LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications

Nick Mitchell, Gary Sevitsky

IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532 USA
{nickm,sevitsky}@us.ibm.com

Abstract. Despite Java's automatic reclamation of memory, memory leaks remain an important problem. For example, we frequently encounter memory leaks that cause production servers to crash. These servers represent an increasingly common class of Java applications: they are *large scale* and they make *heavy use of frameworks*. For these applications, existing tools require too much expertise, and, even for experts, require many hours interpreting low-level details. In addition, they are often too expensive to use in practice. We present an automated, adaptive, and scalable tool for diagnosing memory leaks, called LeakBot.
LeakBot incorporates three new techniques. First, it automatically ranks data structures by their likelihood of containing leaks. This process dramatically prunes the set of candidate structures, using object reference graph properties and knowledge of how leaks occur. Second, it uses Co-evolving Regions to identify suspicious regions within a data structure and characterize their *expected* evolution. Third, it uses the first two methods to derive a lightweight way to track those regions' *actual* evolution as the program runs. These techniques are mutually beneficial: we need only monitor what is highly ranked, and, because the tracking is so cheap, a region's rank can be continually updated with information from production machines. Finally, this whole process can be done without user assistance.
We demonstrate LeakBot's effectiveness on a number of large-scale applications that we have analyzed as part of the ongoing consulting practice our group maintains. We have found that the ranking analysis scales (e.g. written in Java, it analyzes 10^6 objects in 30 seconds with a 300M heap), is selective (e.g. it prunes that set to three candidate leak roots), and is accurate (it discounts non-leaking roots). The CER generation completes in tens of seconds. The lightweight tracking refines the rankings, while lowering throughput by less than 5%.

1 Introduction

Despite automatic garbage collection, memory leaks remain an important problem for many Java applications. A memory leak occurs when a Java program inadvertently maintains references to objects that are no longer needed, preventing the garbage collector from reclaiming space. Memory leaks are easy to spot, but are often difficult to diagnose. We can determine that a memory leak is likely to exist by using a black-box analysis, monitoring the heap after each round of garbage collection. We observe a downward-sawtooth pattern of free space (every collection frees less and less) until the application grinds to a halt for lack of space.

A number of diagnosis tools exist that help the user look inside the black box to determine the root cause of a leak. They rely on a combination of heap snapshot differencing [26,19,30], and allocation and/or usage tracking at a fine level of detail [27,29,11,4,21,24]. We have used tools in both of these categories as part of an active consulting practice our group maintains helping solve memory leaks in large stand-alone Java applications and in IBM customer *e-Business* systems (web-based transaction processing systems built on an application server framework, such as J2EE[1] [28]). We have found that these techniques are not adequate for these large-scale applications.

In our experience, to diagnose a memory leak, the user must look for a set of candidate data structures that are likely to have problems. Finding the right data structures to focus on is difficult; as we will see in Section 2, when exploring the reference graphs of large applications, issues of noise, complexity, and scale make this a daunting task. For example, e-Business servers intentionally retain a large number of objects in caches. Existing approaches require that the user manually distinguish these cached objects from truly leaking ones. In general, these approaches swamp the user with too much low-level detail about individual objects that were created, and leave the user with the difficult task of interpreting complex reference graphs or allocation paths in order to understand the larger context. This interpretation process requires a lot of expertise; even for experts, it usually takes many hours of analysis work to find the root cause of a leak. Moreover, these techniques will in some cases perturb the running application too much to be of practical value, especially in production environments.

1.1 A summary of the contributions of LeakBot

We propose a way around these difficulties: raise the level of analysis from individual objects to regions within data structures. This approach has two beneficial consequences. First, it enables automated discovery and simple presentations of what is really causing the leak. Second, it enables lightweight and automated tracking of how whole data structures evolve. To realize these benefits, we have developed three new techniques: a way to rank data structures automatically by their likelihood of containing leaks, a way to identify suspicious regions within a data structure and characterize their *expected* evolution, and a lightweight way to track those regions' *actual* evolution as the program runs. We present an implementation of these techniques in an automated and lightweight memory leak

[1] J2EE may more accurately be described as a large collection of libraries and frameworks, covering many different areas of functionality such as database access, session management, directory access, asynchronous messaging, etc.

detection tool called LeakBot. We demonstrate that these techniques work well on several large-scale Java applications. Furthermore, we explain how they are especially powerful when used in combination.

Our first step in finding leaks is to identify the few data structures in which leaks are likely to be occurring. We introduce the concept of a *leak root*, which is the head of a data structure containing regions exhibiting unbounded growth. Section 3 discusses why finding candidate leak roots is not straightforward: data structures are complex, and their properties do not have a simple linear effect on the importance of that data structure. We present a method for ranking candidate leak roots which combines, in a non-linear fashion, a collection of structural and temporal properties of the object reference graph. This method uses no knowledge of any particular framework. Section 3.6 demonstrates the effectiveness of the model on three representative IBM customer applications. For example, on a customer form-processing server application, it finds two candidate leak roots from a heap containing one million live objects. In addition, it does so quickly: the ranking process completes in under thirty seconds.

Under one leak root there may be multiple *regions* evolving in different ways. In Section 4 we introduce the notion of Co-evolving Regions, as a way to identify these distinct regions and concisely model the essence of their evolution. To this purpose, we introduce the *owner-proxy* and *change-proxy*, waypoints along each member's path from the leak root. We show how these waypoints are useful for a number of purposes. First and foremost, they classify members into Co-evolving Regions, and allow us to rank regions according to whether they leak. They are also useful for summarizing the structural highlights and severity of each region's growth for the user.

The two previous steps work by analyzing two snapshots[2] of the reference graph taken early in the run. A user could stop there, and operate in an *off-line* mode. But, in order to refine the results of those previous steps, it is sometimes necessary to acquire more information as the application runs. However, in practice, taking additional full reference graph snapshots either often, or late into an application's run, is far too expensive on large-scale applications.

In Section 5 we show how to acquire additional information *selectively*, using the results of the previous steps. In this *on-line* mode, we can be selective in two ways. First, we need only monitor the few most highly ranked regions. Second, we need only track a small subset of an entire region in order to determine how the entire region evolves. We show how to use the owner-proxy and change-proxy to derive a short path that LeakBot can periodically traverse to detect how a region evolves. The number of hops in this traversal is very small in relation to the size of large, leaking data structures.

LeakBot uses the on-line findings to refine the analyses and user presentation. For example, it characterizes each Co-evolving Region according to its actual evolution, and tells the user which containers are growing (i.e. are likely sources

[2] A reference graph snapshot is a list of currently live objects, including, for each object, its identifier, data type, and outgoing references. By live we mean objects that are not collectible by the garbage collector.

of leaks), which are alternating in size (e.g. a cache with a flux of constituent objects, or a database connection pool), and which have reached a plateau (e.g. a data structure fully initialized at application startup time). It uses the characterization trend to update the rankings, and also presents this information to the user to assist in understanding the dynamic behavior of each region. Finally, the updated rankings allow LeakBot to adaptively adjust the frequency with which it explores each region.

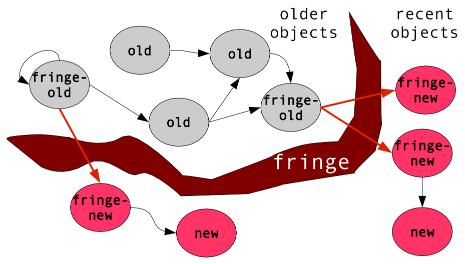


Fig. 1. The *fringe* is the boundary between older and recently created objects.

We have implemented LeakBot in two components. The ranking and CER generation occur in an *analyzer* that runs independently of the target application. The *tracing agent* attaches to the application using JVMPI [15]. In on-line mode, the agent takes the initial snapshots and periodically samples the CERs.

