Diagnosing Memory Leaks using Graph Mining on Heap Dumps

Evan K. Maxwell
emaxwell@cs.vt.edu

Godmar Back
gback@cs.vt.edu

Naren Ramakrishnan
naren@cs.vt.edu

Department of Computer Science
Virginia Tech, VA 24061

ABSTRACT

Memory leaks are caused by software programs that prevent the reclamation of memory that is no longer in use. They can cause significant slowdowns, exhaustion of available storage space and, eventually, application crashes. Detecting memory leaks is challenging because real-world applications are built on multiple layers of software frameworks, making it difficult for a developer to know whether observed references to objects are legitimate or the cause of a leak. We present a graph mining solution to this problem wherein we analyze heap dumps to automatically identify subgraphs which could represent potential memory leak sources. Although heap dumps are commonly analyzed in existing heap profiling tools, our work is the first to apply a graph grammar mining solution to this problem. Unlike classical graph mining work, we show that it suffices to mine the dominator tree of the heap dump, which is significantly smaller than the underlying graph. Our approach identifies not just leaking candidates and their structure, but also provides aggregate information about the access path to the leaks. We demonstrate several synthetic as well as real-world examples of heap dumps for which our approach provides more insight into the problem than state-of-the-art tools such as Eclipse's MAT.

Categories and Subject Descriptors: H.2.8 [Database Management]: Data mining; D.2.5 [Software Engineering]: Debugging aids;

General Terms: Algorithms, Experimentation, Reliability.

Keywords: Memory leaks, heap profiling, graph mining, graph grammars, dominator tree.

1. INTRODUCTION

Memory leaks are a frequent source of bugs in applications that use dynamic memory allocation. They occur if programmers' mistakes prevent the deallocation of memory that is no longer used. Undetected memory leaks cause slowdowns and eventually the exhaustion of all available memory, triggering out-of-memory conditions that usually lead to application crashes. These crashes significantly affect availability, particularly of long-running server applications, which is why memory leaks are one of most frequently reported types of bugs against server frameworks.

Memory leaks are challenging to identify and debug for several reasons. First, the observed failure may be far removed from the error that caused it, requiring the use of heap analysis tools that examine the state of the reachability graph when a failure occurred. Second, real-world applications usually make heavy use of several layers of frameworks whose implementation details are unknown to the developers debugging encountered memory leaks. Often, these developers cannot distinguish whether an observed reference chain is legitimate (such as when objects are kept in a cache in anticipation of future uses), or represents a leak. Third, the sheer size of the heap — large-scale server applications can easily contain tens of millions of objects — makes manual inspection of even a small subset of objects difficult or impossible.

Existing diagnosis tools are either online or offline. Online tools monitor either the state of the heap or accesses to objects in it, or both. They analyze changes in the heap over time to detect leak candidates, which are "stale" objects that have not been accessed for some time. Online tools are not widely used in production environments, in part because their overhead can make them too expensive, but also because the need to debug memory leaks often occurs unexpectedly after an upgrade or change to a framework component, and often when developers believe their code has been sufficiently tested. Offline tools use heap snapshots, often obtained post-mortem when the system runs out of memory. These tools find leak candidates by analyzing the relationships, types, and sizes of objects and reference chains. Most existing heuristics, however, are based solely on the amount of memory an object retains and ignore structural information. Where structural information is taken into account, it often relies on priori knowledge of the application and its libraries.

This paper presents a graph mining approach to identifying leak candidate structures in heap dumps. Our approach is based on the observation that leaks often involve container data structures from which programmers fail to remove unneeded objects that were previously added. Consequently, the heap dump involves many subgraphs of similar structure containing the leaked objects and their descendants. By mining the dump, we can identify those recurring subgraphs,

KDD'10, July 25–28, 2010, Washington, DC, USA.

present the developer with statistics about their frequency and location within the graph. Our key contributions can be summarized as follows:

1. Although analysis techniques are widely used in heap analysis [21–23], our work is the first to employ graph mining for detecting leaking candidates. Specifically, we demonstrate that graph grammar mining used in an offline manner can detect both seeded and known memory leaks in real applications.

2. Compared to other offline analysis techniques, our approach does not require any *a priori* knowledge about which classes are containers, or about their internal structure. It captures containers even when these are embedded into application classes, such as ad-hoc lists or arrays.

3. Our approach can identify leaks even if the leaks' locations within the graph do not share a common ancestor node, or if the paths from that ancestor to the instances are difficult to find by the manual examination that is required in existing tools such as Eclipse Memory Analyzer (MAT).

4. Graph grammar mining can find recursive structures, giving a user insight into the data structures used in a program. For instance, linked lists and trees can be identified by their distinct signatures.

5. Finally, the ability to combine subgraph frequency with location information makes our algorithm robust to the presence of object structures that occur naturally with high frequency without constituting a leak.

2. ANATOMY OF A LEAK

Although memory leaks can occur in all languages that use dynamically allocated memory, they are particularly prevalent in type-safe languages such as Java, which rely on garbage collection to reclaim memory. In these languages, dynamically allocated objects are freed only if the garbage collector can prove that no accesses to them are possible on any future execution path, which is true if and only if there is no path from a set of known roots to the object in the reachability graph. The reachability graph consists of nodes that represent allocated objects and edges that correspond to inter-object references stored in instance variables (fields). Roots, also known as garbage collection (GC) roots, are nodes whose liveness does not depend on the liveness of other heap objects, but on the execution semantics of the program. For instance, in Java, local and global static variables represent roots because the objects referred by them must remain reachable throughout the execution of a method or program, respectively. Hence, memory leaks form if objects remain reachable from a GC root even though the program will no longer access them. Such leaks also occur in programming languages with explicit memory management, such as C++, and our work applies to them. We do not consider leaks arising from memory management errors in those languages (e.g., failing to deallocate unreachable objects).

