // best local trend's duration, slope, exit time
12         while $p2 <= length(P)$ do
13             $exit\_time, r2, dur, slope \leftarrow$ LinearRegression$(ts)$     // fitted
                line's exit time, R2 score, duration, slope
14             if $r2 \geq R2\_score\_min$ and $dur \geq D_b$ and $slope \geq S_b$ then
15                 Update$(D_b, S_b, T_b)$          // update best local values
16             $p2 = p2 + 1$
17         if $T_b \leq C$ then
18             if $D_b \geq D_{max}$ and $S_b \geq S_{max}$ then
19                 Update$(D_{max}, S_{max})$       // update global trend values
20             saveTrend$(D_b, S_b)$, save$(D_{max}, S_{max})$       // save values
21         $p1 = p1 + 1$
22 Function TEST($x = input\_Test\_Ts, C = critical\_time$):
       // Test on the new data to find anomalous memory leak window
23     $a = [0]$            // anomalous empty array of size input_Test_Ts
24     P = CPD(x)                          // get Change-points
25     $len$ = length(P)                   // length of change point indexes
26     while $i \leq len$ do
27         $ts = x[P[len - i] : P[len]]$             // i is a loop variable
28         $exit\_time, r2, dur, slope$ = LinearRegression$(ts)$
29         $D_{max}, S_{max}, Trends$ = get saved values
30         if $exit\_time, \leq C$ and $r2 \geq R_{min}$ then
31             if $slope \geq S_{max}$ and $dur \geq D_{max}$ then
32                 $a[P[len - i] : P[len]] = 1$     // current trend greater than
                    global saved so mark anomalous
33             else
34                 For Each $t$ in $Trends$ if $slope \geq S_t$ and $dur \geq D_t$ then
35                     $a[P[len - i] : P[len]] = 1$       // current trend greater
                        than one of the saved trend so mark anomalous
36         $i = i + 1$
37     return $a$   // list with 0s and anomalous indexes represented by 1
```

Table 2: Synthetically generated timeseries corresponding to each memory leak pattern and their accuracy score.

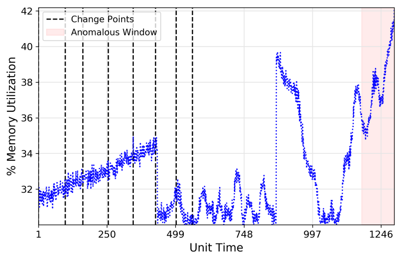
Memory Leak Pattern	**+ve cases**	**-ve cases**	**F1 Score**	**Recall**	**Precision**
Linearly Increasing	30	30	0.933	0.933	0.933
Linearly Increasing(with Noise)	30	30	0.895	1.0	0.810
Sawtooth	30	30	0.830	0.73	0.956
Overall	90	90	0.9	0.9	0.91



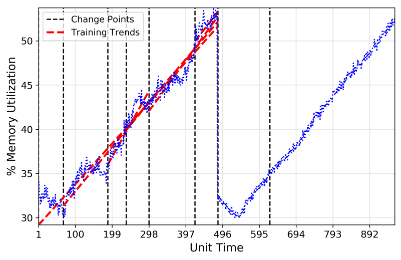
(a) Linearly increasing



(b) Sawtooth linearly increasing



(c) Linearly increasing without trends detected in training data



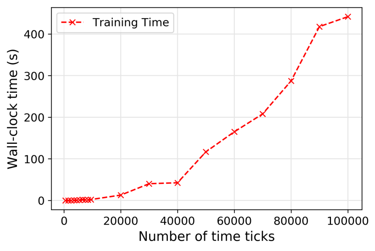
(d) Linearly increasing with similar trend as training data and correctly not detected

Fig. 3: Algorithm result on 3 difficult cases having memory leak (a-c) and one case not having a memory leak (d).