2 Diagnosing Leaks in Large Java Applications

The applications that motivated the work in this paper have properties, common to many Java applications today, that make memory leak diagnosis especially difficult. These applications make heavy use of reusable frameworks and libraries,[3] often from many sources. These *framework-intensive* applications contain large amounts of code where the inner workings are not well understood by the developers, let alone those doing the problem determination. Server-side e-Business applications make use of particularly large frameworks, and introduce additional analysis difficulties due to their high degree of concurrency, scale, and long-running nature.

Before LeakBot, we used existing tools to help diagnose leaks as part of our consulting practice. HPROF [29] works by categorizing each object according to its allocation call path and type, as shown in Table 1. Many tools work by dividing the heap into old and new objects, under the assumption that older objects are expected to be more permanent. The user manually tries to discover why newer, supposedly temporary objects are being retained, by exploring what we call the *fringe*, the boundary between old and new objects, as shown in

[3] For convenience we will use the term framework to mean a framework or library.

	percent		live		alloc'ed		stack	class
rank	self	accum	bytes	objs	bytes	objs	trace	name
1	97.31%	97.31%	10280000	10000	10280000	10000	1995	byte array
2	0.39%	97.69%	40964	1	81880	10	1996	object array
3	0.38%	98.07%	40000	10000	40000	10000	1994	MemoryConsumer
4	0.16%	98.23%	16388	1	16388	1	1295	character array
5	0.16%	98.38%	16388	1	16388	1	1304	character array

Table 1. The output of HPROF on a toy example. It divides the heap objects based on allocation site (`stack trace`) and type.

Figure 1. Below, we describe the many difficulties we encountered when using these tools.

Perturbation: In tracking the call stack of every allocation, HPROF can reduce the throughput of a web-based application by five to ten times. Heap differencing tools that acquire full heap snapshots late into a leak can cause a system with a large heap size to pause for tens of seconds. For servers these slowdowns or pauses can cause timeouts, significantly changing the behavior of the application. On production servers, this level of service degradation is completely out of the question.

Noise: Given a persisting object, it is difficult to determine whether it has a legitimate reason for persisting. For example, caches and resource pools intentionally retain objects for long periods of time, even though the objects may no longer be needed. This is especially relevant to e-Business applications, where numerous resource management mechanisms (such as database connection pools and web page fragment caches) are used behind the scenes to ensure good transaction performance. Some other common examples of noise are: session information that is retained for a fixed time period in web-based systems, in case the user returns later; containers that have lazy removal policies; objects that appear to persist only because they are part of a transaction that was in progress when the application's state was captured. Noise can be especially problematic when diagnosing slow leaks in long-running systems; noise effects can dwarf the evidence needed to diagnose a slow leak until very late in the run.

Existing techniques provide little assistance in this area. An aggregate view, such as that shown in Table 2, dividing the heap into old, new, and fringe objects, gives us little insight for determining which objects exist due to the flux in and out of caches, which are from a transaction in progress, and which are leaks. Existing approaches leave the user with the hard work of digging through reference graphs or call stacks manually. Users must rely on their own, often limited knowledge of how the application and frameworks manage their data, in order to segregate objects by their likely usage.

Data Structure Complexity: Knowing the type of leaking object that predominates, often a low-level type such as String, does not help explain why the leak occurs. This is because these Strings are likely to be used in many contexts, and even may be used for multiple purposes within the same high-level

class name	old	new
java.lang.ref.Finalizer	20246	17084
java.lang.String	223266	9453
xerces...TextImpl	9035	7676
character array	202782	5290
xerces...AttrImpl	6258	5135
object array	17165	3255
java.util.Hashtable$Entry	56745	3244
xerces...NamedNodeMapImpl	3667	2713
xerces...ElementImpl	3204	2123
integer array	4410	2064
java.util.Vector	6394	1993
xerces...DeferredTextImpl	960	1209
java.util.ArrayList	215	1151
com.bank...log.Record	1	1045

class name	fringe-new
java.util.HashMap$Entry	322
java.lang.String	243
java.util.Hashtable$Entry	95
com.ibm...CredentialsImpl	20
com.bank...MessageModel	20
byte array	15
character array	14
xalan...KeyTable	12
java.util.Hashtable	11

Table 2. Number of non-collectible instances in a leaking customer Websphere application. We have divided the heap as shown in Figure 1.

data structure, such as a DOM document. In addition, presented with the context of low-level leaking objects, it is easy to get lost quickly in extracting a reason for leakage. A single DOM object contains many thousands of objects, with a rich network of references among them. Without knowledge of the implementation of frameworks, it is difficult to know which paths in the reference graph to follow, or, when analyzing allocation call paths, which call site is important.

3 Finding Candidate Leak Roots

Consider a buggy e-Business application where each transaction places items into a global `ActiveOrders` structure, but forgets to remove some of them when the transaction is complete. In this simple example, shown in Figure 2, `Book`'s are removed properly, but `CD`'s are inadvertently left connected.

We may distinguish between a data structure that contains a leak, in this example `ActiveOrder`'s, and the actual leaking substructures, in this case the `CD` objects and everything they point to. In general, a single data structure may contain more than one different type of leak, in addition to regions that are stable or are in flux but not growing. In this section we describe a technique for discovering the overall data structures that are likely to contain leaks. Our eventual goal is the discovery and characterization of the leaking regions themselves, which we describe in Section 4 and 5.

Definition 1. *A* leak root *is the object at the head of a data structure which is leaking in one or more ways.*

Our approach to finding leaks is to first identify candidate leak roots. We do this by ranking each object based on a mixture of structural and temporal

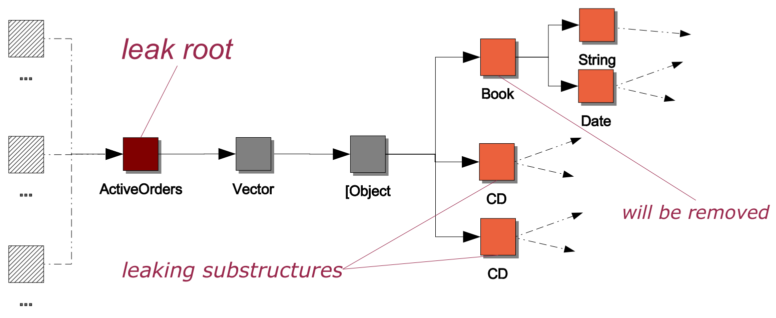


Fig. 2. If every transaction leaks a `CD` object, then `ActiveOrders` is probably the best *leak root*: the most indicative, highest-level object responsible.

properties of the object reference graph, using a small number of snapshots gathered while the application is running. Each candidate leak root may then be used as a proxy for a data structure containing leaks, and ultimately as one of the attributes describing each leaking region.

3.1 An overview of leak root ranking

A number of requirements influence the design of our ranking scheme. First, it must be a good discriminator of leak roots. The discovery process should not propose many more candidates than actual leaks in the program. In addition, it is not enough for the ranking to be merely an ordering; it must be a meaningful ranking as well. If one object is much more likely than another to be a leak root, this should be reflected in those objects relative ranks.

Second, the ranking must be resilient to the time at which the snapshots were taken. We would like the ranking to do a good job with input taken early in a programs run. This allows for quicker turnaround in test environments. and it is a practical concern for production settings, where taking snapshots late in a run with a severe memory leak can be prohibitively expensive.

Finally, it is important that the ranking scale to large object reference graphs, both in its memory and time consumption. We achieve this by filtering many objects down to a small set of candidates in a succession of three steps, each utilizing different criteria. Each ranking step applies an increasingly expensive algorithm to a successively smaller set of candidate leak roots. Each algorithm assigns a number between zero (definitely not a candidate) and one (a highly probable leak root), and each subsequent step only applies to objects ranked above a chosen threshold. The final rank of an object indicates its goodness as a leak root. We term this final rank the `leakroot` rank, or $\mathcal{L}$.

