Identifying the Root Causes of Memory Bugs Using Corrupted Memory Location Suppression

Dennis Jeffrey
Univ. of California, Riverside
jeffreyd@cs.ucr.edu

Neelam Gupta
guptajneelam@gmail.com

Rajiv Gupta
Univ. of California, Riverside
gupta@cs.ucr.edu

Abstract

We present a general approach for automatically isolating the root causes of memory-related bugs in software. Our approach is based on the observation that most memory bugs involve uses of corrupted memory locations. By iteratively suppressing (nullifying) the effects of these corrupted memory locations during program execution, our approach gradually isolates the root cause of a memory bug. Our approach can work for common memory bugs such as buffer overflows, uninitialized reads, and double frees. However, our approach is particularly effective in finding root causes for memory bugs in which memory corruption propagates during execution until an observable failure such as a program crash occurs.

1. Introduction

Software programming is an error-prone activity. Mistakes and misunderstandings can translate to program source code in the form of errors called *bugs*. These bugs can affect the behavior of programs in potentially disastrous ways. Due to the prevalence of software systems presently in use – including those used in critical, life-or-death domains – the task of locating and removing software bugs is a necessity. This task is called *Software Debugging*. Debugging involves a series of steps that can be slow, tedious, and difficult. These steps include: locating bugs in the source code; understanding the nature of the bugs; and modifying the source code to eliminate the bugs without introducing any new bugs. The task of debugging can be especially challenging in large and complex software systems. Techniques to help automate the debugging process can assist developers in more efficiently eliminating bugs, resulting in more robust and reliable software.

Memory-related bugs, such as buffer overflows and uninitialized reads, are an important class of software bugs that are particularly tricky to handle. This is because the corrupted memory locations resulting from these bugs can sometimes make it difficult to link an observed program failure – such as a crash or corrupted value – to the root cause. The effect of one corrupted memory location can propagate (spread) to other memory locations during program execution. By the time an observable program failure occurs due to some corrupted memory location, many other memory locations could have already become corrupted. Thus, the root cause of the bug may be far removed from the point at which the bug becomes apparent. Also, different kinds of memory-related problems can influence each other even though they may all be due to a single root cause. For instance, a program bug could lead to a buffer overflow that unexpectedly corrupts the value of some variable, yet does not immediately exhibit any obvious failure. Later on, the corrupted variable may be used to read from an uninitialized memory location containing an arbitrary value, and that arbitrary value could then be passed to the function "free" which may finally result in a program crash. In a complex case such as this, it may take considerable effort to sort through all of the memory-related problems to isolate the root cause. Tools and techniques for debugging that do not work well for memory bugs, or that specialize in detecting only certain kinds of memory bugs, may therefore have limited effectiveness in general at isolating the root causes of memory bugs.

In this paper, we present a generalized approach that focuses on isolating the root causes of memory bugs. Our approach views a memory-related bug as an instruction that corrupts some memory location, which can in turn corrupt other memory locations as the effect propagates arbitrarily far during execution until an observable program failure occurs. The key assumption motivating our approach is the following: if a program execution reveals a memory bug and the effect of the root cause of that bug is suppressed (nullified) during program execution, then all resulting memory corruption will be avoided and no memory-related failure will occur. Based on this assumption, we propose an approach that searches for the root cause of a memory bug by repeatedly executing a program on the same failing input; on each re-execution, we suppress the direct

cause of an observed memory failure and all of its direct and indirect effects in the execution. This will ensure that the same program failure will be avoided. If the suppression only captures some of the memory corruption but not all of it, then it is assumed that the remaining corrupted memory will lead to a later failure. We therefore repeat this process until finally a particular memory location suppression results in no observed memory-related failures during execution. The statement associated with this latest suppression is then highly likely to be the root cause of the memory corruption. Essentially, our approach isolates the root cause of a memory bug by gradually suppressing all of the instructions involving corrupted memory locations during execution. Our approach is applicable to any kind of memory-related errors that involve corrupted memory locations.

