Detection of Memory Leaks in C/C++ Code via Machine Learning

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

Felix Eichler
Institute of Computer Science
Heidelberg University, Germany
felixgeorg@eichler-pfalz.de

Mohammadreza Ghanavati
Institute of Computer Science
Heidelberg University, Germany
mohammadreza.ghanavati@
informatik.uni-heidelberg.de

***Abstract*—Memory leaks are one of the primary causes of software aging. Despite of recent countermeasures in C/C++ such as smart pointers, leak-related defects remain a troublesome issue in C/C++ code, especially in legacy applications.**
We propose an approach for automatic detection of memory leaks in C/C++ programs based on characterizing memory allocation sites via the age distribution of the non-disposed memory chunks allocated by such a site (the so-called *GenCount*-technique introduced for Java by Vladimir Šor). We instrument malloc and free calls in C/C++ and collect for each allocation site data on the number of allocated memory fragments, their lifetimes, and sizes. Based on this data we compute feature vectors and train a machine learning classifier to differentiate between leaky and defect-free allocation sites.
Our evaluation uses applications from SPEC CPU2006 suite with injected memory leaks resembling real leaks. The results show that even out-of-the-box classification algorithms can achieve high accuracy, with precision and recall values of 0.93 and 0.88, respectively.

***Keywords*—Automated debugging, Memory leaks, Machine learning, C/C++**

I. Introduction

In long-running applications, even non-critical errors can accumulate and negatively influence the performance or functionality. This phenomenon is called *software aging* [34], [16], [13], [3]. One of the primary reasons for software aging is leaking memory. A *memory leak* is a memory fragment or object which is not freed, even if it is never accessed again [18], [27]. Memory leaks occur when dynamically allocated memory or objects become either unreachable and thus cannot be disposed, or when they are still reachable but are not accessed any more (i.e. become obsolete).

The first case can be handled by garbage collection and thus is no longer an issue in modern programming languages and execution environments like JVM or .NET. However, leaks due to non-reachable objects are still possible in C/C++, despite of smart pointers or data structures with automatic memory management, like in STL. Moreover, detecting leaks of the second type is still a considerable challenge. They occur when references to a memory fragment or an object still exist, possibly stored in a global context, but the object is no longer needed. This phenomenon is also called *dead memory* [19]. One of the key detection difficulties is to determine whether an object will be used in the remaining program execution, or not.

For this reason, detection methods tend to focus on the effects of memory leaks, such as growing memory consumption caused by parts of the code (*growth analysis* [30]). However, also such methods have drawbacks. Among several reasons, it can take a long time for the memory leaks to materialize. Occurrence of a memory leak can also dependent on the specific conditions of the production environment. Consequently, memory leaks might not show up during regular software tests, making them an elusive and serious threat for long running applications.

Another aspect of memory leaks concerns their severeness. A single or few small leaking objects will likely not create a noticeable impact on program performance and can therefore be ignored. This is especially true when the program terminates before more objects accumulate. Therefore, effective approaches for memory leak detection should focus on scenarios where many and especially many large objects are continuously leaking.

Methods monitoring memory usage at runtime and performing statistical analysis to identify both types of leaks have been shown to produce good results, with a reasonable performance overhead. In particular, Šor [30] has presented in his PhD thesis an approach for memory leak detection in Java using growth analysis based on the *GenCount* concept. This method can handle both types of leaks (i.e. reachable or not reachable) and focuses on severe leaks, i.e. cases where memory leaks are created repeatedly. The main idea of the *GenCount* metric is to identify code locations which repeatedly create new objects, but do not dispose previously created objects. This method is further enhanced in [31], [32] by characterizing each code location creating objects (so-called *allocation sites*) by a vector of features derived from runtime monitoring (with the *GenCount* metric as one of the features).

In this paper we transfer the above approaches [30], [31] (developed and evaluated for Java) to C/C++ and show their feasibility and precision via evaluation on 14 programs from the SPEC CPU2006 suite. We also extend the original machine learning setup used for Java by introducing additional metrics and features. To get the necessary data we have to intercept calls of functions allocating or deallocating dynamic memory. For the method to be usable in production environments - where real memory leaks are most likely to be encountered - the procedure of gathering data has to limit the time and memory overheads it inflicts on the subject program. Consequently, the memory overhead of our method is low (typically only few

MBytes) and can be parametrized. The runtime overhead is below 5% for programs with at most 100,000 allocation/deallocation events per second and increases to around 20% if this rate surpasses one million. This renders our approach and even its prototypical implementation as practically feasible in production environments.

