IJNRD.ORG ISSN : 2456-4184
INTERNATIONAL JOURNAL OF NOVEL RESEARCH AND DEVELOPMENT (IJNRD) | IJNRD.ORG
An International Open Access, Peer-reviewed, Refereed Journal

"A STUDY ON LEAK DETECTION OF LOW-OVERHEAD MEMORY THROUGH ADAPTIVE STATISTICAL PROFILING"

Kambapu Sai Srinivasa Reddy, Gourav Katoch, Rahul Kaushal
Student, Student, Student
School of Computer Science and Engineering
Lovely Professional University, Phagwara, India

ABSTRACT

Sampling has been effectively used to find chances for performance enhancement. We want to use comparable methods to ensure that the software is proper. Sadly, sampling covers little of the seldom-run code, where problems are most likely to hide. To address this, researcher provide an adaptive profiling system that samples code segment executions at "a rate inversely proportionate to the execution frequency."

Our innovative "memory leak detection program, SWAT," has been put into action to verify our concepts. Using our adaptive profiling architecture, SWAT creates a heap model by tracking program allocations and frees. It then utilizes this model to monitor loads and store these objects with minimal cost. "Stale" items that haven't been used in a "long" period is reported as leaks by SWAT. This enables it to identify any leak that appears while the application is running. The minimal space cost (< 10% in most circumstances, and often less than 5%) and low runtime overhead (< 5%) of SWAT make it useful for tracking production code leaks that take days to appear. Apart from pinpointing the memory leak allocations, SWAT also reveals the last place the application accessed the compromised data, hence aiding in the debugging and leak repair process. Over the last 18 months, "SWAT has been deployed by several Microsoft product groups and has shown to be successful at discovering leaks with a low false positive rate (<10%)."

Keywords: Low-overhead monitoring, Memory leaks. Performance, Reliability, Runtime analysis, Verification.

1. Introduction

Due to the intricate nature of modern software and hardware designs, application profiling is a crucial component of several optimization and performance analysis frameworks. While events-of-interest tracing makes it easy to get coarse-grain program profiles, sampling approaches are typically needed to decrease runtime overhead for fine-grain profiles.

Thanks to the increasing focus on program correctness, there has been a proliferation of static analysis & runtime tools for detecting software defects. Runtime program inspection techniques like Purify (memory leak tool)[1] and Eraser (data race detector)[2] need very detailed monitoring of program events of interest. These tools aren't very useful because of their large runtime overhead (anything from 5 to 30 times).

Sampling is our preferred method of lowering the overhead associated with runtime program testing tools. Regretfully, whereas sampling offers comprehensive coverage of occurrences in often performed code segments, errors frequently exist in seldom executed program regions. To overcome this problem, we provide an adaptive profiling system that samples code segment executions at a rate inversely proportional to the frequency of execution. Consequently, code segments that are performed seldom are efficiently traced, whereas code portions that are executed often are sampled relatively infrequently. This method, while retaining a comparable runtime expense, trades more thorough code coverage for the capacity to gather more samples from frequently performed code regions.

Using two criteria, we provide an informal characterization of the kinds of software faults most suited for sampling-based detection. In the event that program event E is linked to a program bug B, sampling event E will be successful in identifying B if: "(1) event E does not occur when bug B occurs, guaranteeing that sampling does not result in false negatives; and (2) if bug B is absent, the number of instances of event E should surpass the reciprocal of the sampling rate, guaranteeing a low number of false positives reported."[3]

We have put our thoughts into practice by using a new memory leak detecting tool called SWAT. When an allocated object is not released but is never utilized again, memory leakage occurs. Based on an object's historical access history, SWAT makes predictions about its future accessibility. When an object o is expected to remain unaccessed until the program's conclusion, it is referred to be stale and is considered a leak. With its low-overhead adaptive profiling architecture, SWAT tracks program allocations and frees as well as samples data accesses to heap objects. It flags as a leak "stale" heap items that haven't been accessed in a (user-defined) "long" period of time. Compared to Rational's Purify tool, which is commonly utilized, SWAT offers three primary benefits.[4] It finds all leaks that Purify would report in addition to extra leaks (false positive reports are rare) because to its novel approach of finding leaks based on object staleness. Indeed, SWAT's approach ensures that any leak that occurs during the specified program run will be discovered. Furthermore, "the overhead is far less than Purify (5% vs. 3–5X) since it employs sampling. This allows production code leaks that take days to appear to be tracked using SWAT." Last but not least, SWAT indicates which software instruction visited the compromised object most recently, which often facilitates quicker debugging and leak repair. Over the last 18 months, "SWAT has

been deployed by several Microsoft product groups and has shown to be successful at discovering leaks with a low false positive rate (<10%)."

This study makes "three important contributions:"

• it designs and implements an adaptive profiling technique for programs that is suited for sample-based profiling. Furthermore, we provide an informal characterization of the kinds of software mistakes that sampling can discover efficiently (Section 2).

• the creation and use of SWAT, a unique memory leak detector that has several benefits over the most recent runtime leak detectors and leverages "our adaptive profiling framework to achieve minimal overhead (Section 3).

• Evaluation findings, including real-world case studies, demonstrate that, even with a low enough sampling rate to result in a runtime cost of around 5%, SWAT is successful at detecting memory leaks with a low false positive rate (< 10%)" (Section 4).

2. ADAPTIVE PROFILING

Expanding on the bursty tracing architecture outlined Hauswirth, M., & Chilimbi [5] this section details an adaptive profiling implementation. As an expansion of a "sampling architecture outlined by Arnold and Ryder, [6] bursty tracing is a sampling infrastructure that can go from full event tracing to no program monitoring at all with little overhead. Two user-specified sample counters regulate the rate of switching between the two modes and the duration of full tracing." Bursty tracing has a few flaws, one of which is that it could overlook essential but seldom run code regions while sampling. This might lead to program mistakes. To overcome this limitation, adaptive bursty tracing begins with complete "program tracing and then adaptively adjusts the sampling rate for specific segments of code," gradually sampling less often executed areas of code at a lower rate. Before describing adaptive bursty tracing, we provide a short introduction to bursty tracing.