The rest of this paper is organized as follows. The next section describes our general approach for isolating the root causes of memory bugs. Section 3 describes some empirical results obtained by using our implemented approach on a set of benchmark programs. Related work is described in Section 4, and our conclusions and plan for future work are summarized in Section 5.

2. Isolating root causes of memory bugs

2.1. Motivation

There are a variety of types of memory-related bugs that can be present in software. *Buffer overflows* occur when memory locations are accessed that are outside of buffer boundaries. This can cause unexpected corruption of program data and can potentially result in a program crash. *Stack smashing* occurs when the return address for a called function on the call stack is overwritten. When this occurs, the program may crash upon the function return when program control tries to jump to an illegal address. In more dangerous situations, the value used to corrupt the function return address may be carefully chosen by an attacker so that program control passes to malicious code. *Uninitialized reads* occur when a memory location is loaded prior to any store of a proper value into that location. This may lead to memory corruption due to an unexpected, arbitrary value being loaded. *Double frees* occur when a call to function "free" is made on a particular location that has already been previously freed. This situation represents a mishandling of allocated memory, and leads to an abort of the executing program within the call to "free."

Memory bugs often manifest themselves in the form of symptoms that are observed during program execution, such as a program crash or a corrupted (unexpected) value. However, in general the root cause of a bug can be far removed from the points at which problems are observed during execution. This can make it difficult to isolate the root cause when the symptom of a memory bug is observed.

We observe that there is one common trait that memory bugs generally share: they involve corrupted memory locations. When the root cause of a memory bug is traversed during program execution, this leads to a corrupted memory location whose effects can propagate arbitrarily far into the execution, corrupting other memory locations along the way. Eventually, one of these corrupted locations may result in a failure such as a program crash. Intuitively, we can expect that if these corrupted memory locations were actually *not* corrupted during execution of the program, then the program would have proceeded normally without experiencing any memory-related failures. This intuition motivates the key idea of our approach: *to suppress (nullify) the effects of the instructions exercised during a program execution that we know involve corrupted memory locations*. The assumption is that if one memory location becomes corrupted during execution of a program, and the effects of this corruption propagate to subsequent statements that lead to further memory corruption, then identifying and suppressing the initial corruption and its effects will avoid all subsequent memory corruptions. This will prevent any memory-related failures from occurring during program execution. The point at which the initial corruption occurred is then highly likely to be the root cause of the memory bug.

Example. We constructed the example in Fig. 1 to show how a single memory bug can lead to memory corruption that propagates through multiple statements before a program crash occurs. This example involves a piece of code in which there exists an error at line 4. This type of error can happen due to a copy of lines 1 and 2 and a subsequent paste into lines 3 and 4. In this case, the developer forgets to change variable x into variable y at line 4. The effect of this error is that pointers $p2$ and $q2$ mistakenly refer to the same memory location. As a result, when a value is stored in location $*q2$ at line 8, then this clobbers the value originally stored there at line 6. Any subsequent uses of the value at location $*p2$/$*q2$ then make use of a corrupted memory location, which can lead to further corruption at other memory locations (lines 9, 10, and 11). Essentially, the initial bug at line 4 can lead to corruption that propagates through multiple locations until eventually an observable failure such as a program crash occurs (potentially at lines 12, 13, and 15).

Suppose the code in Fig. 1 is exercised on some input. This is represented pictorially in Fig. 2 (A). Initially, pointer $q2$ is corrupted at line 4 since it points to the incorrect memory location. That memory location is then corrupted at line 8, when the value previously stored at that location is mistakenly overwritten. Then, the definition at location a (line 9) is corrupted since it uses the corrupted value from location $*p2$/$*q2$. The definition for location b (line 10) is similarly corrupted. This further results in corruption of lo-