This paper is structured as follows. Our approach for memory leak detection is described in Section II. In Section III we evaluate the approach in terms of the performance overhead and the accuracy of detecting memory leaks. Section IV discusses related work, and Section V summarize the results and give an outlook of future developments.

II. Approach Description

We define an *allocation site* as a location in the source code where a memory or object allocation is performed. We call an allocation site *leaky*, if it leaks memory, i.e., memory fragments or objects which are not disposed (freed) after use.

Overview. Our approach has the following workflow. We collect metrics concerning memory/object allocations and deallocations at runtime for each allocation site. This data is processed off-line to compute features for the classification task. For the training phase, data is collected from applications with artificially injected leaks. With this data we train decision trees, where features are derived from measurements and labels state whether an allocation site is leaky, or not. In the evaluation phase (under cross-validation), we use the trained classifiers to predict whether an allocation site is leaky, and compare the predictions for out-of-sample data against true label values.

A. GenCount-Metric for Growth Analysis in C/C++

One of the key concepts used in our approach is the *GenCount*-metric for memory leak detection based on growth analysis. It has been introduced by Šor in [30] for Java.

Let $GC = (gc_1, gc_2, ..., gc_n)$ denote the set of consecutive garbage collections of an application and A be the set of allocation sites. If an object is created before gc_1 it is said to belong to *generation* 0. If it is created between gc_i and gc_{i+1} it belongs to *generation* i. After any $gc_j \in GC$, for each allocation site $a \in A$, we define a function $G : (a, gc_j) \rightarrow \{0, 1\}^j$, such that for $i \in 0, ..., j$:

$$G(a, gc_j)_i = \begin{cases} 1 & \text{if after } gc_j \text{ there exists a live object of} \\ & \text{allocation site } a \text{ created in generation } i \\ 0 & \text{else.} \end{cases}$$

For integer j and an allocation site a the *GenCount* is then defined as:

$$GenCount(a, j) = \sum_{i=0}^{j} G(a, gc_j)_i$$

In other words, $GenCount(a, j)$ tells us in how many previous generations site a has created objects which are still not disposed in the current generation j. It is expected that for a leaky allocation site a, $GenCount$ grows in an unbounded manner with growing j, as older objects are not disposed. Contrary to this, for a non-leaky site $GenCount$ is expected to remain bounded.

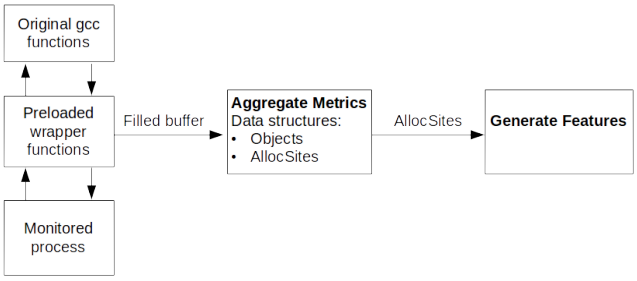

Figure 1. Overview of the data collection and processing.

The *GenCount* was first utilized in a threshold approach, which reports a leak if the *GenCount* of a subset of allocation sites is significantly higher then for others in the same application. An improved approach [31] uses *GenCount* along with other metrics/features (calculated for each allocation site) to train classification models predicting whether an allocation site is leaky, or not.

We embrace and extend the latter method, while porting it from Java to C/C++. To this aim we use features described in Section II-C. Since there are no "generations" in C/C++, we replace them by *epochs*, i.e., time intervals of equal length $interval$ (a parameter which can be adjusted for each application). The epochs are enumerated in ascending order, and the index of the epoch at a given time is defined as:

$$currentEpoch = \lfloor (currentTime - startTime)/interval \rfloor.$$

Thus, the age of a C/C++ object can be expressed in the number of epochs since its creation. In this work, we use in all definitions and concepts the term *epoch* as a replacement for the term *generation* from Java, e.g. our *epochCount* is related to *GenCount* from [30].

B. Data Collection

Figure 1 illustrates the data collection and its processing in our approach. Wrapper functions are used to intercept event data when an allocation or a deallocation function is called. In order to restrict the overhead on the monitored process, the raw data is stored in a buffer of constant size. When the buffer is filled or the process is about to terminate, the buffer content is transfered via a pipe to a separate process, which maintains data structures for tracking existing objects and aggregating the allocation site metrics. A second mode of operation is offered, where the buffers are written to files and then read by the process aggregating the metrics.

We implemented wrapper functions for *malloc*, *calloc*, *realloc* and *free* in a shared library. The wrappers are preloaded using the LD_PRELOAD environment variable [1], which causes the program to use our wrapper functions instead of the original dynamically linked functions.

