Dynamaid

Informative Dynamic Memory Bug Detection in C

Andrew Morrison

Advised by Dr. Robert Dondero

In partial fulfillment of the requirements
for the degree of Bachelor of Arts.

Department of Computer Science
Princeton University
May 2014

This paper represents my own work in accordance with University regulations.

Andrew Morrison
May 5, 2014

Contents

Abstract

Memory bugs in any programming language can be subtle, and frequently have unpleasant consequences for programmers. However, while many tools exist to detect the existence of memory bugs, there are no good options to determine where the bug occurred in the original source code. This makes debugging code far more difficult and time-consuming. This tool provides a way to detect such errors at runtime and reports detailed line number information about the error in the original source code to aid in the debugging process. This makes corrections of such errors substantially easier, especially for students.

Acknowledgements

"If you wish to make an apple pie from scratch,
you must first invent the universe."
— Carl Sagan

Dynamaid is, without a doubt, my proudest accomplishment at Princeton. Finally having a solid, working product after three semesters of work is tremendously validating, as is the thanks and positive feedback from students who used the beta version of the tool. My deepest thanks go to my advisor, Dr. Robert Dondero. Dr. Dondero consistently and patiently met with me throughout the development process, providing key insights and advice every step of the way. He is, without question, one of the best teachers I have ever had, and I am immensely proud of what we have accomplished together.

In any long development process, there are always twists and turns. I frequently fretted that I would not be able to get this tool to work to the extent that I would have liked. When that happened, I had the great fortune of having supportive friends who knew exactly how to talk me down and get me back on track. In particular, I want to thank Jennifer Jang, Shivam Verma, and Justin Wright for always being there when I needed them. Additionally, I owe a great debt of gratitude to my coach, Steve Macy, for making me a better hockey player, a better competitor, and a better person. His toughness and will to win have been a great inspiration to me over the last two years.

Finally, I would like to thank my parents and sisters, who always knew to give me a kick when I got complacent and to give me support and a laugh when I got discouraged. All of my success is because of them.

Introduction

"Computers are like Old Testament gods: lots of rules, and no mercy."

— Joseph Campbell

The C programming language is one of the few popular languages with explicit memory management. Most widely-used languages like Java or Python implicitly handle the bookkeeping of allocating and freeing blocks of dynamic memory, allowing the developer to focus on other tasks. While this automatic memory management is convenient, it comes at a cost. Computing what objects can be safely freed during execution adds overhead at runtime. Furthermore, the user's code is periodically and unpredictably interrupted by a thread that performs "garbage collection" - that is, freeing objects that are out of scope to the programmer.

C takes a different approach to handling dynamic memory. Programmers are responsible for allocating and freeing the memory that their program consumes. This model requires all decisions about when to free memory to be made at compiletime rather than runtime, and so eliminates the overhead of maintaining reference counts for objects. Additionally, programmers are given complete control over *when* their memory gets freed during execution, without being interrupted by a garbage collection thread. If the programmer manages memory correctly, the resultant code is necessarily more efficient than if the management was performed implicitly by the programming environment.

However, as the adage goes, *with great power comes great responsibility.* While C programmers have complete control over how their memory space is arranged and maintained, they must also watch out for memory bugs that cannot occur in managed languages. Should a C programmer mistakenly free an object twice or not at all, corruption of the memory space or heap exhaustion could occur. There is no mechanism in the language to inform the programmer that such a bug has occurred - consequently, these bugs are both common and notoriously difficult to debug.

This is especially true in computer science education. Novice programmers, new to the concepts of pointers and dynamic memory, frequently have great difficulty tracking down memory bugs in their code. It is not uncommon for a student in Princeton's COS 217 to spend hours debugging a single memory leak. Moreover, students often bring buggy code to course staff and teaching assistants for help, reducing the amount of time they might spend helping with other, easier-to-diagnose bugs. My time as a teaching assistant in COS 217 inspired my decision to write software to automate the task of diagnosing these bugs, making life as easy as possible for students and faculty alike.

To address this problem, I have created Dynamaid, a tool that detects common memory bugs and reports on them in a useful way. In particular, Dynamaid is capable of detecting memory leaks, double frees, dangling pointers, and memory overflow. In all cases, Dynamaid knows the names of the variables involved in each bug, and is frequently able to report the exact line number where the bug occurred allowing students to immediately fix their code.

Related Work

"You can get help from teachers, but you are going to have to learn a lot by yourself, sitting alone in a room"

— Dr. Seuss

Given how common memory bugs are in C code, a number of tools have been created to aid in their diagnosis. Unfortunately, most current tools do not tell the programmer exactly what they want to know. *Detection* of memory bugs is not enough - programmers, and especially students, want to know where they need to change their code to fix their bugs, not merely that a problem exists.

This is the primary flaw of a tool like MemInfo in COS 217. Itself a senior thesis, MemInfo is able to detect the existence of memory leaks and memory overflow. Students are provided with a report of errors after their program executes, like this one taken from a buggy student submission of the COS 217 shell assignment:

```
Errors:
   ** 16 un-freed bytes (2 blocks) allocated at dynarray.c:103
   ** 24 un-freed bytes (2 blocks) allocated at dynarray.c:91
Summary Statistics:
   Maximum bytes allocated at once: 220
   Total number of allocated bytes: 7911
Statistics by Line:
          Bytes   Location
            564   command.c:181
            ...
            284   ish.c:196
             40   TOTAL
Statistics by Compilation Unit:
              0   command.c
            902   dfa.c
             40   dynarray.c
           -902   ish.c
             40   TOTAL
```

This informs users of the number of memory leaks that occurred at execution, along with the line number of the call to malloc() that allocated the leaked object. Unfortunately, this is usually not enough to tell students where to look to change their code - frequently, objects are modified in any number of functions, so the leak could be in more or less any part of the codebase. It would be useful to tell students exactly where the object was lost to expedite the debugging process.