The Java program sketched in Figure 1 illustrates how leaks in a program manifest themselves in the reachability graph. In this example, a program maintains a number of objects of type `Legitimate`, which are stored in a hash map

```
public class HiddenLeak
{
  static HashMap legitimateMap;

  static class Legitimate
  {
    HashMap leakyMap = new HashMap();

    static class Leak
    {
      // This object is leaked
    }

    void leak()
    {
      // insert Leak instances into leakyMap
    }
  }

  public static void main(String []av)
  {
     // create N instances of Legitimate
     for (int i = 0; i < N; i++)
     { Legitimate legit = new Legitimate();
       legit.leak();
       legitimateMap.put(i, legit);
     }
  }
}
```

Figure 1: An example of a leak "hidden" underneath legitimate objects.

container. Each `Legitimate` instance, by itself, represents data the program needs to maintain and thus it needs to retain references to each instance. However, `Legitimate` also references a container `leakyMap` that accrues, over time, objects of type `Leak` that should not be stored permanently. Figure 2 shows the resulting heap structure. The hash maps exploit an open hashing scheme, which uses an array of buckets. Each bucket maintains a separate chain of entries corresponding to keys for which hash collisions occurred.

Over time, the space taken up by the leaked object will grow until all available heap space is exhausted. When this limit is reached, the Java virtual machine throws a runtime error (`OutOfMemoryError`). In many production environments, the JVM is run with a flag that saves a snapshot of the heap at this point, which is then fed to a heap analyzer tool.

Most existing tools compute and analyze the dominator tree, which represents a relationship between nodes in the heap that expresses which objects a given object keeps alive. Object 'a' dominates 'b' if all paths from a root to 'b' go through 'a'. If the object graph is a tree, as in this example, then it is identical to its dominator tree. In the example, the static (global) variable `legitimateMap` is the dominator tree root that keeps alive all leaked objects.

However, inspection of the dominator tree for this example with existing tools does not readily point to the leak. For instance, when examining a heap dump with the Eclipse Memory Analyzer tool (available at http://eclipse.org/mat), the tool pointed at 'legitimateMap' as a likely leak candidate, because it keeps alive a large fraction of the heap. None of its children stands out as a big consumer with respect to retained heap size. The leak is "hidden" under a blanket of legitimate objects. At this point, a developer would be required to "dig down," and individually examine

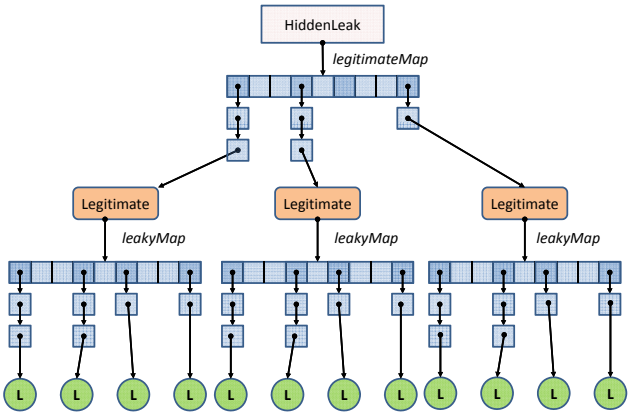

Figure 2: Heap graph after executing the program shown in Figure 1.

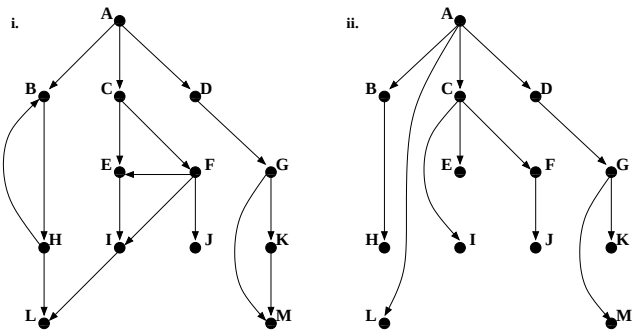

Figure 3: (i) An example graph and (ii) its dominator tree. In practice, the dominator will have a significantly reduced number of edges than the original graph.

paths through each Legitimate instance, which is cumbersome without global information about the structure of the subtrees emanating from these instances. Heap analyzers do support some global information, but is usually limited to histogram statistics that shows how often objects of a certain type occur. This approach often leads to limited insight because `String` objects and `char` arrays are the classes whose objects usually consume most memory in Java programs.

Consequently, there is a need to mine the graph to identify structures that are likely leak candidates, even if these structures are hidden beneath legitimate live objects. Moreover, developers require aggregate information that describe where in the object graph these leak candidates are located.

3. ALGORITHMS

As stated in the introduction, our approach is based on the observation that a leak would manifest as a heap dump containing many similar subgraphs. Rather than directly mine the heap dump, we compute the dominator tree of the heap dump and mine frequent graph grammars [15, 17] in the dominator tree. We describe the rationale behind this approach and algorithmic design decisions in this section.

3.1 Dominator Tree Computation

The computation of dominators is a well studied topic and we adopt the Lengauer-Tarjan algorithm [19] from the Boost graph library implementation. This algorithm runs in $O((|V| + |E|) \log(|V| + |E|))$ time, where $|V|$ is the number of vertices and $|E|$ is the number of edges.

Formally, the dominator relationship is defined as follows. A node x dominates a node y in a directed graph iff all paths to node y must pass through node x. In Fig. 3 (left), all paths to node L must pass through node A, therefore A dominates L. A node x is said to be the unique immediate dominator of y iff x dominates y and there does not exist a node z such that x dominates z and z dominates y. Note that a node can have at most one immediate dominator, but may be the immediate dominator of any number of nodes. The dominator tree $D = (V^D, E^D)$ is a tree induced from the original directed graph $G = (V^G, E^G)$, where $V^D = V^G$, but an edge $(u \rightarrow v) \in E^D$ iff u is the immediate dominator of v in G. Figure 3 shows an example graph and its dominator tree. The computation of D requires the specification of an entry node into G. We therefore introduce a pseudo-root node ρ to G, and add edges $(\rho \rightarrow GC_i)$ where $GC_i \in GC$, the set of Java garbage collection roots (see Section 2).