The wrapper calls the original function with the given parameters and returns the resulting pointer in case of an allocation. For each allocation or deallocation we collect information in a struct called *memEvent* shown in Figure 2 (it requires 32 bytes). A separate process receives and processes the *memEvents* sent from the wrapper library.

```
struct memEvent {
        char                    type;
        unsigned short int      epoch;
        void*                   ptr;
        void*                   allocSite;
        size_t                  size;
};
```

Figure 2. The *memEvent* struct.

C. Metric and Feature Computation

A list of *memEvent* events is used to compute multiple "low level" aggregation metrics per allocation site and epoch (namely *numAlloc*, *numFree*, *sizeAlloc*, *sizeFree*, and *lifetimeFree*). We compute the aggregation metrics incrementally (enabling an online operation of our tool) via an associative array (called *AllocSites*) with allocation sites as keys. The value of *AllocSites* (per allocation site) is another associative array with epochs as keys and a corresponding set of metrics as values.

The aggregation metrics are in turn utilized to calculate the final features (a feature vector per allocation site). The set of features contains all but two features utilized in Java [31] and five features contributed by us. The following two original features are excluded: *leakingAllocationRatio* since it was not used by any decision tree, and *uptime* as it became obsolete due to normalization. All features except for $existingPerEpoch$ and $epochCountGap$ are normalized to values between 0.0 and 1.0 in order to make make features compatible for among the studied SPEC CPU2016 programs.

The features introduced in this work are:

avgLifetime. The average lifetime of objects per application. The *lifetime* of an object is defined (for disposed objects) as number of epochs from allocation to deallocation, or (for live objects) as number of epochs from allocation to the last epoch e_{max}. The feature is normalized via dividing by e_{max}.

existingCountLocalRatio. This feature represents the ratio of the number of existing objects to the number of allocated objects for a given allocation site.

existingSizeLocalRatio. The equivalent to *existingCountLocalRatio*, but using sizes instead of counts.

existingCountGlobalRatio. This feature compares the number of existing objects allocated by this allocation site to the total number of existing objects.

existingSizeGlobalRatio. Same as *existingCountGlobalRatio*, but using sizes instead of counts.

The following features were adapted from the Java approach:

epochCount. The number of previous epochs in which allocation site has created objects still not disposed in the current epoch. Hereafter we call epochs with such property *live epochs*. This feature is normalized via dividing by the total number of epochs (i.e. $e_{max}+1$). It corresponds to *GenCount* in the Java approach.

epochCountGap. The ratio of *epochCount* to the highest lower *epochCount* among all the other allocation sites of an application. This represents the gap between two neighboring allocation sites in the ordered sequence of *epochCounts*.

epochDistUniformity. The standard deviation of the set of distances between indices of neighboring live epochs, normalized by the number of epochs. A lower standard deviation implies that leaking object are created more regularly.

existingPerEpoch. The average number of objects in live epochs. It is calculated by dividing the total number of existing objects of the allocation site by *epochCount*.

D. Leak Detection and Discussion of Alternatives

The features calculated from the metrics can be analyzed in different ways to detect leaks. The approach used in this work is statistical classification. To train a classifier, data is needed from programs containing known leaks, from which labeled features can be derived. In order to be usable in practice, a classifier has to prove to be dependable enough at detecting leaks without causing false alarms and to be transferable to multiple programs which were not part of the training.

Apart from this, the features could be clustered to identify different behaviors of allocation sites, indicating defective or error-free operation. An additional method for correctly classifying obvious non-leaking allocation sites can be achieved by identifying allocation sites where all objects have been freed, i.e. $epochCount = 0$. This can be used as a pruning step before training a model or performing the actual classification.

It can also be noted that we contributed rather simple features, which only use summations over epochs and do not make use of the metrics on the epoch level. More complex features such as value distributions could be calculated, potentially improving the accuracy. On the other side, if the classification performance is sufficient, some metrics could be simplified by direct summation on allocation level, which would simplify the data collection procedure.

III. Evaluation

In this section, we evaluate our approach. We use synthetic leak to evaluate the accuracy of leak detection. We also measure the overhead of our tool on the SPEC CPU2006 benchmark suite. To conduct the evaluation of our approach, we try to answer following research questions:

- RQ1. How accurate is the classification used by our approach?
- RQ2. How can feature visualization improve the accuracy of leak detection?
- RQ3. How large is the overhead of our approach?