Let x and y be pointers to two malloc'ed memory regions, each able to hold two integers.
Let $intArray$ be a heap array of integers.
Let $structArray$ be a heap array of pointers to structs with a field f.

```
1:  int * p1 = &x[1];
2:  int * p2 = &x[0];
3:  int * q1 = &y[1];
4:  int * q2 = &x[0];       // copy-paste error:
                            // should be &y[0]
5:  *p1 = readInt();
6:  *p2 = readInt();        // gets clobbered at line 8
7:  *q1 = readInt();
8:  *q2 = readInt();        // clobbers line 6 definition
9:  int a = *p1 + *p2;      // uses corrupted *p2/*q2
10: int b = *q1 + *q2;      // uses corrupted *p2/*q2
11: int c = a + b + 1;      // uses corrupted a and b
12: intArray[c] = 0;        // buffer overflow
13: structArray[*p2]->f = 0;
                            // NULL dereference
14: free(p2);
15: free(q2);               // double free
```

Figure 1. Example code.

cation c (line 11). Now, suppose that the program crashes at line 12 due to corrupted array index c accessing an illegal address outside the bounds of array $intArray$. This is a buffer overflow error. When the buffer overflow error is observed at line 12, identifying the root cause at line 4 is not obvious since in practice we do not know all of the memory locations that are corrupted. If we only know that the array index c at line 12 is overflowing buffer $intArray$, then there are potentially many possibilities for the root cause. For instance, line 12 itself can be in error. Or, if we assume that variable c has a corrupted (incorrect) value, the problem can be at any earlier statement that influenced the value of variable c at line 12 (such as statements 9, 10, or 11). Or, it is possible that variable c has the correct value but buffer $intArray$ is actually the wrong size. In general, our example shows that it may be very difficult to identify the root causes of memory-related bugs, if the bugs lead to corruption that propagates through many statements before a failure is observed.

As a first step to begin searching for the root cause of the program crash at line 12, we re-execute the program while suppressing the memory corruption we currently know about that directly causes this crash. This is depicted in Fig. 2 (B). To do this, we notice at line 12 that the value at location c is used. Since location c (assumed to be corrupted) was last defined at line 11, then we re-execute the program on the same input, but during execution we suppress the definition at line 11 by *not* performing the store to location c. Accordingly, execution of any subsequent statements directly or indirectly influenced by this definition of c is also suppressed. In our example, only lines 11 and 12 are suppressed when the program is re-executed. However, suppose that now the program gets to line 13 and another crash occurs. This is possible since corrupted location $*p2$ is used as an index into an array of struct pointers. In our example, suppose that $structArray[*p2]$ is actually NULL. Then line 13 will result in a segmentation fault since NULL is dereferenced. The root cause of this fault is still line 4. However, it is yet again not obvious to identify the root cause because the corrupted location $*p2$ used at line 13 does not appear to be related to the actual erroneous line 4. To recognize the root cause, one would need to realize the fact that lines 2 and 4 assign the same memory location to both $p2$ and $q2$.

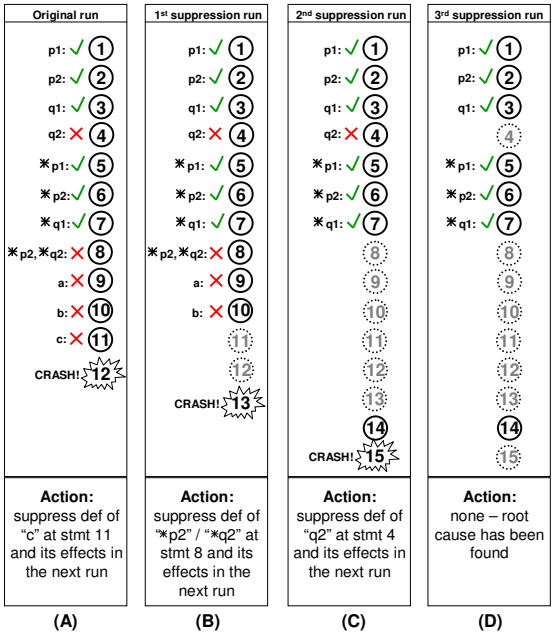

Figure 2. The original run and 3 suppression runs for the code in Fig. 1. Solid circles are executed statements, and dotted circles are suppressed statements. Statements defining a memory location are annotated with information showing whether the location is corrupt (x) or not corrupt (check).