The dominator tree computation on the heap dump graph provides us with several benefits. First, it significantly reduces the number of edges in the graph to be mined. Second, since the resulting graph is a tree, we can apply optimizations in the mining algorithm specific to mining trees. Finally, recall that our goal is not just to find the frequent subgraph representing the leak but also to characterize the source of the leak. Once the dominator tree has been computed, we can search the paths from the entry node of the leaking subgraph up to the root of the dominator tree. This path is generally sufficient to identify the source of the leak. Without the dominator tree computation, tracing all paths to garbage collection roots in the graph is much more expensive and is full of noise.

3.2 Mining the Dominator Tree

In general, frequent subgraphs in the original heap dump need not necessarily be frequent in the dominator tree. To understand this, consider the cases in Fig. 4 which shows example graphs that are frequent in both the dump and the dominator, frequent in the dump but not the dominator, as well as the other two combinations. In particular, the (frequent in dump, infrequent in dominator) combination occurs due to the existence of different routes of entry into a frequent subgraph S in graph G. This situates S into different subgraphs in D that may not be frequent individually. The reverse combination, i.e., (infrequent in dump, frequent in dominator), also happens, because frequent subgraphs in D may contain edges that summarize dissimilar paths in G.

Nevertheless, practical heap dumps have specific degree distribution properties that we can exploit. As we show later, a large majority of nodes in heap dumps have an in-degree of either zero or one. This implies that cases as shown in Fig. 4 (top right) are much fewer in number than cases in Fig. 4 (top left). This is a key distinction because we can guarantee that if a frequent subgraph S in G contains only nodes having in-degree ≤ 1, all instances of S will be completely conserved after the computation of the domi-

nator tree D and will retain their frequencies. Although it is unlikely that a frequent subgraph in the heap dump will comprise *exclusively* of nodes with in-degree ≤ 1, experience shows that it will be composed predominately of such nodes. These observations justify our design decision to compute the dominator tree as a preprocessing step before graph mining and to mine frequent structures in the dominator.

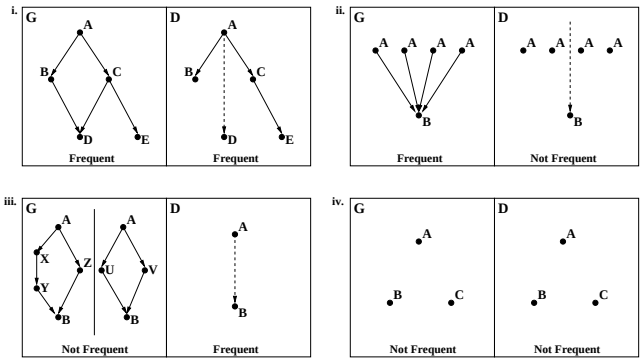

Figure 4: Examples of subgraphs that may be frequent or not frequent in either or both the graph G and its dominator tree D. (i-iv) show all 4 classes of subgraphs defined by the different combinations. Dashed edges represent a "dominates" edge produced only for D.

3.3 Inducing Graph Grammars

To mine patterns in the dominator tree, we explore the use of context-free graph grammars [10, 28] instead of mere subgraphs. Graph grammars are necessary because leaking objects are often recursive in nature and we require the expressiveness of graph grammars. Furthermore, it is not necessarily the number of instances of a subgraph that is important in debugging the leak, but rather the percentage of the heap dump that is composed of instances of the subgraph. An algorithm that finds subgraphs could be misleading because a simple count of the number of instances could over-calculate the composition of the subgraph.

The concept of a graph grammar is akin to a formal language grammar, except that the productions generate subgraphs rather than substrings. We are specifically interested in node-label controlled grammars, as these contain a single non-terminal labeled node which can be replaced with any subgraph. These grammars take the form of $S \rightarrow P$, where S is a single non-terminal node, P can be any subgraph, and the arrow implies a production/replacement rule.

As described in [9, 15, 17], we build graph grammars in *a priori* fashion where we find a production rule capturing a significant portion of the input graph, replacing instances of the production by its non-terminal symbol, and continuing. Candidate productions are evaluated for their ability to compress the graph. We use the alternative size heuristic from [17]:

$$\frac{\text{size}(G)}{\text{size}(S) + \text{size}(G|S)}$$

where $\text{size}(G) = |V| + |E|$. Figure 5 shows two example input graphs G, a top-scoring graph grammar S for each, and the result of compressing G using S (denoted by $G|S$). The first example shows how the graph grammar can describe the same information as a frequent subgraph. The second example demonstrates additional features of graph grammars. The graph grammar mining algorithm works iteratively, where in each iteration the top-ranked graph grammar is used to compress G. The next iteration repeats the process on the newly compressed version of G with non-terminal nodes. In practice, we run the algorithm for just a few iterations (≤ 3) because we focus on the top-ranked grammars when trying to identify a memory leak. For scalability reasons, we use sampling at several key states in the mining process.

1. **Candidate generation:** When we expand an evaluated pattern with k edges to a candidate pattern with $k+1$ edges, we need not generate and explore all candidates because the algorithm uses a beam search, which we limit to 15, to bound the number of patterns generated at each stage [17]. We take a random sample of $< 1\%$ of the instances of the extending pattern to determine the best candidates to fill the beam. The full set of instances will then be explored only for the best candidates determined by the sample.

2. **Scoring candidates:** When a new candidate graph grammar is generated, we must calculate the size heuristic of the candidate. Since it will not have been checked for recursiveness yet, we must ensure that we do not over-estimate the size in the case that instances overlap. Instead of checking all instances of the candidate, we estimate the size heuristic by looking at a random sample of $\sim 5\%$ of all instances to get an idea for how much overlap occurs within the grammar's population. This estimate is used to approximate the actual score.

3. **Recursive Opportunities:** Similar to the sampling technique we use for scoring candidates, we sample the candidate grammar's instances to determine if and how it is recursive. In this case, we use a sample of $< 1\%$ of the instances. We can afford to use a smaller sample size than in the scoring function because scoring requires a higher degree of accuracy.

