Using Cyclic Memory Allocation to Eliminate Memory Leaks

Huu Hai Nguyen and Martin Rinard
Computer Science and Artificial Intelligence Lab
Singapore-MIT Alliance
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

We present and evaluate a new memory management technique for 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. The technique then 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 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 one of the 160 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.

*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.

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 memory 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 memory management technique for eliminating memory leaks. This technique applies to allocation sites [1] that satisfy the following property:

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.*

It is possible to use the following memory management scheme for objects allocated at a given m-bounded allocation site:

- **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:** This memory management scheme 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 both to implement and to operate. Unlike many previously proposed static analysis techniques [14, 9, 18, 16, 12, 23, 21, 20, 15], it does not require the development of a heavyweight static analysis or programmer annotations to detect and/or eliminate memory leaks.

[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.

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 estimates 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 the sequence of observed bounds stabilizes at a value m, we assume that the allocation site is m-bounded and use cyclic allocation for that site.

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 site. In this case the program may overlay two different live objects in the same memory, potentially causing the program to generate unacceptable results or even fail.

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:** Several of our programs contain memory leaks at m-bounded allocation 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 is processes the request. By presenting the program with a sequence of such requests, an attacker can 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 evaluate the accuracy of our empirical bounds estimation approach by running 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 show that this approach is quite accurate: the validation runs agree with the training runs on all but one of the 160 sites that the training runs identify as m-bounded.

- **Reliability:** We also 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 all but one 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). Even for the one program with a single bounds estimation error (and as described further in Section 4.3.2), 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.

- **Bounds Reduction Effect:** To further explore the potential impact of an incorrect bounds estimation, we artificially reduced the estimated bounds at each 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.

Our conclusion is that cyclic memory allocation with empirically estimated bounds provides a simple, intriguing alternative to the use of standard memory management approaches for m-bounded sites. It eliminates the need for the programmer to either explicitly manage allocation and deallocation or to eliminate all references to objects that the program will no longer access. Unlike many previously proposed memory leak detection approaches [17, 13, 10, 14, 9, 18, 16, 12, 23], which simply identify leaks and rely on the programmer to modify the program to eliminate any detected memory leaks, it automatically eliminates the leak without the need for any programmer intervention. It is access-based — many previously proposed approaches analyze object reachability to reason indirectly about memory leaks [17, 13, 14, 9, 18, 16, 12, 23]; our technique, in contrast, reasons about the accesses that the program performs. It is therefore capable of recognizing and eliminating leaks even when the leaked object remains reachable and is therefore appropriate for both garbage collected languages and languages with explicit memory management. One particularly interesting aspect of our results is the indication that it is possible, in some circumstances, to overlay live objects without unacceptably altering the behavior of the program.

1.1 Usage Scenarios

We anticipate that our technique will be prove to be most useful for eliminating leaks in deployed programs, especially when the original developers are not easily available or responsive. Because it is automatic, the technique can successfully eliminate leaks without requiring anyone to understand and modify the program. It is also possible to apply the technique directly to stripped binaries, making it possible to eliminate leaks even when there is no realistic possibility of understanding the program or modifying its source. In this kind of scenario, it is hard to imagine *any* technique that requires programmer intervention successfully eliminating the leak.

During active development, programmers may prefer to use the extracted memory access information to find leaks that they then eliminate by modifying the program source. Or, if they convince themselves that the bounds m are accurate, they can simply use cyclic memory management at the corresponding allocation sites. Note that this last alternative can significantly reduce the programming burden — it eliminates the need for the programmer to explicitly deallocate objects allocated at m-bounded allocation sites (if the program uses explicit allocation and deallocation) or to track down and eliminate all references to objects that the program will not access in the future (if the program uses garbage collection).

1.2 Risk/Reward Analysis for Unsound Transformations

To take a broader perspective, the research suggests that the field may well benefit from exploring a new class of program transformation techniques that trade off soundness in return for other benefits (such as the elimination of memory leaks). In such cases, as with any engineering tradeoff, one must perform a risk/reward analysis to determine if the reward outweighs the risks. Our results indicate that the risks for cyclic memory allocation are apparently

[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).

quite small. Specifically, they indicate that the bounds estimation technique is quite accurate and that the consequences of overlaying live data are usually not too serious. In contrast, the rewards can be significant. Specifically, cyclic memory allocation can eliminate memory leaks that could otherwise limit the lifetime of the program and leave it vulnerable to denial of service attacks. Our expectation is that, over time, researchers will develop many other unsound program transformations for which the rewards (potentially far) outweigh the risks.

1.3 Contributions

This paper makes the following contributions:

- **m-Bounded Allocation Sites:** It identifies the concept of an m-bounded allocation site.

- **Cyclic Memory Allocation:** It proposes the use of cyclic memory allocation for m-bounded allocation sites as a mechanism for eliminating memory leaks at those sites.