A. Accuracy of Leak Detection

In this section, we answer RQ1. The goal of our approach is to train a classifier on labeled data of programs containing known leaks and leveraging it to classify unlabeled data from other programs to detect any undiscovered leaks. This means that we need labeled features collected from multiple programs, i.e., we need to know for each allocation site if it is leaky or not. For this purpose, we apply a decision tree classifier on one subset of the labeled data and test it by

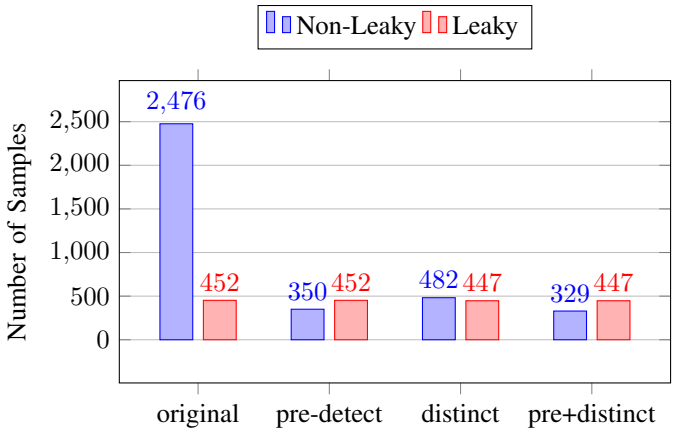

Figure 3. Number of samples before (*original*) and after removing obvious non-leaks (*pre-detect*) or duplicates (*distinct*). The *pre-detect* (or simply *pre*) set was used for all evaluation tasks.

classifying the samples of the other subset, comparing the predicted labels to the actual ones to determine the accuracy of our approach. Here we describe the setup for our experiments, followed by the result on classification accuracy of our leak detection approach.

1) Experiment Setup and Data Collection: We generate the labeled data by injecting synthetic leaks into the data collected from programs in SPEC CPU2006. Then we perform a randomized 10-fold cross-validation, where the training and the test sets consist of randomly selected of 9/10th and 1/10th of all samples, respectively. The procedure is repeated ten times. Each sample becomes a part of a test dataset once. The cross-validation is stratified, which means that the relative amount of leaky and non-leaky allocation sites are approximately equal in both training and test sets.

We use the Weka framework [10] for various data mining tasks, such as training and testing classifiers, performing randomized cross-validation and evaluating features. The J48 decision tree algorithm (the Weka implementation of C4.5 [35]) is used for the classification, because it produced good results in preliminary tests and also the resulting model was interpretable, as it provided information on which features are selected and how they are utilized.

The injection of synthetic leaks is done analogously to [19], where two types of leaks are generated: static and dynamic leaks. For static leaks, a single allocation site is selected associating to 10% of all allocations of the process, and all deallocations of objects originating from this allocation site are ignored. For dynamic leaks, 10% of all deallocations are ignored at random, thus creating a random amount of partially leaky allocation sites.

2) Evaluation Datasets: We collect data for an unmodified run of a program first and then inject the synthetic leaks by removing data of *free* function calls. Therefore we have three runs (unmodified, with static leaks and with dynamic leaks) for 14 benchmarks from SPEC CPU2006[1] with their corresponding feature sets. We assume that the original, unmodified programs have no memory leaks.

[1]We select the same programs as in [19]. However, exclude *bzip2* and *namd* due to very few allocations or deallocation sites. Also *perlbench* is excluded due to compiler errors which could not be resolved.

Table I. CONFUSION MATRIX FOR RANDOMIZED CROSS-VALIDATION

Actual / Predicted	Predicted Leak	Predicted Non-Leak
Actual Leak	399	53
Actual Non-Leak	28	322

Table II. ACCURACY METRICS CALCULATED FROM TABLE I.

Accuracy	**Precision**	**Recall**	**F-Measure**
0.90	0.93	0.88	0.91

Our final future set contains one feature vector for each allocation site of each run of a program, with the label being either *leaky*, if at least one object originating from the allocation site is leaky, or *non-leaky*, if no object of the allocation site is leaked. As all programs have a runtime of only a few minutes, an epoch interval of one second is selected in order to provide enough details.

Two preprocessing steps can be used to prepare the data before training the classifier. First, we can filter out all samples with obvious non-leaks ($epochCount = 0$) in order to prevent obvious *true negatives* from distorting the accuracy metric. Secondly, we can remove all duplicates to get a set of distinct samples[2].

The resulting amounts of leaky and non-leaky samples before and after preprocessing are shown in Figure 3. With each preprocessing method the number of non-leaky sites is greatly reduced, with the side-effect of producing nearly equal set size for both leaky and non-leaky labels. For all evaluation tasks we used the dataset without the obvious non-leaks.