Because the statistical significance of our sample would be largely dependent on the prevalence and recursive nature of a candidate graph grammar, we cannot generalize our method to all cases and we choose our sample sizes empirically without claiming statistical significance. However, our small sample sizes worked well in all of our experiments. We also note that once the top scored graph grammar is returned by the algorithm with sampling, we ensure that the recursion detection and compression of the input graph by the grammar are done exactly.

4. EVALUATION

Our evaluation contains three parts. First, we check whether our algorithm finds seeded structures in a set of synthetically created dumps. Second, we examine its efficacy on heap dumps we obtained from Java developers. Third, we report its scalability and performance with respect to the sizes of the heap graphs considered.

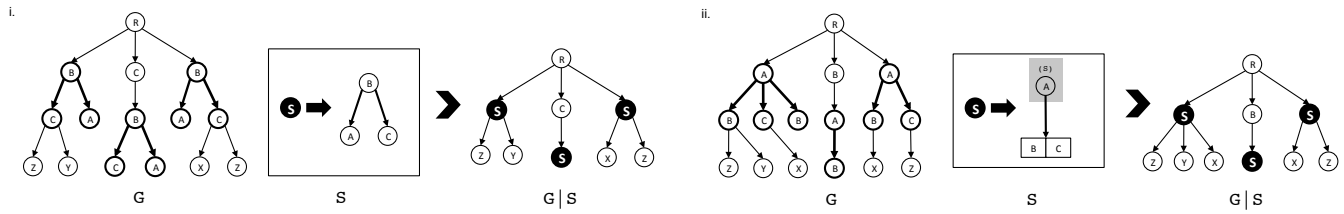

Figure 5: Two examples of an input graph G, a graph grammar S inferred from G, and the result of compressing G on S, denoted $G|S$. (i) A non-recursive grammar which does not contain any embedded non-terminal nodes or edges. (ii) Recursive grammar on node A containing an embedded non-terminal node that can match either nodes of type B or C. The grey box containing the "(S)" label represents a recursive connection instruction.

4.1 Preparing Heap Dumps

We obtained heap dumps from Sun's Java VM (version 1.6) using either the jmap tool or via the `-XX:+HeapDumpOnOutOfMemoryError` option, which triggers an automatic heap dump when the JVM runs out of memory. We use the `com.sun.tools.hat.*` API to process the dump and extract the reachability graph. Each node in the graph corresponds to a Java object, which is labeled with its class. Each edge in the graph corresponds to a reference, labeled with the name of the field containing the reference. We label edges from arrays with `$array$`, ignoring the specific index in which a reference is stored. We remove all edges that correspond to weak and soft references, because weak and soft references to an object do not prevent that object from being reclaimed if the garbage collector runs low on memory.

4.2 Synthetic Examples

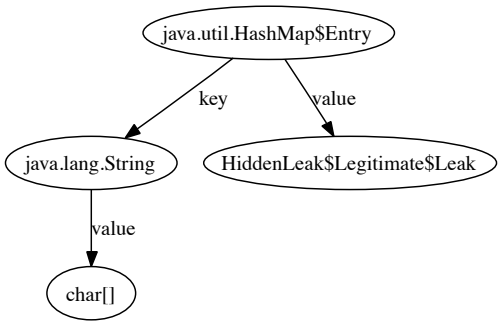

Figure 6: Most frequent grammar mined from HiddenLeak example in Figure 2. The `$` character denotes nested classes. For example, the node `HiddenLeak$Legitimate$Leak` corresponds to Java class `HiddenLeak.Legitimate.Leak` in the source code.

We first present the results for the motivating example presented in Section 2. Figure 6 shows the most frequently occurring mined grammar, which represents a (key, value) pair anchored by an instance of type `HashMap.Entry`. This information directs an expert's attention immediately to a `HashMap` mapping keys of type `String` to values of type `HiddenLeak.Legitimate.Leak`. We found that 70% of the paths from the instances produced by this grammar to GC roots in the original graph exhibit the following structure, where `java.lang.Class` (top) is a root node of the dominator tree, `java.util.HashMap$Entry` (bottom) corresponds to the top node in the best grammar displayed in Figure 6, and edge labels (in parenthesis) stem from the explicit and implicit variable names used in the source code from Figure 1:

```
java.lang.Class
| (legitimateMap)
+-+-> java.util.HashMap
| (table)
+-+-> [Ljava.util.HashMap$Entry;
| ($array$)
+-+-> java.util.HashMap$Entry
| (value)
+-+-> HiddenLeak$Legitimate
| (leakyMap)
+-+-> java.util.HashMap
| (table)
+-+-> [Ljava.util.HashMap$Entry;
| ($array$)
+-+-> java.util.HashMap$Entry
```

The remaining paths contain an additional edge `Entry.next`, which represents the case in which a hash collision led to chaining. This information immediately describes the location of all instances of `Leak` objects in the graph, alerting the developer that a large number of these structures has accumulated underneath each `Legitimate` object. As discussed in Section 2, a size-based analysis of the dominator tree as done in Eclipse's Memory Analyzer would lead only to the bucket array of the `HashMap` object referred to by 'legitimateMap' and require manual inspection of the subtree emanating from it.

Since programmers often choose container types depending on specific space/time trade-offs related to an expected access pattern, we then investigated if the leaking structure would be found if a different container type had been used. We replaced both uses of `HashMap` with class `TreeMap`, which uses a red-black tree implementation. Our algorithm correctly identified a grammar consisting of `TreeMap.Entry` objects that refer to a (key, value) pair, near-identical to the grammar shown in Figure 6. In addition, the aggregated path was expressed by the recursive grammar shown in Figure 7, which covers over 99% of observed paths from a root to the grammar's instances.

