# Detecting and Eliminating Memory Leaks Using Cyclic Memory Allocation

Huu Hai Nguyen and Martin Rinard

Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Singapore-MIT Alliance
Massachusetts Institute of Technology
Cambridge, MA 021139
nguyen@nus.edu.sg,rinard@csail.mit.edu

## Abstract

We present and evaluate a new technique for detecting and eliminating memory leaks in programs with dynamic memory allocation. This technique observes the execution of the program on a sequence of training inputs to find $m$-bounded allocation sites, which have the property that at any time during the execution of the program, the program accesses at most only the last $m$ objects allocated at that site. If the difference between the number of allocated and deallocated objects from the site grows above $m$ throughout the computation, there is a memory leak at that site. To eliminate the leak, the technique transforms the program to use *cyclic memory allocation* at that site: it preallocates a buffer containing $m$ objects of the type allocated at that site, with each allocation returning the next object in the buffer. At the end of the buffer the allocations wrap back around to the first object. Cyclic allocation eliminates any memory leak at the allocation site — the total amount of memory required to hold all of the objects ever allocated at the site is simply $m$ times the object size.

We evaluate our technique by applying it to several widely-used open source programs. Our results show that it is able to successfully detect and eliminate important memory leaks in these programs. A potential concern is that the estimated bounds $m$ may be too small, causing the program to overlay live objects in memory. Our results indicate that our bounds estimation technique is quite accurate in practice, providing incorrect results for only two of the 152 suitable $m$-bounded sites that it identifies. To evaluate the potential impact of overlaying live objects, we artificially reduce the bounds at $m$-bounded sites and observe the resulting behavior. The resulting overlaying of live objects often does not affect the functionality of the program at all; even when it does impair part of the functionality, the program does not fail and is still able to acceptably deliver the remaining functionality.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Dynamic Storage Management; D.2.5 [*Software Engineering*]: Testing and Debugging

***General Terms*** Design, Experimentation, Reliability, Security

***Keywords*** Cyclic Memory Allocation, Failure-Oblivious Computing, Memory Leaks

## 1. Introduction

A program that uses explicit allocation and deallocation has a memory leak when it fails to free objects that it will no longer access in the future. A program that uses garbage collection has a memory leak when it retains references to objects that it will no longer access in the future. Memory leaks are an issue since they can cause the program to consume increasing amounts of memory as it runs. Eventually the program may exhaust the available address space and fail. Memory leaks may therefore be especially problematic for server programs that must execute for long (and in principle unbounded) periods of time.

This paper presents a new technique for detecting and eliminating memory leaks. This technique applies to allocation sites [1] that satisfy the following property:

* This research was supported in part by DARPA Cooperative Agreement FA 8750-04-2-0254, DARPA Contract 33615-00-C-1692, the Singapore-MIT Alliance, and the NSF Grants CCR-0341620, CCR-0325283, and CCR-0086154.



*ISMM'07,* October 21–22, 2007, Montréal, Québec, Canada.


[1] An allocation site is a location in the program that allocates memory. Examples of allocation sites include calls to `malloc` in C programs and locations that create new objects in Java or C++ programs.

DEFINITION 1 ($m$-Bounded Access Property). *An allocation site is $m$-bounded if, at any time during the execution of the program, the program accesses at most only the last $m$ objects allocated at that site.*

If the difference between the number of allocated and deallocated objects from an $m$-bounded site exceeds $m$, the program is consuming more memory than it needs to execute the computation. An increase in the difference over time indicates the presence of a memory leak. It is possible to eliminate this leak as follows:

- **Preallocation:** Preallocate a buffer containing $m$ objects of the type allocated at that site.
- **Cyclic Allocation:** Each allocation returns the next object in the buffer, with the allocations cyclically wrapping around to the first object in the buffer after returning the last object in the buffer.
- **No-op Deallocation:** Convert all deallocations of objects allocated at the site into no-ops.

This cyclic memory management scheme has several advantages:

- **No Memory Leaks:** It eliminates any memory leak at allocation sites that use cyclic memory management — the total amount of memory required to hold all of the objects ever allocated at the site is simply $m$ times the object size.
- **Simplicity:** It is extremely simple to implement and operate. Unlike many previously proposed static analysis techniques [20, 13, 24, 22, 16, 30, 28, 27, 21], it does not require the development of a heavyweight static analysis or programmer annotations to detect and/or eliminate memory leaks.

To use cyclic memory management, the memory manager must somehow find $m$-bounded allocation sites and obtain a bound $m$ for each such site. Our implemented technique finds $m$-bounded sites and obtains the bounds $m$ empirically. Specifically, it runs an instrumented version of the program on a sequence of sample inputs and records, for each allocation site and each input, the bound $m$ observed at that site for that input.[2] If all runs in which the site is invoked produce the same value $m$, we assume that the site is $m$-bounded.

One potential concern is that the bound $m$ observed while processing the sample inputs may, in fact, be too small: other executions may access more objects than the last $m$ objects allocated at the site. In this case the cyclic allocation may overlay two different live objects into the same memory, potentially causing the program to generate unacceptable results or even fail.

[2] In any single execution, every allocation site has a bound $m$ (which may be, for example, simply the number of objects allocated at that site).

To evaluate our technique, we implemented it and applied it to several sizable programs drawn from the open-source software community. We obtained the following results:

- **Memory Leak Elimination:** Our technique found 152 suitable $m$-bounded allocation sites and detected a memory leak at 3 of these sites. Moreover, some of these memory leaks make the programs vulnerable to denial of service attacks — certain carefully crafted requests cause the program to leak memory every time it processes the request. By presenting the program with a sequence of such requests, an attacker could cause the program to exhaust its address space and fail. Our technique is able to identify these sites, apply cyclic memory allocation, and effectively eliminate the memory leak (and the denial of service attack).
- **Accuracy:** We ran the programs on two sets of inputs: a training set (which is used to estimate the bounds) and a larger validation set (which is used to determine if any of the estimated bounds is too small). Our results indicate that this approach is quite accurate: the validation runs agree with the training runs on all but two of the 152 suitable $m$-bounded sites.
- **Reliability:** We performed a long-term test of the reliability of two of our programs (Squid and Pine) by installing them as part of our standard computing environment. In several months of usage, we observed no deviations from the correct behavior of the programs.
- **Impact of Cyclic Memory Allocation:** In two of the programs, the bounds estimates agree with the values observed in the validation runs and the use of cyclic memory allocation has no effect on the observable behavior of the program (other than eliminating memory leaks). For one of the remaining two programs with a bounds estimation error, the resulting overlaying of live objects has no effect on the externally observable behavior of the program during our validation runs. Moreover, an analysis of the potential effect of the overlaying indicates that it will *never* impair the overall functionality of the program. For the final program, the overlaying disables part of the functionality but leaves the remaining functionality intact.
- **Bounds Reduction Effect:** To further explore the potential impact of an incorrect bounds estimation, we artificially reduced the estimated bounds at each suitable $m$-bounded site with $m > 1$ and observed the effect that this artificial reduction had on the program's behavior. In some cases the reduction did not affect the observed behavior of the program at all; in other cases it impaired some of the program's functionality. But the reductions never caused a program to fail and in fact left the program able to execute code that accessed the overlaid objects to continue on to acceptably deliver the remaining functionality.

### 1.1 Usage Scenarios

The great strength of cyclic memory allocation is its ability to eliminate memory leaks without requiring the modification of the application program. Indeed, it is possible to find the bounds $m$ and eliminate memory leaks in stripped binaries with no access to source code and no programmer intervention whatsoever. Moreover, once one has obtained the bounds $m$, it is possible to automatically, seamlessly, and virtually instantaneously detect and eliminate otherwise fatal memory leaks in running programs. We are aware of no other technique that can provide this kind of automatic and immediate response — all previously existing techniques of which we are aware require some form of intervention to eliminate the leak by changing the application program. Of course, the tradeoff is that, if the bounds $m$ are too small, cyclic memory allocation may overlay live data which may, in turn, cause the program to behave unacceptably.