Even the powerful open-source Valgrind [1] has similar problems. Valgrind detects more bugs than MemInfo and operates on compiled binary files. This makes it an excellent general-purpose tool, but because it works on these compiled binaries it has no access to variable name information or line numbers and so cannot tell students where to look for bugs. In an academic setting, when dealing with novice programmers, we as tool designers must do better.

Functionality

"He made an instrument to know
If the moon shine at full or no."
— Samuel Butler

Dynamaid helps students correct four different classes of dynamic memory management errors in their code by reporting the operations that resulted in the bugs and the line number where they occurred. This frequently makes subtle bugs obvious and easy to fix. This section will describe and provide examples of the types of bugs we wish to detect, as well as providing sample executions of the tool and interpretations of its output.

Memory Leaks

Memory leaks occur when the last reference to an object is lost before that object is freed. Once this happens, the programmer has no means to deallocate the object, which will continue to consume heap space for the remainder of the program execution. Should a long-running process continually neglect to free memory, it will eventually face heap exhaustion - future calls to malloc() may fail simply because there is not enough free heap space to meet the request. Programs that unintentionally use too much heap space may also cause the operating system to perform paging operations more often, adversely affecting performance. Concretely, consider the following program that contains a memory leak:

```
#include <stdlib.h>

int main(void)
{
    char *string;
    string = (char *) malloc(10);
    ...
    return 0;
}
```

In this simple program, a ten byte block of memory is allocated and referred to only by the pointer "string." Upon returning from main(), this reference to the allocated block is lost without a corresponding call to free() having been issued. Dynamaid reports that the error occurs when the main() method returns because the variable "string" was lost at that point. In this case, a user would easily notice that a call to free() must happen before the return statement.

During the development process it became useful to distinguish between "pernicious" and "innocuous" memory leaks. A memory leak is innocuous if, as in the above example, the last reference to an object is lost at the end of program execution - these are not really problems, because the program will not be using the heap after it exits. All other leaks are pernicious, in that they happen in the middle of execution rather than at the end; such leaks could lead to heap exhaustion and should be fixed by the programmer. Originally, Dynamaid reported only these pernicious leaks. However, to stay consistent with COS 217's definition of "memory leak" my advisor requested that innocuous leaks be reported as well. Consequently, Dynamaid lists all objects that were still unfreed at the end of execution, providing a complete listing of all leaks that occurred.

Double Frees

A double free occurs when free() is called on a block of memory more than once. This bug corrupts the program's memory management data structures, either causing a crash or causing malloc() to return the same memory address twice at subsequent points during execution, causing some memory to be doubly allocated.

A simple example of a double free would be as follows:

```
#include <stdlib.h>

int main(void)
{
    char *string;
    string = (char *) malloc(10);
    free(string);
    ...
    free(string);
    ...
    return 0;
}
```

As before, a ten byte block of memory is allocated, but is subsequently freed twice, leading to unpredictable behavior after the second free. Dynamaid reports the line number information of both the original and the extra call of free().

Dangling Pointers

Once a block of memory has been freed, a programmer may no longer assume it contains any usable data, because the heap memory may have been repurposed since the block was freed. Pointers to freed memory are called *dangling pointers* and dereferencing one results in undefined behavior. An example of a dereference of a dangling pointer would be as follows:

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
    char *src = "Hello, World!";
    char *dest = (char *) malloc(strlen(src) + 1);

    strcpy(dest, src);
    ...
    free(dest);
    ...
    printf(dest);
    ...
    return 0;
}
```

After dest is freed, the call to printf() produces undefined behavior. The program may print the correct string, it may print garbage output, or it may cause a segmentation fault. Dynamaid modifies all pointers to the object after it has been freed such that they point to kernel space on a Linux system, guaranteeing a segmentation fault. This helps users track down these bugs by guaranteeing that they produce a runtime error.

In an attempt to provide information about where the dereference actually occurred, Dynamaid catches the SIGSEGV signal. When a segmentation fault occurs, Dynamaid receives the faulting address; that is, the address a program incorrectly dereferenced. Dynamaid checks the symbol table to see whether an object corresponds to that address. If it does, Dynamaid reports that a dangling pointer was dereferenced along with the location of the last successful operation Dynamaid performed. Normally, this is enough to point users to within a few lines of where the dangling pointer was dereferenced.

Overflow and Underflow

Once a user has received a block of memory from a call to malloc(), he or she may only use as many bytes as were requested. If a user writes before or after the boundary of the block, Dynamaid reports memory underflow or overflow respectively. Writing past the bounds of a block can corrupt any data that follows on the heap, leading to unpredictable behavior.

The following is a common example of memory overflow:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
    char *src = "Hello, World!";
    char *dest = (char *) malloc(strlen(src));

    strcpy(dest, src);
    printf(dest);
    free(dest);
    return 0;
}
```

This program forgets to allocate enough memory for the trailing null byte copied by strcpy(). Consequently, the copy operation overwrites the byte immediately following the allocated block, possibly corrupting the heap and causing unpredictable bugs later in execution. Dynamaid checks for overflow and underflow when the block is freed, and reports the line number where the call to free() occurred, informing the user which object had its boundary overwritten.