2.1 Bursty Tracing

Continuous program monitoring with little overhead is possible using bursty tracing, a sampling-based technique. Bursty tracing differs from conventional sampling in that it may record full details of program execution (a "trace sample") for brief intervals. "How often a trace sample is obtained and its size are both controlled by two movable sampling counters." When compared to traditional sampling methods, bursty tracing offers a significant improvement in the amount of temporal execution data available for use in debugging and correcting software faults. Bursty tracing goes a step further by letting you tweak both the collection frequency and the trace sample extent, giving you even more control and flexibility.

Bursty tracing is defined in more depth elsewhere,[7] but we provide a quick summary here. The bursty tracing architecture involves duplicating the code of every method. Although "both versions of the code have the initial

instructions, only one of them has been configured to profile relevant events. At procedure entrances or loop back-edges, control is regularly transferred to dispatch checks in both versions of the code. To choose which version of the code to continue executing, the dispatch checks employ two counters, nCheck and nInstr."

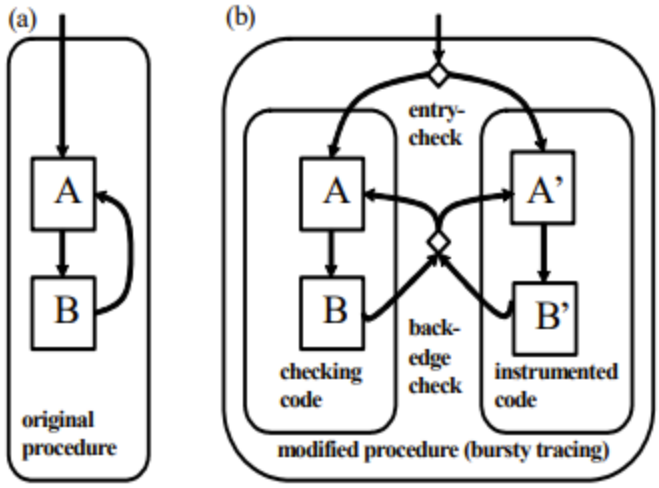

Figure 1: Low-overhead temporal profiling instrumentation

"Both nCheck and nInstr are set to zero at launch. Typically, the checking code is run and nCheck is decreased with each check. The dispatch check hands control to the code that is being instrumented when it reaches zero, and nInstr is initialized with nInstr0 (where nInstr0<<nCheck0). Each time the instrumented code is checked, nInstr is decreased. When it hits zero, the checking code is called again and nCheck is initialized with nCheck0."

The deterministic bursty tracing profiling framework works regardless of the operating system or hardware. Since it does not need recompilation or access to program source code, we were able to build it using Edward,[8] "a binary rewriting system for the x86." It is simple to manage the profiling overhead: after the initial cost of the dispatch checks, which is proportional to nInstr0/(nCheck0+nInstr0), the overhead grows exponentially with the sampling rate. We have complete control over the burst duration and sampling rate via nCheck0 and nInstr0. From what we can see, the dispatch checks have a fundamental runtime cost of "less than 5% and a sample rate of 0.1% makes the instrumentation overhead almost disappear."

2.2 Adaptive Bursty Tracing (ABT)

"Bursty tracing" records the timing of execution of frequently used code segments, yet many bugs in programs only show up in seldom used code areas that sampling would miss. Adaptive bursty tracing (ABT) uses a sample rate that is "inversely proportional to the execution frequency of code segments to overcome this problem." As a result, sections of code that are infrequently performed are basically traced, whereas areas of code that are regularly executed are sampled extremely slowly.[9]

Instead of using a uniform sample rate across all dispatch check locations, ABT uses "a per-dispatch check sampling rate." Initially, complete tracing (a sampling rate of 100%) is applied to all dispatch checks. Until a minimum sampling rate is attained, "the sampling rate is dropped by a configurable fractional amount (Decr) with each execution of a dispatch check. We apply the following formula: nCheck0(n) = (Decr(n - 1) - 1)*nInstr0"

The following sampling rates may be obtained using the above parameters: "Decr = 10, Min = 0.1%, nCheck0(1) = 0, nCheck0(2) = 9*nInstr0, nCheck0(3) = 99*nInstr0, and nCheck0(4) = 999*nInstr0. Here, r is the sampling rate, which is defined as (nInstr0)/(nCheck0+nInstr0) or 10%." "In the steady state," the lower bound sampling rate is used for often performed code segments, whereas virtually tracing (sampling at close to 100%) is used for seldom executed code segments. In exchange for more thorough code coverage, this method sacrifices the capacity to gather more samples from frequently performed code parts. By taking around the same amount of trace samples overall, it keeps the runtime overhead constant.

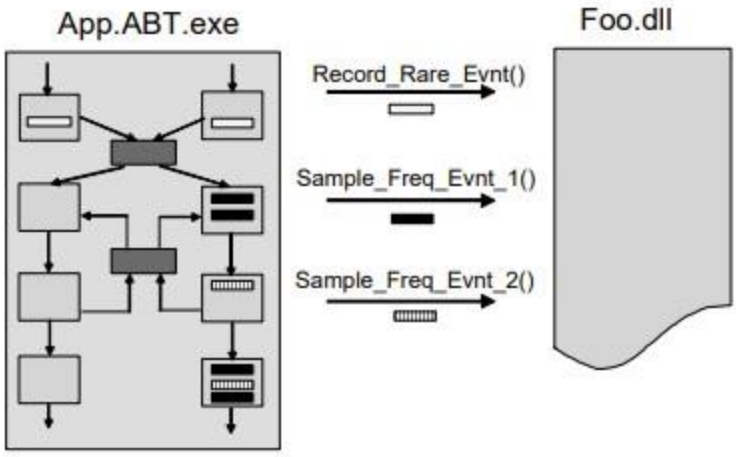

Figure 2: Framework for Adaptive Bursty Tracing

"The bursty tracing profiling framework's overhead and the quantity of profiling information it produces are controlled by the counters nCheck0 and nInstr0. The ABT framework's equivalent is nCheck0, nInstr0, Decr, Min. To illustrate, if you set nCheck0 to 9900 and nInstr0 to 100, the sampling rate will be 1% (100/10000=1%) and the burst duration will be 100 active checks."

Figure 2 serves as an illustration of the ABT structure. Traditional instrumentation (Record Rare Event()) is used to track "infrequent program events like dynamic memory allocations and lock acquisitions." Data references & branch executions are examples of rare yet costly occurrences that are tracked by ABT (Sample_Freq_Evnt_i()). "A custom dll called Foo.dll receives all of these events and either writes them to a log file for later processing or does online analysis and reports faults."