We therefore anticipate usage scenarios in which the consequences of memory leaks are severe enough and the risks associated with the possibility of unacceptable behavior manageable enough to justify the use of the technique. We note that the usage context of the program has a significant impact on this tradeoff. We outline several usage scenarios below. All scenarios first perform a set of training runs to find $m$-bounded allocation sites as outlined above.

- **Preemptive Leak Elimination:** In this scenario, the production runs use cyclic memory allocation at every allocation site with a bound $m$. The primary advantage of this approach is its simplicity — the application of cyclic memory allocation is straightforward and the resulting memory allocation algorithm is clear. A drawback is that this approach may overlay live objects even when there is no danger of running out of memory.
- **Offline Leak Elimination:** In this scenario, the production runs initially execute with standard memory management algorithms. If a run fails because of a memory leak at an $m$-bounded allocation site (the production runs identify such sites by tracking the difference between the number of allocated and deallocated objects for each site), it reexecutes using cyclic memory allocation to eliminate the leak at that site. For programs that are designed to process an input, produce a result, and then halt, it is possible to use cyclic memory allocation only for inputs that have already caused a production run to fail. For server programs that are designed to process an (in principle) unbounded number of dynamically presented inputs or requests, it may be preferable to use cyclic memory allocation for all future production runs.

  The advantage of this approach is that it minimizes the possibility of overlaying objects by using cyclic memory allocation only after the program has demonstrated that it cannot execute successfully with standard memory allocation algorithms. The disadvantage is that the program must fail before cyclic allocation eliminates the leak.
- **Latent Leak Elimination:** In this scenario the production runs track the difference between the number of allocated and deallocated objects for each site. If this difference grows significantly beyond $m$ for an $m$-bounded allocation site, future production runs use cyclic allocation at that site. The goal is to use evidence from successful production runs to find sites with *latent* memory leaks — leaks that may cause future runs to fail, but were not serious enough to cause the completed runs to fail. The use of cyclic memory allocation at such sites can eliminate such leaks before they can affect future runs.

  This approach does not risk overlaying live objects unless there is at least some evidence that there is a potentially dangerous leak for cyclic memory allocation to eliminate. It also has the advantage that it may be able to find and eliminate leaks before they cause the program to fail. Note, however, that it is still possible for a leak to cause a production run to fail if previous production runs provided no evidence of the presence of the leak.
- **Online Leak Elimination:** In this scenario, the program starts out using standard memory management algorithms, then dynamically switches to cyclic memory allocation when presented with evidence that it may be leaking memory from a specific allocation site. As it runs, the program monitors the amount of outstanding memory for each $m$-bounded allocation site. When this amount increases to the point that it may start to threaten the viability of the program, the program switches to cyclic memory allocation at that site only. The program continues to execute seamlessly through the switch — there is no need to restart the program or interrupt its execution in any way. After allocating at least $m$ objects in the cyclic buffer, the program can then reclaim all of the objects allocated at that site before the switch to cyclic memory allocation.

  By deploying cyclic memory allocation only in response to evidence of a potentially fatal leak, this approach runs the risk of overlaying live memory only when there is evidence that the program would otherwise fail. We therefore anticipate that this approach will be useful when continued program execution is an important priority and obtaining intervention is problematic, either because of programmer availability issues or response time requirements. Two of our sample programs, for example, (Xinetd and Squid) process external requests delivered via the Internet. Both of these programs have memory leaks that can be exercised by carefully crafted requests; these leaks make both programs vulnerable to denial of service attacks. The use of cyclic memory allocation in response to excess memory usage at specific allocation

sites can eliminate the vulnerability with no service outage whatsoever.

- **Static Memory Usage Bounds:** It is often useful to be able to obtain a guaranteed bound on the maximum amount of memory that a given program will ever consume. Such bounds can be especially useful for programs in embedded systems, which must typically execute 1) within a fixed amount of physical memory determined when the hardware is manufactured and 2) without any direct human oversight or control (and therefore without the possibility of human intervention should anything go wrong). The acquisition of a bound $m$ for every allocation site, in combination with the application of cyclic memory allocation at every allocation site in the program, can provide such an upper bound — the total amount of memory that any execution of the program will ever require is simply the total amount of memory required to hold all of the preallocated buffers plus the amount required for the call stack (which, in the absence of recursion, is usually straightforward to compute).

  We anticipate that the decision to apply cyclic memory allocation in this way will be driven largely by an analysis of the benefit of eliminating one of the primary failure modes (running out of memory) as compared to the risk of introducing unacceptable behavior by overlaying live objects. Given the severe consequences of system failure in many embedded systems, we anticipate that developers of embedded systems may choose to be more aggressive in their application of cyclic memory allocation than developers operating in contexts in which the consequences of failure are less severe. Note that even if the developer chooses not to use cyclic memory allocation at all sites, its use at some sites can simplify the overall memory usage analysis by enabling the developer to focus only on the remaining sites that do not use cyclic memory allocation.

- **Checked Results:** Many programs are used simply as a tool to generate a given result. A word processor, for example, may be used simply to produce a document; a game program may be used simply to provide the experience of playing the game. In many cases the acceptability of the result may be immediately apparent upon inspection or via an automated check [8, 6]. In such cases, cyclic memory allocation may enable the program to produce a result that can be then checked for acceptability. The program would first execute with the standard memory management algorithms. If it failed with a memory leak, it would then execute with cyclic memory allocation, which would ideally allow it to produce a result which would then be checked for acceptability. Note that the acceptability check eliminates any risk associated with the use of cyclic memory allocation — once one has obtained a result that passes the acceptability check, the specific mechanism used to generate the result is irrelevant.

It is also possible to use cyclic memory allocation to provide a quick fix for newly discovered memory leaks while programmers work to change the program to eliminate the leaks. It is possible to apply this technique at any allocation site and even in the absence of any *a priori* indication of what reasonable values for the bounds $m$ might be — one can simply use testing to search for bounds that enable the program to execute acceptably for an appropriate test suite. And of course, it is possible to use cyclic memory allocation without risk once programmers or an automated analysis have verified that the bounds $m$ are adequate for all executions.

We note that cyclic memory allocation may not be appropriate in all circumstances. In particular, if successful human intervention is readily available within an acceptable period of time, if it is feasible to recover from the leak by restarting the program (and it is acceptable to lose service during the restart), or if the consequences of an unacceptable result are so severe that one would rather have no result at all than run the risk of silently obtaining an unacceptable result, then it may be preferable to simply let the program exhaust its address space and fail.

### 1.2 Other Resource Leaks

It is, of course, possible for programs to leak other resources such as file descriptors or processes. These kinds of leaks can have equally disabling effects — a program that exhausts its supply of file descriptors may be unable to write an output file or create a socket to write a result to a client; a server that exhausts its supply of processes may be unable to service an incoming request. It is straightforward to generalize the memory leak elimination technique in this paper to eliminate these kinds of resource leaks as well — the program could respond to its inability to allocate a new file descriptor or process by simply closing or terminating an outstanding file descriptor or process, then reallocating the resource to satisfy the allocation request. Experience using a variant of least-recently used allocation to manage a fixed-size pool of file descriptors indicates that it is an effective file descriptor management technique (at least for the tested applications) that eliminates any possibility of a file descriptor leak [18].