Usage

Dynamaid is invoked by the user at build time to produce an instrumented binary file that performs error checking for the above bugs. Like the MemInfo tool currently in use in COS 217, Dynamaid is invoked similarly to the gcc compiler. In nearly all cases, a student can replace "gcc217" with "gcc217dm" in their commands to get their instrumented binary file. For example, if a user would normally invoke the command:

```
gcc217 mycode.c -o mybinary
```

to compile their code, with Dynamaid they would only need to invoke the command:

```
gcc217dm mycode.c -o mybinary
```

to get their instrumented code. The only difference is that Dynamaid accepts an extra flag that gcc217 does not: a user may pass the “-kill” flag to instruct Dynamaid to terminate a program as soon as the first memory management bug is found. This allows a user to avoid being overwhelmed by error messages in a program that has many bugs. In this case, the user would write:

```
gcc217dm -kill mycode.c -o mybinary
```

Though Dynamaid is invoked at build time, error detection happens at runtime. Consequently, if a program’s execution does not follow the logical path where a memory bug exists, that bug will not be reported (since, from Dynamaid’s perspective, it never happened!) Therefore, Dynamaid is a supplement to, and not a replacement for, thorough testing.

Output

Dynamaid provides users with a great deal of information when a bug has been found. The messages are terse, in the interest of not cluttering their terminal output unnecessarily. When a bug is detected, the output is as described below.

Memory Leak

The output when a memory leak is detected is as follows:

```
-------------------------------------------------------------
Dynamaid: <Error Message>
Assigning from <RHS> to <LHS> *
File: <File Name>
Line: <Line Number>
Variable: <Variable Name>
Depth: <Depth>
Object size: <Size>
This object has been pointed to or returned by:
-        <Names>
------------------------------------------------------------
```

* denotes optional field.

- Error Message
  - “Memory leak detected during free”
    When an object is freed, any pointers it contained are lost, decreasing the reference counts of all objects referenced by those pointers. If a free causes

a reference count to become zero for an object that has not been freed, this is the message displayed.

  - "Memory leak detected within object during assignment"
    When the last reference to an object is lost via an assignment, any pointers that object contained are lost, decreasing the reference counts of all objects referenced by those pointers. If this causes a reference count to become zero for an object that has not been freed, this is the message displayed.
  - "Memory leak detected within object leaving scope"
    When the last reference to an object is lost by leaving scope, any pointers that object contained are lost, decreasing the reference counts of all objects referenced by those pointers. If leaving scope causes a reference count to become zero for an object that has not been freed, this is the message displayed.
  - "Memory leak detected during assignment."
    During an assignment, one reference to the left hand side of the expression is lost. If an assignment causes a reference count to be zero for an object that has not been freed, this is the message displayed.
  - "Memory leak detected: last reference out of scope."
    When a local variable pointing to an allocated object exits block scope, one reference to that object is lost. If the end of scope causes a reference count to become zero for an object that has not been freed, this is the message displayed.

- RHS, LHS
  If any of the above situations arise as the result of an assignment operation, Dynamaid informs a user that a bug occurred while assigning from "RHS" to "LHS", where RHS and LHS are the right-hand side and left-hand side of the assignment expression, respectively. If the leak was not the result of an assignment, this line does not appear.

- File Name
  The file in which the last reference to the object was lost.

- Line Number
  The line number at which the last reference to the object was lost.

- Variable Name
  The variable name involved in the operation that triggered the detection of the leak.

- Depth
  If an object being freed or lost causes a leak, the leaked object exists somewhere

in the object graph created by the user. Depth is how many pointer traversals Dynamaid had to make from the original object to detect the leak.

- Size
  The size in bytes of the leaked object.

- Names
  The set of all variables that have pointed to this object at any point during execution.

Double Free

The output when a double free is detected is as follows:

```
-------------------------------------------------------------
Dynamaid: Double Free detected.
File: <Second File Name>
Line: <Second Line Number>
Variable: <Second Variable>
Object first freed in file "<First File Name>" at line
<First Line Number>, variable "<First Variable Name>"
------------------------------------------------------------
```

- File Name
  First File Name is the file where the first call to free() occurred. Second File Name is the file where the second (double) call to free() occurred.

- Line Number
  First Line Number is the line at which the first call to free() occurred. Second Line Number is the line where the second (double) call to free() occurred.

- Variable Name
  First Variable Name is the variable name used in the first call to free(). Second Variable Name is the variable name used in the second (double) call to free().

Dangling Pointer Dereference

The output when a dangling pointer is dereferenced is as follows:

```
------------------------------------------------------------
Dynamaid: Dangling pointer dereferenced (Segfault occurred).
Object size: <Size>
This object has been pointed to or returned by:
-        <Names>

The last successful assignment operation before the
segfault was from <RHS> to <LHS> in file '<File Name>'
at line <Line Number>
Terminating execution...
------------------------------------------------------------
```

- Size
  The size in bytes of the dereferenced object.
- Names
  The set of all variables that have pointed to this object at any point during execution.
- RHS, LHS
  The right and left-hand sides of the last successful assignment Dynamaid performed.
- File Name
  The file where the last successful assignment Dynamaid performed happened.
- Line Number
  The line at which the last successful assignment Dynamaid performed happened.

Memory Overflow/Underflow

The output when memory overflow or underflow is detected is as follows:

```
------------------------------------------------------------
Dynamaid: Memory overflow/underflow detected.
File: <File Name>
Line: <Line Number>
Variable: <Variable Name>
------------------------------------------------------------
```

- File Name
  File Name is the file where the object whose boundaries had been overwritten got freed.
- Line Number
  Line Number is the line at which the object whose boundaries had been overwritten got freed.
- Variable Name
  Variable Name is the variable used in the call to free() that exposed overflow.

Design

"It is a mistake to think you can solve any major problems just with potatoes."
— Douglas Adams

This senior thesis is the continuation of a junior independent work project from Spring 2013. In this sense it is the product of three semesters of work. However, the final product bears little resemblance to the previous independent work submission. Dynamaid works by instrumenting user code to maintain reference counts of objects and execute our error-detection logic at key points at runtime. This section discusses the evolution of the tool at a general level and describes the exact algorithms used for bug detection.