3) Randomized Cross-Validation: The resulting confusion matrices and evaluation metrics for randomized cross-validation are shown in Tables I and II. The results show a relatively high value for accuracy metrics for the classifier used in our approach.

B. Feature Visualization

In this section, we answer RQ2. A first step for analyzing the dataset is to visualize the feature values and look for patterns. We use scatter plots to visualize the relationship between two features in connection with the label. Three combinations with noticeable patterns are shown in Figure 4. In each plot the $existingSizeLocalRatio$ on the y-axis is plotted against another feature on the x-axis.

Comparing $epochCount$ shows a clear line of leaky sites at about $y = 0.1$ along the range of x. This shows the 10% of dynamic leaks per program, which in most cases caused 10% of an allocation site to be leaky.

A line of static leaks can be seen at $y = 1$ where none of the allocated objects are deallocated. The non-leaking sites are mostly in the y range below 10%, but there is also a spot of non-leaking sites where all objects still exist ($y = 1$) and which are allocated in a small number of epochs. These are

[2]Apart from duplicate feature vectors made up of 0 and 1 values, duplicates can happen for multiple other reasons: For example when a program run contains multiple allocation sites showing the same behavior or when the features of an allocation site are not affected by the leak injection.

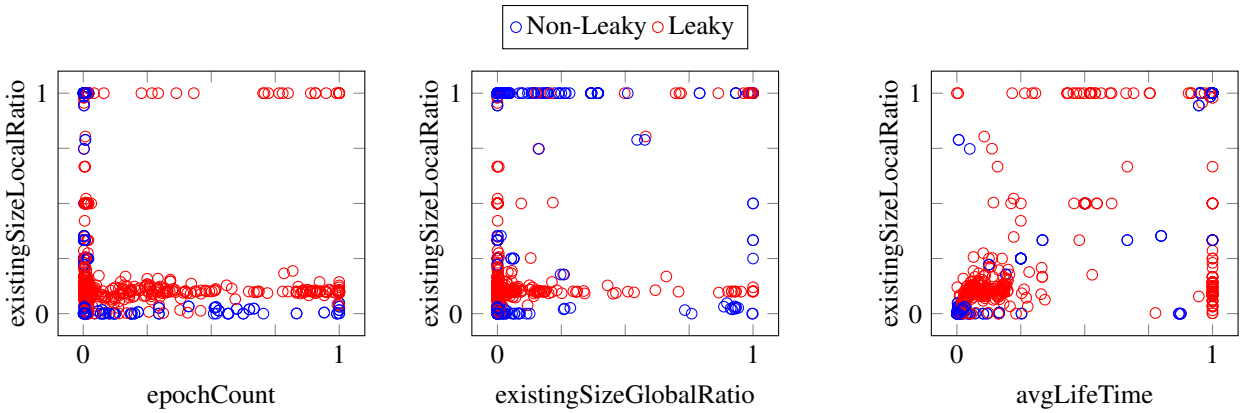

Figure 4. Scatter plots of feature combinations for all samples of *pre-detect* dataset.

the objects which are allocated at program start and are needed during the entire execution.

Plotting the global and local size ratios against each other shows that most of the allocation sites with dynamic leaks share less than 25% of the total existing size, which makes sense as the leaking objects are shared over multiple allocation sites. The static leaks are concentrated in the upper right corner, which shows that the objects of a single allocation site where all objects are leaking largely outweigh all other existing objects.

Lastly, the comparison with $avgLifeTime$ shows an area of leaks with a slightly higher size and lifetime than the non-leaks concentrating near the origin. The sites with dynamic leaks are mostly split between those with average lifetime below 0.25, where objects seem to be created over the whole runtime, and those with average life time at about 1, where the objects were created at the start of the program. The static leaks show varying average lifetimes at the top of the size range. The concentration of non-leaking sites which has been at $(0, 1)$ in the first plot is now at $(1, 1)$, so these objects are in fact allocated at program start and existed over the whole runtime.

C. Performance Overhead

In this section, we answer RQ3. An experiment is carried out to determine the effects of collecting data on the runtime of subject programs. The SPEC CPU2006 benchmark suite was run three times with and without our shared library being preloaded. We select the same benchmarks as [19], but without considering *perlbench* because of compile errors. The system under test has a 2.5 GHz Intel Core i5-3210M CPU and 8 GB of memory. The runtime overhead represents the relative amount of additional time needed for the modified program to terminate in comparison to the original version. It is calculated as follows:

$$RuntimeOverhead = \frac{time_{modified} - time_{original}}{time_{original}}$$

To calculate the overhead for each benchmark, the average of three execution times is taken for two versions: One piping the data to a separate process and one writing it to files. The results are shown in Figure 5. In most cases the overhead stays below 10% for the piped version and below 5% for the persistent version. Only three benchmarks shows an overhead over 10% for both versions, the maximum being *omnetpp* with 21%. Some programs produces negative overheads which indicates that normally occurring fluctuations outweigh the effects of our modification.