### 1.3 Unsound Transformations In General

Since its inception, the field of automated program transformation has focused almost exclusively on sound transformations that do not change the semantics of the program. But soundness is a harsh restriction that significantly limits the broad range of potentially viable transformations that researchers could, in principle, investigate. It may therefore be worthwhile to adopt a more balanced perspective that recognizes the potential benefits that unsound transformations can deliver. Cyclic memory allocation is a concrete example of

an unsound transformation whose benefits can easily (in at least some contexts) outweigh the risks associated with its ability to change some aspect of the program's behavior. An increased willingness to consider unsound transformations might very well enable researchers to discover more useful transformations that have this desirable property.

### 1.4 Contributions

This paper makes the following contributions:

- **$m$-Bounded Allocation Sites:** It identifies the concept of an $m$-bounded allocation site and presents a technique for identifying memory leaks at these sites.
- **Empirical Bounds Estimation:** It presents a methodology for empirically estimating the bounds at each allocation site. This methodology consists of instrumenting the program to record the observed bound for an individual execution, then running the program on a range of training inputs to find allocation sites for which the sequence of observed bounds is the same.
- **Automatic Memory Leak Elimination:** It proposes the use of cyclic memory allocation for $m$-bounded allocation sites as a mechanism for eliminating memory leaks at those sites.
- **Experimental Results:** It presents experimental results that characterize how well the technique works on several sizable programs drawn from the open-source software community. The results show that our technique can detect and eliminate memory leaks in these programs and that the programs can, in many cases, provide much if not all of the desired functionality even when the bounds are artificially reduced to half of the observed values. One intriguing aspect of these results is the level of resilience that the programs exhibit in the face of overlaid data.

The remainder of the paper is structured as follows. Section 2 presents an example that illustrates our approach. Section 3 describes the implementation in detail. Section 4 presents our experimental evaluation of the technique. Section 5 discusses related work. We conclude in Section 6.

## 2. Example

Figure 1 presents a (simplified) section of code from the Squid web proxy cache version 2.4.STABLE3 [4]. At line 8 the procedure `snmp_parse` allocates a buffer `bufp` to hold a `Community` identifier. At lines 18 and 19 the procedure `snmpDecodePacket` writes a reference to the allocated buffer into a structure `checklist` allocated on the stack; at line 25 it writes a reference to the buffer into its parameter `rq`. The procedure `snmpDecodePacket` passes both `checklist` and `rq` on to other procedures. This pattern repeats further down the invoked sequence of procedures.

The procedure `snmpDecodePacket` is called by the procedure `snmpHandleUdp`, which passes a pointer to itself as an argument to `CommSetSelect`, which then stores

```
 1:  u_char *
 2:  snmp_parse(struct snmp_session * session,
 3:             struct snmp_pdu * pdu,
 4:             u_char * data,
 5:             int length)
 6:  {
 7:    int CommunityLen = 128;
 8:    bufp = (u_char *)xmalloc(CommunityLen+1);
 9:    return (bufp);
10:  }
11:  static void
12:  snmpDecodePacket(snmp_request_t * rq)
13:  {
14:    u_char *Community;
15:    aclCheck_t checklist;
16:    Community =
17:      snmp_parse(&Session, PDU, buf, len);
18:    checklist.snmp_community =
19:      (char *) Community;
20:    if (Community)
21:    allow = aclCheckFast(
22:      Config.accessList.snmp, &checklist);
23:    if ((snmp_coexist_V2toV1(PDU))
24:         && (Community) && (allow)) {
25:      rq->community = Community;
26:      snmpConstructReponse(rq);
27:    }
28:  }
29:  void
30:  snmpHandleUdp(int sock, void *not_used)
31:  {
32:    commSetSelect(sock, COMM_SELECT_READ,
33:                  snmpHandleUdp, NULL, 0);
34:    if (len > 0)
35:      snmpDecodePacket(snmp_rq);
36:  }
```

**Figure 1.** Memory leak from Squid

a reference to `snmpHandleUdp` in a global table of structs indexed by socket descriptor numbers. The program then uses the stored reference as a callback.

Any analysis (either manual or automated) of the lifetime of the `bufp` buffer allocated at line 9 in `snmp_parse` would have to track this complex interaction of procedures and data structures to determine the lifetime of the buffer and either insert the appropriate call to `free` or eliminate all the references to the buffer (if the program is using garbage collection). Any such analysis would, at least, need to perform an inter-procedural analysis of heap-aliased references in the presence of procedure pointers. In this case the programmer either was unable to or failed to perform this analysis. The program uses explicit allocation and deallocation, but (apparently) never deallocates the buffers allocated at this site and therefore contains a memory leak [1].

When we run the instrumented version of Squid on a variety of inputs, the results indicate that the allocation site at

line 9 is an $m$-bounded site with bound $m = 1$ — in other words, the program only accesses the last object allocated at that site. The use of cyclic memory allocation for this site with a buffer size of one object eliminates the leak and, to the best of our ability to determine, does not harm the correctness of the program. In particular, we used this version of Squid in our standard computational environment as a proxy cache for several months without a single observed problem. During this time Squid successfully served more than 100,000 requests.

## 3. Implementation

Our memory management technique contains three components. The first component instruments the program to observe all memory allocations and accesses when the program runs. The second analyzes the resulting data to find memory leaks and $m$-bounded allocation sites. The third component replaces, at each $m$-bounded allocation site, the invocation of the standard allocation procedure (`malloc` in our current implementation) with an invocation of a procedure that implements cyclic memory management for that site. This component also replaces the standard deallocation procedure (`free` in our current implementation) with a modified version that operates correctly in the presence of cyclic memory management by discarding attempts to explicitly deallocate objects allocated in cyclic buffers. It also similarly replaces the standard `realloc` and `calloc` procedures.

### 3.1 Instrumentation

The instrumented version of the program records the allocation site, address range, and sequence number for each allocated object. The address range consists of the beginning and ending addresses of the memory that holds the object. The sequence number is the number of objects allocated at that site prior to the allocation of the given object. So, the first object allocated at a given site has sequence number 0, the second sequence number 1, and so on.

As the program runs, the instrumentation maintains the following values for each allocation site:

- The number of objects allocated at that site so far in the computation.
- The number of objects allocated at that site that have been deallocated so far in the computation.
- An observed bound $m$, which is a value such that 1) the computation has, at some point during the execution, accessed an object allocated at that site $m-1$ allocations before the most recent allocation, and 2) the computation has never accessed any object allocated at that site more than $m-1$ allocations before the most recently allocation.

The instrumentation uses the Valgrind `addrcheck` tool to obtain the sequence of addresses that the program accesses as it executes [5]. It processes each accessed memory address and uses the recorded address range information to determine the allocation site and sequence number for the accessed object. It then compares the sequence number of the accessed object with the number of objects allocated at the allocation site so far in the computation and, if necessary, appropriately updates the observed bound $m$.

### 3.2 Finding $m$-Bounded Allocation Sites

Our technique finds $m$-bounded allocation sites by running the instrumented version of the program on a sequence of training inputs of increasing size. When it finishes running the program on all of the training inputs, it compares the observed bounds $m$ for each allocation site. If all of these bounds are the same for all of the inputs, it concludes that the site is $m$-bounded with bound $m$.

### 3.3 Finding Leaking Allocation Sites

Consider an allocation site with an observed bound $m$. If the difference between the number of objects allocated at that site and the number of deallocated objects allocated at that site is larger than $m$, there is a memory leak if the difference either 1) increases during a single run or 2) increases as the size of the input increases.

### 3.4 Cyclic Memory Management Algorithm

We have implemented our cyclic memory management algorithm for programs written in C that explicitly allocate and deallocate objects (in accordance with the standard C semantics, each object is simply a block of memory). Each $m$-bounded allocation site is given a cyclic buffer with enough space for $m$ objects. The allocation procedure simply increments through the buffer returning the next object in line, wrapping back around to the beginning of the buffer after it has allocated the last object in the buffer.

