Automated Memory Leak Diagnosis by Regression Testing

Mohammadreza Ghanavati, Artur Andrzejak
Institute of Computer Science
Heidelberg University, Germany
{mohammadreza.ghanavati, artur.andrzejak}@informatik.uni-heidelberg.de

***Abstract*—Memory leaks are tedious to detect and require significant debugging effort to be reproduced and localized. In particular, many of such bugs escape classical testing processes used in software development. One of the reasons is that unit and integration tests run too short for leaks to manifest via memory bloat or degraded performance. Moreover, many of such defects are environment-sensitive and not triggered by a test suite. Consequently, leaks are frequently discovered in the production scenario, causing elevated costs.**

In this paper we propose an approach for automated diagnosis of memory leaks during the development phase. Our technique is based on regression testing and exploits existing test suites. The key idea is to compare object (de-)allocation statistics (collected during unit/integration test executions) between a previous and the current software version. By grouping these statistics according to object creation sites we can detect anomalies and pinpoint the potential root causes of memory leaks. Such diagnosis can be completed before a visible memory bloat occurs, and in time proportional to the execution of test suite.

We evaluate our approach using real leaks found in 7 Java applications. Results show that our approach has sufficient detection accuracy and is effective in isolating the leaky allocation site: true defect locations rank relatively high in the lists of suspicious code locations if the tests trigger the leak pattern. Our prototypical system imposes an acceptable instrumentation and execution overhead for practical memory leak detection even in large software projects.

***Index Terms*—Automated debugging, memory leak, regression testing, software tests, version comparison**

I. Introduction

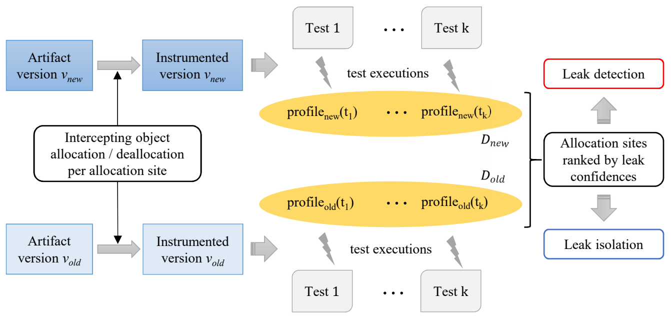
Memory leaks are most prominent type of memory management defects. They occur if objects remain in heap memory but are never accessed again. Also managed languages such as Java, C# or Python suffer from memory leaks. The reason for this is that garbage collectors of these languages over-approximate object liveness by its reachability [6]. Consequently, a reachable object is not disposed even if it will not be used again. The most common scenario for such bugs are forgotten references in collection data structures (e.g. lists or maps) [37].

Leaks are notoriously hard to detect, reproduce, and fix. One of the reasons is long latency between leak triggering and manifestation of visible symptoms such as memory bloat or performance degradation [13]. A further problem is their sensitivity to inputs and execution environments [6]. As a consequence, many of such defects escape in-house quality assurance measures including unit, integration, and even performance testing. If discovered in customer usage, they can have a significant economic impact. For example, a "latent memory leak bug" has caused a partial outage of Amazon's EC2 cloud service on 22 October 2012 [1], affecting operations of hundreds of EC2 customers.

A number of tools [11, 13, 24] and research techniques help developers to diagnose leaks. One strategy is to apply staleness analysis to identify "dead" objects - those which can not be accessed for a long time [6, 14, 17, 28, 37]. Another group of works is based on analyzing heap growth [7, 16, 33, 34], or analysis of captured state [9, 24, 25, 35]. Most of these works assume that a leak has been already observed and test code triggering the leak is available. They help the developer with isolating the root causes of a leak at a cost of a proprietary execution environment (e.g. modified JVM [6]) or significant execution slowdown (e.g. 300-400% for Java [37]). Recent approaches for C/C++ focus on performance efficiency and promise slowdown of <3% [17]. This makes them usable in a production environment and allows leak detection at customer sites.

However, none of these works address the fact that virtually all non-trivial software projects today (1) are developed as a series of relatively small code changes, and (2) are accompanied by an extensive suite of software tests which check (primarily) functional properties of the artifact. In this work we exploit (1) for an anomaly-detection based approach for leak diagnosis, and (2) for triggering memory leaks during in-house testing. Our approach supports diagnosis of memory leaks during the software development phase and helps to pinpoint the root causes if a leak is detected. It requires only small modifications of the software testing framework and no changes in sources of tests and software. This is an important factor for its acceptance and practicability in context of existing projects. Since our method is based on anomaly detection, it is not necessary to execute test code until significant memory bloat occurs (such a bloat is a prerequisite for most existing methods). In this way, diagnosis time remains proportional to the time for executing project's test code.

Inspired by the Delta Debugging technique [39] for isolation of "crashing" errors we use software version comparison to uncover memory-related anomalies of the current (latest) software version. Figure 1 outlines the approach. Given an older and current version of software under development, we try to identify differences in memory allocation and deallocation behavior for each allocation site between these versions. The



Artifact version v_{new}
Instrumented version v_{new}
Test 1
Test k
test executions
$profile_{new}(t_1)$ … $profile_{new}(t_k)$
Intercepting object allocation / deallocation per allocation site
D_{new}
D_{old}
$profile_{old}(t_1)$ … $profile_{old}(t_k)$
Artifact version v_{old}
Instrumented version v_{old}
test executions
Test 1
Test k
Leak detection
Allocation sites ranked by leak confidences
Leak isolation

Figure 1. Overview of our approach

workloads used here are (unmodified) unit and/or integration tests. We assign each such allocation site an anomaly score (denoted as *leak confidence* LC, Section II-C) and rank the sites by this value. Unusually high LC values of top-ranked sites might indicate a new memory leak. Detection can be performed by manual evaluation of such top LC scores. For automated leak detection, the leak confidence of top-ranked sites can be compared against a threshold. If alert is triggered, ranking of sites supports debugging by indicating which allocation sites should be checked first.