In our example above, the algorithm would identify the `ActiveOrders` object as a likely leak root. A number of considerations lead us to rank this object higher than other objects in the graph (for example, higher than the `Vector`). The ranking algorithms which led to this choice are the realization of the following four observations.

Observation 1 (Binary Metrics) *Structural and temporal reference graph attributes can definitively rule out many objects, but definitively rule in none.*

We can easily eliminate some of the objects in Figure 2 from any further consideration. For example, in Java, array objects do not automatically grow. Therefore, we can rule out the object array, `[Object`. Section 3.3 provides a collection of binary (yes or no) metrics which eliminate a large number of objects from further consideration. We will show how eight binary metrics typically reduces the set of candidates from a million to a few hundred objects.

Of the remaining candidates, however, we cannot with absolute assurance assign them a rank of one. Based on a few heap snapshots, we cannot know that the application will not eventually remove the `Order` objects from the `Vector` (just as e-Business applications eventually clear out cached user sessions). At best, we can say the an object is a good candidate and, with additional evidence, that it is increasingly likely to be so. Also, for the reasons of noise described in Section 2, there are typically many such *possibly* leaking structures in the reference graph. We should prioritize those not immediately ruled out.

Observation 2 (Mixture Metrics) *For those objects not ruled out yet, some reference graph attributes are positive indicators, in favor of candidacy, and some are negative indicators. But no one attribute either stands out or always applies.*

Data structures and leaks have many forms, thus the importance of each attribute varies. Consider the importance of the size of a candidate, treated as a data structure.[4] Increasing size is a positive indicator, in favor of candidacy. But it is not always indicative in every leak situation, since not all big data structures leak. For example, we analyzed a customer's business-to-business gateway application. Typical of many applications, the top five data structures by size were all caches and resource pools. They ranged from 200kB to 1.5MB in size. The known leak root, on the other hand, was (at one point) only 64kB large. In another typical example, an e-Business form processing application with two leaks, one leak showed up as the largest data structure, while the other leak showed up only as the 85th-largest; the second case turned out to be a slow leak.

Observation 3 (Gating Functions) *Some positive indicators are much more positive than others; ibid for negative.*

If the binary metrics prune the candidates down to a hundred, the mixture model must do better than just ordering those hundred by likelihood. Instead, we starkly differentiate those that are very likely from those that are less so. We accomplish this differentiation by applying nonlinear gating functions to the values of the reference graph attributes.

For example, one criterion that helps rank an object is the number of objects it owns which are referenced by on-stack variables. Owning such objects is a negative indicator, because it implies that this data structure is changing in size

[4] Note that this alone would be an advance over existing tools.

only because the heap sample happened to capture some operations in progress. We'd like this indicator to follow a very sharp curve: owning just a few on-stack roots should highly discount the candidate; owning none shouldn't discount the candidate at all. This is an example of applying a "low-pass" gating function to a reference graph attribute. Section 3.4 shows how other attributes are gated with high-pass or band-pass gates.

Observation 4 (Fixpoint Metrics) *The rank of an object depends on the rank of other objects.*

We have found two main cases when metrics based only on reference graph attributes are insufficient. First, when one data structure leaks, all of its enclosing leak as well. However, if the *only* reason we think the enclosing data structures leak is due to that one data structure, then we have falsely identified multiple leak roots for a single leak. However, there are common cases where, looking only at the members and reachability of a candidate leak root, we will be left in this situation (for example, when a candidate has multiple parents).

The second reason stems from the need to combat the noise effects described in Section 2. Consider a leak of the form that objects of type B leak under an object A, and where each B is itself a complex data structure which is populated during (but not after) a transaction. Therefore, if a graph snapshots is acquired concurrently with transactions, then it will appear as if objects of type B are leaking: e.g. in one snapshot they are empty (newly created), and in the second they are fully populated. In this common scenario, object A will appear to leak (because of the true leak of B's into A); but a large number of B's will also be identified, falsely, as candidate leak roots by the attribute-based metrics.

3.2 Reference Graph Attributes

The first two ranking steps use metrics based on a collection of reference graph attributes. While most of these attributes have a well-understood meanings [17], we define them here, to avoid confusion.

single-entry equivalence Given an arbitrary graph G, we compute a reduced graph G' where a node in G' represents all nodes in G in the same single-entry (but not necessarily single-exit) region. The edges are collapsed in the obvious way. In the applications we have studied, the collapsed graph has about one eighth as many nodes of the original graph. For example, many character arrays are each pointed to by a single `String` object; we can collapse each pair of objects into a single node in G'.

GC roots Those objects referenced by sources other than fields of Java objects. Examples of these GC roots include references from local variables currently on the Java or native stack, JNI references from native code, references from currently-held monitors.

reachability The *reachers* of an object o is the union of all paths from some set of objects to o. To make this computation efficient, the ranker computes reachability on the single-entry collapsed graph. In addition, rather than computing all-points reachability, the ranker only computes a small reachability vector. Each element of the vector counts the number of GC roots of a particular type that reach that single-entry subgraph.

unique ownership One object *o dominates* o' if any path from a GC root which includes o' also includes o. In the other direction, the objects *uniquely owned by* an o is the set of all objects dominated only by it; we denote this by M_o. Again, so that this analysis scales, the ranker computes dominance on the single-entry reduced graph (see Appendix A).

age The age of an object is the snapshot in which that object was first witnessed by LeakBot. The fringe of an object reference graph is the set of objects in the latest generation immediately pointed to by objects in earlier generations. We say an object is *on the fringe* if it is a new object pointed to by an older one. In this discussion, we say an object is *new* if it comes from the latest generation, and otherwise it is *older*.

size We distinguish between the allocation size and the data structure size of an object. The latter is the total size of its uniquely owned objects.

3.3 Binary Metrics (ranking step 1)

The step-1 rank of a candidate object is the *product* of the eight metrics of that object. Each metric is computationally easy-to-compute and each takes on a value of zero or one. Thus, if any metric assigns a value of zero, then that the object is not a candidate. Otherwise, the binary metrics are "agnostic" to that candidate, and it is passed on to the next ranking step. The following binary metrics evaluate to zero for objects with certain structural (S_1 through S_4) and temporal (T_1 through T_4) reference graph attributes. We show how, together, these metrics quickly eliminate most objects from further consideration.

Binary Metrics based on Structural Graph Attributes

S_1 **leaf nodes**: these objects cannot possibly be the root of a leaking data structure. Note that a leaf node may eventually point to another object, and commence leaking. But we rely on the fact that it has not yet.

S_2 **arrays**: objects which are arrays; in Java, arrays are allocated with a fixed size, therefore, a leak involving growth of an array must have that array as part of a larger data structure (which reallocates the array when it reaches its maximum size).

S_3 **internal nodes**: objects which aren't the head of a single-entry region. From every single-entry region, we choose one (the head) as a representative of that region, and disregard the rest. For example, the `Vector` in Figure 2 is filtered out using this property. In many cases, this will keep us from identifying more than one leak root for the same leak.

S_4 **non-owners**: objects which uniquely own nothing. An object may be a non-leaf node, but only share ownership of objects with many others. These objects tend to be located close to GC roots, such as class loaders. We ignore them, and instead favor the objects they point to (directly or indirectly) which *do* own objects. The parents of the `ActiveOrders` object in Figure 2 have this property.

	# objects	fraction remaining			
		S_1	S_2	S_3	S_4
`phone company`	267,956	67%	59%	9%	6%
`IDE`	350,136	61%	55%	9%	7%
`brokerage`	838,912	65%	62%	7%	3%
`brokerage2`	1,015,112	71%	70%	2%	1%
`finance`	1,320,953	60%	56%	11%	8%

Table 3. For five applications, this table shows the *cumulative* effectiveness (from left to right) of the four binary metrics based on structural graph attributes.