A key issue our implementation must solve is distinguishing references to objects allocated in cyclic buffers from references to objects allocated via the normal allocation and deallocation mechanism. The implementation performs this operation every time the program deallocates an object — it must turn all explicit deallocations of objects allocated at $m$-bounded allocation sites into no-ops, while successfully deallocating objects allocated at other sites. The implementation distinguishes these two kinds of references by recording the starting and ending addresses of each buffer, then comparing the reference in question to these addresses to see if it is within any of the buffers. If so, it is a reference to an object allocated at an $m$-bounded allocation site; otherwise it is not.

### 3.5 Variable-Sized Allocation Sites

Some allocation sites allocate objects of different sizes at different times. We extend our technique to work with these kinds of sites as follows. We first extend our instrumentation technique to record the maximum size of each object allocated at each allocation site. The initial size of the buffer is set to $m$ times this maximum size — the initial assumption

is that the sizes observed in the training runs are representative of the sizes that will be observed during the production runs.

At the start of each new allocation, the allocator has a certain amount of memory remaining in the buffer. If the newly allocated object fits in that remaining amount, the allocator places it in the remaining amount, with subsequently allocated objects placed after the newly allocated object (if they fit). If the newly allocated object does not fit in the remaining amount but does fit in the buffer, the allocator places the allocated object at the start of the buffer. Finally, if the newly allocated object does not fit in the buffer, the allocator allocates a new buffer of size $max(2 * m * r, 3 * s)$, where $r$ is the size of the newly allocated object and $s$ is the size of the largest existing buffer for that site.

Note that although this extension may allocate new memory to hold objects allocated at the site, the total amount of memory devoted to these objects is a linear function of the size of the largest single object allocated at the site, not a function of the number of objects allocated at the site.

### 3.6 Applying Cyclic Memory Allocation

As discussed above in Section 1.1, there are a range of ways to use cyclic memory allocation in practice. For evaluation purposes, however, we developed the concept of a *suitable* $m$-bounded allocation site and applied cyclic memory allocation at all such suitable sites. An $m$-bounded site is suitable if the total number of objects allocated at the site is either one or at least three times the bound $m$ in all training runs in which the site is invoked. The rationale is to apply cyclic memory allocation to allocation sites that produce a significant number of dead objects in the training runs. Another goal is to capture sites that allocate singleton objects during initialization.

### 3.7 Failure-Oblivious Computing

Overlaying live objects has the potential to introduce execution anomalies such as out of bounds memory accesses, null pointer dereferences, multiple deallocations of the same object, and infinite loops. In standard program execution environments, such anomalies can easily cause the program to fail.

Failure-oblivious computing is a collection of techniques that are designed to enable programs to execute through such anomalies to continue to deliver acceptable service to their users. We have previously applied failure-oblivious computing to several widely-used open-source servers [26]. Our results show that failure-oblivious computing 1) eliminates security vulnerabilities caused by buffer-overflow errors in these servers and 2) enables these servers to execute successfully through attacks that trigger these buffer-overflow errors. The servers then continue on to correctly service subsequent requests.

We apply failure-oblivious computing to targeted sites in the code as appropriate to ameliorate (and in many cases even eliminate) any global effect of any anomalies that overlaying live objects may introduce. Specifically, we discard any writes via null or out of bounds pointers and apply a technique to heuristically exit infinite loops. This last technique bounds the maximum number of iterations of each loop at $10^3 i$, where $i$ is the largest previously observed number of iterations of the loop (when the loop exits normally and not because of the bound on the number of iterations).[3] We use training runs to obtain the initial observed values $i$; if a loop does not execute during the training runs, we impose no bound on the number of iterations the first time the loop executes. Subsequent executions of the loop use the largest observed number of iterations $i$ from previous executions to bound the number of iterations to be at most $10^3 i$. Both techniques (discarding out of bounds writes and heuristically exiting infinite loops) preserve the default flow of control in that execution continues with the next statement after the write or loop.

## 4. Evaluation

We evaluate our technique by applying it to several sizable, widely-used programs selected from the open-source software community. These programs include:

- **Squid:** Squid is an open-source, full-featured Web proxy cache [4]. It supports a variety of protocols including HTTP, FTP, and, for management and administration, SNMP. We performed our evaluation with Squid Version 2.4STABLE3, which consists of 104,573 lines of C code.
- **Freeciv:** Freeciv is an interactive multi-player game [2]. It has a server program that maintains the state of the game and a client program that allows players to interact with the game via a graphical user interface. We performed our evaluation with Freeciv version 2.0.0beta1, which consists of 342,542 lines of C code.
- **Pine:** Pine is a widely used email client [3]. It allows users to read mail, forward mail, store mail in different folders, and perform other email-related tasks. We performed our evaluation with Pine version 4.61, which consists of 366,358 lines of C code. We used Pine to access the Unix mail file and Pine mail files such as folders. All of these files are stored on the same machine running Pine.
- **Xinetd:** Xinetd provides access control, logging, protection against denial of service attacks, and other management of incoming connection requests. We performed our evaluation with Xinetd version 2.3.10, which consists of 23,470 lines of C code.

[3] In some very small number of cases (typically the main event-processing loop of the program), the programmer may actually intend a loop to execute forever. We allow the programmer to identify such loops and disable the loop exiting technique for these loops.

Note that all of these programs may execute, in principle, for an unbounded amount of time. Squid and Xinetd, in particular, are typically deployed as part of a standard computing environment with no expectation that they should ever terminate. Memory leaks are especially problematic for these kinds of programs since they can affect the ability of the program to execute successfully for long periods of time.

Our evaluation focuses on two issues: the ability of our technique to eliminate memory leaks and on the potential impact of an incorrect estimation of the bounds $m$ at different allocation sites. While we did not obtain detailed performance results, our experience using the programs made it clear that the use of cyclic memory allocation does not significantly affect the performance of our programs — all programs exhibited basically the same performance with or without cyclic memory allocation. We perform the following experiments for each program:

- **Training Runs:** We select a sequence of training inputs of increasing size and run the instrumented version of the program on these inputs to find suitable $m$-bounded allocation sites and to obtain the estimated bounds $m$ for these sites as described in Section 3.2.
- **Validation Runs:** We select a validation input. This input is different from and larger than the training inputs and is intended to exercise strictly more of the functionality of the program than the training inputs. We run the instrumented version of the program (both with and without cyclic memory allocation applied at suitable $m$-bounded sites) on this input. We use the collected results to determine 1) the accuracy of the estimated bounds from the training runs and 2) the effect of any overlaying of live objects on the behavior of the program (this overlaying would be caused by observed bounds $m$ that are too small).
- **Conflict Runs:** For each suitable $m$-bounded allocation site with $m > 1$, we construct a version of the program that uses the bound $\lceil m/2 \rceil$ at that site instead of the bound $m$. We then run this version of the program on the validation input. We use the collected results to evaluate the effect of the resulting overlaying of live objects on the behavior of the program.
- **Long-Term Usage:** Squid and Pine are part of the standard computing environment of the first author of this paper. This author replaced the standard versions of these programs with versions that use cyclic memory allocation for all suitable $m$-bounded sites identified during the training runs. He then used the versions with cyclic memory management exclusively for several months prior to the submission of this paper.