This path grammar identifies the leak as hidden in a tree of trees and provides a global picture that would be nearly impossible to obtain by visual inspection. The use of recursive productions enabled the algorithm to identify a classic container data structure (a binary tree) without any a priori knowledge. The use of recursive grammars is also essential for other recursive data structures, such as linked lists. The following example demonstrates that our mining approach

```
+-+-> java.lang.Class
| (legitimateMap)
+-+-> java.util.TreeMap
| (root)
{
   +-+-> java.util.TreeMap$Entry
   | ( right | left )
   +-+-> java.util.TreeMap$Entry
}*
| (value)
+-+-> leaks.TreeMapLeaks$Leak
| (leakyMap)
+-+-> java.util.TreeMap
| (root)
{
   +-+-> java.util.TreeMap$Entry
   | ( right | left )
   +-+-> java.util.TreeMap$Entry
}*
```

Figure 7: Resulting root path grammar if a TreeMap container is used for the example shown in Figure 2.

easily captures such data structures, even when they occur embedded in application classes (rather than in dedicated collection classes). Class `OOML` in Figure 8 embeds a link element `next` and an application-specific `payload` field. The main method contains an infinite loop which will add elements to a list held in a local variable 'root' until heap memory is exhausted.

```
public class OOML {
    OOML next;          // next element
    String payload;

    OOML(String payload, OOML next) {
        this.payload = payload;
        this.next = next;
    }

    // add nodes to list until out of memory
    public static void main(String []av) {
        OOML root = new OOML("root", null);
        for (int i = 0; ; i++)
            root = new OOML("content", root);
    }
}
```

Figure 8: A singly linked list embedded in an application class.

Figure 9 shows the most frequent subgraph, which contains a recursive production $OOML \rightarrow^{next} OOML$, representing a single linked list. The root path aggregation showed the location of its instances in the graph:

```
+-+-> Java_Local
| (root)
{
   +-+-> OOML
   | (next)
   +-+-> OOML
}*
```

4.3 Web Application Heapdumps

4.3.1 Apache Tomcat/J2EE

We obtained a series of heap dumps that resulted from recurring out-of-memory situations during the development of the LibX Edition Builder, a complex J2EE web application

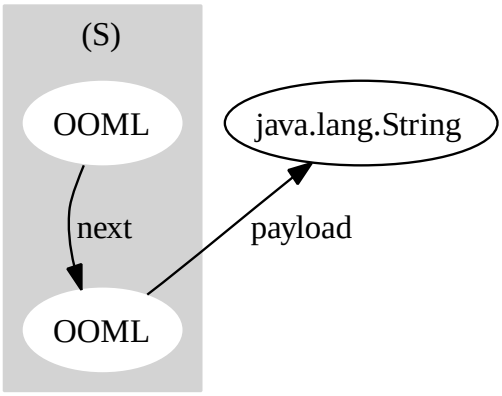

Figure 9: Most frequent grammar in synthetic linked list example.

that makes heavy use of multiple frameworks [2] including the Apache Tomcat 5.5 servlet container. These heap dumps were generated over a period of several months. When the server ran out of memory during intense testing, developers would simply save a heap dump and restart the server, without further immediate investigation of the cause.

We obtained a total of 20 heapdumps, varying in sizes from 33 to 47MB. In all of these dumps, the grammar shown in Figure 10 percolated to the top. This grammar represents instances of type `BeanInfoManager` that reference a `HashMap` through their `mPropertyByName` field. 80% of the root paths are expressed via the following grammar:

```
+-+-> org.apache.catalina.loader.StandardClassLoader
| (classes)
+-+-> java.util.Vector
| (elementData)
+-+-> [Ljava.lang.Object;
| ($array$)
+-+-> java.lang.Class
| (mBeanInfoManagerByClass)
+-+-> java.util.HashMap
| (table)
+-+-> [Ljava.util.HashMap$Entry;
| ($array$)
+-+-> java.util.HashMap$Entry
| (value)
+-+-> org.apache.commons.el.BeanInfoManager
```

This grammar shows that the majority of these objects are kept alive via a field named `mBeanInfoManagerByClass`. Since the field is associated with a node of type `Class`, it represents a static field. Examination of an actual path instance reveals that this static field belongs to class `org.apache.commons.el.BeanInfoManager`.

Similar to the HiddenLeak example, the Eclipse analyzer reported the (legitimate!) `HashMap.Entry` array stored in the `mBeanInfoManagerByClass` class as accumulation point, but could not provide insights into the structure of the objects kept in this table without tedious manual inspection. We eventually found that this leak had already been reported by another developer against the Tomcat Apache server (Bug 38048: Classloader leak caused by EL evaluation). Interestingly, the original bug report had received little attention, likely because the bug reporter included only a single trace to a leaked object reachable from the `BeanInfoManager` class.

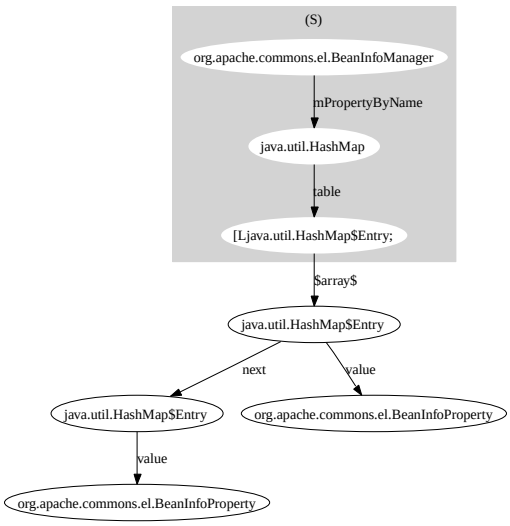

Figure 10: Most frequent grammar in Tomcat 5.5 heap dump.

This leak was subsequently fixed in a 6.x release of Tomcat. After updating the server, we periodically took heap dumps via the jmat tool. We subjected these heap dumps, which contain no known leaks, to our analysis. Unsurprisingly, the subgraph anchored by `HashMap.Entry` rose to the top, reflecting the ubiquitous use of hash maps. However, path aggregation showed these hash maps were located in very different regions of the object graph, thus making it less likely for them to be leaks.