Evolution

In Spring 2013, the Dynamaid tool was only intended to detect memory leaks and double frees, with dangling pointer handling as a stretch goal. The main difficulty involved in memory leak detection is instrumenting arbitrary C code with function calls to maintain reference count information at runtime. A tool that was sensitive to the syntax of C would be required to handle truly arbitrary code. However, such a project is well beyond the scope of a one-semester project. To ease this burden, Dynamaid users would have been required to include annotations in their code and refrain from using certain (otherwise valid) C coding conventions.

This approach worked reasonably well considering the timeframe allowed to implement it. However, if the user's annotations were incorrect or outdated, Dynamaid would not work - ironically, these "pseudobugs" were more difficult to identify than the memory leaks themselves. Naturally, any solution acceptable for use in COS 217 would need to be significantly more robust and require little to no user intervention.

To truly achieve the goal of working on arbitrary C code, a full-blown C parser capable of understanding and manipulating any valid C syntax was required. With two semesters available to complete this senior thesis, my advisor and I decided that a difficult but attainable goal would be to construct such a parser. With this goal in mind, I spent the first month of the fall semester learning to use the Flex and Bison

UNIX utilities. Flex and Bison [2] are designed to generate parsers from user-input grammar files. Because they are so powerful, they are used in a variety of compilers. However, this should give some sense of the difficulty of the project we were about to undertake - if successful, we would have implemented a large part of a C compiler. Due to the project's difficulty, my advisor suggested that I might talk to Professor David Walker, a member of Princeton's COS faculty with a specialty in programming languages, before beginning to write actual Flex and Bison code. After considering our project, Professor Walker suggested that we look into the open-source CETUS project from the University of Purdue.

CETUS [3] is a C source-to-source conversion tool originally designed to safely parallelize sequentially-written C code. The main page for the tool explains that "CETUS" is not an acronym - rather, it says, *"The constellation Cetus is a sea monster. A C monster."* Written in Java, CETUS' underlying code provides an object model for the C programming language with objects that represent assignment expressions, function calls, and other C constructs. CETUS represents a C program as a tree of these objects - building on CETUS involves traversing and modifying this tree. This process is significantly more robust and maintainable than modifying C grammar files.

Of course, CETUS has limitations. One issue is that it supports only the C90 standard, so input files that mix declarations and code cause the tool to crash. This is not an issue in the context of COS 217, which requires student code not only to adhere to the C90 standard but also to compile without even the most pedantic warnings from the compiler. The expectation, in other words, is that student code will be relatively simple and clean. Another issue from a maintenance perspective is that the tool's documentation is generally poor, with a limited tutorial and a sparsely-commented Javadoc. As a result, most of my understanding of this tool comes from actual visual inspection of the source code and experimentation with simple input programs. Nevertheless, once understood, CETUS performs reliably and therefore has become the backbone of this thesis project.

CETUS allows Dynamaid to instrument student code to maintain the reference count information and perform bug checks at runtime. The following section describes in detail how the Dynamaid code accomplishes this.

Implementation

The Dynamaid tool is comprised of three main parts:

1. C data structures and functions that perform runtime checks and report bugs.

2. Java code that instruments student C code to use these structures and functions.

3. A Python script that performs preprocessing, calls our Java code, and invokes the gcc compiler to produce the executable binary.

Preprocessing

One of the most inconvenient parts of using CETUS for this project was that line number information is stripped away when the program tree is created. In order to report line numbers for each bug, a mechanism was required to force CETUS to include line number information in the program tree, without changing anything about how the program itself executes.

The natural solution to this problem is to use pragmas. Dynamaid first runs a pass over all the student code and removes the comments while preserving line numbers. Then, it generates a new file where after every line from the original student code comes a line of the form

```
#pragma MemMgr:LINENUM:
```

where LINENUM is the line number in the original code of the line that follows. A CETUS pass will retain this pragma information, so line number information is available (albeit in a roundabout way) to our instrumentation code.

Instrumentation

Once line number information has been embedded in the student source code, Dynamaid can begin instrumentation. Once CETUS has been invoked, Dynamaid receives the program tree, the nodes of which represent all code from all files that the student gave at the command line. Any code outside of function calls is handled at compiletime, as per the C90 standard. Therefore, since all of our instrumentation involves operations that happen at runtime, Dynamaid only considers code that resides within functions. Global variables may be safely ignored, except as they appear within functions.

For every function in the student submission, the first step is to generate enterScope() and leaveScope() calls for each formal parameter at the beginning and the end of the function. This allows us to keep track of references that are created as the result of a function call - if the pointers correspond to an object that has been allocated, it is important that Dynamaid be able to increment its reference count. An enterScope() call is generated for all formal parameters, even if they are not pointer variables - our C backend must, therefore, be smart enough to ignore spurious calls to this function. We briefly considered an approach that would only generate enterScope() calls for pointer types, but this complicated the handling of typedef'd pointer types unnecessarily.

As an implementation wrinkle, not all types of parameters can be passed to enterScope(), which takes a void* as argument. Some variable types cannot be cast to void* - in particular, Dynamaid does not generate function calls for enumerable variables. This is perfectly fine, since an enum will never reasonably be a pointer in the context of COS 217.

In order to handle variables leaving scope within a function, Dynamaid then walks the body of the function and collect information on each compound statement. A mapping is created between compound statements and the variables declared within them. This mapping allows us to place leaveScope() calls at the end of the correct compound statement, effectively figuring out scope information for the object (which CETUS does not provide for us out of the box.) Additionally, Dynamaid can use this information to tell which variables are in scope at any point in the code. This is vital when it comes to handling return statements, which force all local variables out of scope. To handle return statements, Dynamaid walks the tree looking for returns, identifies all variables in scope at that point, and adds leaveScope() calls right before the return statement.