Table 1 presents the percentage of executed allocation sites that the training runs identify as suitable $m$-bounded sites, the percentage of memory allocated at these sites (in both the training and validation runs combined), and the percentage of invalidated sites (sites for which the observed bound $m$ was too small) for each of our programs. In general, the training runs identify roughly half of the executed sites as suitable $m$-bounded sites, there is significant amount of memory allocated at those sites, and there are almost no invalidated sites — the training runs deliver observed bounds that are consistent with the bounds observed in the validation runs at all but two of the 152 sites with observed bounds $m$ in the entire set of programs.

| Program | % $m$-bounded | % memory | % invalidated |
|---|---|---|---|
| Squid | 55.7 | 86.0 | 2.9 |
| Freeciv | 48.3 | 84.9 | 0.0 |
| Pine | 60.0 | 15.0 | 1.5 |
| Xinetd | 58.8 | 89.8 | 0.0 |

**Table 1.** Memory Allocation Statistics

We next discuss the interaction of cyclic memory allocation with each of our benchmark programs. To evaluate the impact of cyclic memory allocation on any memory leaks, we compare the amount of memory that the original version of the program (the one without cyclic memory allocation) consumes to the amount that the versions with cyclic memory allocation consume.

### 4.1 Squid

Our training inputs for Squid consist of a set of HTTP links that we obtained from Google news, CNN, BBC, MSN, a set of FTP links from the mirrors for Apache, OpenSSH, the ftp server at the NUS School of Computing, and a set of SNMP queries that we generated from a Python script that we developed for this purpose. The training inputs have from 122 to 863 links and from 20 to 80 SNMP queries. The number of attributes queried ranges from 2 to 8. Our validation input consists of a larger set of links (3041) from the same sites mentioned above and a larger set of SNMP queries (110) from our Python script. The validation SNMP queries contain more variables than the training queries.

**Training and Validation Runs**: Our training runs detected 34 suitable $m$-bounded allocation sites out of a total of 61 allocation sites that executed during the training runs; 30.7% of the memory allocated during the training runs was allocated at suitable $m$-bounded sites. Table 2 presents a histogram of the observed bounds $m$ for all of the suitable $m$-bounded sites. This table indicates that the vast majority of the observed bounds are small (a pattern that is common across all of our programs). The validation run determined that the bound $m$ was too small for one of the 34 $m$-bounded sites.

| m | 1 | 2 | 3 | 14 |
|---|---|---|---|---|
| # sites | 30 | 2 | 1 | 1 |

**Table 2.** $m$ distribution for Squid

**Effect of Overlaying Live Objects**

The SNMP module in Squid stores objects in a tree, with each object named remotely by a path through the tree. Each node in the tree holds a data value that remote nodes use to identify and query that node in paths through the SNMP tree. The objects allocated at the invalidated site hold the data values in the SNMP tree nodes. Overlaying these objects makes Squid unable to find SNMP objects whose corresponding path through the tree contains nodes whose data value objects have been overwritten by subsequently allocated data value objects from other nodes. The net effect is that the SNMP module sometimes fails to locate an object named in a remote query, returning instead an indication that the specified SNMP object was not found.

**Memory Leaks**: Our results show that Squid has a memory leak in the SNMP module; this memory leak makes Squid vulnerable to a denial of service attack [1]. Our training runs indicate that the allocation site involved in the leak is an $m$-bounded site with $m$=1. The use of cyclic allocation for this site eliminates the leak. Figure 2 presents the effect of eliminating the leak. This figure plots Squid's memory consumption as a function of the number of SNMP requests that it processes with and without cyclic memory allocation. As this graph demonstrates, the memory leak causes the memory consumption of the original version to increase linearly with the number of SNMP requests — this version leaks memory every time it processes an SNMP request. In contrast, the memory consumption line for the version with cyclic memory allocation is flat, clearly indicating the elimination of the memory leak. Note that it is possible, by sending sufficiently many SNMP requests, to make the memory consumption of the original version arbitrarily larger than the memory consumption of the version with cyclic memory allocation.

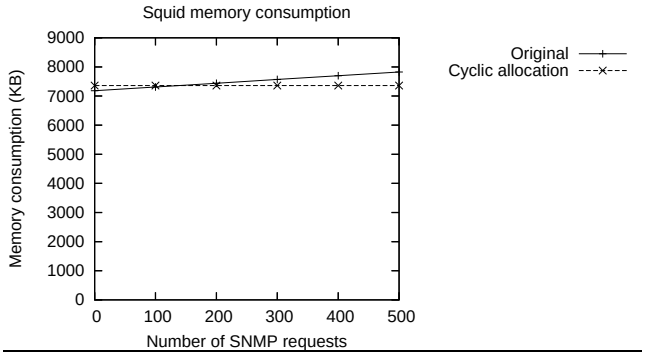


**Figure 2.** Squid memory consumption

**Conflict Runs**: For Squid, the training runs find a total of four suitable $m$-bounded allocation sites with $m$ greater than one. One of these sites is the invalidated site discussed above. Artificially reducing the size of the buffer from 14 objects to 7 objects increases the number of queries that the Squid SNMP module is unable to service, with the remaining functionality otherwise unaffected. We next discuss our results from the conflict runs for the remaining three sites. These results provide additional insight into the potential effect of overlaying live objects in this program.

The first site we consider holds metadata for cached HTTP objects; the metadata and HTTP objects are stored separately. When we reduce the bound $m$ at this site from 3 to 2, the MD5 signature of one of the cached objects is overwritten by the MD5 signature of another cached object. When Squid is asked to return the original cached object, it determines that the MD5 signature is incorrect and refetches the object. The net effect is that some of the time Squid fetches an object even though it has the object available locally; an increased access time is the only potential effect.

The next site we consider holds the command field for the PDU structure, which controls the action that Squid takes in response to an SNMP query. When we reduce the bound $m$ from 2 to 1, the command field of the structure is overwritten to a value that does not correspond to any valid SNMP query. The procedure that processes the command determines that the command is not valid and returns a null response. The net effect is that Squid is no longer able to respond to any SNMP query at all. Squid still, however, processes all other kinds of requests without any problems at all.

The next site we consider holds the values of some SNMP variables. When we reduce the bound $m$ from 2 to 1, some of these values are overwritten by other values. The net effect is that Squid sometimes returns incorrect values in response to SNMP queries. Squid's ability to process other requests remains unimpaired.

**Long-Term Usage**: During the long-term usage period, the version of Squid with cyclic memory allocation served more than 100,000 requests with a variety of content types (html, images, binaries, ...) and languages (English and Vietnamese). We observed no problems, errors, or anomalies.

### 4.2 Freeciv

Freeciv allows both human and AI (computer implemented) players to compete in a civilization-building game. Our training inputs for Freeciv consist of from 2 to 30 AI players. The sizes of the game map range from size 4 to size 15 and the games run from 100 to 200 game years. Our validation input consists of 30 AI players and a map size of 20. The game runs for 400 game years.

**Training and Validation Runs**: Our training runs detected 42 suitable $m$-bounded allocation sites out of a total of 87 allocation sites that executed during the training runs; 81.2% of the memory allocated during the training runs was allocated at suitable $m$-bounded sites. Table 3 presents a histogram of the observed bounds $m$ for all of the suitable $m$-

bounded sites. As for the other programs, the vast majority of the observed bounds are small. All of the observed bounds in the validation run are consistent with the observed bounds in the training runs; the use of cyclic memory allocation therefore does not change the behavior of the program.

| m | 1 | 2 |
|---|---|---|
| # sites | 39 | 3 |

**Table 3.** $m$ distribution for Freeciv

**Memory Leaks**: Freeciv has a memory leak associated with an allocation site repeatedly invoked during the processing of each AI player. This allocation site allocates an array of boolean flags that store the presence or absence of threats from oceans. The training runs determine that this allocation site is an $m$-bounded site with $m$=1. Cyclic memory allocation eliminates this memory leak.