Our work comprises the following contributions:

- We propose an approach for diagnosis of memory leaks [Section II] which is based on regression testing (comparison of software versions under development) and exploits existing test code for leak discovery. Contrary to other approaches, such diagnosis can be completed before a visible memory bloat occurs, in time proportional to the execution of a project's test code.
- We validate our approach on 7 real cases found in 5 large projects. We perform empirical evaluations on the accuracy and efficiency of the approach and provide estimations on execution time and memory overheads [Section IV].
- The results show that if the leaky code is exercised by the (unit) tests, our approach can accurately diagnose memory leaks with a low rate of false positives [Section IV]. Furthermore, the overheads on our prototypical testbed indicate that the approach is feasible for performing leak detection during the development phase of real-world projects.

II. Approach Description

Our approach is based on comparing object allocation and deallocation behavior of a previous version v_{old} and a target (usually most recent) version v_{new} of a software artifact under development (Figure 1). Our method attempts to diagnose leaks which have been *newly introduced by code evolution between* v_{old} *and* v_{new}. We do not assume that v_{old} is leak-free but leaks already present in v_{old} are less likely to be discovered. Thus, the choice of v_{old} should consider its reliability, especially whether memory bloat has been observed during prolonged execution. Our approach works with both C/C++ and managed languages, with the only difference being technicalities of code instrumentation. The details of our prototypical implementation in Java are outlined in Section III-A.

A. Instrumentation and Data Collection

Given software versions v_{old} and v_{new}, we identify (by static analysis) in each version all code locations which can allocate heap memory. In Java, such an allocation is triggered by object instantiation; in C/C++ this can be also caused by calling `new()` or related functions. We denote such a code location as an *allocation site* as and identify it by a source file ID, line number and (in case of Java) the class of the instantiated object.

In the subsequent phase of the diagnosis we execute a series of software tests on instrumented versions of both v_{old} and v_{new} and collect allocation-related data from each test run. For clarity, we speak in the following about unit tests (symbol ut) but in fact any other type of test or terminating code can be used. In detail, for a given unit test ut and software version v the following data is logged for each allocation site as:

- number n_a of objects allocated at as during the whole execution of ut
- among all objects created at as, the number n_d of objects deallocated during the execution of ut.

Of particular interest is the number of *residual objects* which have not been deallocated upon termination (counted for a particular allocation site). Given a fixed as, we denote their number by $n_r = n_a - n_d$.

The data obtained after running ut under artifact version v are the tuples (ID_{as}, n_a, n_d) for all allocation sites, where ID_{as} is data described above which uniquely identifies an allocation site. We call this set of tuples (over all allocation sites) an *allocation profile* and denote it by $profile_{old}(ut)$ or just $profile_{old}$ for v_{old} and by $profile_{new}(ut)$ (or $profile_{new}$) for v_{new}. Note that test execution is not always deterministic, and so the allocation profile might depend on a particular run.

The full results of the dynamic data collection are a set D_{old} of all allocation profiles (i.e. over all unit tests) executed under v_{old}, and an analogous set D_{new} for v_{new}.

B. Types of Allocation Sites

Some of the allocation sites present in $profile_{old}$ do not exist or have not been executed in $profile_{new}$ and vice versa. This gives rise to the following grouping of allocation sites:

- *Old allocation sites.* These are allocation sites which are recorded only in the allocation profile $profile_{old}$. As we are interested in leak discovery in the newer software version v_{new}, such allocation sites can be safely ignored.
- *New allocation sites.* These are allocation sites which are visible only in the new allocation profile $profile_{new}$.
- *Matching allocation sites.* These are allocation sites which appear in both old and new profiles.

Line insertions or deletions due to changes between previous and new software version change line numbering for all subsequent lines in the respective source file. This creates a technical problem for pairing (matching) allocation sites between versions. For example, if in source file F a new line after the line number k has been added, each line with number $j > k$ in the older version of F corresponds to line with number $j + 1$ in the new version of F. Since line numbers are part of data identifying an allocation site, the line numbers must be adjusted to identify correctly all new and matching sites.

We solve this problem with an algorithm which analyses the differences between source codes of v_{old} and v_{new} and adjusts line numbers in all profiles *profile*$_{new}$. The input of this algorithm are patches (`*.diff` - files) expressing code differences between v_{old} and v_{new} obtained by querying a software repository for the project. However, also any other data comparison tool which produces output in *unified diff* format can be used for preprocessing.

C. Leak Confidence Score

This section describes the computation of the scalar anomaly score called *Leak Confidence LC* from the complete sets of allocation profiles D$_{old}$ and D$_{new}$. This score maps each allocation site as encountered in v_{new} to a numerical value $LC(as)$ in $[0, 1]$, with higher values indicating higher defect probability. The general form of $LC(as)$ is:

$$LC(as) = A(as) * B(as) * C(as) \tag{1}$$

with terms $A(as)$, $B(as)$, and $C(as)$ described below. Their definitions are based on our empirical observations and our prior research on memory leak detection [19].

To simplify the notation, we introduce for $x \neq 0$, $y \neq 0$ the *normalized harmonic mean of x and y $H(x, y)$* defined by:

$$H[x, y] = \frac{1}{1/x + 1/y}.$$

We set $H = 0$ if $x = 0$ or $y = 0$.

1) Factor $A(as)$: This factor captures the overall strength of an allocation site as in terms of residual memory/objects. It only considers version v_{new}. It exploits as a core idea the observation that a leaky allocation site as is likely to deallocate only *few* of its allocated objects. This should hold for any unit test ut and will yield a high "relative" number of residual objects $n_r(as, ut)/n_a(as, ut)$.