The memory overhead for the subject process is fixed, as we do not allocate dynamic memory. We select a buffer size of 100,000 elements, each having a size of 32 Bytes (one *memEvent*), which means that the memory overhead produced by the buffer is fixed at 3.2 MB.

As the runtime overhead shows, writing to disk is in most cases faster than piping the data. The procedure could therefore be improved by passing the data via disk to the aggregating process, which deletes it after it has been processed. Maintaining live aggregations and discarding the raw data shows a clear benefit when looking at *omnetpp*: With about 17 GB it produces the most raw data, which is compressed to about 20 KB for the aggregations.

Further improvements would be necessary to reduce the highest overheads, which comes from a rising number of allocation/deallocation events per second: The number stays below 100,000 for benchmarks left of *gcc* in Figure 5 and rises up to about one million for *omnetpp*.

We searched for the bottleneck using this benchmark. Results are depicted in Figure 6. The cause of 16 percentage points of the 22% overhead is due to writing the filled buffers to files. In comparison, fetching the system time to determine the current epoch causes 2 percentage points of overhead. The procedure could be parallelized to mitigate this problem, so the subject program can continue while a separate thread handles the buffer. This would reduce the time overhead but increase the memory overhead, as the unchanged writing speed means more data has to be buffered. Therefore it would still be necessary to find a faster way to transfer the data or to reduce its amount.

IV. Related Work

The problem of memory leak is recognized as an important research problem and is investigated by many groups from

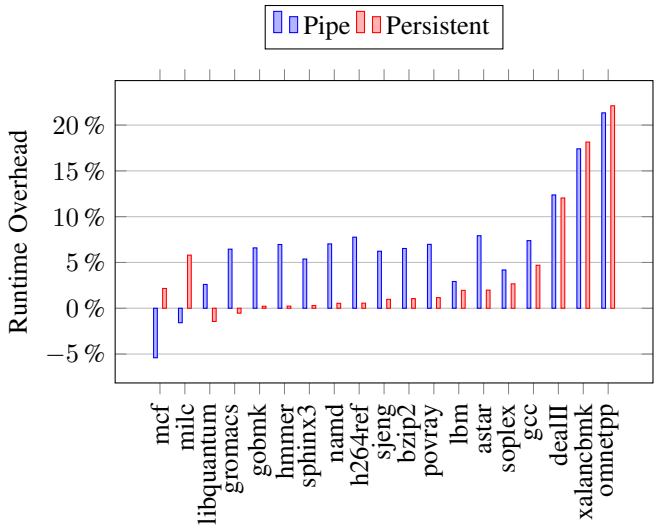

Figure 5. Runtime overhead for SPEC CPU2006 benchmarks, passing the raw data through a pipe or writing it to persistent files.

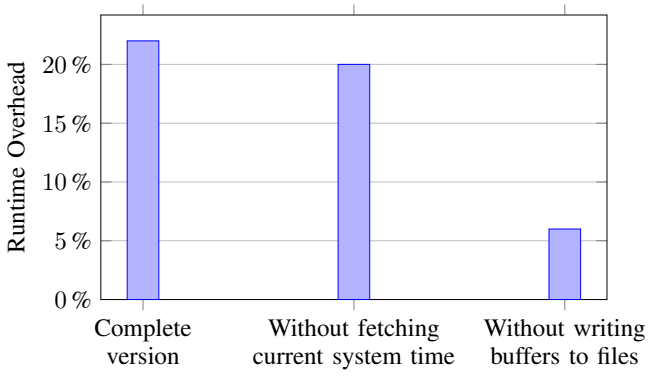

Figure 6. Runtime overheads caused by data collection for *omnetpp*: The full version, a version without requesting the current system time for every *memEvent* and a version without writing the buffer to files.

academia and industry. We classify the existing works into four broad areas.

Software rejuvenation. Memory leaks as one important type of software aging defects impose high negative impact on running application. Software rejuvenation is a technique to control such kind of effects. This approach introduced in work [16], tries to control the effects of aging defects via scheduled restarts. Other research here include case studies [22], [37], [4], modeling of performance degradation [3] and limiting the application downtime due to the scheduled restarts [6], [36], [2].