**Conflict Runs**: Freeciv has three suitable $m$-bounded allocation sites with $m$ greater than 1; all of these sites have $m$=2. All three of these sites are part of the same data structure: a priority queue used to organize the computation associated with path-finding for an AI player. Each priority queue has a header, which in turn points to an array of cells and a corresponding array of cell priorities. The training and validation runs both indicate that, at all three of these sites, the program accesses at most the last two objects allocated. Further investigation reveals that (at any given time) there are at most two live queues: one for cells that have yet to be explored and one for cells that contain something considered to be dangerous. During its execution, however, Freeciv allocates many of these queues.

The first allocation site we consider holds the queue header. Reducing the bound for this site from 2 to 1 causes the size field in the queue header to become inconsistent with the length of the cell and priority arrays. The application of failure-oblivious computing enables the program to execute successfully through the resulting out of bounds array accesses. Reducing the bounds for the other two sites causes either the cell arrays or the cell priorities to become overlaid. In all three cases the program is able to execute successfully without a problem. While the overlaying may affect the actions of the AI players, it is difficult to see this as a serious problem since it does not cause the AI players to violate the rules of the game or visibly degrade the quality of their play.

### 4.3 Pine

Pine is a widely-used email program that allows users to read, forward, and store email messages in folders. Our training inputs have between 1 and 4 mail folders containing between 10 and 97 email messages. The number of attachments ranges from 0 to 4. Our validation input has 24 mail folders that contain more than 2,500 mail messages.

**Training and Validation Runs**: Our training runs detected 66 suitable $m$-bounded allocation sites out of a total of 110 allocation sites that executed during the training runs; 15.0% of the memory allocated during the training runs was allocated at suitable $m$-bounded sites. Table 4 presents a histogram of the observed bounds $m$ for all of the suitable $m$-bounded sites. As for the other programs, the vast majority of the observed bounds are small. The validation run revealed that the observed bound $m$ was too small for 1 of the 66 $m$-bounded allocation sites. In this case we say that the validation run *invalidated* this site. Neither the training nor validation runs revealed a memory leak in Pine.

| m | 1 | 3 | 8 |
|---|---|---|---|
| # sites | 64 | 1 | 1 |

**Table 4.** $m$ distribution for Pine

**Effect of Overlaying Live Objects**: The objects allocated at the invalidated site implement a circular doubly-linked list of status messages for Pine to display on the status line. Overlaying these objects causes Pine to dereference a null pointer. The application of failure-oblivious computing enables Pine to execute through these null pointer dereferences with no visible negative effect on the behavior of the program. An analysis of the relevant code indicates that overlaying these objects may have the potential to cause Pine to fail to display a status message. We did not, however, observe any missing messages during our training or validation runs.

**Conflict Runs**: Pine has two suitable $m$-bounded allocation sites with $m$ greater than 1. The first site is the invalidated site discussed above in Section 4.3. The training runs indicate that this site has an observed bound of $m$=3, but the validation runs indicate that Pine may access as many as the last 4 objects allocated at this site. Reducing the bound $m$ from 3 to 2 causes a write access via a null pointer. As discussed above in Section 4.3, the application of failure-oblivious computing enables Pine to execute successfully through these null pointer dereferences with no changes in the observable behavior of the program.

The other site has a bound $m$=8. This site allocates nodes that store content filters that Pine uses to convert special characters or format an input stream for display. These nodes are stored in a singly-linked list. Reducing the bound $m$ from 8 to 4 causes the list to become cyclic. In the absence of our technique for exiting infinite loops (see Section 3.7), this cyclicity would cause a loop that processes this list to fail to exit. The application of our infinite loop exiting technique causes the execution to proceed beyond this loop, which enables Pine to process most messages without any observable difference. For some messages that contain HTML tags, however, Pine inserts some additional incorrect characters. Note that the insertion of these characters does not substantially impair the legibility of the message.

**Long-Term Usage**: During the long-term usage period, the version of Pine with cyclic memory allocation processed over 2,500 mail messages stored in 11 mail folders. It successfully processed messages with a variety of formats (text, html, attachments, single and multiple receivers). It also successfully performed the full range of mail commands (read messages, save attachments, reply to messages, move messages between folders, delete messages, et cetera). We observed no problems, errors, or anomalies.

### 4.4 Xinetd

Our training inputs for Xinetd consist of between 10 and 200 requests. Our validation input consists of 500 requests. All of these requests are generated by a Perl script we developed for this purpose.

**Training and Validation Runs**: Our training runs detected 10 suitable $m$-bounded allocation sites out of a total of 17 allocation sites that executed during the training runs; 89.7% of the memory allocated during the training runs was allocated at suitable $m$-bounded sites. Table 5 presents a histogram of the observed bounds $m$ for all of the suitable $m$-bounded sites. All of the observed bounds $m$ are 1. All of the observed bounds in the validation run are consistent with the observed bounds in the training runs; the use of cyclic memory allocation therefore does not change the observable behavior of the program.

| m | 1 |
|---|---|
| # sites | 10 |

**Table 5.** $m$ distribution for Xinetd

**Memory Leaks**: Xinetd has a leak in the connection-handling code — whenever Xinetd rejects a connection (it is always possible for an attacker to generate connection requests that Xinetd rejects), it leaks a connection structure 144 bytes long. Our training runs indicate that the allocation site involved in the leak is an $m$-bounded site with $m$=1. The use of cyclic allocation for this site eliminates the leak. Figure 3 presents the effect of eliminating the leak. This figure plots Xinetd's memory consumption as a function of the number of rejected requests with and without cyclic memory allocation. As this graph demonstrates, the memory leak causes the memory consumption of the original version to increase linearly with the number of rejected requests. In contrast, the memory consumption line for the version with cyclic memory allocation is flat, clearly indicating the elimination of the memory leak.

Note that because none of the suitable $m$-bounded allocation sites in Xinetd have $m$ greater than one, we do not investigate the effect of reducing the bounds.

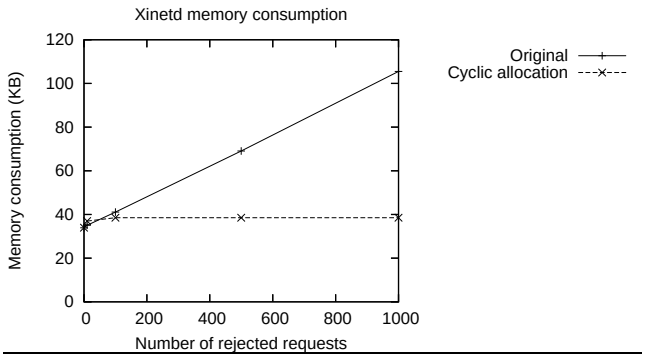


**Figure 3.** Xinetd memory consumption

### 4.5 Discussion

Memory leaks are an insidious problem — they are difficult to find and (as the discussion in Section 2 illustrates) can be difficult to eliminate even when the programmer is aware of their presence. Our experience with our four programs underscores the difficulty of eliminating memory leaks — despite the fact that all of these programs are widely used, and in some cases, crucial, parts of open-source computing environments, three of the four programs contain memory leaks.

Our results indicate that cyclic memory allocation enabled by empirically determined bounds $m$ can play an important role in eliminating memory leaks. Our results show that this technique eliminates a memory leak in three of our four programs. If the bounds $m$ are accurate, there is simply no reason not to use this technique — it is simple, easy to implement, and provides a hard bound on the amount of memory required to store the objects allocated at $m$-bounded sites. In this situation there are two key questions: 1) how accurate are the observed bounds, and 2) what are the consequences if the observed bounds are wrong?