We can achieve a higher robustness if we consider the set $UT(as)$ of all unit tests which cover as. This motives the definition of the *rate of residuals ResidR*: it is the fraction of allocated objects which are not deallocated during the execution of relevant unit tests:

$$ResidR(as) = \frac{\sum_{ut \in UT(as)} n_r(as, ut)}{\sum_{ut \in UT(as)} n_a(as, ut)}. \tag{2}$$

Our experiments have shown that leaky allocation sites have higher *absolute* number of residual objects $\sum_{ut \in UT(as)} n_r(as, ut)$ (summed over all relevant unit tests). The final formula for $A(as)$ combines both expressions via the normalized harmonic mean:

$$A(as) = H[ResidR(as, UT), \sum_{ut \in UT} n_r(as, ut)]. \tag{3}$$

2) Factor $B(as)$: This factor captures how easily a memory leak is triggered at an allocation site as by a unit test that exercises it. It only considers version v_{new}. If an allocation site as becomes a leak cause then it is likely to create residual objects under many different execution patterns (represented by different unit tests). We define *("dirty") test rate $TestR(as)$* as the fraction of unit tests ut (among unit tests in $UT(as)$) for which the number of residual objects $n_r(as, ut)$ is greater than zero.

The metric $TestR(as)$ can be inaccurate in case of small $|UT(as)|$, i.e. if allocation site as executed only few times. To dampen the impact of such cases, we use harmonic mean of $TestR(as)$ and $|UT(as)|$:

$$B(as) = H[TestR(as), |UT(as)|]. \tag{4}$$

3) Factor $C(as)$: This factor measures the "leakiness" of an allocation site as in the new version v_{new} compared to the old version v_{old} (and hence considers both versions). If an allocation site as becomes a leak cause due to evolution between versions v_{old} and v_{new}, the number of its residual objects $n_r(as)$ is likely to increase in v_{new}. We define the $NresidChR(as)$ as the relative change in the number of residual objects of *as* in the older version to the newer version, i.e.:

$$NresidChR(as) = \frac{n_r(as, v_{new}) - n_r(as, v_{old})}{n_r(as, v_{new})}$$

Note that for new allocation sites, $n_r(as, v_{old})$ is equal to zero, and so we have $NresidChR(as) = 1$ in such cases.

Allocation sites with larger value of numerator $\Delta(as) := n_r(as, v_{new}) - n_r(as, v_{old})$ have higher probability to be a memory leak. Therefore we define $C(as)$ as the harmonic mean of $NresidChR(as)$ and $\Delta(as)$:

$$C(as) = H[NresidChR(as), \Delta(as)]. \tag{5}$$

D. Leak Detection and Isolation

The sets D$_{old}$ and D$_{new}$ of all allocation profiles are used to compute the *Leak Confidence LC* (Section II-C) for each allocation site triggered in the current software version v_{new}. In the next analysis step, the allocation sites in v_{new} are ranked by their LC-values in decreasing order. In this way we obtain a *ranked list of suspects* with most suspicious sites being top-ranked.

Leak detection (i.e. alerting about existence of any potential new leaks) can be performed in two ways. The first option is a "manual" inspection of the top values in the ranked list. If the top values are substantially higher than previously observed for this application, or if previously unknown allocation sites surface to the top, a manual alert can be triggered.

The second option is to raise an alert automatically if the highest LC value in the list is above a threshold, i.e. $LC(as) >$ LC$_{th}$. The value of such a threshold needs to be adjusted for each application under development which is shown in Section IV-A1. Clearly, much more sophisticated methods than a simple threshold (like adaptive thresholds, or a combination of a threshold and "novelty" of top-ranked sites) are possible. The investigation of such schemata and automated leak detection is beyond the scope of this work.

Leak isolation (i.e. finding the root causes) is performed analogously to automated debugging: a developer is given a ranked list of suspects and investigates the code and behavior of the top-ranked allocation sites in this list. As discussed in Section IV-A, our method is sufficiently accurate by placing the true root causes (i.e. defect-inducing allocation sites) close to the top of the list.

III. Implementation and Evaluation Environment

All experiments were executed in a virtual machine with 8 GB physical memory under Ubuntu 12.04 running on a 2.9 GHz Intel Dual Core i7-3520M CPU. We have used Java 1.6.0_27 with 4 GB heap size. Our framework, data and evaluation results of our approach are available at http://1drv.ms/1GU3Pfn.

A. Leak Detection in Java

We describe here the instrumentation in our Java prototype needed to record the allocation profiles introduced in Section II-A. In Java, an allocation site corresponds to a unique location in the bytecode. We specify it by the name of the corresponding source file, the line number of this allocation site in the source file, and the class of instantiated object. In this way, the developer can identify (during leak isolation) the code location more easily than via a bytecode position.

Object allocations. To monitor and record object allocations we use the library *java-allocation-instrumenter* [22]. It performs static code analysis and instruments bytecode at each allocation site. This instrumentation calls our proprietary code hook which inspects the current stack trace. We retrieve the code location of the caller (i.e. allocation site file and line number), and save this information together with the class of the instantiated object in a hash map. Our hook also checks whether the allocation site is located in one of the source files of interest (we exclude code in the third-party libraries).

Object deallocations. The final function of the code hook is to prepare notifications of object deallocations. To this end we link via *phantom references* (using Java's `sun.misc.Cleaner.create()` method) each newly allocated object with a proprietary callback method. This method executes exactly once after the object has been disposed. The method can identify the allocation site as for the object and updates the deallocation count for it.

After a test has finished but the JVM is still alive, we enforce a garbage collection and record the statistics n_a and n_d for each monitored allocation site.

B. Evaluation Environment and Applications

We performed the evaluation of our approach on the reported real memory leaks from different Java applications. We collected seven real leaks shown in the first column of Figure 2. The corresponding subject programs are listed in Table I. Three out of seven cases are from Apache Hadoop. The rest are collected from four other applications: Snappy-Java, Apache Solr, Apache Nutch, and Apache Thrift.

Table I
Subject programs.

Subject Program	# LOC	# unit tests
Hadoop-Common	94k	234
Hadoop-Yarn	163k	85
Hadoop-HDFS	200k	315
Snappy-Java	2.5k	6
Apache Thrift	6k	18
Apache Solr	38k	19
Apache Nutch	27k	31

Apache Hadoop [2] is a large open source software project containing several thousands lines of code of Java with a lot of development revisions. It also comes with several hundreds of unit tests. These features make Hadoop a suitable test environment for the evaluation of efficiency and accuracy of our approach in detection and isolation of memory leaks in large-scale software systems.