We re-execute the program again to suppress the newly-revealed memory corruption directly involved in the crash at line 13. This is depicted in Fig. 2 (C). This time, we suppress the last definition of location $*p2$ – which is at line 8 – plus the other statements that are directly or indirectly influenced by the definition at line 8. Note that line 8 is the appropriate last definition, since pointer $q2$ actually refers to the same location as $p2$. In our example, during execution

input:
Program P and test case t causing a memory-related failure.
output:
Stmt identified as root cause for the memory failure.
algorithm SearchForMemoryFailureRootCause
begin
1: S_{def} := "undefined";
2: S_{supp} := {};
3: **while** an observable memory failure f occurs during execution of P using t **do**
4: S_{use} := the stmt instance at which f occurs;
5: L := the used mem location causing f at S_{use};
6: S_{def} := the stmt instance defining L prior to its use at S_{use};
7: **re-execute** P using t while suppressing
(1) stmt instances in S_{supp};
(2) the def of L at S_{def}; and
(3) any subsequent stmt instances directly or indirectly influenced by the def of L at S_{def};
8: **augment** set S_{supp} with the additional stmts suppressed at line 7 above;
endwhile
9: **report** the stmt associated with the latest S_{def};
end SearchForMemoryFailureRootCause

Figure 3. General approach to identify the root cause of a memory-related bug.

we therefore suppress lines 8, 9, 10, 11, 12, and 13. Note that lines 14 and 15 are not suppressed since the corrupted location defined at line 8 (which happens to be pointed to by both $p2$ and $q2$) does not actually influence the locations of the pointers $p2$ and $q2$ themselves used at lines 14 and 15. With these new suppressions, however, the program crashes yet again. The problem here is that at line 15, the program aborts due to a double free of the same memory location.

Finally, we re-execute the program a third time as shown in Fig. 2 (D). Here, we further suppress the definition of pointer $q2$ at line 4, plus its subsequent use at line 15. In total, we therefore execute only lines 1, 2, 3, 5, 6, 7, and 14. In this case, the program proceeds normally (without any crash) since the uses of all corrupted memory locations have been suppressed during execution. In other words, only the statements involving non-corrupted memory locations are exercised. As a result, we conclude that the most-recently suppressed statement – line 4 – must be the root cause of all the memory corruption that led to the program crashes.

2.2. General approach

Our general approach for identifying the root causes of memory-related bugs is described in Fig. 3. The input to our approach is a program and an associated test case from

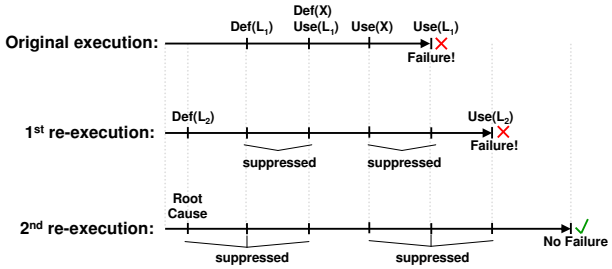

Figure 4. Example showing an abstract view of running our approach using a total of three executions.

which an observable memory failure occurs. Our approach then iteratively searches for and reports the root cause of the failure. The main loop comprising our approach is shown in lines 3 – 8 in Fig. 3. This loop iterates when an observed memory failure occurs. On each iteration, the corrupted memory location and associated instruction instance causing the program failure are identified (lines 4 and 5). Then, the statement instance that last defined this corrupted memory location is identified (line 6). The key step of our approach is then performed (line 7). In this step, the program is re-executed on the same input, but the effects of the statement instances directly or indirectly involving the corrupted definition identified at line 6, are suppressed. Additionally, any statement instances suppressed in prior loop iterations (if any) are also suppressed in this step. The effect is as if the suppressed statement instances are not executed. Finally, we update a set that is maintained to keep track of which statement instances have been suppressed thus far (line 8). If another failure is observed in the suppressed program execution, then the loop iterates again. On each loop iteration, at least one additional statement instance is suppressed during program execution. Thus, the loop will eventually terminate when the program execution produces no observed memory failures (in the extreme case, an empty program execution will not produce any memory-related failures). The statement associated with the last identified corrupted memory location definition is then reported as the likely root cause for the memory bug (line 9).