- **Empirical Bounds Estimation:** It proposes 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.

- **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 cyclic memory allocation can 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.

- **New Tradeoff:** It introduces the concept of trading off soundness in return for other benefits, in this case trading the possibility of overlaying live data in return for the elimination of memory leaks. As researchers examine the risks and rewards of unsound program transformations more closely, we expect the field to produce many other unsound but beneficial transformations.

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 (transitively) invoked sequence of procedures.

```
 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:    }
37:  }
```

Figure 1: Memory leak from Squid

The procedure snmpDecodePacket is called by the procedure snmpHandleUdp, which passes a pointer to itself as an argument to CommSetSelect, which then stores 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 explict 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 1 object eliminates the memory leak and, to the best of our ability to determine, does not harm the correctness of the program. In particular, we have used this version of Squid in our standard computational environment as a proxy cache for the last 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 two components. The first component locates m-bounded allocation sites and obtains the bound m for each site. The second component replaces, at each m-bounded allocation site, the invocation of the standard allocation procedure (`malloc` in our current implementation) with an invocation to 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 Finding m-Bounded Allocation Sites

Our technique finds m-bounded allocation sites by running an instrumented version of the program on a sequence of training inputs of increasing size. 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, 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 also 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.

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.

When the technique 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. In this case, it generates a production version of the program that uses cyclic allocation for that allocation site with a buffer size of m objects.

3.2 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 may be a memory leak at that site. Note that our technique collecte enough information to recognize such sites.

It would be possible to use cyclic memory allocation for only such sites. Our current implementation, however, uses cyclic memory allocation for all sites with an observed bound m. We adopt this strategy in part because it increases the number of objects allocated in cyclic buffers (thereby simplifying the overall memory management of the program) and in part because gives us a more thorough evaluation of our technique (since it uses cyclic allocation for more sites).

3.3 Cyclic Memory Management

We have implemented our cyclic memory management algorithm for programs written in C that explicitly allocate and deallocate objects (in accordance with the 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.

3.4 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.

3.5 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 compting [19] 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 [19]. 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 can then continue on to correctly service subsequent requests [19].

We therefore apply failure-oblivious computing [19] as apropriate to ameliorate (and in many cases even eliminate) any global effect of any anomalies that overlaying live objects may introduce. Specifically, we simply discard any writes via null or out of bounds pointers and apply a technique to heuristically exit infinite loops. This last technique simply bounds the maximum number of iterations of each loop to be $10^3 i$, where i is the largest previously observed number of iterations of the loop (when the loop exits normally and not because of the imposition of the $10^3 i$ 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.

- **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.

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. 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 m-bounded allocation sites and to obtain the estimated bounds m for these sites as described in Section 3.1.

- **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 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 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 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 m-bounded sites, the percentage of memory allocated at these sites, 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 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 one of the 160 sites with observed bounds m in the entire set of programs.

Program	% m-bounded	% memory	% invalidated
Squid	62.5	43.3	0.0
Freeciv	50.0	75.2	0.0
Pine	60.0	15.0	1.5
Xinetd	64.7	94.8	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

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

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.

4.1.1 Training and Validation Runs

Our training runs detected 41 m-bounded allocation sites out of a total of 66 allocation sites that executed during the training runs; 43.3% of the memory allocated during the training runs was allocated at m-bounded sites. Table 2 presents a histogram of the observed bounds m for all of the 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).

m	1	2	3
# sites	38	2	1

Table 2: m distribution for Squid

4.1.2 Memory Leaks

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.

4.1.3 Conflict Runs

For Squid, the training runs find a total of three m-bounded allocation sites with m greater than one. We next discuss our results from the conflict runs when we artificially reduce the sizes of the observed bounds at these 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.

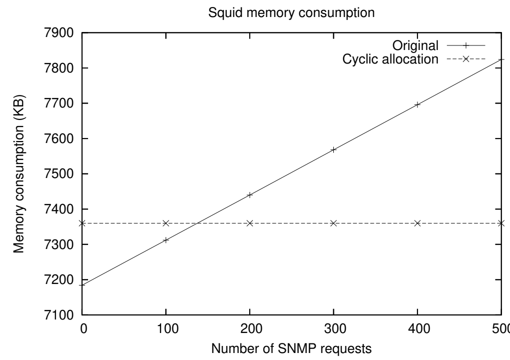

Figure 2: Squid memory consumption

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 completely unimpaired.

4.1.4 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 is designed to allow 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.

4.2.1 Training and Validation Runs

Our training runs detected 42 m-bounded allocation sites out of a total of 84 allocation sites that executed during the training runs; 75.2% of the memory allocated during the training runs was allocated at m-bounded sites. Table 3 presents a histogram of the observed bounds m for all of the 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 observable behavior of the program.