Snappy-Java [32] is a port of Snappy program which is used in many projects and frameworks such as Apache Spark, Big Table and MapReduce for compression and decompression. Apache Solr [4] is an open source enterprise search platform mainly for full-text searching. Apache Nutch [3] is an open source, scalable, feature-rich web search engine. Apache Thrift [5] is a software framework which is designed for development of scalable and efficient cross-language services.

For six out of seven cases, the developers fixed the leak defects by applying new patches to the leaky version. We used these patches to find the actual root cause of the memory leaks. Note that to fix the memory leaks, developers have changed multiple lines of code in different files. However, in each case we marked as the root cause of a memory leak an allocation site which was generating the leaking objects.

We reproduced the leaky version for each case from the corresponding issue in the bug repository. For non-leaky versions, we manually found a reasonably recent non-leaky version *prior* to the leaky one by comparing the source codes. In this way, we obtained seven pairs of software versions (one per issue): (non-leaky older version, leaky newer version).

Although manual finding a non-leaky version is quite tedious, this step is needed only in the evaluation and it is not a limitation of our approach.

C. Experimental Evaluation

To evaluate our approach, we designed experiments to address the following research questions:

RQ1: ***How accurate is our approach in root cause isolation of memory leaks?***

To answer this research question, we perform the leak confidence analysis for each of the seven issues listed in Figure 2 on two application variants: on a faulty version (i.e. with memory leak), and a non-faulty version (prior to the faulty version). We report 1) the leak confidence score for the root cause of the memory leak and 2) the position of the root

issue	(a) information				(b) leak isolation			(c) LC analysis	
	issue status	leaky version	non-leaky version	#trig. ut (#total ut)	rank	#candidates		LC	LC_{max}
						$LC > 0$	$LC > 0.6$		
hadoop-8632	fixed	2.0.0a	0.20.0	135(234)	1	3079	484	0.99	0.99
hdfs-5671	fixed	2.2	2.0.6	2(315)	157	1574	242	0.64	0.99
yarn-1382	open	2.2	0.23.11	23(85)	125	1214	311	0.84	0.98
thrift-1468	fixed	0.5.0	0.4.0	0(18)	-	207	12	-	0.88
snappy-91	fixed	1.1.1.4	1.1.1.3	2(6)	1	11	1	0.61	0.61
solr-1042	fixed	1.3	1.2.1	11(19)	4	129	13	0.84	0.86
nutch-925	fixed	1.2	0.8	13(31)	9	261	28	0.86	0.93

Figure 2. Results of leak confidence analysis and leak isolation for real cases using new site analysis. Section (a) describes the issues: status of the issue, leaky and non-leaky versions of each subject programs as well as the number of unit tests which trigger the leak pattern (Column #trig. ut). Section (b) shows the result of the leak isolation: rank of the defect-inducing allocation site and the size of the ranked list of suspects using two filtering criteria on the leak confidence value LC an allocation site: $LC > 0$ and $LC > 0.6$. Section (c) reports as LC the leak confidence score for the defect-inducing allocation site and as LC_{max} the largest leak confidence value among all sites in the ranked list.

cause of the memory leaks in the ranked list of the suspicious allocation sites (Section IV-A).

RQ2: ***What is the runtime performance of our approach?*** To answer RQ2, we evaluate our approach in terms of runtime and memory overhead (Section IV-B). For each unit test we collect these two statistics after the test is executed. Finally, we aggregate the collected measures over all unit tests for the program in question and report the overall results for each case.

IV. Experimental Results

This section presents our experimental results followed by a discussion of them.

A. Answer to RQ1: Accuracy of Defect Isolation

To evaluate the accuracy we collect and analyze data before and after leak-inducing changes in each of the seven cases listed in Figure 2 (Section III-B). In this figure, Section (a) provides information about the versions used for each of the applications and the number of unit tests which trigger the leak pattern. For each case, we executed all of the unit tests for both non-leaky and leaky versions of the program in question. Then we applied the leak confidence analysis to find the root cause of the memory leaks.

The collected data revealed that in all seven cases the failure-inducing allocation site was of type *new site* (see Section II-B). Therefore we considered only new sites in our analysis. Section 2(b) shows the result of leak isolation for each case. In 4 out of 7 cases, the root cause of the memory leaks were ranked in top 10. The root cause in two cases, hadoop-8632 and snappy-91 were ranked first. The other two issues, solr-1042 and nutch-925 were ranked 4 and 9, respectively.

For issues hdfs-5671 and yarn-1382 our approach assigns low ranks to the failure-inducing allocation sites: 157 and 125, respectively. This essentially means that failure isolation is not successful in these cases. The reasons for this are discussed in Section IV-A2. In case of the issue thrift-1468, our approach did not include the root cause of the memory leak in the list of suspects. This can be attributed to the fact that the leak-activating allocation site is not triggered by the unit tests at all. Consequently, this allocation site did not appear in the list of known allocation sites. In Section IV-A2 we provide a detailed analysis for each leak issue.

1) Other aspects of defect isolation: Developers usually investigate only few first entries of a ranked list of suspects for defect isolation. Since this list would originally contain 1000's of entries, we propose to filter it. Figure 2 Section (b) shows the lengths of filtered lists for two criteria: $LC > 0$ and $LC > 0.6$. Except for issue snappy-91, the criterion $LC > 0.6$ seems to produce lists of appropriate lengths.

In Section (c) of Figure 2 we show the actual leak confidence values of the defect-inducing allocation sites (symbol LC). For comparison we also include as LC_{max} the largest leak confidence value among all sites in the ranked list. Obviously for the four accurately isolated root causes (i.e. hadoop-8632, snappy-91, solr-1042, and nutch-925) the differences between LC and LC_{max} are small $(0, \ldots, 0.07)$. In cases of inaccurate isolation of the root causes (hdfs-5671 and yarn-1382) the differences are larger (0.35 and 0.14, respectively) but not extreme.

Also, LC_{max} values show that the top values of the leak confidence scores vary between applications, ranging between 0.61 (snappy-91) and 0.99 (hadoop-8632, hdfs-5671). This makes it indeed necessary to use application-specific threshold values for automated (threshold-based) leak detection discussed in Section II-D (see also Section IV-C2).