Binary Metrics based on Temporal Graph Attributes

T_1 **no age intersection**: the object owns only older, or only new objects. If we have witnessed no objects added to a data structure in any of the reference graph snapshots, then this object is a very likely the owner of a pool, or some other unchanging structure. Likewise, if we see no older objects as of the latest graph sample, then we very likely have caught a transient data structure, perhaps due to an in-progress transaction. In either case, we can safely ignore this object.

T_2 **new arrays only**: the object owns only new object arrays, but no new objects inside those arrays. For example, an empty hashtable used only during program initialization may still own a large, but empty array.

T_3 **no fringe**: the object owns no objects on the fringe. Some objects may own both new and older objects, but they own none on the fringe. This is likely to be an artifact of shared ownership. To avoid these artifacts, we favor the objects which own both older, new, *and* fringe objects.

T_4 **no datatype intersection**: the set of data types of older owned objects intersected with the same for new objects is empty. For example, a generic object cache may contain ten strings in one reference graph sample and fifteen integers in a later sample. This data structure passes most of the other binary filters, but nonetheless isn't very likely to be leaking.

Table 3 and Table 4 give five examples of the effectiveness of the binary metrics, for the structural and temporal metrics respectively. These five examples come from engagements we've been involved with; four are large IBM customers, and one (IDE) is an internal IBM application. In each case, the input to the binary metrics was a pair of full reference graph snapshots. We had warmed up the applications various amounts (e.g. with `financial` we warmed up the system

			number remaining			
	# objects	S_i	T_1	T_2	T_3	T_4
`phone company`	267,956	16,346	73	73	72	29
`IDE`	350,136	25,653	99	99	29	10
`brokerage`	838,912	26,291	97	82	81	67
`brokerage2`	1,015,112	12,020	102	102	64	17
`finance`	1,320,953	106,900	579	519	518	242

Table 4. The cumulative filtering effectiveness (from left to right) of the four binary metrics based on temporal graph attributes (T_1, T_2, T_3, T_4). S_i shows the number of objects left after applying all four structural metrics; see Table 3.

with only five minutes of typical load, whereas for `brokerage2` we warmed up the system with 30 minutes of typical load). We took the first snapshot, performed additional load (roughly the same as the warmup load), and took the second snapshot.

On the `finance` application, the second snapshot had around 1.3 million live objects, and the combination of the eight binary metrics filtered out all but 242 objects. This number is somewhat higher than for the other applications because we had warmed up the application for a much shorter period of time than for the others. Nonetheless, the binary metrics are effective. As we discussed earlier, resilience to input early in a program's run is an important design criteria. Our experience has shown that, given input from early in a program's run, the binary metrics typically filter down to several hundred candidates.

3.4 A Mixture of Gated Metrics (ranking step 2)

Of the (typically) several hundred remaining candidates, not all are equally likely. Thus, we rank them by the unweighted sum of a collection of gated metrics.

As pointed out in Observations 2 and 3, no one metric is an overwhelming indicator of candidacy, but selected reference graph attributes can be very strong negative indicators. To reflect this observation, we *gate* each attribute. The particulars of each gating function depend on the attribute, but each has the following characteristics. For extreme values of the attribute, gates are either strongly against or agnostic to that object's candidacy (but never strongly in favor). If an attribute has a strongly negative extreme, the gate assigns a negative rank. By agnostic, we mean that, all other things being equal, we should assign the object a rank of one. In-between, the gates use a superposition of cubic exponential gating functions to implement either high-pass, low-pass, or band-pass filters on the attribute's value. Space does not permit a full exposition of each attribute's gate, but we describe them for several of the following attributes.

G_1 **on-stack ownership**: We discount data structures that are growing only because we caught operations in progress based on the number of objects owned that are referenced by on-stack GC roots.

G_2 **on-stack reachability**: We discount those objects reachable from on-stack roots, because the entire data structure may be transient.

G_3 **ownership counts**: S_4 has already filtered out objects which own nothing. Here, we favor objects which own both a greater number and size of objects. We consider number and size separately: owning one large array isn't as indicative of problems as owning many smaller objects. But comparing two objects which own the same number, we somewhat favor the one of larger data structure size.

G_4 **new ownership**: T_1 has already filtered out objects which own no new objects. Here, we favor objects which own a greater number of newer objects.

G_5 **array ownership**: The larger the number of object arrays compared to objects, the less likely the candidate. Also, if a data structure contains no object arrays, we have found it to be less likely (though not entirely unlikely) to be a root of leaks. Therefore, for this criterion, the number of object arrays in a data structure, we must apply a band-pass gating function to array ownership: not too large a fraction of object arrays, and not too small.

G_6 **fringe ownership**: If an object owns many objects on the fringe, that is a sign that the leak is progressing quickly. All other things being equal, we favor these candidates over others.

G_7 **fringe datatype uniformity**: Single leaks tend to have a fairly uniform datatype on the fringe. [5] If there is only a plurality of datatypes on the fringe, this is an indication either that this data structure may have multiple leaks, or that it is a general-purpose data structure with a constantly-changing constituency (like a cache). In the former case, we'd like to favor the smaller data structures which contain the individual leaks (if not heavily discounted by other metrics). We want to ignore the latter case entirely.

G_8 **datatype intersection**: As explained earlier, we strongly discount objects without high overlap in owned datatypes from one sample to the next.

G_9 **dominance frontier**: Data structures that are highly embedded in larger ones tend not to leak. Rather, leaking data structures extend ownership all the way down to graph leaf nodes. Therefore, we discount an object which owns many objects with a non-empty dominance frontier.

3.5 Iterative Fixpoint Ranking (ranking step 3)

Finally, LeakBot updates the step-2 rank to account for the interactions identified in Observation 4. We account for interactions using an iterative algorithm which inflates or discounts the rank of one object based on its rank relative to the rank of related candidates. The algorithm starts with all objects whose rank, so far, lies above a specified threshold (see Appendix B). It then iterates until no candidate's rank changes appreciably. We have found that, in nearly every case, no more than three iterations are required.

Initially, the step-3 rank of every object equals its step-2 rank. At each iteration, choose a candidate o, and compute the three metrics from o.

[5] This is the *change proxy* introduced in Section 4.

F_1 **immediate domination residue**: the sum of the step-3 ranks of each object o immediately dominates.

F_2 **by-type immediate domination residue**: as F_1, but sum the maximum by datatype.

F_3 **immediate dominator residue**: the maximum of the step-3 ranks for every object in immediate dominators from, but not including o.

Let r_o be the current step-3 rank of o. Update r_o as follows. If $F_1 \approx 0$, then no sub-structures are better candidates than the current object; continue to the next iteration with no changes. If $F_1 = r_o$, then o is a candidate mainly because exactly one of its sub-structures is a good candidate; discount r_o by 50%. Otherwise (if $F_1 > r_o$) multiple of o's sub-structures contribute to o's candidacy; if $F_2 = F_1$ then there are two independent problems in sub-structures, and so discount r_o by 50%; otherwise, we are witnessing the falsely-identified leaks described in Observation 4, and so discount each of the falsely identified candidates by 50%. We perform similar updates based on F_3. If $F_3 = 0$, then no larger structure is a good candidate, so continue with no changes. If $F_3 \gg r_o$, then there is an enclosing data structure which is a much better candidate than o; discount r_o by 50%.

3.6 Examples of the Ranker in Use

We have applied LeakBot in off-line mode to a variety of applications, both large GUI applications and e-Business applications. We have used it for a number of purposes: to diagnose known leaks, to check whether an application has leaks before shipping it, and to verify that fixes for known leaks do in fact work. Here, we share three of these experiences. In each of these examples, the input to LeakBot was a trace containing two snapshots of the heap, with a number of suspected leaking operations separating the two snapshots.