All that remains is to replace assignments with function calls. As with enterScope() calls, assign() calls cannot be generated for variables that cannot be cast to type void*. Dynamaid therefore ignores assignments involving variables of enumerable type since they will never reasonably be pointers. Dynamaid again walks the tree finding assignment expressions. CETUS gives us both sides of the expression for free, making it a simple matter to construct the call to assign. For a standard assignment operation of the form

```
ptr1 = ptr2;
```

Dynamaid generates a call to the assign function that looks like:

```
assign(&ptr1,ptr2,sizeof(ptr2),"ptr1","ptr2","filename.c",linenum);
```

The first argument is the address of a pointer, since the assign() function must change the contents of ptr1. Dynamaid must also pass in the size of the object because whatever is being assigned is automatically promoted to a 4-byte void*, even if it was originally only a 1-byte char. Without the sizeof() mechanism the assign() function would always copy 4 bytes to the left-hand side of an assignment, resulting in (ironically enough) memory overflow.

Rather than handle each one of the C assignment operators with its own function (e.g., having an assignSubtract() function corresponding to the “-=” operator), Dynamaid simply modifies the second argument of the assign function instead. For example, if Dynamaid encounters an assignment of the form

```
a -= b;
```

then it would make the second argument to the assign function

```
a - b
```

This pattern continues for all ten of C’s assignment operators.

To find the filename for any tree element, Dynamaid needs to walk up the tree to the root. The children of the root node in the program tree are of type TranslationUnit, and directly correspond to the files CETUS was invoked with. Every other element in the tree has a TranslationUnit as an ancestor - finding this ancestor allows Dynamaid to get the filename that it contains.

To find the line number corresponding to an object, Dynamaid traverses the tree keeping track of the last pragma that it found. After the preprocessing step, it can be sure that there is a pragma for every line in the code - the most recent one found during the instrumentation pass corresponds to the line number of the current tree element.

The last step is to override the standard library dynamic memory management functions. Dynamaid walks the tree one last time looking for function calls to malloc(), calloc(), realloc(), or free(), and replace them with calls to Dynamaid's C functions named MemMgr_malloc(), MemMgr_calloc(), MemMgr_realloc(), and MemMgr_free(). Usefully, CETUS allows us to do this in a much more robust way than the MemInfo tool does. MemInfo overrides the library functions with a set of fragile preprocessor directives - CETUS allows us to not only make direct, reliable changes to the function name but also to its arguments. For instance, since many bugs can be detected on a call to free(), error location information is included in the call. In the solution, calls like

```
free(pointer);
```

become

```
MemMgr_free(pointer, "pointer", linenum, "filename.c");
```

Making this change without access to a C parser would have required at the very least a complicated regular expression, and would have been a debugging nightmare.

Runtime Error Detection

Once the instrumentation step is complete, Dynamaid has successfully wired the student's code together with our runtime error-detection logic. What remains is to use the information from the instrumentation to report runtime issues.

The most important runtime component of Dynamaid is the global symbol table that stores all information about allocated objects. The symbol table maps from pointer addresses to info_t structs that contain important information about the object. In particular, the info_t struct looks like:

```
typedef struct {
    size_t refCount;
    int isAllocated;
    size_t size;
    name_t *name;
    addr_t *addr;
    char *freedFile;
    char *freedName;
    int freedLine;
} info_t;
```

- refCount
  The number of references currently pointing to this object.
- isAllocated
  False if the object has been freed, true otherwise.
- size
  The size in bytes of the object.
- name
  A linked list of all variable names that have pointed to this object.
- addr
  A linked list of all pointer addresses that have pointed to this object.
- freedFile, freedName, freedLine
  If the object has been freed, these are the filename, variable name, and line number of the original call to free.

With this information, Dynamaid can to detect and report each of the four classic memory bugs. The algorithms for each are detailed below.

Memory Leaks The problem of detecting and reporting memory leaks is simply described. If any operation would cause an allocated object's reference count to reach zero, then Dynamaid has found a leak and should report that operation. This implies that Dynamaid must successfully perform two tasks:

1. Accurately maintain reference counts, and
2. Accurately determine how many references are lost during an operation

The practical upshot of this is that users are informed immediately after a leak has occurred and receive the exact location of the error. In fact, this method would also allow us to implement garbage collection - instead of reporting a leak, Dynamaid would simply free the object.

There are four main ways that reference counts can change. The first three are simple and their effects trivial to predict.

1. A formal parameter is passed into a function.
   If a pointer is passed into a function as a formal parameter, it becomes, for all intents and purposes, a local variable to that function. If the pointer currently points to an object, that object's reference count is increased by one. This is handled by calling enterScope() for all formal parameters.

2. A local variable leaves scope.
   This can happen either when a variable reaches the end of its block scope, or as the result of a return statement. Whenever a pointer to a valid object leaves scope, that object's reference count is decremented by one. If this operation results in a reference count of zero for an allocated object, Dynamaid reports a leak. This is handled by the leaveScope() function. There is one exception to the above. If a reference is returned from a function, Dynamaid does not report a leak for that object even if there are temporarily no variables that point to an allocated object. This is to prevent spurious warnings in function's whose purpose it is to allocate, initialize and return an object. These functions are common in COS 217. Consider the following function that creates a (useless) linked list:

```
#include <stdlib.h>
struct node {
struct node *next;
};

static struct node *getList(){
struct node *first;
struct node *second;
struct node *third;
struct node *fourth;

first  = (struct node *) malloc(sizeof(struct node));
second = (struct node *) malloc(sizeof(struct node));
third  = (struct node *) malloc(sizeof(struct node));
fourth = (struct node *) malloc(sizeof(struct node));

first->next = second;
second->next = third;
third->next = fourth;
fourth->next = NULL;

return first;
}
```

When this function returns, there is temporarily no pointer to the first node in the list - however, the list is not lost because a reference is returned to the caller. Bugs where students forget to catch a returned object from a function are rare, so Dynamaid ignores this possibility in order to reduce spurious warning messages. Of course, this means that if a student really does forget to catch an object being returned from a function, Dynamaid will not report it.