2) Detailed discussion of each issue:

HADOOP-COMMON: Issue *HADOOP-8632*[1] reported that a newly introduced variable *CACHE_CLASSES* in the *configuration* class caused a leak on the class loaders. This variable is a part of a patch for solving a performance regression issue (issue *HADOOP-6133*). Therefore the changes in the *HADOOP-6133* could be the root cause of this memory leak. However, the memory leak was reported three years after *HADOOP-6133* was submitted in issue *HADOOP-8632*. Consequently, a developer has modified the code with converting the strong reference of the class to its *ClassLoader* to a weak reference:

[1]https://issues.apache.org/jira/browse/HADOOP-8632

```
- private static final Map<ClassLoader, Map<String,
-   Class<?>>> CACHE_CLASSES = new
-   WeakHashMap<ClassLoader, Map<String, Class<?>>>();

+ private static final Map<ClassLoader, Map<String,
+   WeakReference<Class<?>>>> CACHE_CLASSES = new
+   WeakHashMap<ClassLoader, Map<String,
+   WeakReference<Class<?>>>>();
```

Although there is a large amount of changes between the leaky and non-leaky version, our leak confidence analysis could pinpoint the instantiation site of the variable *CACHE_CLASSES* correctly with high leak confidence score, placing it at position 1 in the ranked list of suspects. The reason is that the leak pattern in this case is exercised by many unit tests.

HADOOP-HDFS: Issue *HDFS-5671*[2] reported that there is a leak in the *getBlockReader* method of the *DFSInputStream* class. This issue is also duplicated by issue *HDFS-5697* in the bug repository. When a client requests a file's block to *DataNode*, the *BlockReader* will be called from *DFSInputStream* class. If the cache is missing, a new pair for BlockReader will be created. However, if an *IOException* is thrown during creation of a new *BlockReader* with the given pair, then the TCP socket used by the *regionserver* will not be closed. This causes too many *close-wait* status which is a socket (connection) leak and finally result in a huge memory footprint. To solve this issue, developers changed *DFSInputStream* class and moved the statement which creates a new *BlockReader* into a try block which closes the peer when there is no new *BlockReader*.

```
Peer peer = newTcpPeer(dnAddr);
- return BlockReaderFactory.newBlockReader(
-   dfsClient.getConf(), file, block, blockToken,
-   startOffset, len, verifyChecksum, clientName, peer,
-   chosenNode, dsFactory, peerCache,
-   fileInputStreamCache, false, curCachingStrategy);

+ try {
+  reader = BlockReaderFactory.newBlockReader(
+    dfsClient.getConf(),file,block,blockToken,startOffset,
+ ...
+ } finally {
+  if (reader == null) {
+   IOUtils.closeQuietly(peer);
+  }
+ } }
```

We applied our approach to detect the site which calls the new *BlockReader*. We used the version reported by the issue as the leaky version. However, for the correct version we chose a much earlier version prior to the leaky one because issue HDFS-5671 was not reported as a regression error. We exercised this leak to check the reaction of our approach to the huge amount of changes between the two software versions (i.e., leaky and non-leaky versions).

Our approach reported the suspicious statement, however with a low leak confidence score and low ranking position. The main reason for this low accuracy is the low number of unit tests which trigger the memory leak. The anomaly score (leak confidence) introduced in the Section II-C is directly dependent on the number of unit tests which trigger a memory leak pattern. The more triggering unit tests, the higher value of the leak confidence. However in this case in contrary to issue *HADOOP-8632*, only two unit tests (from the total number of 315 unit tests) exercised the memory leak pattern which contributed to low accuracy.

[2]*https://issues.apache.org/jira/browse/HDFS-5671*

```
+ inputBuffer = inputBufferAllocator.allocate(inputSize);
+ outputBuffer = inputBufferAllocator.allocate(outputSize);
```

Figure 3. The leak-inducing changes in the Snappy-Java.

HADOOP-YARN: Issue *YARN-1382*[3] reports that *NodeListManager* class in *Hadoop-Yarn* contains a memory leak when a node in the unusable nodes set never comes back. This issue is reported in the comments of another issue (*YARN-1343*) and is still an open issue during the writing of this paper. Based on the description of the issue, although this is not a huge memory leak, it can be accumulated if the *NodeManager* are configured with ephemeral ports which assumes that the nodes are still new when they are released.

We applied the analysis to check whether our approach can find the root cause of this leak. Although we could pinpoint the potential root cause with high leak confidence score, the position of the root cause in the ranked list was low.

The main reason for the low accuracy is the large amount of changes between the leaky and non-leaky version. Our approach requires a prior version (non-leaky version) to the current version (i.e., potentially leaky version) of the application in order to categorize the allocation sites and to perform the leak isolation using the leak confidence analysis. If these two versions have a huge source code difference (i.e., many versions between these two versions), the total number of newly introduced allocation sites increases substantially. This considerably affects the rank of the defect-inducing allocation site and also increases the number of potential leak suspects.

SNAPPY-Java: Snappy-Java suffered from a severe memory leak after updating from version 1.1.1.3 to the version 1.1.1.4. This memory leak was reported in issue *Snappy-91*[4]. It had negative effects on Apache Spark 1.2.0 which updated the Snappy-Java library to the version 1.1.1.4. Due to the memory leak introduced in the newer version of the Snappy-Java, the Spark developers roll-backed to the previous version of the Snappy-Java. Figure 3 shows the code excerpt which contains the leak-inducing changes in the *SnappyOutputStream* class. The *outputBuffer* is allocated from the *inputBufferAllocator*, however it is released to the *outputBufferAllocator*.

After applying our approach to the Snappy-Java, it could catch the instantiation site of the *inputBufferAllocator* as a suspicious allocation site with high leak confidence score and also assigned it the highest rank in the list of suspects.