Discovering and diagnosing a leak In this example, we analyzed a large GUI integrated development environment, heavily dependent on frameworks, for leaks. We tested opening and closing an editor window, thinking that this should be a "round trip" scenario: all resources for the editor window should go away when it is closed. We performed a total of three operations: we warmed up the IDE with two operations, took a heap snapshot, then performed one more operation, and finally took a second heap snapshot. Table 5(a) shows that, from 350 thousand live objects, the ranker chooses only three with non-zero `leakroot`, and only one with `leakroot` above 0.5. We were surprised to find these highly-ranked suspects. It turns out that LeakBot had identified a previously unreported leak. With a 90MB heap, the structural computation takes 5 seconds, and the metric computation takes another 5 seconds (on a 1.2GHz Pentium3-M).

For comparison, we enabled HPROF, and exercised the application similarly. We started the application with the option `-Xrunhprof:heap=sites`, and used the IBM 1.3.1 JVM. After we had issued eighteen leaking operations, we examined the HPROF output. Recall that HPROF aggregates allocations by call site

class name	$\mathcal{L}$
(*)WorkbenchPage	0.719
WidgetTable	0.446
ResourceBundle	0.31

(a) `IDE`

class name	$\mathcal{L}$
WidgetTable	0.430
DeltaDataTree (#1)	0.322
DeltaDataTree (#2)	0.320

(b) `IDE- bug fixed`

class name	$\mathcal{L}$
DDRMain	0.396
ibm.LogUtil	0.265

(c) `auction- no leak`

class name	$\mathcal{L}$
APCache	0.830
TemplateCache	0.805
AntiVirus	0.757
Record	0.596
(*)XSLTransform	0.582

(d) `brokerage`

class name	$\mathcal{L}$
(*)EventNotifier	0.848
ibm.CachedTargets	0.579
(*)FormProperties	0.572

(e) `brokerage2`

class name	$\mathcal{L}$
ibm.CachedTargets	0.271
ibm.ORB	0.234

(f) `brokerage2-bug fixed`

Table 5. Examples of the leak root ranking, showing the objects with highest rank ($\mathcal{L}$), with those at the head of actual leaks annotated (*).

and type. It ranks these aggregations by the number of non-collectible bytes due to that call site and type. Table 6 shows the top five, all of which are primitive arrays. The first leaking application-typed aggregate, of type `StyleRange`, has rank 45. However, given the rate at which this application was leaking `StyleRange` objects, we would have to perform around 200 leaking operations before its aggregate floated to the top. In contrast, as we have shown, LeakBot is robust to the quality of the input: `leakroot` does well with many fewer extant leaks: after only three leaking operations, it has identified the leak as the top suspect.

Checking for a leak before shipping In this case, we applied the ranker to a high-volume e-Business application. We now know this application is leak free. At the time, however, we needed to verify this before the application went into production. We applied the ranker to a previously acquired trace, collected while the application was running a workload mix of various web transactions. Of seven hundred thousand objects, the ranker assigns 11 a non-zero `leakroot`; it assigns only two objects a `leakroot` above 0.25, and none above 0.4. Table 5(c) shows the `leakroot` metric for those two objects. With a 300MB heap, the structural analysis (computing the reduced object reference graph, and the dominator and reachability relations) took 10 seconds, the metric computation took 15 seconds.

Verifying that a fix to a known leak works Our third demonstration is from a leaking e-business form-processing application. The developers had already implemented fixes to two leaks, but wanted two types of assurance: first, that the patches indeed fixed the problem, and second, that there were no

	percent		live		alloc'ed		stack	class
rank	self	accum	bytes	objs	bytes	objs	trace	name
1	5.27%	5.27%	639600	39	2279600	139	1522	character array
2	4.57%	9.84%	554488	7339	559752	7540	2262	character array
3	4.35%	14.20%	528192	6762	589504	7294	1530	character array
					...			
640	0.01%	85.45%	1152	18	1152	18	19766	EditorManager$Editor

Table 6. A subset of the output of HPROF on the IDE application from Table 5(a). The head of the structures which are leaking is ranked 640th.

remaining leaks. The customer could not afford to discover, after deploying the fixes and running in production for several days, that there were still leaks. We first applied LeakBot's ranker to the server running a known-buggy version of the code. Table 5(e) shows the result: from one million live objects, the ranker finds ten with non-zero `leakroot`, five with `leakroot` above 0.3, and only three above 0.5. With a 300MB heap, the structural and metric computations took 15 seconds each (on a 1.2GHz Pentium3-M). The same analysis applied to the fixed code appears in Table 5(f). This time, the ranker assigns 9 objects a non-zero `leakroot`, and it assigns no objects a `leakroot` greater than 0.3.

4 Co-evolving Regions: patterns within leaking structures

Section 3 identifies leaks by finding candidate leak roots, objects which head data structures that possibly contain leaks. However, there are several reasons why this information is too coarse. For example, one leak root may identify more than one leak. In addition, leakage isn't the only way a data structure can change. There is a variety of ways in which evolution happens. For example, one data structure can have distinct regions that evolve as leaks (grow without bound), as caches or pools (bounded size, changing constituency), that may never change (e.g. a preloaded data structure), that may shrink (e.g., if used for initialization), or that may switch between these various types of evolution. This section refines from the level of a data structure to the level of *regions* within that structure.[6] We desire to identify regions that are as big as possible, but that still evolve in a single, coherent way. We term such regions *Co-evolving Regions* (CERs).

Coherency of evolution is determined by several factors. First, the region should only exhibit one type of evolution: either monotonic growth, monotonic shrinkage, bounded-changing constituency, and bounded-fixed constituency. Second, as a region evolves, different of its elements are, or once were, on the fringe. Those fringe elements must share a similar to each other. Third, all members of a region must share a similar relationship to the region's leak root.

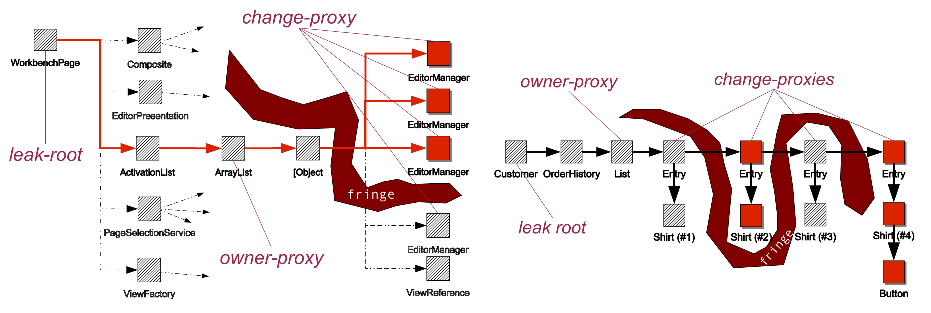
This section presents a mechanism for finding *likely*[7] Co-evolving Regions. To find CERs, we develop an equivalence relation for objects owned by a leak

[6] In [7], they discuss the "geometry of containment".

[7] Likely, not certainly. Section 5 shows how to adapt regions as the program runs.

root. To every object owned by a candidate leak root we assign a Region Key, a tuple of features reflecting the important components of equivalence. We define similarity of Region Keys, and classify a leak root's members based on Region Key similarity. Finally, we describe how to prioritize the Co-evolving Regions using a simple ranking algorithm.

4.1 Region Keys



(a) (b)

Fig. 3. The leak-path does not solely indicate co-evolution. The elided path of leak root, owner-proxy, and change-proxy is a better indicator.

Two members are part of the same CER based in part on their paths from their leak roots. As there may be many such paths, we identify one.