4.2.2 Memory Leaks

It turns out that Freeciv has a memory leak associated with an allocation site repeatedly invoked during the processing of each

m	1	2
# sites	39	3

Table 3: m distribution for Freeciv

AI player. Specifically, this allocation site allocates an array of boolean flags that store the presence or absence of threats from the oceans. The training runs determine that this allocation site is an m-bounded site with m=1. Cyclic memory allocation completely eliminates this memory leak.

4.2.3 Conflict Runs

Freeciv has three 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.

4.3.1 Training and Validation Runs

Our training runs detected 66 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 m-bounded sites. Table 4 presents a histogram of the observed bounds m for all of the m-bounded sites. As for the other programs, the vast majority of the observed bounds are small. The validation run determines 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.

m	1	3	8
# sites	64	1	1

Table 4: m distribution for Pine

4.3.2 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 either our training or our validation runs.

4.3.3 Memory Leaks

Neither the training nor validation runs revealed a memory leak in Pine.

4.3.4 Conflict Runs

Pine has two m-bounded allocation sites with m greater than 1. The first site is the invalidated site discussed above in Section 4.3.2. The training runs indicate that this site has an observed bound of bound 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 4.3.2, 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 formatted 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 3.5), 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.

4.3.5 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, ...). We observed no problems, errors, or anomalies.

4.4 Xinetd

Our training inputs for Xinetd consist of between 10 and 500 requests. Our validation input consists of 1000 requests. All of these requests are generated by a Perl script we developed for this purpose.

4.4.1 Training and Validation Runs

Our training runs detected 11 m-bounded allocation sites out of a total of 17 allocation sites that executed during the training runs; 94.8% of the memory allocated during the training runs was allocated at m-bounded sites. Table 5 presents a histogram of the observed bounds m for all of the 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	11

Table 5: m distribution for Xinetd

4.4.2 *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 m-bounded allocation sites in Xinetd have m greater than one, we do not investigate the effect of reducing the bounds.

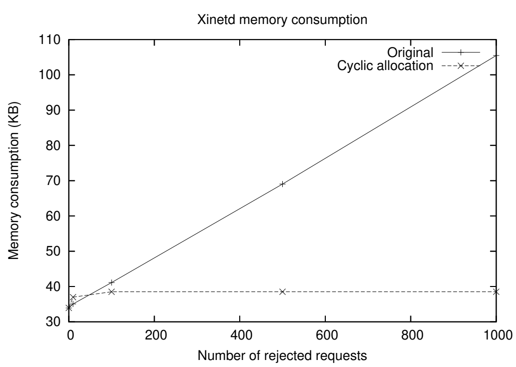

Figure 3: Xinetd memory consumption

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?

Our results indicate that the observed bounds are reasonably accurate — the validation inputs invalidate only one of the 160 m-bounded allocation sites. Moreover, our conflict runs indicate that overlaying live data has a surprisingly small effect on the execution of the program. Of the 8 sites considered in the conflict runs, none causes the program to fail when the bound is artificially reduced; for 6 of these 8 sites, the bounds reduction leaves the entire functionality of the program intact! Even without failure-oblivious computing, only 2 of the 8 sites cause the program to fail when the bound is artificially reduced, with 4 of the 8 sites leaving the entire functionality of the program completely intact.

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 [7, 8] 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.

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 [17, 13] 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. The drawback is the possibility that our transformation may cause the program to overlay live data. Our results indicate that 1) the chance of overlaying live data is apparently quite small because the observed bounds are apparently quite accurate, and 2) the consequences of overlaying live data do not appear to be that severe in practice.

Chilimbi and Hauswirth [10] 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 determine 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, we expect that it should be possible to drive the false positive rate down to a level where the rewards of eliminating the memory leak would outweigh the risks of premature deallocation.

5.2 Static Memory Leak Detection

Evans [14], Bush, Pincus, and Sielaff [9], Heine and Lam [18], Hackett and Rugina [16], Chou [12], and Xie and Aiken [23] 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. Note that all of these analyses are appropriate only 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 developer to provide additional annotations [14, 23]. 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.

We note that 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 that the programmer may inadvertently introduce new errors.

Another factor is the quality of the information that the leak detector provides. Both Hiene and Lam's analysis and Hackett and Rugina'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 [21]. 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 [20].

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 [15]. 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 collection. 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 [22, 6, 11]. 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 led us, in part, to develop the approach we present in this paper.

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 show 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.

Since its inception, the field of automated program transformation has focused almost exclusively on sound transformations that do not affect the semantics of the program. We believe that this focus has led the field to ignore many potentially useful unsound transformations (such as the memory leak elimination technique that this paper presents). As the field matures, we expect to see researchers increasingly develop and deploy viable transformations that happen to be unsound. The deployment decision will turn on whether an appropriate risk/reward analysis shows that the rewards that the transformation delivers outweigh the risks associated with the possibility that the transformation may introduce errors.

7. 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] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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