SOLR: Issue *Solr-1042*[5] reports a memory leak in the *DataImportHandler*. If *SqlEntityProcessor* executes *DataImport* many times, the instances of *TemplateString* will be cached. This causes the memory footprint to grow steadily until it reaches an *OutOfMemory* exception. The solution for this memory leak is to use the *TEMPLATE_STRING* as a non-static

[3]*https://issues.apache.org/jira/browse/YARN-1382*
[4]*https://github.com/xerial/snappy-java/issues/91*
[5]*https://issues.apache.org/jira/browse/SOLR-1042*

variable. This is applied by the developers to the next version of *SOLR*:

```
- private static final TemplateString TEMPLATE_STRING =
-   new TemplateString();

+ private final TemplateString templateString =
+   new TemplateString();
```

Our approach could pinpoint the instantiation site of the variable *TEMPLATE_STRING* as a suspicious allocation site by ranking it among the top 5 potential root causes. The leak confidence score of the root cause was 0.84 - a narrow difference to the site with the highest leak confidence score (0.86).

NUTCH: Issue *Nutch-925*[6] reports a severe memory leak in the plugin repository cache used in the *PluginRepository* class of *NUTCH*. The *plugins* are stored in a *WeakHashMap* <*conf, plugins*>. Each time *plugins* are needed, a new *Class* and *ClassLoader* will be created. Since *Class and ClassLoader* are stored in the permanent heap space they cannot be garbage collected. Therefore the *OutOfMemory* exception will be thrown eventually. Following is the partial changes applied by the developers to fix this issue:

```
- private static final WeakHashMap<Configuration,
-   PluginRepository> CACHE = new WeakHashMap<
-   Configuration, PluginRepository>();

+ private static final WeakHashMap<String,
+   PluginRepository> CACHE =
+   new WeakHashMap<String, PluginRepository>();
```

We performed analysis to detect this memory leak. Our approach could isolate the leak root cause among the top 10 entries of the ranked list with a relatively high leak confidence score of 0.86.

THRIFT: Issue *Thrift-1468* [7] reports a memory leak which is captured during running of Apache HCatalog. According to the description of this issue, if a HCatalog server runs for a long time with continuous client requests, the memory footprint of the *metastore-server* grows continuously until it reaches an *OutOfMemory* exception. The *HCatalog-server* uses Apache Thrift. There is a *WeakHashMap* which maps *TTransport* objects to their wrapped *TSaslServerTransport* instances in the class *TSaslServerTransport* of the Apache Thrift. However, in the *WeakHashMap* the value has a hard reference back to the key. Therefore the entry persists for all time, since the key can not be garbage collected. This causes an increase in the memory usage when the *HCatalog-server* is running. Following is a part of the patch that is applied by the developers to fix this issue:

```
- private static Map<TTransport, TSaslServerTransport>
-   transportMap = Collections.synchronizedMap(new
-   WeakHashMap<TTransport, TSaslServerTransport>());

+ private static Map<TTransport,
+   WeakReference<TSaslServerTransport>> transportMap =
+   Collections.synchronizedMap(new
+   WeakHashMap<TTransport,
+   WeakReference<TSaslServerTransport>>());
```

[6]*https://issues.apache.org/jira/browse/NUTCH-925*
[7]https://issues.apache.org/jira/browse/THRIFT-1468

However our approach was not successful in case of Apache Thrift. After further investigations, we found that the correspondent function triggering this memory leak was not exercised by the test suites provided with the Apache Thrift. Naturally our approach was unable to isolate the root cause.

This memory leak is highly environment-sensitive and hence it can be only triggered by exercising a specific pattern. Although this case shows a limitation of our method, it simultaneously points potential extensions. We can use test generation techniques with optimization objective to find patterns triggering memory leaks. This is an important research direction which is also mentioned by the other works [30, 36].

B. Answer to RQ2: Performance Evaluation

To answer RQ2, we measured the performance of our approach using POSIX-conform operating system commands. *Run time* and *Resident Set Size* (RSS) are the metrics that we collected for each unit test.

To compute the overhead of our approach, we collected the runtime and RSS after execution of each unit test with and without instrumentation. Then for each subject programs we aggregated the obtained results from execution of all of the unit tests corresponded to the subject program. The overhead is then computed as follows:

$$overhead = \frac{\sum_{ut \in UT} metric_{inst} - \sum_{ut \in UT} metric_{no\, inst.}}{\sum_{ut \in UT} metric_{no\, inst.}}$$

where metric can be the runtime or RSS ($metric_{inst}$ is the value of instrumented version, and $metric_{no\, inst.}$ without instrumentation).

Figure 4 shows the runtime and RSS overhead of our approach on the subject programs. The result shows that our approach imposes a moderate overhead on both runtime and RSS which makes it applicable for development phase.

The overhead runtime of our approach is imposed by the instrumentation used for collecting the heap profiles. It causes delays in both instantiation and deletion of allocation objects which results in an increase in the overall execution time of the unit tests. As shown in Figure 4, the execution time of the unit tests with instrumentation is on median from 0.55 to 3.75 times more than the runtime of the unit tests without instrumentation. One solution to decrease the runtime overhead is the selective instrumentation. For example, we can instrument only the part of the code which are only relevant to the recent code changes.

Resident set size (RSS) is also measured for each of the subject programs. Figure 4 shows that the aggregate RSS for each subject programs with instrumentation is on median from 1.2 to 3.97 times more than aggregate RSS of the that program without instrumentation.

C. Discussion

In this subsection we describe our observations during the experiments and later discuss possible threats to validity for our evaluation.

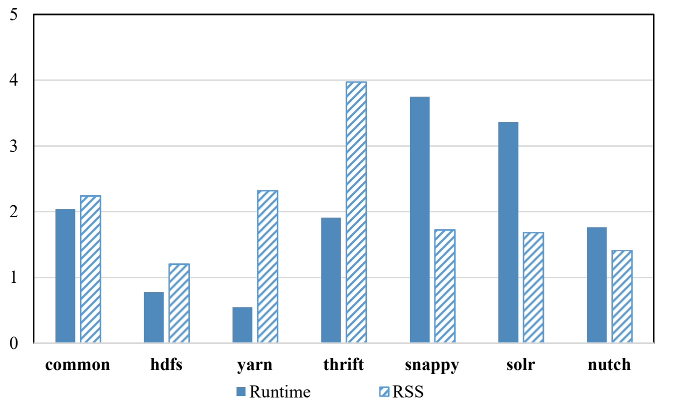


Figure 4. Runtime and RSS overhead of subject programs. Overhead of 1 (y-axis) means that the instrumented version has twice the runtime or RSS of the non-instrumented version.