In practice, the accuracy of the bounds is a function of the input training set. Our results indicate that the observed bounds in our experiments are reasonably accurate (the validation inputs invalidate only two of the 152 suitable $m$-bounded allocation sites). Moreover, the incorrect bounds at the two invalidated sites are due to (in retrospect) obvious training set deficiencies. This suggests that information about $m$-bounded allocation sites could be used to uncover previously unsuspected deficiencies in test suites designed to comprehensively explore the space of relevant program behaviors. Specifically, one could examine each $m$-bounded site and ask if it seems reasonable for the site to have the same bound in all possible executions. For the two invalidated sites in our evaluation, such an investigation immediately reveals important program behaviors that the training set failed to exercise.

The conflict runs also provide some insight into the effect of overlaying live objects. With standard C semantics, overlaying live objects caused the conflict runs to fail for five

of the nine allocation sites. But the application of failure-oblivious computing shows that the errors that caused the programs to fail do not reflect the presence of any fundamental damage to the program's ability to execute successfully. Indeed, failure-oblivious computing enabled all of the conflict runs to execute without failure, with the programs remaining fully functional for six of the nine allocation sites! These results indicate that overlaying live objects at the nine allocation sites in the conflict runs tends to produce localized errors from which the program can easily recover if one uses failure-oblivious computing to eliminate the inherent brittleness otherwise present in standard computing environments.

One aspect of our implementation that tends to ameliorate the negative effects of overlaying objects is the fact that different $m$-bounded allocation sites have different cyclic allocation buffers. The resulting memory management algorithm will typically preserve basic type safety even when the system overlays live objects — the objects sharing the memory will tend to have the same basic data layout and types and satisfy the same invariants. This property makes the program less likely to encounter a completely unexpected collection of data values when it accesses data from an overwriting object instead of the expected object. This is especially true for application data, in which the values for each conceptual data unit tend to be stored in a single object, with the values in multiple objects largely if not completely independent. Even if overlaying the objects allocated at those sites causes the program to lose the data required to implement the full functionality, it does not harm the ability of the program to execute code that accesses the overlaid objects. The program can therefore execute through this code without failing, preserving its ability to deliver other functionality.

Core data structures, on the other hand, tend to have important properties that cross object boundaries. Overlaying objects allocated at these sites tends to cause the program to violate these properties. In the absence of failure-oblivious computing, these violations may leave the program vulnerable to failures or infinite loops. In our experiments, however, failure-oblivious computing enabled our programs to execute successfully through these anomalies to deliver their full functionality to their users in spite of the data structure inconsistencies. It may also be possible to use data structure repair [11, 12] to eliminate any residual inconsistencies and enable the program to continue to execute successfully.

Interestingly enough, in some of the cases in which bounds reduction has no effect on the observable behavior, the program is actually set up to tolerate inconsistent values in objects. In one program (Squid) the program anticipates the possibility of inconsistent data and contains code to handle that case. In the other program (Freeciv) the program is able to successfully execute with a range of data values. These two examples suggest that many programs may already have some built-in capacity to fully tolerate inconsistent or unexpected data.

Finally, we note that some programs tend to have multi-purpose allocation sites, each of which allocates objects for a different conceptual purpose in the program. This can occur, for example, if the program uses several instances of a given data structure in different parts of the program. In this case a single allocation site inside the data structure implementation may allocate memory for different data structure instances in unrelated parts of the program. If the different data structure instances have different bounds $m$, our current implementation may miss an opportunity to apply cyclic memory allocation. This phenomenon can also cause our current implementation, in the absence of test suites that adequately explore the behavior of the different parts of the system, to incorrectly conclude that the bounds observed for some instances of the data structure reflect the (unexplored in the test suite) usage patterns of other instances. One obvious way to ameliorate this problem is to use the calling context to obtain different logical allocation sites for the different instances.

## 5. Related Work

We discuss related work in dynamic memory leak detection, static memory leak detection, and static memory leak elimination.

### 5.1 Dynamic Memory Leak Detection

Purify, Insure++, and other dynamic analysis tools [23, 19] provide dynamic memory leak detectors for programs with explicit memory management. The basic approach is to track object reachability to provide a list of unreachable objects that the program failed to deallocate. It is then the responsibility of the programmer to analyze the program, find the root cause of the leak, and modify the program to eliminate the leak. Note that these techniques are not designed to find memory leaks that involve reachable objects that the program will never access in the future.

Our approach, in contrast, automatically applies a transformation that eliminates the leak. The potential benefits include the elimination of the need for a programmer to analyze the program to find the leak, the elimination of the programming effort required to fix the leak, and the elimination of the possibility of an incorrect fix introducing additional errors into the program source. Because our approach is based on how the program accesses data (rather than reachability properties), it can detect and eliminate leaks even when the leaked objects remain reachable.

Jump and McKinley present a system that helps programmers find memory leaks by identifying data structures that grow systematically during the execution of the system [25]. Chilimbi and Hauswirth [14] present a dynamic approach that tracks allocations and frees, then periodically samples the memory accesses of the program to find "stale" objects which have not been freed and have not been accessed recently. It then identifies such objects as comprising potential memory leaks. It is the programmer's responsibility to de-

termine if the identified objects, in fact, comprise a memory leak and, if so, to modify the program to eliminate the leak.

Note that it is possible to extend Chilimbi and Hauswirth's approach to automatically eliminate leaks — simply deallocate the stale objects which their technique identifies as comprising potential memory leaks. With an appropriately tuned sampling and leak identification policy and the application of techniques such as failure-oblivious computing that ameliorate the negative effects of internal errors, it may be possible to drive the false positive rate down to a level where, for applications that place a premium on continued execution, the rewards of eliminating the memory leak could outweigh the risks of premature deallocation.

### 5.2 Static Memory Leak Detection

Hackett and Rugina [22], Heine and Lam [24], Chou [16], Bush, Pincus, and Sielaff [13], Evans [20], and Xie and Aiken [30] have all developed static analyses that discover memory leaks in programs with explicit memory management. All of the analyses check that the program correctly frees allocated objects before the object becomes unreachable. The analyses differ in the techniques they use to track the referencing relationships in the program: Evans' analysis tracks annotations that identify unique references to objects, Bush, Pincus and Sielaff's analysis symbolically simulates candidate execution traces, Hiene and Lam's analysis tracks synthesized ownership properties, Hackett and Rugina use an efficient shape analysis, Chou's analysis uses symbolic reference counting, and Xie and Aiken's analysis directly models references between objects to reason about how objects escape procedure call contexts. These analyses are designed for programs that use explicit memory management — a garbage collector would correctly reclaim all of the unreachable leaking objects that they identify.

In comparison with dynamic techniques (such as those discussed above and the technique that we present in this paper), the great advantage of static techniques is the elimination of the need to exercise the program on an input that exposes the leak. It is even possible to analyze incomplete programs or fragments of complete programs. Disadvantages include the need to implement a heavyweight static analysis, the possibility that the analysis will not scale, and the possibility that the inevitable analysis imprecisions may introduce false positives or false negatives. Some analyses also require the programmer to provide additional annotations [20, 30]. Because each static analysis is designed to recognize leaks that arise because of an interaction between a specific kind of reachability property and the memory management actions of the program, such analyses will fail to recognize leaks that involve reachable objects or objects with reachability properties that the analysis is not designed to analyze.

There is a tension between leak detection and leak elimination, especially when the leak elimination technique is potentially unsound (as ours is). During development there is usually an ample supply of programmers who understand the program and are readily able to modify it. Unless the leak elimination requires the development of new data structures to more precisely track object liveness, the costs of modifying the program to eliminate the leak may be quite low. After the program is deployed, however, the costs of modifying the program typically rise dramatically as the supply of programmers who understand the program dwindles. In this case automatic memory leak elimination via cyclic memory allocation can be much more effective than attempting to modify the program to eliminate the leak — it eliminates the need to invest programmer time and effort to understand and modify the program and eliminates the risk of inadvertently introducing new errors.