2.3 Discussion

While "sampling methods like the ABT" methodology may catch most computer flaws, they aren't foolproof. The class of software problems detectable by a sampling technique with two criteria may be described informally. To effectively discover bug B related with program event E, sampling event E must be done in the following way:

Condition of soundness:

$$B \Longrightarrow -E$$

This condition guarantees that sampling will not provide false negatives by stating that event E should not occur if bug B occurs.

Condition of precision:

$$-B \Longrightarrow |E| > \frac{1}{(samplingrate)}$$

According to this, in the case that bug B is absent, the frequency of "occurrences of event E should be more than the reciprocal of the sampling rate. This will guarantee that there are few false positives recorded."

Both requirements are met by many liveness properties. Sampling stores to find uninitialized variables meets soundness requirements, but not precision. Utilizing Eraser's lockset technique to detect data races and sample "shared accesses may not meet soundness and may not meet preciseness requirements." Both conditions are satisfied by sampling and verifying memory accesses to detect buffer overruns.

3. SWAT: DETECTING MEMORY LEAKS

We have created a unique memory leak detection program, SWAT, which makes use of our ABT framework in order to evaluate our concepts. Before describing SWAT, "we first provide some basic information on leak detection."

3.1 Background

When an allocated object is not released but is never utilized again, memory leakage occurs. "A memory leak detector's main query is: Given an object o at a time t in a program's execution, has o been leaked?" Usually, it is difficult to know whether or not item o has been released at time t. We need to know whether the object will be utilized later on or released before the program ends in order to respond to this query.

A definitive response to the query can only be provided at the conclusion of a program run. Anything that isn't liberated (immortal) at that time is automatically a leak. This method has the disadvantage that, "because the operating system will reclaim the whole heap of the ending process," objects are often not released on purpose at

the conclusion of a program run. Therefore, a leak detector that uses eternal object identification will find a lot of boring leaks. Furthermore, items that are not utilized and are only released at program termination should generally be considered leaks for server programs that must run continuously.

At any time while the program is running, a conservative response to the query may also be provided. It is possible to discern between items that are certain to leak and those that may leak. Theoretically, figuring out if "an item o is dead at time t is the most accurate conservative response. If the program is unable to reach any future state where object o is going to be accessed, then object o is dead." On the other hand, a living thing could or might not leak. This is due to the fact that an object may only be considered alive if there is only one potential program route in the future that has access to it; nevertheless, the actual execution may follow a different "path that does not visit the object."

"A more pragmatic, if less accurate (it will identify fewer leaks) method to provide a conservative response is to ascertain whether an object o is inaccessible at time t. If there isn't a reference chain from the root set to an object, it can't be reached (it can't be reached from the global and stack variables). Any unreachable object o is inevitably leaked (and dead) at time t." On the other hand, an accessible object could leak or not.

"Keep in mind that the set of leaked items at any given time t always consists of the set of dead objects, and the set of dead objects always consists of the set of inaccessible objects."

When determining "whether object o is leaked at time t, the sample-based method for memory leak detection discussed in this work does not always provide the right response." It merely offers a well-informed estimate. However, in other cases, that estimate might be even more helpful than the accurate cautious response, which is often "unknown."

3.2.Overview

Memory leaks are identified by SWAT using this simple principle: if an item in the heap "has not been accessed in a "long" period of time, there is a memory leak. This straightforward invariant guarantees that SWAT finds every leak that occurs during the specified runtime execution. Nevertheless, putting this staleness policy into a useful memory leak tool faces two major challenges." First of all, it may be too costly to monitor every access to heap data. Subsequently, there may be a significant number of false positives in the disclosed leaks. Maybe for these reasons, this "staleness" strategy isn't used by any memory leak tool that currently exists.

Our low overhead adaptive profiling system is used by SWAT to monitor heap accesses. It has an average runtime overhead of less than 5% due to its 0.1% sampling rate. "Our experience with SWAT suggests that it is necessary to adjust the "time elapsed" before an unaccessed heap object is reported as a leak in order to reduce false positives. Furthermore, developers are interested in a large number of the remaining false positives since long-inactive objects often point to wasteful memory utilization. There doesn't seem to be any discernible difference in the amount of false positives while sampling the heap data accesses. This is due to the fact that most heap

objects are visited more than once, and for an active object to be incorrectly identified as a leak," the sample must miss all of these visits. Later on, we provide corroborating empirical data.

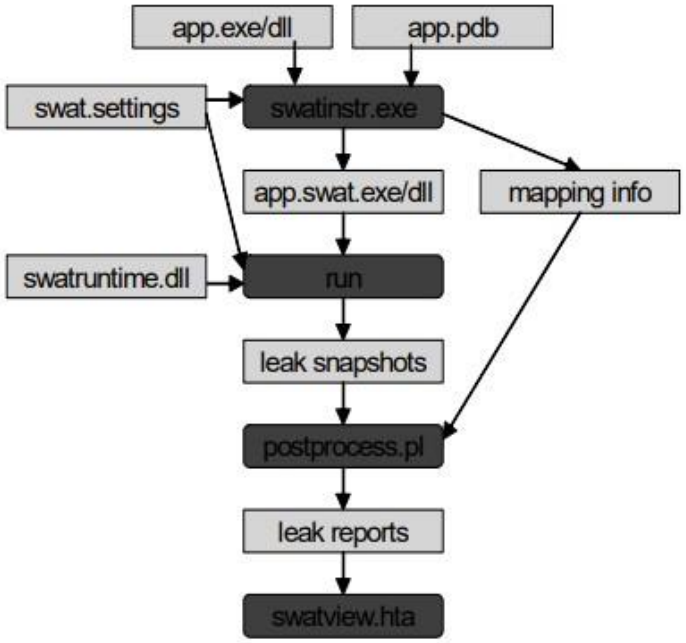

Figure 3: The infrastructure of SWAT

Figure 3 illustrates how SWAT works in greater detail. An instrumented version of the program is made by swatinstr.exe and is used instead of the original. To facilitate the association of "program counter values with the original copy's source code," a mapping file is also generated.