Definition 2. *The* leak-path*, P_m, of m owned by leak root o is the reverse of the path of immediate dominators from the m to o.*

The entire leak path is too rigid a specification to be useful for classifying objects into regions. For example, Figure 3 shows the structure of two leaks, an array leak (from the application covered in Section 3.6) and a linked list leak. In the array example, both the `EditorManager` objects and all their constituents should be part of the same CER, and yet their paths are, in large part, different. In the linked list example, even the leak-paths of only the highest-level leaking objects (the `Entry`'s) can be very different. And yet, in both examples, members of one region have *somewhat* similar leak-paths.

Eliding leak-paths via the owner-proxy and change-proxy. While the leak-path in its entirety does not indicate co-evolution, there is an elided version which does. We identify important "waypoints" in every leak-path, that indicate similarity of evolution [12]. Every object between the waypoints is effectively a wildcard for determining in which region a member belongs. The only parts of the path which do matter are the leak root, two concepts we now introduce: the owner-proxy, and the change-proxy.

The owner-proxy is a stable object on the old side of, and in close proximity to, the fringe. The change-proxy is that indicator of updates to the region; for this, we choose the largest stable object on the new side of the fringe. For example, Figure 4 shows the leak-paths for several of the objects in Figure 3(b). It illustrates how the waypoints define wildcarded subpaths of the each leak-path. The change-proxy for the `Shirt` and `Button` objects is the same, because, every leak to that region is indicated by the addition in a similar way.

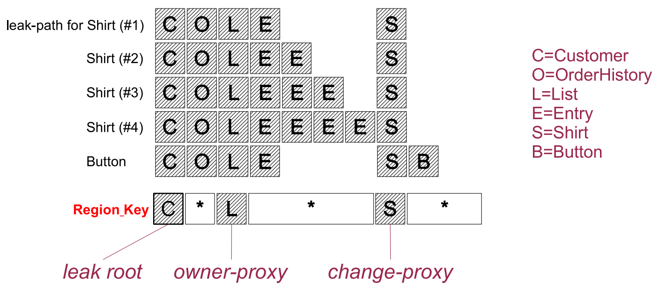


Fig. 4. Some leak-paths from Figure 3(b). The Region Keys for the `Shirt`s and `Button`s are the same, so they are part of one Co-evolving Region.

Definition 3. *The* distance-from-fringe *of an object o on a leak-path P_m is the number of hops from o to an object on the other side of the fringe. It is positive for objects on the old side of the fringe, and negative for those on the new side. We'll denote this by $d_{o,m}$. Also let $c_{o,m}$ be the number of fringe crossings along P_m from m to o. Finally let t_o be the expected lifetime of object o.*

For example, along the highlighted leak-paths in Figure 3(a), `[Object` has a distance of 1, and the `EditorManager` objects have distance -1. When computing the Region Key for m, we have found that a simple model of expected lifetime works very well in practice. Assume that arrays and objects with the same data type as the chosen change-proxy have an expected lifetime of 0, that new objects have 1, and that old objects have 10.

Definition 4. *The* change-proxy *of a new member m, C_m is that object o in P_m that maximizes $-t_o/d_{o,m}/(1+c_{o,m})$. The* owner-proxy, *O_m, maximizes $t_o/d_{o,m}$.*

Consider finding the owner-proxy and change-proxy for the `Shirt` object in the linked list example shown in Figure 3(b). We use the lifetime model defined above. It's leak-path contains all seven objects in the figure. In determining the change-proxy, the ratios specified in the above definition for each element in P_m are $(-0.833, 1.25, -2.5, +0.333, -5, +1, +0.5)$, indicating the best choice of change-proxy is the right-most `Entry` object. For the owner-proxy, the ratios specified in the above definition for each element in P_m are $(+3.33, +5, 0, 0, 0, 0, -0.125)$ indicating the best choice of owner-proxy is the `List` object. A similar process for the leak in Figure 3(a) will determine that the `ArrayList` from is the owner-proxy of every `EditorManager` object: the `[Object` array is closer to the fringe, but has a much shorter expected lifetime, and the objects further upstream have equal lifetimes but larger distances.

Definition 5. *The Region Key of an object m belonging to leak root L is the tuple (L, O_m, C_m). The Region Key for m and m' are equal (i.e. objects m and m' belong to the same co-evolving region) if $L = L'$, $O_m = O_{m'}$, and the datatype of C_m is the datatype of $C_{m'}$.*[8]

Using Region Keys to find Co-evolving Regions. For every leak root candidate L whose rank lies above a desired threshold, we compute Co-evolving Regions as follows. To L we associate a set of regions. To each region, we associate two numbers to measure a region: the total number of bytes which belong to that region, and the number of distinct data structures within that region. The latter is a useful metric, because it estimates the number of leaking operations which led to that region's current constituency.

Definition 6. *The* dump-size *of a region is the number of Region Keys that map to that region (using Definition 5's). The* proxy-size *of a region is the number of distinct change-proxy objects over all Region Keys which map to that region.*

Then, for each $m \in M_L$ whose P_m has spans the fringe (i.e. $c_{L,m} > 0$), do the following. Compute m's Region Key as described above, and insert it into L's region set. If an equivalent Region Key already exists, increment that region's dump-size. If an equivalent Region Key with the same change-proxy exists, increment that region's proxy-size.

For example, using this process, in the IDE application from Section 3.6, the highest-ranked candidate leak root (the `WorkbenchPage` object) has two regions. The proxy-size of the known leaking region is precisely the number of leaking operations that had been performed: three (recall from that section that we had performed three suspected leaking operations).

4.2 Ranking Regions

Finally, not all regions are equally likely to leak, so we rank them. When comparing one region to another, we consider three criteria. First, if one region's leak root has been ranked higher than another, this influences the relative ranking of the regions similarly. Second, if one region has a higher proxy-size than another, we favor the larger one. We do not use dump-size, because we'd prefer to rank based on an estimate of the number of leaking operations which have

[8] More generally, either C_m is assignable to the datatype of $C_{m'}$ or vice versa.

been performed, rather than the byte-size of the leak. Finally, if one region's proxy-size is growing faster than another's, we favor the faster one. This third criterion allows for updating the region ranks as we gather more information from the running application. A region's rank is the the unweighted average of these three elements. Unlike the leak root ranking described in Section 3 where a root's rank was bounded at one, we allow a region's rank to grow without bound. This allows for differentiating regions based on their leak rate, whereas bounding at one would asymptote all leaking regions to the same rank.

5 Cheap, Adaptive, Online Exploration of Regions

We have described the analyses of Sections 3 and 4 assuming an off-line usage scenario: acquire snapshots, find candidate leak roots, and then find co-evolving regions within highly-ranked candidates. We could very well stop here. An immediate benefit of having found regions that are likely to co-evolve is that LeakBot can present a high-level *schematic* of the suspected problematic regions of the reference graph. However, we'd like to know more than just the structure of problematic regions. We'd also like to know how those regions actually evolve.

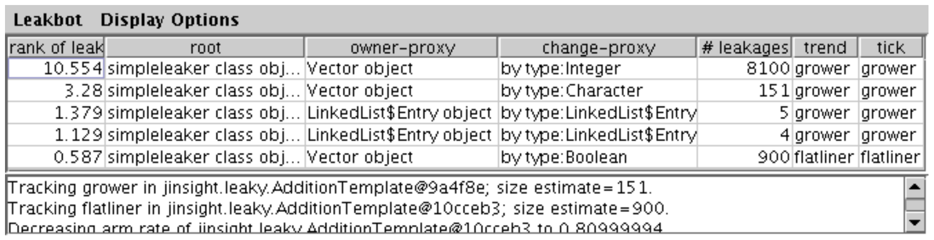
In off-line mode, our estimates of actual evolution are limited by the information in initial snapshots. Recall that we desire to acquire initial snapshots as early into the run as possible. On the other hand, if LeakBot remains connected to the program under analysis, it can present a more refined view of how regions actually evolve.