3. An assignment is evaluated.
   If there is an assignment of the form

   ```
   ptr1 = ptr2;
   ```

   then whatever ptr2 points to gains a reference, and whatever ptr1 was pointing to loses a reference. In the case where ptr1 and ptr2 are the same, no reference counts change.

However, the fourth way a reference count can change is significantly more complicated. Suppose, as in our linked list example, that object A contains the last reference to object B. Further suppose that a user frees object A. Then object B has been lost, and there is a memory leak. Therefore, when freeing an object Dynamaid must take into account the pointers that object contains. To do this, when an object is freed or leaked, Dynamaid walks over the memory it contains, looking at every four consecutive bytes and checking if they (taken as a pointer) correspond to another object. If they do, it decrements the reference count for that object. If that object is itself leaked as a result of this changing reference count, this process repeats recursively, walking the user's object graph via depth-first search until there are no more leaked objects.

Unfortunately, the problem is even more complicated. Consider a circular linked list, as opposed to the NULL-terminated linked list in the last code example. A client of this linked list would hold a single pointer to the first node. If this pointer were ever lost, the entire list would be lost to the user - all four nodes would be lost at once. However, the reference count for the first node *would not be zero*; after all, the fourth node in the list still contains a reference to the first one! In other words, the client's loss of their reference to the first node counts double. Not only is the first reference lost, but also any reference to the first node from later in the list is too. This problem generalizes to any user object graph that contains a cycle.

Early versions of the Dynamaid tool did not handle cycles in the object graph. The solution was to handle the loss of any reference in two passes. The first pass was a prediction step. This would run depth-first-search on the user's object graph starting at the object that was losing a reference. At each node, the number of references to the original object would be counted. The total number of references to the original object after all nodes had been visited is the number of references to the original object that will be lost. If this number (plus one for the original reference loss that caused us to look at the object in the first place - e.g. the user losing their handle to the first node in the list) is equal to the current reference count, Dynamaid concludes that the first object has indeed leaked. If it has been leaked, a second round of depth-first-search begins from the root node exactly as described above.

Dangling Pointers Dangling pointers were originally a stretch goal for this thesis, and for good reason: it is difficult to come up with a good way to handle them. Simply *having* a dangling pointer in your code is not a bug. These are only bugs if they are dereferenced. Even with CETUS, instrumenting every single memory access in user code (which is the only way to truly detect the bug immediately) was too difficult and unreliable to implement.

One of the reasons that dangling pointers are so difficult to detect is that they produce unpredictable behavior. Frequently, even after an object is freed, its contents remain in the same place on the heap. Therefore, subsequent memory accesses could return the correct bytes, and the program would continue unscathed. Our strategy, therefore, was to force programs that dereferenced dangling pointers to cause a segmentation fault and crash the program.

The first step to achieving this was maintaining a list of pointer variables that pointed to each allocated object. The addr linked list in the info_t struct for an object contains the addresses of each pointer that had ever pointed to this object - this list was updated every time the assign() function was called. Then, when an object was freed, Dynamaid would set all of these pointers to a strategic value that pointed to kernel-space on the nobel cluster, guaranteeing a segmentation fault on a subsequent dereference.

There were a few difficulties implementing this approach. The first was that it was difficult to know when to remove a pointer from the list of addresses. Suppose one of the pointers was a local variable to a function that had just returned. When the function returns, the stack shrinks and that pointer address is no longer in writable memory. In a sense, we had our own (intentional) dangling pointers to addresses on the stack! Removing these pointer addresses required knowing what the active stack frame was before and after a function returned - practically an impossible task from Dynamaid's point of view. Instead, Dynamaid simply never removes an address from the list. To prevent Dynamaid from crashing when accessing an out-of-bounds address, we needed some way to predict whether a memory access would cause a segmentation fault.

Remarkably, this can be checked easily with the following code [4]:

```
int fd[2];
if (currAddr->oldaddr == NULL)  continue;
if (pipe(fd) >= 0) {
    if (write(fd[1], currAddr->oldaddr, 128) > 0)
        if (*(currAddr->oldaddr) == ptr)
            *(currAddr->oldaddr) = modPtr(ptr);
    close(fd[0]);
    close(fd[1]);
}
```

Here, Dynamaid effectively uses the return value of the write() system call to ask the operating system whether the address at oldaddr is in bounds. If it is, then the pointer is set to our strategic value.

Originally, Dynamaid reasonably set these pointers to NULL after a free. Unfortunately, this proved too destructive. In particular, it completely prevented Dynamaid from detecting double frees. After the first free() completed, all references to the object became NULL, including any references that were subsequently freed.

In order to handle double frees and dangling pointers simultaneously, a more clever solution was required. We needed a transformation of dangling pointers that guaranteed a segmentation fault, but was also reversible so that MemMgr_free() could still determine whether the pointer corresponded to something that had been freed before.

The solution relies on the fact that on the nobel cluster, malloc() always returns a memory address that is a multiple of four. Consequently, the low-order two bits are always zero - these are free bits that Dynamaid can use in this transformation. Additionally, any pointer whose high-order two bits are ones is a guaranteed segmentation fault on the nobel cluster. The transformation applied therefore takes the highest order two bits and stores them in the two free bits at the end of the pointer. The high two bits are set to ones. This is a pointer that guarantees a segmentation fault, but where the original pointer is recoverable - simply replace the high two bits with the original bits, and set the low two bits to zeros.