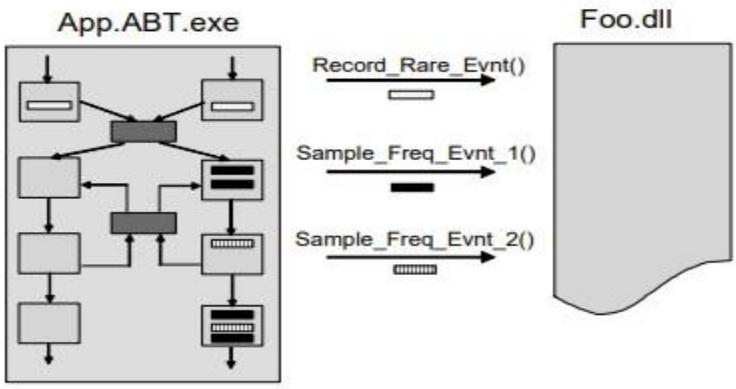

Figure 4: Instrumentation of SWAT

As seen in Figure 4, the instrumented application communicates "to a runtime dynamically loaded library (swat.dll) all heap allocations/frees (Record_Alloc()) and a sampling set of memory accesses (Sample_Load/Store()) collected using our adaptive profiling methodology. The heap model is maintained by the runtime DLL using the heap allocation/free information, which it updates with the heap access information." Every now and again, swat.dll does a snapshot, visiting every object in its heap models and reporting as leaks

any objects that meet its staleness criteria. Each heap item identified as a leak may be linked to “the responsible heap allocation, any heap frees that released objects produced at that allocation site, and—most importantly—the last access." Based on our experience, it is quite helpful to have this last access information in order to troubleshoot and correct reported breaches rapidly. Furthermore, the latest access data makes it possible to quickly identify false positives. A graphical user interface is used to postprocess and display the leak snapshots. A source code browser integrated into the GUI shows the most recent access to an item that has been disclosed.

3.1 Heap Model

The heap model's job is to record details about every item that is allocated. An object descriptor is the data that is kept for an object. “The allocation site, last access time, and last access site” of an object are included in its object descriptor. To keep this data updated and make it accessible, “the heap model must implement the following interface:”

AllocateObject(ip,startAddress,size)

FreeObject(ip, startAddress)

FindObject(ip, address)

GetObjectIterator()

Every time an object is allocated or deleted, the instrumentations call the first two functions. The heap model updates the object descriptor with the allocation details whenever AllocateObject is invoked. Every instruction that reads from memory uses the FindObject method. They are not limited to using the object's start address; “the address they provide that function may refer to any location within the object.” With every FindObject call, “the heap model revises the last access site (ip) and time.” This function must have little overhead because it is used so often. Lastly, the GetObjectIterator allows you to get all the accessible information for each object by iterating over all the objects that are presently allocated.

There are a lot of potential implementations of this interface, but the most of them are impractical. “An inline implementation, which keeps the object descriptor in the object's header, is the simplest.” One major problem with this method is that it doesn't tell you where to start when you have a reference within the object. Alternatively, you might keep a hash table of all the potential addresses to the object's description that spans the provided address. A significant amount of space would be wasted if this table had one entry for every assigned item. A more sophisticated method would include storing a hash table just for the beginning addresses of each item, with a distinct little data structure that can be used to map any address to the beginning address. Given the diminutive size of many objects, this structure might store the disparity between an address and its related object's start address in only one or two bytes.

We take a different tack in our real use of the heap model. An address tree, a binary tree structure, is what we keep track of (Figure 5). One of the 32 bits that make up an address is represented by each level of this 32-level tree. There is a representation of the most important bit at the root node. Its two offspring stand in for the two types of addresses, where the initial bit might be either 0 or 1. The FindObject method traverses the tree from its root to the leaf specifying the provided address in order to get the object descriptor that spans that address.

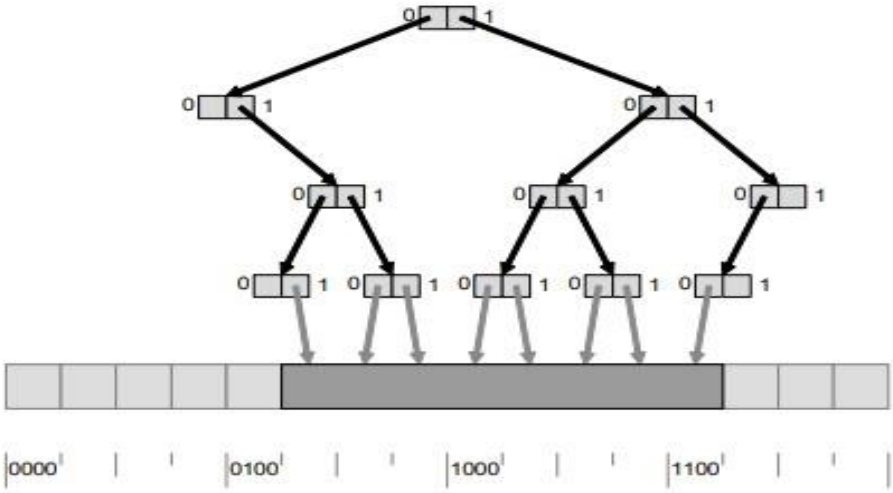

Figure 5: Part of the whole address tree for an 8-by-1 object having an initial address of 0101.

It would take a lot of memory to keep a full "binary tree with a leaf node for every heap address" that is utilized. To cut down on this unnecessary space consumption, we let an internal node that has many addresses pointing to the same item directly redirect to that object's description (refer to Figure 6). As a result, several leaf nodes and their progenitors are removed. Large, properly oriented items benefit greatly from the space savings. "Objects' start addresses and sizes are often multiples of 4, and allocators commonly align objects to 4 byte boundaries, thus we basically preserve all nodes on the lowest two tiers." Our method is adaptable and may continue to function even when objects aren't perfectly aligned, unlike other methods that may take advantage of the 4 byte object alignment. The object descriptor is directly referenced by inside nodes, and all four superfluous nodes are eliminated.

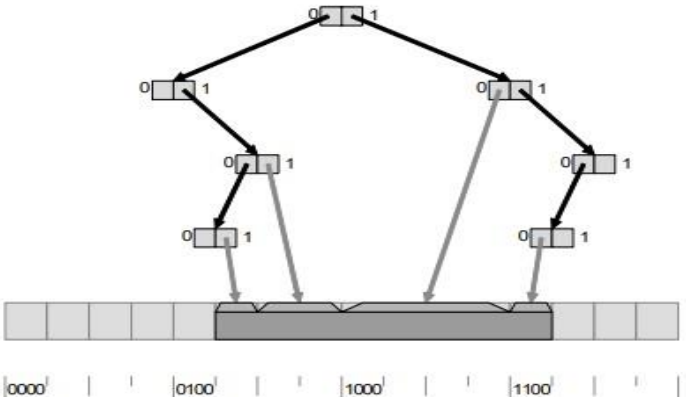

Figure 6: A portion of "an optimized address tree" for an 8-by-1 item with a starting address of 0101.