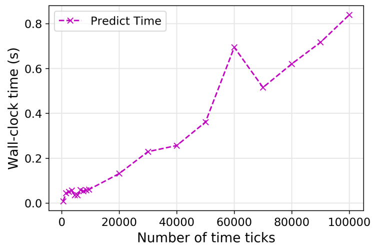
Furthermore, we present the detailed results of the algorithm on the selected 4 cases shown in the Figure 3 : simple linearly increasing memory utilization, sawtooth linearly increasing pattern, linearly increasing pattern with no trends detected in training data, and linearly increasing with similar trend as training data. The figure also shows the change points, training trends and the detected anomalous memory leak window for each of the cases.

For the first case shown in Fig. 3a, memory utilization is being used normally until it suddenly starts to increase linearly. The algorithm detected one training trend and reported the complete test set as anomalous. The test set trend is having similar slope as training trend but with a longer duration and higher memory usage hence it is reported as anomalous.

In the second case (Fig. 3b), the trend represents commonly memory leak sawtooth pattern where the memory utilization increases upto a certain point



(a) Training Time



(b) Prediction Time

Fig. 4: Precog's prediction method scale linearly.

and then decreases (but not completely zero) and then again it start to increase in the similar manner. The algorithm detected three training trends and reported most of the test set as anomalous. The test set follows a similar trend as captured during the training but with the higher memory utilization, hence it is reported.

In the third case (Fig. 3c), no appropriate training trend was detected in the complete training data but, the algorithm is able to detect an increasing memory utilization trend in the test dataset.

In Fig. 3d, the VM does not have a memory leak but its memory utilization was steadily increasing which if observed without the historic data seems to be a memory leak pattern. However, in the historic data, the same trend is already observed and therefore it is a normal memory utilization pattern. Precog using the historic data for detecting the training trends and then comparing them with the test data correctly reports that trend as normal and hence does not flag the window as anomalous. It is also to be noted that, if the new data's maximum goes beyond the maximum in the training data with the similar trend then it will be regarded as a memory leak.

4.2 Q2. Scalability

Next, we verify that our prediction method scale linearly. We repeatedly duplicate our dataset in time ticks, add Gaussian noise. Figure 4b shows that Precog' predict method scale linearly in time ticks. Precog does provide the prediction results under 1 second for the data with 100,000 time ticks. However, the training method shown in Figure 4a is quadratic in nature but training needs to conducted once a week or a month and it can be done offline as well.

4.3 Q3. Parameter Sensitivity

Precog requires tuning of certain hyper-parameters like R2 score, and critical time, which currently are set manually based on the experts knowledge. Figure 5 compares performance for different parameter values, on synthetically generated dataset. Our algorithm perform consistently well across values. Setting minimum R2 score above 0.8 corresponds to stricter fitting of the line and that is why the

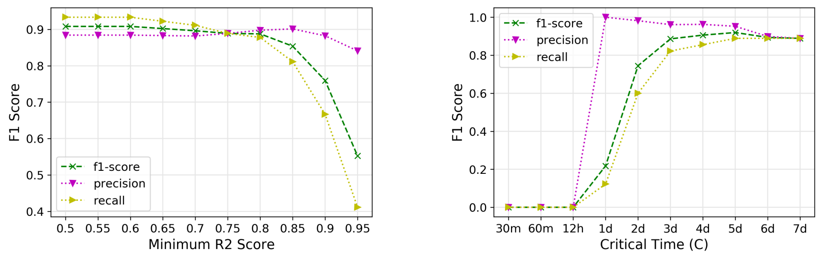


Fig. 5: Insensitive to parameters: Precog performs consistently across parameter values.

accuracy drops. On the other hand, our data mostly contains trend lines which would reach threshold withing 3 to 4 days, therefore setting minimum critical time too less (less than 3 days) would mean the trend line never reaching threshold within the time frame and hence decreasing the accuracy. These experiments shows that these parameters does play a role in the overall accuracy of the algorithm but at most of the values algorithm is insensitive to them. Furthermore, to determine these automatically based on the historic data is under progress and is out of the scope of this paper.