1) Summary: From the results of the experiments we found that our approach is generally effective in isolation of memory leaks. By analyzing the results obtained from the experiments we can conclude that our approach can diagnose memory leaks if the execution pattern triggering the leak is exercised by the (unit) tests.

We examined our approach on several large-scale real-world applications to verify the scalability of our detection approach. Our approach included the leak root cause in the list of suspects in 6 out of 7 cases. In 4 cases, the root causes were ranked in the top 10 of our ranking report. As such, for RQ1, we can assess that *our approach is applicable to large, real programs and is able to diagnose memory leaks whenever the leak pattern be executed by the (unit) tests. Also it can significantly reduce the number of suspected memory leaks by applying the introduced leak confidence analysis.*

To evaluate the performance efficiency we measured the run time and resident set size of the execution of the unit tests. Based on the collected data we can assess for RQ2 that *our approach imposes moderate overhead in terms of run time and memory consumption. This makes our approach feasible for using in the testing process.*

2) Choice of the Leak Confidence Threshold: Our leak confidence metric can be used to detect potential memory leaks by comparing its value against a predefined threshold LC_{th} (Section II-D). The choice of the threshold value will generally impact the number of false positives and negatives. Very low threshold values can introduce many false positives, while high values might lead to missing leaky allocation sites.

In particular, results in Section (c) of Figure 2 show that a suitable threshold value differs between applications (see Section IV-A1). In general, a developer needs to adjust such threshold according to the experiences with the number of false positives and false negatives on a particular project.

3) Threats to Validity: Threats to external validity arise when the results of the experiments cannot be generalized for any arbitrary program. We evaluated our approach on the limited number of applications. Therefore we can not claim that our approach can isolate memory leaks in all types of real-world applications. However, we are confident that our approach can be applied to a variety of Java applications (and C/C++ applications with a suitable code instrumentation). This can help developers in the root cause analysis of the memory leaks and also narrowing down the list of suspicious allocation sites.

Threats to construct validity arise when our approach is unable to pinpoint the leaky allocation site because of lack of the tests which trigger the memory leak pattern. That means the accuracy of our approach is influenced by the ability of the test code to trigger the leak defects. This can be avoided by having test suites with high code coverage. This is a typical goal in quality assurance of each project and can be assumed in typical real-world applications.

Threats to internal validity arise when our approach instrument the source code of the application in question. Delay in the instantiation and the destruction of the allocated objects due to the code instrumentation might affect the run time of the testing process. However, the overall overhead of our approach is moderate and is comparable to the existing leak detection approaches.

V. Related Work

There is a large body of research dedicated to the problem of memory leaks, see [33, Chapter 3] for a comprehensive survey. We restrict our discussion to the techniques related to our approach and to the most significant results.

Static analysis. Static analysis has been used predominantly for languages with manual memory management like C/C++. It can detect defects such as double or missing calls of `free()`. Techniques include e.g. reachability analysis via a guarded value flow graph [8], backward dataflow analysis [29], or detecting violations of constraints on object ownership [15]. A major problem of static analysis is lack of scalable and precise reachability/liveness analysis for heap objects in managed languages like Java. A recent approach named LeakChecker [38] attempts to overcome this problem by focusing on important loops specified by the developer.

Automated resource management [10, 27] is another direction in static analysis. FACADE [27] is a compiler and runtime framework which decreases the cost of runtime memory management by transforming the data path of the big data applications. Using code transformation and static approximation of resource lifetime, CLOSER [10] determines the higher-level resources which contain references to other resources in source code of application. Then it inserts disposal calls at appropriate points in the source code of application to release the resources with expired lifetime.

Dynamic analysis. The major lines of approaches include *detecting staleness*, *growth analysis*, and *analysis of captured state*. Our work is related to the last two groups.
Staleness detection. Staleness (lack of recent read/write accesses) is the most direct and distinguishing property of leaked memory. It has been exploited in series of excellent works, pioneered by the SWAT approach [14]. The key problem is

the overhead of monitoring object accesses. Multiple remedies have been proposed: path-biased sampling [14], page-level sampling [28], modifications of the JVM [6], or focusing only on container accesses [37]. A recent work Sniper [17] is able to reduce the total runtime overhead for C/C++ to less than 3% by exploiting hardware units in modern CPUs. For Java, the lowest overhead is still about 80% [37], despite that only containers are monitored and code annotations are required.
Growth analysis. Cork [16] finds growth of heap data structures via a directed graph (TPFG) where each object traces all references pointing to itself. FindLeaks [7] tracks object creation and destruction and if more objects are created than destroyed per class (i.e. number of residuals grow), a suspect is found (runtime overheads are not reported). This resembles our approach, but we essentially compare the number of residual objects (per allocation site) *between software versions*. By utilizing this prior information we are less prone to false positives such as allocation sites with large number of residuals (e.g. caches). Works [33, 34] (and in a modified form the NetBeans Profiler) exploit the observation that for a perpetually leaking class many different generations of its instances exist. Machine learning techniques helps here for low latency leak detection (with runtime overheads of about 40% [33]).
Analysis of captured state. LeakBot [25] uses complex, multi-phase object ranking to find suspicious regions of heap objects which grow over time (using multiple heap snapshots). LeakChaser [35] exploits invariants among lifetimes of objects. It requires a developer to specify regions or objects belonging to the same "transaction" in order to detect invariant violations. LEAKPOINT [9] uses dynamic tainting to track heap memory pointers. It is implemented on top of Valgrind [26] which results in a runtime overhead of 100–300 *times*. The Valgrind tool Omega [24] uses related approaches (similar overheads).