When provided with "a 32-bit address and the ensuing 32-bit address tree," the time overhead of the FindObject method remains constant. In the worst scenario, the object descriptor is reached after traversing 32 address tree nodes. Keep in mind that the data cache houses the address tree's top level and the nodes that are used the most often, lowering the cost of memory access. To further take advantage of location, "we batch and display tree operations in address sorted order. Finally, to reduce the effect of the FindObject operation on the total program execution time, we use a low sampling rate of 0.1% in our studies."

3.2 Predicates of Staleness

Every now and then, "our leak detector searches the heap model for things that have been exposed. The precise time between traversals may be adjusted by the user. Interactive applications would likely benefit from scheduling these traversals for when the program is idle and awaiting user input," even if this is not presently done. Furthermore, during program closure, a leak report is always created. Instead than measuring staleness using wall clock time, we utilize accesses. This prevents marking interactive application objects that are inactive for a whole night as stale. The heap model preserves data in each heap object's descriptor, as was mentioned in the preceding subsection. "A staleness predicate determines whether or not the object has been leaked based on this information. An overview of the three distinct staleness predicates" is provided in Figure 7.

Every item "that has never been accessed is regarded as a leak by the Never Accessed staleness predicate." The Constant Time staleness predicate makes an educated approximation based on how long it has been since an object was last accessed (as shown by observed data accesses). The item is deemed to have been leaked if that time exceeds a defined threshold. The constant time predicate and the active time predicate are comparable. The threshold is now n times the object's active duration rather than a constant, which is the difference. The interval between an object's first and most recent access is known as its active time. This predicate makes sense intuitively as it permits an object that has been in operation for a considerable amount of time to become dormant before it is deemed to be a leak.

3.3 Reporting on Leaks

What should be reported to the user is a fundamental issue for any workable memory leak detection. The program's missing deallocation locations are the reason for the detector's identification of leaking items. The detector is unable to identify source code locations that are accountable for the leaked objects since these deallocation sites are missing. As a result, the majority of leak detectors merely reveal an object's allocation place. The moment at which the leaked object became inaccessible is likewise reported by Insure++ [17]. Where an item was leaked is one of the most aggressive details our statistical leak detection reveals. We are able to disclose not just "the allocation site for each leaked object, but also the last observed access site because we sample object accesses and record the last access for staleness prediction." In addition, we provide all deallocation locations of any objects—if any—that were assigned to the same site as the item that was leaked. This may help to further focus on the specific code segment where the item that was leaked should have been deallocated.

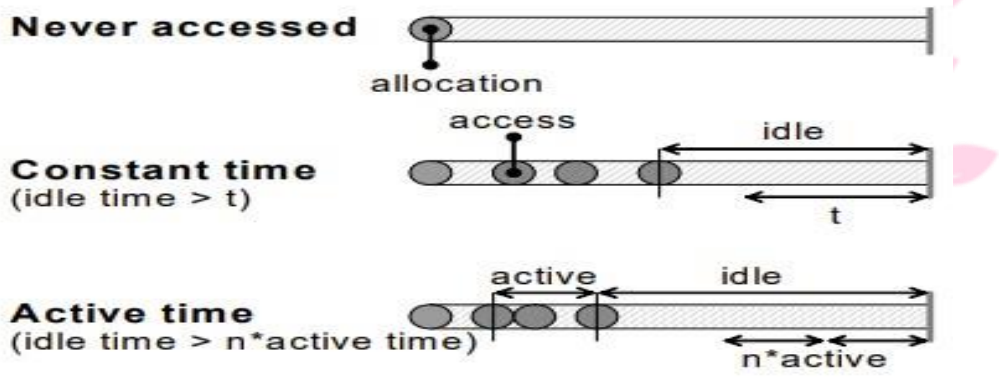

Figure-7: Staleness Predicates

3.4 Ordering Leaks

Not every time is our leak detection accurate. Occasionally, it flags an item as compromised even when future access to it is still planned. Furthermore, it aggressively flags as leaked any object that is unused (unlike earlier detectors that merely flag as leaked any object that is inaccessible or immortal). "This gives rise to the issue of false positives, which we resolve by arranging the list of leak reports in a certain manner."

At the level of specific items that we link to a last access location, leaks are found. A grouping based on allocation site is used. We allow you to sort leak lists in three distinct ways: by drag, by number of items, and by number of bytes. Real leaks are more likely to occur on an allocation site with several items leaked. The higher the level of the leaks in terms of difficulty, the more spilled bytes they include. Lastly, "the space-time product of the number of leaked bytes multiplied by the amount of time the leaked objects occupied space represents the full cost of a leak, not simply the number of bytes that were lost. The term "leak's drag" refers to this product, which is the opportunity cost of not reusing the stolen memory. The priciest leaks are ranked first on the leak list when

it is arranged by drag. Based on our experience, the best results are obtained by first sorting the allocation sites by the number of leaked objects, and then concentrating on high ranking leaks with the biggest drag."

3.5 Discussion

Collecting bursts instead of point samples doesn't really help with leak detection, but it can provide helpful context for troubleshooting. At the final access point for leaked items, SWAT may capture program burst history.

An essential part of SWAT is the adaptive tracking of code portions that are infrequently performed. This is necessary to prevent the false detection of data objects as leaks, even if they are not used in regularly performed sections of code. Our first effort at a leak prediction based on staleness was plagued by false positives due to its use of bursty tracing.

4. EXPERIMENTS

We looked into SWAT from several angles. We tested our staleness predicates, "SWAT's runtime and space overhead, the effect of sampling on leak identification," and three more sets of tests. Lastly, we provide case studies that prove SWAT can effectively identify memory leaks while minimizing "false positives."

4.1 Benchmarks

For our investigations, "we used the SPECInt2000 benchmarks. The benchmarks were executed on a single CPU of a dual-processor 2.7 GHz Pentium 4 PC with 2 GB RAM and Windows XP in order to conduct the measurements." Runs of the SPEC benchmarks were conducted using their most extensive dataset (ref). We averaged the findings after running each timed run five times. There was less than a 2% difference in the results across all runs. In order to test our staleness predicates, examine "the effect of sampling on leak detection," and determine SWAT's runtime overhead, we conducted three sets of experiments.