Static analysis. Techniques include e.g., reachability analysis via a guarded value flow graph [8], backward dataflow analysis [33], or detecting violations of constraints on object ownership [15]. However because of using static analysis, they introduce many false positives.

Dynamic analysis. The major lines of approaches include *staleness detection*, *growth analysis*, *analysis of captured state*, and *regression testing*. Our work is related to *growth analysis* and *analysis of captured state*.

Staleness detection. Staleness (lack of recent read/write accesses) is the most distinguishing property of memory leak. It has been purposed in work [14]. The key challenge is the overhead of monitoring object accesses. Multiple works try to overcome this problem: path-biased sampling [14], page-level sampling [29], modifications of the JVM [5], or focusing only on container accesses [39]. A recent work Sniper [19] is able to reduce the total runtime overhead to less than 3% by exploiting hardware units of modern CPUs.

Growth analysis. Several works compare heap usage between different versions of application [17], [7], [31], [30]. Šor [31] proposes the *genCount* metric which exploits the age distribution of memory fragments allocated by a single allocation site to detect potential leaks. This approach is subsequently improved in [30] by applying machine learning for higher detection accuracy with a runtime overhead of about 40%. Growth analysis on the level of aggregated memory metrics has been also exploited to detect software aging [23], [24], [25].

Analysis of captured state. This direction is followed in previous works [28], [38], [9]. The Valgrind tool Omega [26] uses similar approach to detect memory leaks, however with a high overhead.

Regression testing. Comparing behavior between evolving code versions is the key idea of regression testing. Many techniques use this concept for debugging of crashing (i.e., non-latent) errors [12]. Our previous works utilize version comparison for memory leak detection or isolation. Works [20] and [24] use cumulative memory consumption metrics (such as Heap Usage or RSS) for detecting the introduction of memory leaks in newer version of the application. Works [21], [11] combine the version comparison approach with integration and unit testing to isolate and detect memory leaks.

V. Conclusion and Future Work

We presented an approach for memory leak detection in C/C++ using growth analysis, specifically the *GenCount* concept proposed by Šor for Java [30]. We extended it beyond the original approach by introducing additional metrics and features.

The memory overhead of our method is low (typically only few MBytes) and can parametrized by setting the size of an event buffer. The runtime overhead is below 5% for programs with at most 100,000 allocation/deallocation events per second and increases to around 20% if this rate surpasses one million. To evaluate our approach, we injected synthetic leaks into multiple SPEC CPU2006 applications and collected data from defective and error-free allocation sites. We then trained decision trees and evaluated the classification accuracy using cross-validation, yielding f-measure values of up to 0.9.

Our future work will include lowering the instrumentation overhead by sampling and by online aggregation of the events. We will also attempt to improve the accuracy by adding more features and using Gradient Boosting Machines and Deep Neural Networks as classification algorithms.

References

[1] Ld.so(8) - linux programmer's manual, 2015. (Cited from: II-B)

[2] J. Alonso, L. M. Silva, A. Andrzejak, and J. Torres. High-available grid services through the use of virtualized clustering. In *IEEE-GRID*, Austin, USA, September 2007. (Cited from: IV)

[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. (Cited from: I and IV)

[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*, pages 1411–1416, 2011. (Cited from: IV)

[5] M. D. Bond and K. S. McKinley. Leak Pruning. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–288, 2009. (Cited from: IV)

[6] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HotOS*, pages 125–130. IEEE Computer Society, 2001. (Cited from: IV)

[7] K. Chen and J.-B. Chen. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *IEEE International Conference on Computers, Software & Applications (COMPSAC)*, pages 23–28, 2007. (Cited from: IV)

[8] S. Cherem, L. Princehouse, and R. Rugina. Practical Memory Leak Detection Using Guarded Value-flow Analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 480–491, 2007. (Cited from: IV)

[9] J. Clause and A. Orso. LEAKPOINT: Pinpointing the Causes of Memory Leaks. In *International Conference on Software Engineering (ICSE)*, pages 515–524, 2010. (Cited from: IV)

[10] E. Frank, M. A. Hall, and I. H. Witten. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, 4th edition, 2016. (Cited from: III-A1)

[11] M. Ghanavati and A. Andrzejak. Automated memory leak diagnosis by regression testing. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM*, pages 191–200. IEEE, 2015. (Cited from: IV)

[12] M. Ghanavati, A. Andrzejak, and Z. Dong. Scalable isolation of failure-inducing changes via version comparison. In *IEEE International Workshop on Program Debugging (IWPD) at ISSRE*, pages 150–156, 2013. (Cited from: IV)