4.3.2 MVEL

In a separate project, one of the authors developed a rule-based system for the application of software engineering patterns to enhance code [29]. During the development of this project, out of memory situations occurred when certain input was fed to the rule engine, which was written using the Drools rule engine framework. In this system, rules can contain expressions written in the MVEL scripting language (mvel.codehaus.org).

Mining the resulting heap dump showed the grammar in Figure 11, which contains a recursive production $RegExMatch \rightarrow^{nextASTNode} RegExMatch$. This mined grammar mirrors the synthetic linked list discussed in Section 4.2. All RegExMatch objects are contained in a single list held in a local variable:

```
+-+-> Java_Local
| (??)
+-+-> org.mvel.ASTLinkedList
| (firstASTNode)
{
   +-+-> org.mvel.ast.RegExMatch
   | (nextASTNode)
   +-+-> org.mvel.ast.RegExMatch
}*
```

This path indicates that the cause of the memory exhaustion was the unbounded growth of a singly-linked list of `RegExMatch` objects, likely due to a bug in the MVEL parser. Although this information does not directly lead to the underlying bug, it rules out a number of other scenarios, such as memory exhaustion due to a large object kept alive by a long list of `RegExMatch` objects.

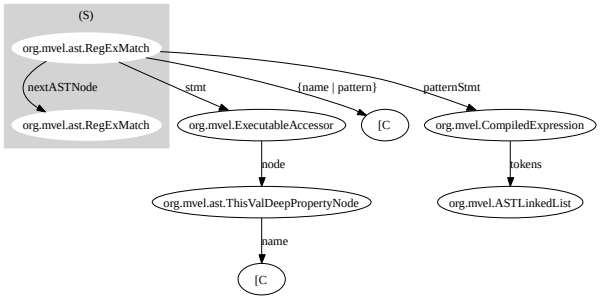

Figure 11: Most frequent grammar in MVEL heap dump.

4.4 Scalability and Other Quantitative Aspects

In this section, we study some quantitative aspects of our graph mining approach to illustrate its effectiveness at mining heap dumps. First, we study the indegree distribution of nodes from 24 real (i.e., not synthetic) heap dumps. Assessing the % of nodes that have indegree ≤ 1 across these dumps, we obtain statistics of a minimum of 84%, an average of 89%, and a maximum of 99.8% (from the MVEL dumps). This suggests that assessing frequent subgraphs in the dominator tree should not cause significant loss of information as compared to the original heap dump. At the same time, Table 1 illustrates the reduction gained in the number of edges and the overall size of the graph by choosing to focus on the dominator tree.

Fig. 12 illustrates the time taken for diagnosing leaks as a function of the size of the dominator. This time does not include loading and pre-processing (removal of weak and soft references) and computation of the dominator. It does include the time to mine the top (best) graph grammar, for graph reduction, and for summarizing root paths. The lower cluster of points are drawn from the Tomcat/J2EE dumps. These are processed faster because they do not involve recursive constructs and there are fewer instances of the mined grammar in the dump. We see that these are processed in

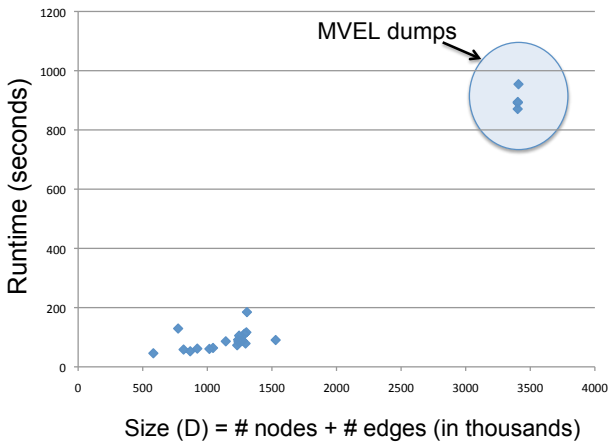

Figure 12: Runtime statistics on real heap dumps.

Table 1: Summary statistics of some of the heap dumps analyzed in this work. Each heap dump is named by the type of leak it contains and the date it was created. Dumps labeled as 'maintenance' were taken before the production server was shut down for maintenance; they appear to be healthy and leak-free.

Name	Memory (MB)	# nodes	# edges (G)	# edges (D)	% edge reduction	% size reduction	Grammar size (avg)
tomcat.jul03	39	713076	1243152	536628	57%	36%	10.3
tomcat.jul05	41	732089	1288883	555829	57%	36%	7.0
tomcat.jun02	33	603941	1035000	440201	57%	36%	11.7
tomcat.jun04	33	588559	974322	427347	56%	35%	9.7
tomcat.may02	41	745434	1442146	561377	61%	40%	31.0
tomcat.may15	40	788482	1545443	609702	61%	40%	31.0
tomcat.sep20	40	734027	1295724	561956	57%	36%	10.3
tomcat.oct20	36	655251	1133970	487573	57%	36%	14.3
tomcat.nov05	43	742729	1336582	559642	58%	37%	24.3
tomcat.oct24	41	707913	1218834	533087	56%	36%	11.3
tomcat.nov03	42	715032	1202226	522893	57%	35%	23.7
tomcat.nov06	42	710730	1250427	535371	57%	36%	25.7
tomcat.oct28	42	717464	1260223	540283	57%	36%	9.0
tomcat.oct06	38	700479	1212796	530758	56%	36%	10.3
tomcat.oct03	47	854365	1584715	674692	57%	37%	11.0
tomcat.oct14	40	722417	1278437	550516	57%	36%	10.3
maintenance.feb06	34	321786	464017	260367	44%	26%	15.0
maintenance.nov08	35	519435	635638	348284	45%	25%	11.0
maintenance.nov17	35	547759	689958	374595	46%	25%	11.0
maintenance.nov09	34	493408	577236	322739	44%	24%	14.7
mvel.feb12	69	1704284	3832262	1698297	56%	39%	15.0
mvel.feb13	69	1704286	3832264	1698296	56%	39%	13.0
mvel.feb14	69	1704810	3832897	1698405	56%	39%	13.0
mvel.feb19	69	1707258	3837542	1701496	56%	39%	15.0

about 2–3 minutes. Conversely, the MVEL dumps involve significant recursion and several hundreds of thousands of instances. Furthermore, the path summarization for the MVEL constructs require greater work since we must traverse up the linked list to observe the root path for even a single instance.