A principal constraint of LeakBot is that it must minimally perturb the program's behavior. Region Keys, in addition to helping us find Co-evolving Regions, can also help us derive lightweight probes to discover how these regions actually evolve. To this gather information, the LeakBot agent periodically traverses selected subgraphs of the object reference graph of the running application. It reports important structural changes back to the analyses of the previous sections. With the updated analysis (closing the feedback loop), we update the traversals as described below.

For example, to efficiently detect leaks of data structures into an array, we needn't keep track of every element in those leaking data structures. Instead, it is sufficient to periodically examine the references from the array, to a depth of one. There is no need to look any deeper into each leaking data structure, since we can just count the array contents by datatype. However, there is another case we must consider. In Java, an array is of bounded size. Thus, if the Co-evolving Region has monotonic growth, we would expect occasional reallocations of the array; when adding to an `ArrayList`, the underlying array is a transient object. Therefore, we must start the traversal from `ArrayList`, not the array. Observe that the same traversal also detects elements having been removed from the array. In addition, it can be used to inform us of when a *relinking* has occurred — that is, when one path element has been replaced by a new one. Observe that this traversal (in this case) follows precisely the path between owner-proxy and change-proxy.

This example shows that, to know how a Co-evolving Region evolves, we must derive a set of traversals that detect certain *updates*: additions, removals, and relinking. In some cases, one traversal can detect more than one of these updates. If, when doing the actual traversal, we witness an evolution, we say that update has been detected.

For every region, we keep a histogram of detected updates. We use this to estimate a region's evolution trend. For example, if only addition updates have been detected for a region, we say that region is a monotonic grower. If a roughly equal mixture of addition and removal updates have been detected, we classify the region as an *oscillator*. If only removal updates, it is a *shrinker*. If no updates, then it is a *flatliner*. Figure 5 shows the actual tool in action. The table has one row per Co-evolving Region, and indicates for each its current proxy-size and evolution trend.



rank of leak	root	owner-proxy	change-proxy	# leakages	trend	tick
10.554	simpleleaker class obj...	Vector object	by type:Integer	8100	grower	grower
3.28	simpleleaker class obj...	Vector object	by type:Character	151	grower	grower
1.379	simpleleaker class obj...	LinkedList$Entry object	by type:LinkedList$Entry	5	grower	grower
1.129	simpleleaker class obj...	LinkedList$Entry object	by type:LinkedList$Entry	4	grower	grower
0.587	simpleleaker class obj...	Vector object	by type:Boolean	900	flatliner	flatliner

```
Tracking grower in jinsight.leaky.AdditionTemplate@9a4f8e; size estimate=151.
Tracking flatliner in jinsight.leaky.AdditionTemplate@10cceb3; size estimate=900.
Decreasing arm rate of jinsight.leaky.AdditionTemplate@10cceb3 to 0.80999994
```

Fig. 5. Screenshot of the LeakBot tool. Each row corresponds to one suspect region. For each region, we show its region rank ("rank of leak"), the proxy-size ("# leakages"), and a summary of the trend and tick of that region's evolution.

Note that, in some cases, such as Figure 3(b), that traversal would be much longer than necessary: as the list in that example grows, so does the traversal. The solution to this problem of finding *efficient* traversals involves defining a family of short traversals which explore the fringe as it evolves. Space does not permit discussion of this topic, and therefore we leave it to future work. Nonetheless, the owner-proxy and change-proxy can still be used to define these traversals without a full dump of the object reference graph.

5.1 Tracing Adaptively

LeakBot first publishes the traversals to the tracing agent. The tracing agent spawns a thread that cycles through the unique traversals and periodically (once per second) performs at most one every time it wakes up. The agent assigns, to each unique traversal, a *sample bias* which lies between 0 and 1, and is initially 1. The bias is the probability that, when a traversal's turn comes up, the agent will actually perform the traversal. For every traversal, the agent determines whether any of that traversal's associated updates have been detected. It reports any detected updates (i.e. as an element having been added, removed, or relinked) back to the analyzer. For example, when an addition update is detected, the analyzer updates the proxy-size of that region.

LeakBot adaptively adjusts the sample bias of the traversals. Since we are interested in tracking leaks, we increase the bias whenever an addition template fires, and decrease it whenever no template fires, and decrease it even more so when a removal template fires. LeakBot ensures that no CER is completely ignored, in case the CER's mode changes at some point.

5.2 Implementation of the Tracing Agent

LeakBot works with either full reference graph snapshots acquired earlier, or it selectively acquires this information via a live connection. LeakBot's analyzer can parse previously acquired trace files in the Sun heapdump format, the IBM heapdump format, or the Jinsight [18] format. The agent relies on the JVMPI [15] profiling interface to gather information from the JVM.

JVMPI identifies objects by their memory address. Therefore, to maintain unique object identifiers in between initial snapshots, the agent needs to listen to object move and free events. This slow down garbage collection by as much as a factor of two. Luckily, LeakBot allows this interval to be very short. Once we have identified CERs, we no longer listen to move and free events. Instead, we use weak references to maintain persistent identifiers just for the elements of traversal paths — a very small number of weak references in relation to the entire reference graph. Therefore, once tracing begins, we do not measurably perturb the garbage collection. In addition, because the sampling process itself is so infrequent and selective, the cost of the sampling is also very small. In fact, the only measurable slowdown is the cost of having a JVMPI agent *attached*, not of listening to events. For example, when attached, some JVMs use a slower object allocator. This overhead can run as high as several percent, which still meets our design constraint.

6 Additional Related Work

This section categorizes and discusses the related research we have encountered.

Type Assistance: Some recent work use static semantics to enforce and detect ownership using ownership types[6,5]. Perhaps, knowing aspects of data structure flux at compile-time, such as the allocation site of constituents, and the sites at which members are added or removed, or substructures relinked, we could insert instrumentation at just those sites.

Heap Analysis: Some have studied the interaction between the application's and the runtime's use of objects [22,23]. They break an object's lifetime into several phases, such as the time after allocation and before first use, and the time between last use and collection ("drag"). Glasgow Haskell [10] as of version 5.03 has built-in support for this type of analysis, which it calls "biographical profiling." Other works study how liveness information [1,13] or reachability [14] can benefit conservative garbage collection. As with static analysis, it is conceivable that LeakBot could leverage these findings in its ranking. However, they gather the information via a pre-pass profiling run, which doesn't fit with our goal of analysis on production machines.

Debugging Memory Problems by Tracing: Others have explored instrumenting allocation sites [3,11,29] and instrumenting field access and modification [24,25] in order to assist in tracking down memory-related errors. Some of these [24,25] using the notions introduced by [22], strive to automatically fix the problems, by instrumenting field updates to remove "dragging" references to array elements. In addition to high overhead, these techniques do not address the issues of ease of tool use in framework-intensive environments. While static assistance could possibly guide the instrumentation points to reduce runtime cost, it would have to be very precise in order to be useful.

Glasgow Haskell supports a form of heap analysis called "retainer profiling" to help fix memory consumption problems due persisting, but unevaluated closures. At least as of version 5.04, this profiling is based only on reachability, has a fixed maximum-size retainer set, and exploring multi-hop ownership requires multiple passes. In addition, retainer profiling is done on a per-class basis, with classes specified on the application command-line. For long-running, framework-intensive servers we think this approach is too restrictive.

Debugging via Tracing: Some approaches instrument the application [16] or debugger [2]) to alert the tool user to specific conditions specified by the tool user. Based on the specified conditions, these tools insert or activate instrumentation to gather the relevant information. This is akin to the interaction between the analyzer and tracing agent in LeakBot, except that in LeakBot the requests for information gathering have been automated, and are specific to the evolution of whole data structures.