4.1.1 Overhead for Runtime

With our adaptive profiling method, we tested SWAT's runtime overhead with "a Decr value of 10 and Min values set to 0.01% and 0.1%, respectively. To generate bursts of length 10, nInstr0 was set to 10." Figure 8 displays the outcomes. For the majority of the benchmarks, the runtime overhead in both scenarios was around 5%. Although the execution duration was not mentioned, "SWAT was programmed to only issue one leak report" at the conclusion of these runs. Therefore, the figures solely represent the time spent building the heap model and monitoring access.

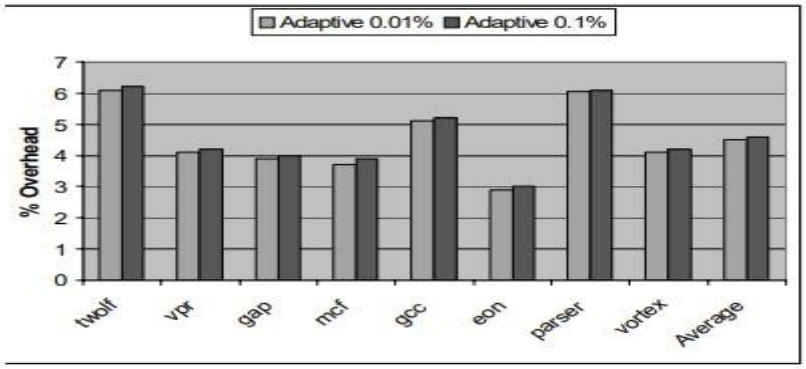

Figure 8: SWAT's Runtime Overhead

4.1.2 Space Overhead

The space above that runs SWAT is shown in Figure 9. In order to get the data, we averaged many heap snapshots taken while the application ran. The address tree data structure that was used to describe the heap contributes to some of this cost, while the remaining portion is caused by the information that is linked to each heap item in order to detect objects that have been leaked and their last known access location.

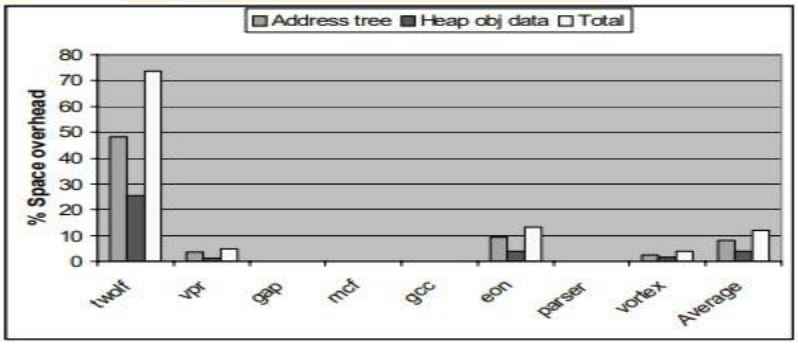

Figure-9: SWAT's overhead space

Averaged out to less than 10%, total space overhead is little with the exception of twolf, which contains an enormous amount of incredibly tiny items. Less than 1% of the benchmarks' space overhead is attributable to gaps, mcf, gcc, and parser.

4.1.3 Adaptive Profiling's Effect on Leak Detection

In order to assess the impact of sampling-induced false positives, we conducted the following experiment. To start, we removed 10% of all dynamic deallocations at random, which introduced leaks into the benchmarks. After that, we used “an IdleGt100Million references staleness predicate to tally the discovered leaks and set the sampling rate to 100%.” Next, we tested various adaptive sampling rates to determine the amount of extra leaks that were reported, also known as false positives. Pictured below are the outcomes (Figure 10). False positive

rates are very low at 1% and 0.1% sample rates. Given our discovery in Section 2.3 that sampling is suitable for mistakes related "with events that occur often enough to be observed despite sampling," this is not entirely unexpected. Therefore, it is very improbable that non-leaked items would be reported as false positives until their access frequency exceeds 100 or 1000 times, respectively.

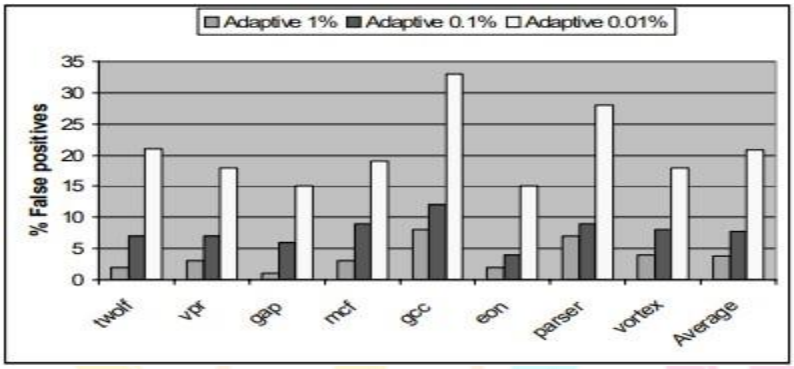

Figure-10: False positives introduced by sampling.

4.1.4 Assessment of the Staleness Predicate

Using the same methods as before, we removed "10% of all dynamic deallocations at random and compared the false positive rate to the leaks indicated by complete tracing" in order to assess the various staleness predicates. Figure 11 displays the outcomes. Although it works effectively, the IdleGt1Billion predicate could overlook leaks in applications that don't run for long enough. Both the IdleGt10*active and the IdleGt100Million predicate function similarly, with the latter somewhat underperforming in most situations and significantly outperforming in others.