Version comparison. Comparing behavior between evolving code versions (here called *version comparison*) is the key idea of regression testing. Many techniques use this concept for debugging of crashing (i.e. non-latent) errors: Delta Debugging [39] attempts to find a minimal set of failure-inducing code changes, works [12, 20, 31] exploit it for automated debugging, and in the work [40] it is used for diagnosing configuration errors.
To our best knowledge, only our own previous works utilize version comparison for memory leak detection or isolation. Papers [18] and [23] use cumulative memory consumption metrics (such as Heap Usage or RSS) for *detecting* whether memory leaks have been introduced in a newer version; the isolation of defects is not addressed. While paper [18] suggests a simple visual detection scheme, work [23] evaluates more sophisticated anomaly detection methods, in particular Control Charts.
The short paper [19] is most closely related to this work. It performs leak isolation first by ranking the allocation sites based on the type of the allocation sites (i.e., new and matching allocation sites) and then with comparing the number and cumulative size of residual (not deallocated) objects per allocation site between software versions. This work considerably extends the paper [19] and differs from it in multiple aspects. First, our ranking criterion (leak confidence, Section II-C) for suspicious allocation sites is completely different. It is based on more refined metrics, essentially the (absolute) numbers of allocated and deallocated objects measured for *many different* (unit) tests. Contrary to this, in paper [19], only a single (integration) test was used. Second, the evaluation presented here is based on real memory leaks while in paper [19], only injected memory leaks was evaluated. Finally, the robustness and accuracy of the method presented in this work is substantially higher.

VI. Conclusion and Future Work

Although only 1% of all defects in large open-source projects are related to issues such as memory leaks [21], they can substantially increase the cost of ownership of large-scale software systems. Unfortunately, their inherent characteristics (namely long latency before manifestation and weak link between defects and symptoms) substantially hamper their detection and isolation via traditional unit and integration tests.

We have presented an approach for memory leak diagnosis which exploits existing test code. It is based on version comparison approach in combination with data analysis. Our technique can alert about suspected presence of memory leaks and provides a ranked list of suspicious allocation sites as an automated debugging support for the developers.

Our approach has multiple advantages. First, the effort of setup and integration into existing testing processes is low since no test modifications are required and existing test processes and frameworks can be used. Thus, the method can be integrated into the regression testing phase without any modification on the existing test suites. In this phase, developers can apply our approach to find the memory leaks in a specific version and after fixing them they can assume the fixed version as a non-leaky version. Secondly, the diagnosis is sufficiently accurate which can substantially shorted time for fixing such defects. Finally, the execution overhead (even in our prototypical system) is acceptable and makes it feasible to use our method for automated memory leak checks once in every 2 to 4 runs of a "normal" full test suite.

Our future work will target three directions. First, we will perform a detailed study on the impact of different threshold values on a variety of applications to find the optimal threshold value for memory leak detection. Secondly, we want to optimize the performance of our approach in order to reduce the runtime and memory overheads. For example, we can instrument only the part of the code which is relevant to the recent code changes and also we can execute the unit tests which only correspond to the changed code. With such optimizations, one can run our approach after each commit of the code changes. Thirdly, we will focus on cases with low detection accuracy and attempt to improve our technique here. A promising approach is to modify unit tests to achieve higher coverage of modified code regions, since lack of triggering of leaking sites turned out to be one the primary reasons for low accuracy.

ACKNOWLEDGMENTS

We would like to thank Lutz Büch, Zhen Dong and Diego Costa from Heidelberg University for their fruitful feedback. We also thank the SCAM reviewers for their valuable and thorough comments.

REFERENCES

[1] Amazon AWS. Summary of the October 22,2012 AWS Service Event in the US-East Region. https://aws.amazon.com/de/message/680342, 2012.

[2] Apache hadoop. http://hadoop.apache.org/.

[3] Apache nutch. http://nutch.apache.org/.

[4] Apache solr. http://solr.apache.org/.

[5] Apache thrift. http://thrift.apache.org/.

[6] 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.

[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.

[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.

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

[10] I. Dillig, T. Dillig, E. Yahav, and S. Chandra. The closer: Automating resource management in java. In *ACM International Symposium on Memory Management (ISMM)*, pages 1–10, 2008.

[11] T. E. Foundation. The Eclipse Memory Analyzer (MAT). Version 1.2.

[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.

[13] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX*, pages 125–138, 1991.

[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.

[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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] C. Liu, J. Yang, L. Tan, and M. Hafiz. R2fix: Automatically Generating Bug Fixes from Bug Reports. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 282–291, 2013.

[21] F. Machida, J. Xiang, K. Tadano, and Y. Maeno. Aging-related bugs in cloud computing software. In *IEEE International Workshop on Software Aging and Rejuvenation (WOSAR) at ISSRE*, pages 287–292, 2012.

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

[23] 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.

[24] B. Meredith. Omega - An Instant Leak Detection Tool for Valgrind, 2008. Version 1.2.

[25] 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.

[26] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[27] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 675–690, 2015.

[28] 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.

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

[30] A. Orso and G. Rothermel. Software testing: A research travelogue (2000–2014). In *Conference on Future of Software Engineering (FOSE)*, pages 117–132, 2014.

[31] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. DARWIN: An Approach to Debugging Evolving Programs. *ACM Transactions on Software Engineering and Methodology*, 21(3):19:1–19:29, July 2012.

[32] Snappy-java. http://github.com/xerial/snappy-java/.

[33] V. Šor. *Statistical approach for memory leak detection in Java applications*. PhD thesis, University of Tartu, Estonia, Tartu, Estonia, 2014.

[34] V. Sor, 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, 2013.

[35] 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.

[36] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FSE/SDP Workshop on Future of Software Engineering Research (FoSER)*, pages 421–426, 2010.

[37] 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.

[38] D. Yan, G. Xu, S. Yang, and A. Rountev. LeakChecker: Practical Static Memory Leak Detection for Managed Languages. In *International Symposium on Code Generation and Optimization (CGO)*, pages 87–97, 2014.

[39] A. Zeller. Yesterday, My Program Worked. Today, It Does Not. Why? In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 253–267, 1999.

[40] S. Zhang and M. D. Ernst. Which Configuration Option Should I Change? In *International Conference on Software Engineering (ICSE)*, pages 152–163, 2014.