5 Conclusion

Memory leak detection has been a research topic for more than a decade. Many approaches have been proposed to detect memory leaks, with most of them looking at the internals of the application or the object's allocation and deallocation. The Precog algorithm for memory leak detection presented in the current work is most relevant for the cloud-based infrastructure where cloud administrator does not have access to the source code or know about the internals of the deployed applications. The performance evaluation results showed that the Precog is able to achieve a F1-Score of 0.85 with less than half a second prediction time on the real workloads. This algorithm can also be useful in the Serverless Computing where if a function is leaking a memory then its successive function invocations will add on to that and resulting in a bigger memory leak on the underneath system. Precog running on the underneath system can detect such a case.

Prospective directions of future work include developing online learning-based approaches for detection and as well using other metrics like CPU, network and storage utilization for further enhancing the accuracy of the algorithms and providing higher confidence in the detection results.

ACKNOWLEDGEMENTS

This work was supported by the funding of the German Federal Ministry of Education and Research (BMBF) in the scope of the Software Campus program. The authors also thank the anonymous reviewers whose comments helped in improving this paper.

References

1. Ataallah, S.M.A., Nassar, S.M., Hemayed, E.E.: Fault tolerance in cloud computing - survey. In: 2015 11th International Computer Engineering Conference (ICENCO). pp. 241–245 (Dec 2015). https://doi.org/10.1109/ICENCO.2015.7416355
2. Chen, K., Chen, J.: Aspect-based instrumentation for locating memory leaks in java programs. In: 31st Annual International Computer Software and Applications Conference (COMPSAC 2007). vol. 2, pp. 23–28 (July 2007). https://doi.org/10.1109/COMPSAC.2007.79
3. Clause, J., Orso, A.: Leakpoint: pinpointing the causes of memory leaks. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering. vol. 1, pp. 515–524 (May 2010). https://doi.org/10.1145/1806799.1806874
4. Gokhroo, M.K., Govil, M.C., Pilli, E.S.: Detecting and mitigating faults in cloud computing environment. In: 2017 3rd International Conference on Computational Intelligence Communication Technology (CICT). pp. 1–9 (Feb 2017). https://doi.org/10.1109/CIACT.2017.7977362
5. Jain, N., Choudhary, S.: Overview of virtualization in cloud computing. In: 2016 Symposium on Colossal Data Analysis and Networking (CDAN). pp. 1–4 (March 2016). https://doi.org/10.1109/CDAN.2016.7570950
6. Jump, M., McKinley, K.S.: Cork: Dynamic memory leak detection for garbage-collected languages. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 31–38. POPL '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1190216.1190224, http://doi.acm.org/10.1145/1190216.1190224
7. Mitchell, N., Sevitsky, G.: Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In: Cardelli, L. (ed.) ECOOP 2003 – Object-Oriented Programming. pp. 351–377. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
8. Pooja, Pandey, A.: Impact of memory intensive applications on performance of cloud virtual machine. In: 2014 Recent Advances in Engineering and Computational Sciences (RAECS). pp. 1–6 (March 2014). https://doi.org/10.1109/RAECS.2014.6799629
9. Rudafshani, M., Ward, P.A.S.: Leakspot: Detection and diagnosis of memory leaks in javascript applications. Softw. Pract. Exper. **47**(1), 97–123 (Jan 2017). https://doi.org/10.1002/spe.2406, https://doi.org/10.1002/spe.2406
10. Sor, V., Srirama, S.N.: A statistical approach for identifying memory leaks in cloud applications. In: CLOSER (2011)
11. Sor, V., Srirama, S.N.: Memory leak detection in java: Taxonomy and classification of approaches. Journal of Systems and Software **96**, 139–151 (2014)
12. Sor, V., Srirama, S.N., Salnikov-Tarnovski, N.: Memory leak detection in plumbr. Softw., Pract. Exper. **45**, 1307–1330 (2015)
13. Vilk, J., Berger, E.D.: Bleak: Automatically debugging memory leaks in web applications. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 15–29. PLDI 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3192366.3192376, http://doi.acm.org/10.1145/3192366.3192376
14. Xie, Y., Aiken, A.: Context- and path-sensitive memory leak detection. SIGSOFT Softw. Eng. Notes **30**(5), 115–125 (Sep 2005). https://doi.org/10.1145/1095430.1081728, http://doi.acm.org/10.1145/1095430.1081728