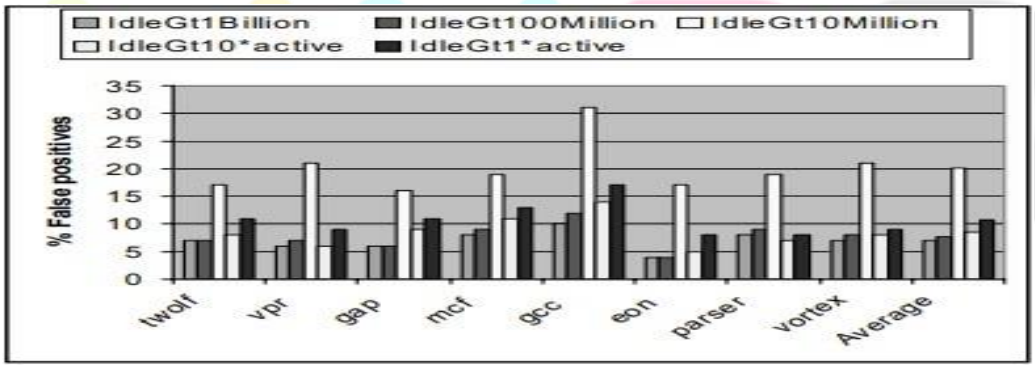

Figure11: Different staleness predicates' introduction of false positives.

4.2 Case Studies

Over the last 18 months, several Microsoft product groups have used SWAT, and the tool has shown to be efficient in detecting breaches while maintaining a low false positive rate. Two personal computer games, a multimedia application, and a big, interactive web software application are the four case studies that we offer.

4.2.1 Comprehensive Interactive Web App

We ran a big interactive web program for a few hours after randomly skipping frees at runtime to intentionally insert 34 leaks, so we could test SWAT. With the exception of one "potential" false positive—which really was a leak—SWAT accurately detected all thirty-four injection leaks. The next step was to replace the original program with one that had SWAT applied to it, which we then distributed to our study group members for everyday usage. We couldn't disclose the runtime overhead since we didn't have a way to record and play back external inputs. On the other hand, the application's overhead was so little that it couldn't have been distinguished from the original. Table 1 further shows that the space above was quite low. Over the last six months, SWAT has identified about twenty "potential" leaks; the developers have not yet confirmed these claims.

	% Runtime Overhead		% Space Overhead		
Benchmark	Adaptive 0.01%	Adaptive 0.1%	Address tree	Heap object data	Total
Interactive Web App	NA	NA	0.18	0.27	0.45
PC Game (Strategy)	4.3	4.71	8.2	2.21	10.41
Multimedia App	4.78	5.23	0.0	0.13	0.13
PC Game (Simulation)	4.42	4.97	3.25	0.78	4.03

Table 1: SWAT overhead in real-world applications

4.2.2 3rd Party PC Game (Strategy)

A few months before the ship date, we were called and requested to do a SWAT analysis in order to find any memory leaks. The game's SWAT instrumented version had an overhead of less than 5%, which made it playable. This was a very positive outcome, as modern PC games are resource-intensive and strain desktop performance to the maximum. About 10% of the entire area was above. Two hours or so of real gaming were used to create the

leak report. Many memory leaks were detected by SWAT. Many of them were "false positives," but they also pointed to heap items that had been inactive “for a very long period, which suggested inefficient use of memory.” For instance, things that were cached but were never retrieved again. Even though the delivery date was so near, “the lead developer nevertheless decided to change the memory management methods to better monitor heap allocation after seeing the SWAT leak report.”

4.2.3 Multimedia Application

One month before to the multimedia application's planned release, we implemented SWAT on the most recent beta version. As before, the application's SWAT instrumented version was identical to the original, allowing for extensive testing scenarios that accurately represented real-world usage. With no false positives, SWAT found six leaks in the application's visual browser component. Before release, these were brought to the attention of the developers and corrected.

4.2.4 PC game (simulation) from the first party

SWAT was requested of us as a part of a code quality assessment. There was about two hours of gaming in the test scenario. “To calculate the runtime overhead, we recorded and reenacted the situation. The results are shown in Table 1. Runtime and space overhead were both below five percent.” Five "potential" leaks were found by SWAT; developers have not yet confirmed this. After carefully going over the pertinent code, we think there aren't any false positives in this.

4.2.5. Discussion

Every time, “the overhead of executing the SWAT instrumented binary was little enough to allow real-world” use tracking instead of relying on truncated, fictitious situations. Based on our expertise, SWAT can be operated profitably on a continuous basis. Sampling is often accused for omitting crucial details. These criticisms, in our opinion, are misguided since the amount of information collected is determined by multiplying the “sampling rate by the number of program instructions that are being watched. lesser sample rates encourage more individuals to run monitored apps longer since they suggest lesser overhead, which might result in the production of more information overall.”

The leak reports' "last access" data proved very helpful in identifying and repairing breaches “as well as differentiating genuine leaks from false positives.” The percentage of false positives was very low for three of the apps. But even false positives may be intriguing, as the example of “the PC game Strategy” shows. Instead of patching the leaks, the SWAT data in this instance was utilized “to re-architect and rebuild the memory management” procedures.

5. RELATED WORK

Memory leak detection and bursty tracing are the two most closely connected fields of research.

5.1.Bursty Tracing

Arnold and Ryder first proposed the concept of sampling as a means to lower the cost of potentially overhead-heavy instruments. [10] In order to create temporal profiles, Chilimbi and Hirzel[11] enhanced this infrastructure to sample bursts of instrumentations. In order to dynamically identify and prefetch hot data streams, they used their bursty tracing framework.[12] We enhance their system by continuously expanding the instrumentation coverage in infrequently performed code while maintaining minimal overhead.

"The Arnold-Ryder framework is extended in a new dimension by Liblit et al.[13] A way to trigger samples based on a geometric distribution is introduced instead of evenly spacing samples. Their data is a statistically sound random sample," therefore it may be used for a wide variety of statistical tests. In order to find bugs in programs, they also use their framework. Nevertheless, their main emphasis is on determining the root of mistakes over several program runs, while ours is "on low-overhead program checkers that identify certain kinds of flaws during individual program runs."

5.2. Memory Leak Detection

Memory leak detection may be done in three different ways. To begin, static analysis tools[14,15] allow one to find potential leak spots prior to executing the program. While they are able to detect leaks before to program deployment and do not add any overhead during runtime, their inability to include dynamic information causes them to provide conservative findings, which may include false positives, and they are unable to detect all potential leaks.

Furthermore, a number of dynamic tools[16-18] enable interactive study of memory utilization by taking, comparing, and inspecting heap snapshots. "In runtime settings with a garbage collector," they aid in the hunt for the reasons of leaked objects by providing information "about the amount of memory allocated at different locations and for different kinds or object sizes. Additionally, they display the connectedness of heap items.