Another factor is the quality of the information that the leak detector provides. Both Hackett and Rugina's analysis and Hiene and Lam's analysis are sound and (because they are designed to recognize specific programming patterns that leak memory) are able to identify the location in the program that discards the last reference to the leaked object. In this case the modification to eliminate the leak is straightforward (and in fact, could be applied automatically). Unsound techniques or techniques that provide less of an indication why the leak occurred require much more programmer effort and the modification runs a much larger risk of introducing new errors.

### 5.3 Static Memory Leak Elimination

Shaham, Kolodner, and Sagiv present a static analysis designed to recognize and eliminate memory leaks that occur in data structures that maintain arrays of references to objects [28]. The basic idea is to find array elements that will always be overwritten before they are next read, then set such references to NULL, thereby potentially making the referenced object unreachable and enabling the garbage collector to reclaim the object. Shaham, Yahav, Kolodner, and Sagiv use a shape analysis to eliminate memory leaks in garbage-collected Java programs. The basic idea is to find and eliminate eliminate references that the program will no longer use [27].

Gheorghioiu, Salcianu, and Rinard present a static analysis for finding allocation sites that have the property that at most one object allocated at that site is live during any point in the computation [21]. The compiler then applies a transformation that preallocates a single block of memory to hold all objects allocated at that site. Potential implications of the technique include the elimination of any memory leaks at such sites, simpler memory management, and a reduction in the difficulty of computing the amount of memory required to run the program.

Interestingly enough, these analyses all consider the future referencing behavior of the program to find objects that the program will no longer access regardless of whether they are reachable or not. These analyses are therefore (in principle) capable of eliminating leaks regardless of whether the program uses explicit memory management or garbage col-

lection. This is in contrast with the static memory leak detection algorithms discussed above in Section 5.2. Because these analyses all find objects that become unreachable before they are deallocated, they are not appropriate for programs that use garbage collection.

Researchers have used escape analysis to enable stack allocation for objects that do not escape a given procedure call context [29, 7, 15]. Because these analyses were developed for programs that use garbage collection, they would typically not eliminate any memory leaks — the stack-allocated objects would become unreachable and reclaimed when the enclosing procedure call context exits even if they were allocated in the heap. But it should be possible to apply these analyses to programs with explicit memory allocation, in which case the transformation could eliminate memory leaks. In particular, stack allocation would eliminate memory leaks if the untransformed original program failed to explicitly deallocate the stack-allocated objects.

An advantage of all of these analyses is their soundness — the analysis considers all possible executions and does not apply the transformation unless the program will never allocate more than one live object from the site. Drawbacks include the need to develop a sophisticated static program analysis, the need to target specific usage patterns that leak memory, and the possibility that the analysis may not scale or may (because of the inevitable imprecisions in any static analysis) fail to find an important leak. For example, the Gheorghioiu, Salcianu, and Rinard analysis will eliminate a memory leak at a given allocation site only if at most one object allocated at the site is live at any point during the computation and if the program never stores a reference to an object allocated at that site into the heap. In practice, these restrictions severely limit its utility as a memory leak eliminator. In particular, this analysis would eliminate none of the leaks described in this paper. Our dissatisfaction with the limitations of these kinds of analyses, in part, led us to develop the approach we present in this paper.

### 5.4 Conservative Garbage Collection

Conservative garbage collection [10] can eliminate memory leaks in programs written in languages with explicit memory deallocation. In comparison with cyclic memory allocation, conservative collectors have the advantage that they are designed to reclaim only unreachable memory, which ensures that they cannot cause the allocator to overlay live data (assuming that the collector correctly locates all object references). A disadvantage of conservative collectors is that, like standard garbage collectors, they are unable to eliminate memory leaks in which the program retains (or appears to retain [9]) references to objects that it will not access in the future. Other potential issues include time and space overheads [17, 31].

It is possible to combine conservative garbage collection and cyclic memory allocation to eliminate memory leaks in languages with explicit deallocation. Specifically, one could use the conservative collector to reclaim unreachable memory, then dynamically switch to cyclic allocation at specific sites only in the presence of evidence that the program may be leaking memory at that site. Given the perceived safety of conservative collection, such an approach could further minimize the risk of overlaying live objects while still eliminating the possibility of severe memory leaks at sites with a bound $m$.

## 6. Conclusion

Memory leaks are an important source of program failures, especially for programs such as servers that must execute for long periods of time. Our cyclic memory allocation technique observes the execution of the program to find $m$-bounded allocation sites, which have the useful property that the program accesses at most only the last $m$ objects allocated at that site. It then exploits this property to preallocate a buffer of $m$ objects and cyclically allocate objects out of this buffer. This technique caps the total amount of memory required to store objects allocated at that site at $m$ times the size of the objects allocated at that site. Our results show that this technique can eliminate important memory leaks in long-running server programs.

One potential concern is the possibility of overlaying live objects in the same memory. Our results suggest that the risk of overlaying live objects is small, that the consequences of overlaying live objects are not severe (and that failure-oblivious computing can significantly ameliorate any negative consequences), and that the reward (eliminating important memory leaks) can be significant.

## 7. Acknowledgements

We would like to thank the anonymous reviewers for their useful feedback on the content and ideas in this paper.

## References

[1] CVE-2002-0069. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0069.

[2] Freeciv website. http://www.freeciv.org/.

[3] Pine website. http://www.washington.edu/pine/.

[4] Squid Web Proxy Cache website. http://www.squid-cache.org/.

[5] Valgrind website. http://www.valgrind.org/.

[6] K. Arkoudas and M. Rinard. Deductive runtime certifi cation. In *Proceedings of the 2004 Workshop on Runtime Verification (RV 2004)*, Barcelona, Spain, April 2004.

[7] B. Blanchet. Escape analysis for object oriented languages. application to Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[8] Manuel Blum and Sampath Kannan. Designing Programs That Check Their Work. In *STOC*, pages 86–97, 1989.

[9] H. Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1993.

[10] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9), September 1988.

[11] Brian Demsky and Martin Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '03)*, October 2003.

[12] Brian Demsky and Martin Rinard. Data Structure Repair Using Goal-Directed Reasoning. In *Proceedings of the 2005 International Conference on Software Engineering*, May 2005.

[13] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software — Practice & Experience*, 30(7), June 2000.

[14] T. Chilimbi and M. Hauswirth. 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*, October 2004.

[15] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[16] A. Chou. *Static Analysis for Bug Finding in Systems Software*. PhD thesis, Stanford University, 2003.

[17] D. Detlefs, A. Dosser, and B. Zorn. Memory Allocation Costs in Large C and C++ Programs. Technical Report CU-CS-665-93, University of Colorado, Boulder, August 1993.

[18] Octavian-Daniel Dumitran. Fixing file descriptor leaks. Master's thesis, Massachusetts Institute of Technology, June 2007.

[19] Cal Erikson. Memory leak detection in c++. *Linux Journal*, (110), June 2003.

[20] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, June 1996.

[21] Ovidiu Gheorghioiu, Alexandru Salcianu, and Martin Rinard. Interprocedural compatability analysis for static object preallocation. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, January 2003.

[22] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'05)*, January 2005.

[23] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, December 1992.

[24] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.

[25] Maria Jump and Kathryn S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *POPL*, pages 31–38, 2007.

[26] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, December 2004.

[27] Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Mooly Sagiv. Establishing Local Temporal Heap Safety Properties with Applications to Compile-Time Memory Management. In *The 10th Annual International Static Analysis Symposium (SAS '03)*, June 2003.

[28] R. Shaham, E. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in java. In *Proceedings of the International Conference on Compiler Construction (CC '00)*, March-April 2000.

[29] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[30] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of ESEC/FSE 2005*, September 2005.

[31] B. Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23(7), July 1993.