Discovering Static Properties with Tracing: Several works collect and analyze traces to establish properties of programs which are difficult to establish with static analysis [8,9]. As with other work we have discussed, collecting the required traces can be expensive. Perhaps detecting invariant properties of data structures could assist LeakBot in ranking the objects or refining the CERs.

7 Conclusion

We had two design goals for LeakBot. First, it should be usable by non-experts, and by experts without many hours to burn. Second, it should be feasible to apply on production machines. In our consulting practice with IBM customers, we happily found that these two goals are not at odds. In fact, we found a synergy between them. Being selective about what the tool presents to the user (because those are the aspects which best explain the problem) facilitates being selective about what it traces.

In this paper we have presented a tool called LeakBot, which we have demonstrated to achieve these goals. We have used LeakBot successfully in our consulting practice to assist in problem determination on large IBM customer applications. It has quickly identified that leaks do or do not exist, verified that bug fixes actually fix problems, and assisted in diagnosing and expediting the resolution of known leaks. We were able to analyze applications with millions of live objects, and trace the ongoing evolution of suspicious data structures with negligible impact on the transaction throughput of those server applications. LeakBot implements four contributions presented in this paper:

Raising the level of analysis. Our technique presents results, and performs analysis at the level of data structures, not individual objects. This enables discovery and presentation of high-level properties, both structural and temporal, of those data structures.

A way to automatically find problematic data structures. We define the concept of leak roots, objects that are likely to be at the head of these structures: they contain one or more leaks. We identify a set of important structural and temporal properties of data structures, and introduce a nonlinear combination of them that prioritizes objects based on the likelihood that they are leak roots. Without any knowledge of specific frameworks, it quickly finds a small set of candidate leak roots out of millions of live objects.

Concise models of data structure evolution. We introduce Co-evolving Regions, sub-structures whose members exhibit coherent evolution behavior. We identify salient features — the leak root, the owner-proxy, and the change proxy — that concisely describe this similarity. These features not only group members into regions, but they also aid in explaining the regions to a user, and in observing the actual evolution in a cheap way.

A lightweight tracing agent. We show how to use Co-evolving Regions to automatically derive tracing schemes to detect when elements are added to, removed from, or repositioned within a region. To track these changes, we derive traversals from the owner-proxy and change-proxy.

An adaptive loop that combines ranking and evolution tracing to mutual benefit. Ranking informs the tracing: it identifies a small set of data structures that we need to track. Tracing informs the ranking: it identifies new temporal properties of those data structures. These new findings refine the rankings, which in turn allows the tracing agent to focus on only the relevant regions.

We continue to validate these techniques in our consulting practice. For example, as new frameworks come along, we may need to refine the model LeakBot uses for object ranking; new data structure or temporal properties, or new ways of combining them may be appropriate. We also continue to refine the CERs, and their use in the adaptive agent. Other areas of future work include using the high-level properties that LeakBot uncovers to develop better presentations, and exploring additional applications of CERs beyond memory leak diagnosis.

8 Acknowledgements

We appreciate the assistance of Matthew Arnold, Jong-Deok Choi, and Harini Srinivasan at IBM T.J. Watson Research Center in the refinement of the presentation of this paper.

References

1. O. Agesen, D. Detlefs, and J. E. B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Programming Language Design and Implementation*, 1998.
2. J. K. A. W. Appel. Traversal-based visualization of data structures. In *Symposium on Information Visualization*, pages 11–18, 1998.
3. D. R. Barach, D. H. Taenzer, and R. E. Wells. A technique for finding storage allocation errors in c-langage programs. *ACM SIGPLAN Notices*, 17(5), May 1982.
4. Borland software corporation optimizeit$^{\mathrm{TM}}$.
5. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Object-oriented programming, systems, languages, and applications*, 2002.
6. D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *European Conference on Object-Oriented Programming*, 2001.
7. J. S. Dong and R. Duke. The geometry of object containment. *Object-oriented Systems*, 2:41–63, 1995.
8. M. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering pointer-based program invariants. In *International Conference on Software Engineering*, 1999.
9. S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, May 2002.
10. The Glasgow Haskell compiler user's guide. `http://haskell.cs.yale,edu/ghc`.
11. R. Hastings and B. Joynce. Purify — fast detection of memory leaks and access errors. In *USENIX Proceedings*, pages 125–136, 1992.
12. B. Hayes. Using key object opportunism to collect old objects. In *Object-oriented programming, systems, languages, and applications*, 1991.
13. M. Hirzel, A. Diwan, and A. Hosking. On the usefulness of liveness for garbage collection and leak detection. In *European Conference on Object-Oriented Programming*, 2001.
14. M. Hirzel, J. Hinkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *International Symposium on Memory Management*, 2002.
15. `http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html`.
16. R. Lencevicius. On-the-fly query-based debugging with examples. In *Automated and Algorithmic Debugging*, 2000.
17. S. S. Muchnik. *Advanced Compiler Design and Implemtnation*. Morgan Kaufmann, 1997.
18. W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of Java programs. In *Software Visualization, State-of-the-art Survey*, volume 2269. Springer-Verlag, 2002.
19. W. D. Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. *Concurrency: Practice and Experience*, 12:1431–1454, 2000. previously appeared in ECOOP 1999.
20. G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems*, 24(5):455–490, 2002.
21. Rational software corporation quantify$^{\mathrm{TM}}$.
22. N. Rojemo and C. Runciman. Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In *International Conference on Functional Programming*, pages 34–41, 1996.

23. C. Runciman and N. Rojemo. New dimensions in heap profiling. *Journal of Functional Programming*, 6(4):587–620, July 1996.
24. R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in java. In *Computational Complexity*, pages 50–66, 2000.
25. R. Shaham, E. K. Kolodner, and M. Sagiv. Estimating the impact of heap liveness information on space consumption in Java. In *International Symposium on Memory Management*, June 2002.
26. Sitraka Inc. JProbe$^{\mathrm{TM}}$Profiler with Memory Debugger ServerSide Suite.
27. Heap Analysis Tool. `http://java.sun.com/people/billf/heap`.
28. Java 2 Platform, Enterprise Edition. `http://java.sun.com/j2ee`.
29. Sun Microsystems HPROF JVM profiler.
30. S. Wilson and J. Kesselmann. *Java^{TM}Platform Performance Strategies and Tactics*. Addison Wesley, June 2000.

A Handling multiple ownership

We detail two solutions to multiple ownership. One generalizes the definition of ownership, and the other prunes the reference graph so that multiple ownership does not occur.

First, we can generalize unique ownership to k-ownership, which is the number of objects owned by this object and k other objects. Thus, 0-ownership is equivalent to the unique ownership described above. We can derive this new relation by weakening dominance to allow for k-dominance. In this case, the dominator tree becomes a diamond-free graph (i.e. a tree with neither cross nor back edges).

In practice, however, computing k-dominance may be too expensive. Alternatively, we can use a technique similar to [20]: reduce the graph to a depth-first spanning tree. Since there are many such trees for one graph, we provide heuristics which guide which non-tree edges are more favorable to prune than others. For example, we populate the start set (of graph roots) in priority order: we give highest preference to first class objects, then objects referenced by on-stack GC roots, then every other graph root. Another example heuristic: for objects on the fringe, we make sure that if we clip any incoming edges, we clip edges which do not cross the fringe in preference to those which do. Note, however, that by pruning these edges, the ranking algorithm loses shared ownership information. Therefore, this pruning trades off speed of analysis with completeness of information.

B Choosing a threshold for leak root ranks

In our implementation, we have designed the gating functions so that most objects with a step-2 rank below 0.4 are fairly certain not to be leak roots. In off-line mode, we tend to set the threshold at least to 0.4 under the assumption that the snapshots we have are all the information we will get. In online mode, we can leverage the online evolution tracker described in Section 5: set the threshold lower, and use evolution tracking to quickly confirm or deny the candidacy of all non-zero ranked leak roots.