And last, at the conclusion of the program run, automated "dynamic tools provide a list of leaked items and leak locations." Only everlasting things can be captured by some of those instruments. [19,20] So, the allocation functions are all that need to be instrumented. Other methods [21,22,23] manage to catch items that are really difficult to reach. The allocation routines are instrumented and a mark-and-sweep garbage collection technique [24] is used to do this. To find out when an object's final reference vanishes, Insure++ [25] use "runtime pointer tracking, a reference counting garbage collection technique." These tools aren't very useful since they either have a large overhead or miss a lot of leaks.

The kind and liveness accuracy of a reachability-based leak detector's traversal greatly affect the quality of its findings, as shown by Hirzel et al. [26,27] For them, the most crucial thing is how accurate the liveness is. We get around the issue of accurate liveness analysis by continuously monitoring accesses to objects and determining whether or not they are live. "To the best of our knowledge, no other dynamic memory leak" detection program actively seeks for and records leaks caused by object accesses. While capturing potentially accessible leaking items, our technique does this with little effort, making it suitable for use "in a deployed system."

6. FUTURE WORK

We want to investigate potential uses for our adaptive profiling technology in addressing various software defects. Taking use of the adaptive profiling framework's minimal overhead monitoring feature may be as simple as developing a well-known remedy for certain sorts of issues. Sometimes, as in SWAT, it may need taking a fresh tack, made possible by our adaptive profiling architecture, to tackle an issue.

Using a less-than-ideal but more compact heap model is something we'd want to assess in relation to SWAT. The number of address tree nodes in our present heap model is still somewhat high, especially for applications that include several little objects. If the little "constant amount of space (e.g., a fraction of a byte) available for each object is insufficient to retain the offset, we may use a compact encoding to map from any given address to the beginning of the object that spans that address, sacrificing accuracy in the process."

In addition, it would be fascinating to see how our leak detector's adaptive bursty tracing method stacks up against Liblit's random sampling method.

Contrasting the more cautious Insure++ method of determining the site the object became inaccessible with our technique of determining the last site the leaked object was visited would be an interesting empirical comparison. Our method may determine, for a certain run, the earliest feasible deallocation location (immediately after the final access) for the object. Before losing all references to an object, the Insure++ method finds the final conceivable place where it may be deallocated. It would be fascinating to merge the two methods so that consumers "could see the allocation site, the site of the most recent access, and the site the object became" inaccessible for every leak. In addition, the user would be able to see all deallocation sites for additional items that were assigned to the same site. Adding on to this would be to provide a record of the program route that was used to go from "the last access site to the site where the object became inaccessible. This would provide the user with all the program locations where the deallocation might be put to stop the leak."

7. CONCLUSIONS

We have described an adaptive profiling architecture that may be used to develop lightweight runtime program verification tools. Depending on how little the overhead is, these program checkers may identify issues with

production code and report them. In this document, we lay out the broad strokes of what our methodology can detect as programming errors. To make sure our theories hold water, we used SWAT, a novel memory leak detection method. Several Microsoft product groups have used SWAT within the last 18 months. It has shown effectiveness in discovering breaches while maintaining a low false positive rate.

References

1. Bond, M. D., & McKinley, K. S. (2006). Bell: Bit-encoding online memory leak detection. ACM SIGARCH Computer Architecture News, 34(5), 61-72.
2. Xie, Y., & Aiken, A. (2005, September). Context-and path-sensitive memory leak detection. In Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (pp. 115-125).
3. Sui, Y., Ye, D., & Xue, J. (2014). Detecting memory leaks statically with full-sparse value-flow analysis. IEEE Transactions on Software Engineering, 40(2), 107-122.
4. Sui, Y., Ye, D., & Xue, J. (2014). Detecting memory leaks statically with full-sparse value-flow analysis. IEEE Transactions on Software Engineering, 40(2), 107-122.
5. Hauswirth, M., & Chilimbi, T. M. (2004, October). Low-overhead memory leak detection using adaptive statistical profiling. In Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (pp. 156-164).
6. Matthew Arnold and Barbara Ryder. A framework for reducing the cost of instrumented code. In ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), 2001
7. Martin Hirzel and Trishul M. Chilimbi. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO), pages 117-126, December 2001.
8. Edwards, A., Srivastava, A., & Vo, H. (2001). Vulcan: Binary transformation in a distributed environment. Microsoft Res., Redmond, WA, USA, Tech. Rep. MSR-TR-2001-50.
9. Ibid.cit.no.5
10. Ibid.cit.no.6
11. Ibid.cit.no.7
12. Trishul M. Chilimbi and Martin Hirzel. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 199-209, June 2002
13. Liblit, B., Naik, M., Zheng, A. X., Aiken, A., & Jordan, M. I. (2005). Scalable statistical bug isolation. Acm Sigplan Notices, 40(6), 15-26.
14. David L. Heine and Monica S. Lam. A Practical FlowSensitive and Context-Sensitive C and C++ Memory Leak Detector. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2003

15. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. In Software Practice and Experience, 2000; 30:775-802.
16. Borland OptimizeIt Profiler. http://www.borland.com/optimizeit/optimizeit_profiler/index.html
17. EJ-technologies' JProfiler. http://www.ej-technologies.com/ products/jprofiler/overview.htmlPaul Moeller. Win32 Java Heap Inspector. 1998. http://www.geocities.com/moellep/debug/HeapInspector.html
18. Quest jProbe Memory Debugger. http://www.quest.com/jprobe/debugger.asp
19. Erwin Andreasen and Henner Zeller. LeakTracer. August 2003. http://www.andreasen.org/LeakTracer
20. Benjamin Zorn and Paul Hilfinger. A memory allocation profiler for C and Lisp programs. In Proceedings of the Summer USENIX Conference, pages 223-237, 1988
21. R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In Proceedings of the Winter USENIX Conference, pages 125-136, 1992.
22. GreatCircle. http://www.geodesic.com/
23. Jeremy Dion and Louis Monier. ThirdDegree. http://research.compaq.com/wrl/projects/om/third.html
24. Hans Boehm, Alan Demers, and Mark Weiser. A garbage collector for C and C++. http://www.hpl.hp.com/personal/ Hans_Boehm/gc/
25. Parasoft Insure++. http://www.parasoft.com/products/insure
26. Martin Hirzel and Amer Diwan. On the Type Accuracy of Garbage Collection. In International Symposium on Memory Management (ISMM), pages 1-11, October 2000.
27. Martin Hirzel and Trishul M. Chilimbi. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO), pages 117-126, December 2001.