[13] M. Grottke, R. Matias, and K. S. Trivedi. The fundamentals of software aging. In *2008 IEEE International Conference on Software Reliability Engineering Workshops (ISSRE Wksp)*, pages 1–6, Nov. 2008. (Cited from: I)

[14] M. Hauswirth and T. M. Chilimbi. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 156–164, 2004. (Cited from: IV)

[15] D. L. Heine and M. S. Lam. Static Detection of Leaks in Polymorphic Containers. In *International Conference on Software Engineering (ICSE)*, pages 252–261, 2006. (Cited from: IV)

[16] 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. (Cited from: I and IV)

[17] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 31–38, 2007. (Cited from: IV)

[18] C. Jung, S. Lee, E. Raman, and S. Pande. Automated Memory Leak Detection for Production Use. In *International Conference on Software Engineering (ICSE)*, pages 825–836, 2014. (Cited from: I)

[19] C. Jung, S. Lee, E. Raman, and S. Pande. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering*, 2014. (Cited from: I, III-A1, 1, III-C, and IV)

[20] F. Langner and A. Andrzejak. Detecting software aging in a cloud computing framework by comparing development versions. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 896–899, 2013. (Cited from: IV)

[21] F. Langner and A. Andrzejak. Detection and root cause analysis of memory-related software aging defects by automated tests. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 365–369, 2013. (Cited from: IV)

[22] L. Li, K. Vaidyanathan, and K. Trivedi. An approach for estimation of software aging in a web-server. In *ISESE'02*, pages 91–102, 2002. (Cited from: IV)

[23] F. Machida, A. Andrzejak, R. Matias, and E. Vicente. On the effectiveness of Mann-Kendall test for detection of software aging. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 269–274, Nov. 2013. (Cited from: IV)

[24] R. Matias, A. Andrzejak, F. Machida, D. Elias, and K. Trivedi. A systematic approach for low-latency and robust detection of software aging. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2014. (Cited from: IV)

[25] R. Matias, G. Sena, A. Andrzejak, and K. S. Trivedi. Software Aging Detection Based on Differential Analysis: An Experimental Study. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 71–77, Oct. 2016. (Cited from: IV)

[26] B. Meredith. Omega - An Instant Leak Detection Tool for Valgrind, 2008. Version 1.2. (Cited from: IV)

[27] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. *Lecture Notes in Computer Science*, 2743:351–377, 2003. (Cited from: I)

[28] N. Mitchell and G. Sevitsky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 351–377, 2003. (Cited from: IV)

[29] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and Precisely Locating Memory Leaks and Bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 397–407, 2009. (Cited from: IV)

[30] V. Šor. *Statistical approach for memory leak detection in Java applications*. PhD thesis, University of Tartu, 2014. http://dspace.ut.ee/bitstream/handle/10062/43817/shor_vladimir.pdf. (Cited from: I, II-A, IV, and V)

[31] V. Šor, P. Ou, T. Treier, and S. Srirama. Improving Statistical Approach for Memory Leak Detection Using Machine Learning. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 544–547, Sept. 2013. (Cited from: I, II-A, II-C, and IV)

[32] V. Šor, S. N. Srirama, and N. Salnikov-Tarnovski. Memory leak detection in Plumbr. *Software: Practice and Experience*, 45(10):1307–1330, Oct. 2015. (Cited from: I)

[33] M. Orlovich and R. Rugina. Memory Leak Analysis by Contradiction. In *International Static Analysis Symposium (SAS)*, pages 405–424, 2006. (Cited from: IV)

[34] D. L. Parnas. Software aging. In *Proceedings 16th International Conference on Software Engineering (ICSE '94)*, pages 279–287, may 1994. (Cited from: I)

[35] R. Quinlan. C4.5: Programs for machine learning. Morgan Kaufmann Publishers, San Mateo, CA., 1993. (Cited from: III-A1)

[36] 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. (Cited from: IV)

[37] L. M. Silva, H. Madeira, and J. G. Silva. Software aging and rejuvenation in a SOAP-based server. In *IEEE International Symposium on Network Computing and Applications (IEEE-NCA)*, pages 56–65, Los Alamitos, CA, USA, 2006. IEEE Computer Society. (Cited from: IV)

[38] G. Xu, M. D. Bond, F. Qin, and A. Rountev. LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–282, 2011. (Cited from: IV)

[39] G. Xu and A. Rountev. Precise Memory Leak Detection for Java Software Using Container Profiling. *ACM Transactions on Software Engineering and Methodology*, 22(3):17:1–17:28, July 2013. (Cited from: IV)