Finally, we compared the runtime of our algorithm when used on the dominator tree versus the original heap dump graph. We chose one particular dump, tomcat.sep20 from Table 1, to make the comparison. We used this dump because the Tomcat leak graph grammar does not display significant recursion and this dump is one of many good representatives of the leak class. In order to compare the runtimes, we excluded the path summarization step that was included in the runtime plot in Fig. 12 because this step would not be comparable between graphs. We note that the preprocessing step of removing weak and soft references will differ between graphs as well (in fact it is less complex in the dominator tree), but the resulting graphs are comparable. Further, we found that the most frequent graph grammar in the dominator tree is a subgraph of the most frequent graph grammar in the full heap dump graph, thus requiring more iterations of candidate generation and therefore more runtime for discovery. To enable a fair runtime comparison, we considered the time required to generate the most frequent single-edge graph grammar from the dominator tree versus from the full graph. We found that the dominator tree accomplished this task in 21 seconds versus 38 seconds in the full graph. We also compared the complete runtime for the graph grammar in the full heap dump graph for tomcat.sep20 versus the runtime on a dominator tree from a similar heap dump, tomcat.nov05, because the discovered graph grammar from the dominator tree in tomcat.nov05 was closer in size to that found in the full heap dump graph for tomcat.sep20 and composed of the same leak – but was actually larger and more frequent. We found that in the dominator tree the runtime was 130 seconds versus 426 seconds in the full heap dump graph. Our results show that we obtain $\sim$46% runtime reduction for identifying small graph grammars and $\sim$69% runtime reduction for large graph grammars when the percentage of size reduction is only 36%. This suggests that mining the dominator tree is not only quicker due to size reduction, but also because its tree structure contains less noise and redundancy, thereby simplifying the mining process.

5. RELATED WORK

Our research combines ideas from software engineering and graph mining. We discuss related work in each of these areas.

5.1 Memory Leak Detection Tools

One of the first systems to debug leaks exploited visualization, allowing a user to interactively focus on suspected problem areas in the heap [26]. Most recent existing leak detection tools use temporal information, including object age and staleness, that is obtained by monitoring a program as it runs. For instance, IBM's Leakbot [21–23] acquires snapshots at multiple times during the execution of a program,

applies heuristics to identify leak candidates, and monitors how they evolve over time.

Minimizing both the space and runtime overhead of dynamic analyses have been the subject of intense study. Space overhead is incurred because object allocation sites and last access times must be recorded; runtime overhead because this information must be continuously updated. Bell and Sleigh [3] use a novel encoding to minimize space overhead for allocation sites. To minimize the runtime cost, statistical profiling approaches have been developed [13]. Cork [16] combines low-overhead statistic profiling with type-slicing. Some profilers, notably the NetBeans profiler, use information already kept by generational collectors to determine object age. Lastly, hardware support for monitoring memory access events has been proposed in [30].

By contrast, our approach explores mining information from a single heap dump, which is often the only source of information available when out-of-memory errors occur unexpectedly, which is the common case in production environments in which dynamic tools are rarely deployed. Our work is complementary to dynamic approaches. Mined structural information is likely to enhance information these tools can provide, especially in the common scenario in which software engineers diagnose suspected leaks in codes with which they are not familiar. Moreover, the ability to identify data structures could be exploited to automatically infer which operations are add/delete operations on containers, which could benefit approaches that rely on monitoring the membership of object containers to identify leaks [33].

In the context of languages with explicit memory management, several static analyses have been developed that identify where a programmer failed to deallocate memory [8, 14, 25, 32]. Similarly, trace-based tools such as Purify [12] or Valgrind [24] can identify unreachable objects in such environments. By comparison, the garbage collected languages at which our analysis aims do not employ explicit deallocation; we aim to identify reachable objects that are unlikely to be accessed in the future. Lastly, rather than eliminating the source of leaks, some systems implement mitigation strategies such as swapping objects to disk [4].

5.2 Graph Mining

Graph mining is a well studied field that expands far in both breadth and depth. Initial works such as [18, 35] focused on the problem of discovering frequent subgraphs and these algorithms have been expanded in several directions over the past decade [6, 7, 34, 36, 37]. Building upon FP-tree data structures [11], fast data structures [1] have also been developed. Similar to our work, graph mining algorithms have been tailored toward specific application domains where the structure of the desired subgrahs can be exploited in the discovery process.

Cook and Holder's work [9] takes a different approach to graph mining by finding a single, "best" subgraph as opposed to all subgraphs frequent above some threshold. This approach uses a scoring function based on the minimum description length (MDL) principle and is therefore computationally more complex. This work has also been expanded upon for the use of finding highly descriptive graph grammars [15, 17] which consider the recursive nature of the subgraphs and allow for variability in the edge and node labels.

Graph grammars are synonymous with other formal language grammars, with the difference being that the "sentences" being generated are connected graphs. Graph grammars have an array of applications, but have generally been researched from a theoretical perspective for graph generation [10, 28] as opposed to inference problems as studied here. The benefit of graph grammars is that they can capture richer information about the connectivity of subgraphs than traditional frequent subgraphs. Although these graph grammars are primarily context-free and therefore lossy, they provide a more descriptive representation of a subgraph than just a frequency count. Our work follows the graph grammar philosophy but applies it toward the characterization of dominator trees, which has not been studied before.