This was successful in forcing segmentation faults for dangling pointer dereferences - unfortunately, it was not particularly useful in describing where the bad dereference occurred. To offer some debugging help in this situation, Dynamaid remembers the information of the last operation it performed. A handler was installed to catch the SIGSEGV signal. This handler outputs the information about the last operation, along with information about the object responsible for the fault. This is determined by looking up the faulting address in the symbol table. This information is surprisingly useful in debugging dangling pointers, and is enough to point students to the area where the fault occurred.

Double Frees Double frees are significantly easier to detect than the above issues. After reversing the transformation applied to detect dangling pointers, a simple lookup in the global symbol table suffices to see if the object has been freed before. If it has, report a double free along with information about where the first free occurred.

Memory Overflow As was the case with dangling pointers, memory overflow is difficult to detect immediately after it occurs. Once again, this is because in order to detect overflow Dynamaid would have to intercept every memory access and check whether the user is writing to a part of the heap that it owns. This is too much for Dynamaid to handle.

However, Dynamaid can easily detect overflow and underflow after the fact. When a user requests memory via a call to malloc() or calloc(), eight bytes are allocated in addition to the amount the user actually asked for. This allows for two four-byte sentinels, one before and one after the block of memory being returned to the user. The reason for using four-byte sentinels instead of one-byte sentinels is to ensure that the address returned to the client is still a multiple of four so that dangling pointer detection and double free detection still works. When a user frees an object, Dynamaid checks to ensure these sentinels are intact. If they are not, then overflow or underflow has been detected, and Dynamaid reports this along with the corresponding call to free, variable name, and object information.

Testing Methodology

"We have to continually be jumping off cliffs and developing our wings on the way down."
— Kurt Vonnegut

When students in COS 217 submit their assignments, they are generally told that their work will be graded "...on quality from the user's and programmer's points of view." In particular, "From the user's point of view, [a] program has quality if it behaves as it should." This section, discusses the testing process to determine whether Dynamaid indeed behaves as a user would expect.

Of course, Dynamaid is not a COS 217 assignment. There is no reference solution to match, nor is there any test suite provided to be reasonably sure the code is correct. Instead, testing was done on a more ad-hoc basis. As development continued and new features were added to the tool, the tests we put it through became more sophisticated. Because Dynamaid aims to accept arbitrary C code as input, a three-tiered testing approach suggested itself, with each tier comprising more difficult tasks than the one before it. The three tiers were as follows.

- Tier 1: Unit Testing
  The tests in this tier were short, often nonsensical programs that usually contained no more than one or two functions. Most of the time, these were composed to test a single feature in the context of a certain data structure or code layout.

  As an example of a unit test, consider the following code designed to test memory overflow detection.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(void)
{
    char *to_copy = "Hello, World!\n";
    char *not_enough;

    /* Forgotten about null byte */
    not_enough = (char *) malloc(strlen(to_copy));

    strcpy(not_enough, to_copy);

    printf(not_enough);
    /* Should report overflow */
    free(not_enough);

    return 0;
}
```

This test is sufficient to determine whether, in the most basic of circumstances, Dynamaid can detect memory overflow.

This type of testing has two important advantages. First, tests like the above can be composed very quickly and executed very cheaply. This lends itself nicely to regression testing. After any significant change to the codebase, these small tests can be run to provide a sanity check that previous functionality was not negatively impacted. Second, if these tests fail, then they are generally the simplest programs that cause the incorrect output - this is very valuable when debugging, since there are relatively few parts of the input that could have caused the error. In fact, a number of these test cases are the result of fault-driven testing; when Dynamaid bugs were found at higher tiers, a tier 1 test was created for that bug. Constructing the minimal example that causes the error was my first step when bugs were found on more complicated inputs.

- Tier 2: Fault Injection
  Unit tests by themselves are unlikely to expose any but the most superficial bugs in Dynamaid. To increase confidence that the tool was behaving as it should, more complicated test cases were required. Ideally, whatever tests we created would be similar to COS 217 assignments that Dynamaid would eventually be exposed to. At this stage, in order to decrease the number of coding problems that could reasonably affect Dynamaid, faults were only injected into COS 217

assignments that were known to be bug-free. In particular, simple faults were injected into the reference solutions to the courses Symbol Table and Shell assignments. The code for these tests cannot be included in this document in order to preserve the integrity of the assignments for future classes, but they generally represented the most common instances of each of the four memory bugs that Dynamaid detects.

These tests were considerably more stressful on Dynamaid. The input files were larger, and there were many more dynamic memory operations than in the tier-1 tests. The advantage to this type of testing is that it is far more likely to expose errors in Dynamaid - the disadvantage is that their cause would be much more difficult to track down. Consequently, whenever this fault injection testing caused Dynamaid problems, a simple test case was added to the first tier to handle it.

- Tier 3: Real Student Submissions
  This is by far the most stressful type of test for Dynamaid. Low-scoring submissions from the Fall 2013 semester of COS 217 were selected and run through Dynamaid. Usually, these did not reveal bugs in error detection. When Dynamaid failed on these files it was usually the result of a crash during the instrumentation step. Students frequently use strange coding constructs that violated assumptions Dynamaid made about coding style. Most of the time when these were found, the strange coding practices were accommodated - those that were not are listed in the known limitations section below.

Known Limitations

- Combined declaration statements and initializations are not supported.
  This is a bug in Dynamaid, but not a terribly intrusive one. In particular, statements like

  ```
  char *string = NULL;
  ```

  cause the tool to fail if “string” will ever point to memory that will be managed dynamically. Instead, users should write

  ```
  char *string;
  string = NULL;
  ```

  to accomplish the same task.

- Only one assignment operator may be used per statement.
  In particular, statements of the form

  ```
  a = b = c;
  ```

  cause Dynamaid to crash during the instrumentation phase. Instead, users should write

  ```
  b = c;
  a = b;
  ```

  This is also a more explicit style that may help students track down other bugs in their code, anyway.

Evaluation

"Three things are to be looked to in a building: that it stand on the right spot; that it be securely founded; that it be successfully executed."
— Johann Wolfgang von Goethe

It is one thing to know whether the Dynamaid tool works as specified. The testing methodology from the previous section convinces us, to the extent possible, that Dynamaid functions correctly. It reports each of the four main bugs reliably in all test cases - in this sense, the tool was "successfully executed."

It is another thing altogether to know whether it "stands on the right spot" - that is, whether the tool is indeed useful when it comes to debugging code. This is more difficult to assess. Simply asking students whether they used the tool and found it helpful is unlikely to produce reliable results; most students have little experience with code that requires explicit memory management, and so have nothing to compare Dynamaid to. Without context, student feedback would not be objectively helpful.

Originally, the plan to objectively evaluate Dynamaid had been to run experiments on the grading process of the shell assignment for the Spring 2014 semester. Each grader could have been assigned to a group that used Dynamaid to track down bugs in student submissions, or to a control group that could only use the current course tools. Any time difference between the two groups could be attributed to the use of the tool. This only assumes that on average, the graders in each group would be equally skilled, and that student submissions by precept are likely to be equally buggy. Unfortunately, this report was due to be completed before the assignment's due date.

Consequently, we contented ourselves with a much simpler process. At its core, our evaluation was still a time-trial. My version of the shell assignment from the Fall 2011 semester was injected with a number of dynamic memory management bugs. This version was given to my advisor, who was instructed to time how long it took him to debug the program using anything *but* Dynamaid. Of course, Dynamaid detected all of the bugs immediately, making fixing them relatively easy.

The code for the assignment was relatively clean. It passed all style checks from splint and Dynamaid's cousin, critTer, and received a nearly perfect score at submission-time in the Fall 2011 semester. Additionally, most of the five inserted bugs were inspired by common mistakes students often make in their assignments - bugs which my advisor was very familiar with. Nevertheless, from start to finish the debugging process took him 27 minutes. If one of the most experienced debuggers in the COS 217 course took that long to discover all the bugs in a relatively clean version of the shell assignment, it is a safe bet that it would take students significantly longer.

In reality, of course, most students likely would not solve all the bugs at all, instead bringing their code to their preceptor or course lab TAs looking for help. According to the Princeton registrar, 176 students were enrolled in COS 217 in the Fall 2013 semester. If even a small subset of these students saved more than 20 minutes of debugging (likely much, much more), this would considerably ease the strain on course staff at busy times of year.

Out of sheer curiosity, after the debugging trial ended my advisor compiled his fixed version of the shell assignment with Dynamaid and ran it to see if any additional bugs remained. This was unlikely - after all, the code had just been subjected to a battery of official grading scripts and been reported clean. Remarkably Dynamaid correctly reported a dangling pointer in the code! This was not a fault injection - it was a bona fide bug that had been in my original submission when I had taken the course. Effectively, this bug had eluded me, my original grader, and my thesis advisor, surviving two full grading sessions. Dynamaid detected it immediately, and provided enough information to quickly correct the issue. We conclude that Dynamaid is indeed useful to the COS 217 class.

Future Work

"I am tomorrow, or some future day, what I establish today. I am today what I established yesterday or some previous day."
— James Joyce

The Dynamaid development process went remarkably well. Originally, the tool was only intended to detect memory leaks and double frees, and act as a supplement to the MemInfo tool which also detected memory overflow. This goal alone was daunting. However, not only did we accomplish both of these tasks, but we also implemented our stretch goals of detecting dangling pointer dereferences and memory overflow as well. Dynamaid now implements all the features of MemInfo, and reports bugs in a much more informative and useful way. It now stands to replace MemInfo in coming offerings of COS 217.

In this sense, Dynamaid is a complete product. There were no more stretch goals left to implement at the time of this submission. However, there are still a few ways in which future developers may wish to improve the final product.

The first is to tackle the known issues described in the testing section. The issue regarding expressions that evaluate multiple assignments is very low-priority, but it would be useful to address the fact that variables cannot be declared and initialized in the same statement. Such a construct is common in student code when they would like to set a newly declared pointer variable to NULL. Incidentally, this problem may have been fixed already. There was simply not enough time to test whether the issue has been resolved.

Such bug fixes are unlikely to be enough to motivate a subsequent independent work project on their own. However, certain Dynamaid output could be made significantly more informative if a future developer would take the time to instrument every memory access within student code. At present, Dynamaid does not report precisely where dangling pointer dereferences or memory overflow occur, opting instead to report an area of the code likely to be responsible. If all memory accesses could be intercepted, it would be possible to report exactly where these bugs occur in the same manner as memory leaks and double frees. This would require modifying the Java code that builds on CETUS to instrument all ways memory can be modified in C - simply comprising this list alone is likely to be tricky. Nevertheless, I believe it should be possible to implement as an independent work project.

An interesting extension of the project would be to detect bugs beyond memory management issues. Perhaps, for instance, it would be possible to detect whether a program is vulnerable to buffer overrun attacks. It may also be possible to build on CETUS in a way that allows more intelligent and informative code profiling to help diagnose performance issues. These are things that are not truly tested for in COS 217, but it would be interesting to create a whole suite of diagnostic tools like Dynamaid.

Bibliography

[1] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[2] John R. Levine. *flex and bison - Unix text processing tools.* O'Reilly, 2009.

[3] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.

[4] Caf. After segfault: Is there a way, to check if pointer is still valid?, December 2010. Available at `http://stackoverflow.com/questions/4401193/after-segfault-is-there-a-way-to-check-if-pointer-is-still-valid`.