5.3 Data Mining for Software Engineering

Data from programming projects (code, bug reports, documentation, runtime snapshots, heap dumps) is now so plentiful that data mining approaches have been investigated toward software engineering goals (see [31] for a survey). Graph data, in particular, resurfaces in many guises such as call graphs, dependencies across subprojects, and heap dumps. Graph mining techniques have been used minimally for program diagnosis. For instance, program behavior graphs have been mined for frequent closed subgraphs that become features in a classifier to predict the existence of "noncrashing" bugs [20]. Behavior graphs were also mined with the LEAP algorithm [34] in [5] to identify discriminative subgraphs signifying bug signatures. However, to the best of our knowledge, nobody has investigated the role of mining heap dumps for detecting memory leaks or used a graph grammar mining tool.

6. CONCLUSION

We have presented a general and expressive framework for diagnosing memory leaks using frequent grammar mining. Our work extends the arsenal of memory leak diagnosis tools available to software developers. For the KDD community, we have introduced the notion of dominators and how they possess sufficient statistics for mining certain types of frequent subgraphs. The experimental results are promising in their potential to debug leaks when other state-of-the-art tools cannot. We expect our algorithm to be used in complement of tools like Eclipse MAT.

Our future work revolves around three themes. First, we seek to embed our algorithm in a runtime infrastructure so that it can track leaking subgraphs as they build up over time. Second, we seek to investigate the theoretical properties of dominators and whether they can support a frequent pattern growth [11] style of subgraph mining. This approach will allow us to process larger heap dumps than our current approach. Third, we plan to perform a quantitative evaluation comparing the quality of our reports to existing tools. Such quantitative comparisons require the definition of a metric, which could be derived by approximating the number of lines of code a user would have to investigate to verify the presence or absence of a bug, as proposed in [27].

Acknowledgements

This work is supported in part by US NSF grant CCF-0937133 and the Institute for Critical Technology and Applied Science (ICTAS) at Virginia Tech. Also, we would like to thank Jongsoo Park, the developer of the dominator tree

algorithm used in the Boost C++ libraries, for his ready responses to our questions and comments.

7. REFERENCES

[1] M. Akbar and R. Angryk. Frequent pattern-growth approach for document organization. In *CIKM '08*, pages 77–82, 2008.

[2] A. Bailey and G. Back. LibX–a Firefox extension for enhanced library access. *Library Hi Tech*, 24(2):290–304, 2006.

[3] M. Bond and K. McKinley. Bell: bit-encoding online memory leak detection. In *ASPLOS-XII '06*, pages 61–72, 2006.

[4] M. Bond and K. McKinley. Tolerating memory leaks. In *OOPSLA '08*, pages 109–126, 2008.

[5] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *ISSTA '09*, pages 141–152, New York, NY, USA, 2009.

[6] H. Cheng, X. Yan, J. Han, and P. Yu. Direct discriminative pattern mining for effective classification. In *ICDE '07*, pages 169–178, 2008.

[7] C. Chent, X. Yan, F. Zhu, and J. Han. gApprox: Mining frequent approximate patterns from a massive network. In *ICDM '07*, pages 445–450, 2007.

[8] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07*, pages 480–491, 2007.

[9] D. Cook and L. Holder. Substructure discovery using minimum description length and background knowledge. *JAIR*, 1:231–255, 1994.

[10] J. Engelfriet and G. Rozenberg. Graph grammars based on node rewriting: an introduction to nlc graph grammars. In *Graph grammars and their application to computer science: 4th Intl. Workshop*, pages 12–23, 1991.

[11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD '00*, pages 1–12, 2000.

[12] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in c and c++ programs. In *Winter USENIX Conference*, pages 125–138, 1992.

[13] M. Hauswirth and T. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI '04*, pages 156–164, 2004.

[14] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI '03*, pages 168–181, 2003.

[15] I. Jonyer, L. Holder, and D. Cook. MDL-based context-free graph grammar induction and applications. *IJAIT*, 13(1):65–79, 2004.

[16] M. Jump and K. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *POPL '07*, pages 31–38, 2007.

[17] J. Kukluk, L. Holder, and D. Cook. Inference of node and edge replacement graph grammars. In *ICML Grammar Induction Workshop '07*, 2007.

[18] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM '01*, pages 313–320, 2001.

[19] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[20] C. Liu, X. Yan, H. Yu, J. Han, and P. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *SDM '05*, pages 286–297.

[21] N. Mitchell. The runtime structure of object ownership. In D. Thomas, editor, *ECOOP '06*, 2006.

[22] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In *ECOOP '03*, 2003.

[23] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA '07*, pages 245–260, 2007.

[24] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, pages 89–100, 2007.

[25] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Lecture Notes in Computer Science*, volume 4134, pages 405–424. Springer, 2006.

[26] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in java. *Concurrency - Practice and Experience*, 12(14):1431–1454, 2000.

[27] M. Renieris and S.Reiss. Fault localization with nearest neighbor queries. In *ASE '03*, pages 30–39, 2003.

[28] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. 1997.

[29] E. Tilevich and G. Back. Program, enhance thyself! demand-driven pattern-oriented program enhancement. In *AOSD '08*, pages 13–24, April 2008.

[30] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA '07*, pages 273–284, 2007.

[31] T. Xie, S. Thummalapenta, D. Lo, and C. Liu. Data Mining for Software Engineering. *IEEE Computer*, Vol. 42(8):35–42, Aug 2009.

[32] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *ESEC/FSE-13*, pages 115–125, 2005.

[33] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE '08*, pages 151–160, 2008.

[34] X. Yan, H. Cheng, J. Han, and P. Yu. Mining significant graph patterns by leap search. In *SIGMOD '08*, pages 433–444, 2008.

[35] X. Yan and J. Han. gSpan: graph-based substructure pattern mining. In *ICDM '02*, pages 721–724, 2002.

[36] X. Yan and J. Han. CloseGraph: mining closed frequent graph patterns. In *SIGKDD '03*, pages 286–295, 2003. 956784.

[37] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Out-of-core coherent closed quasi-clique mining from large dense graph databases. *ACM TODS*, 32(2), 2007.