Fig. 4 is an example showing three program executions involved when running our approach to isolate the root cause of a memory bug. In the original execution (top horizontal line), a memory-related failure occurs due to accessing corrupted location L_1. This location is last defined at the point labeled "Def(L_1)," which also directly influences the point labeled "Def(X),Use(L_1)" and indirectly influences the point labeled "Use(X)." Thus, when the program is re-executed (middle horizontal line), the effects at all four of these program points are suppressed. However, another failure is observed due to accessing another corrupted location L_2, which is last defined at the point labeled "Def(L_2)." On

the next re-execution (bottom horizontal line), the effects of these two program points are then suppressed in addition to the other four suppressions performed in the previous re-execution. This leads to the result that no failures are observed. The statement associated with the most recent corrupted memory definition that is suppressed, "Def(L_2)," is then identified as the root cause of the memory bug.

Our approach for identifying the root causes of memory bugs assumes that during program execution, suppressing the root cause of a memory bug – as well as all of its direct and indirect effects – will prevent the memory-related failure from occurring as it will avoid the propagation of the memory corruption leading to the failure. Assumedly, if the root cause of a bug is not suppressed but only a portion of the total corrupted memory is suppressed, then the non-suppressed portions of corrupted memory would lead to other subsequent memory failures. Our approach takes advantage of these subsequent memory failures to gradually isolate the root cause. We believe this approach is promising because in general, root causes can propagate corrupted memory in a distributed fashion – with each corrupted memory location potentially influencing multiple other memory locations – rather than in a straight-line fashion. Thus, even though suppressing some corruption may avoid one failure, there is a chance that any remaining corruption would still lead to subsequent failures.

One observation to make about our approach is that it is general and can be used to isolate root causes for a variety of memory bugs. This is because it views memory bugs in terms of accesses to corrupted memory locations during program execution, and this trait is shared by most memory bugs. One notable exception to the generality of our approach, however, is memory leak bugs. For memory leaks, memory locations may actually not be corrupted; the only problem may be that certain memory locations are not freed when they should be. As a result, our abstraction of viewing memory bugs as accesses to corrupted memory locations does not directly apply to memory leaks.

Another observation about our approach is that it can be effective even when multiple independent memory bugs exist in a program. This is because when our approach is performed, it will eventually identify the root cause for *some* memory bug in the program (even though suppressions may be performed for multiple corrupted locations associated with different distinct bugs). Once this first identified root cause is corrected, our approach can then be run again if necessary to identify other remaining root causes.

2.3. Applications by example

Our approach can be effective for isolating the root causes of different kinds of memory bugs that involve corrupted memory locations. To show this, we now describe a few examples of real memory bugs found in software, and illustrate how our approach can be used to identify the root causes of the bugs.

```
      ...
167:  # define MAX_PATH_LEN 1024
      ...
233:  char ifname[MAX_PATH_LEN];
      ...
1009: strcpy(ifname, iname);     // potential overflow
      ...
1719: free(env), env = NULL;     // potential crash
```

Figure 5. Buffer overflow bug in `gzip-1.2.4`.

Buffer Overflow Example. Program `gzip-1.2.4` contains a known bug in which a global buffer overflow can occur if a specified file name is unexpectedly too long. The overflow occurs at a call to function "strcpy" at line 1009 in file *gzip.c*. However, in one execution instance, an observable failure is not observed until the program crashes later on in a call to "free" at line 1719 in *gzip.c*. The relevant code is illustrated in Fig. 5.

In this code at line 1009, source buffer $iname$ is a string containing an inputted file name. In the event that this file name is $>=$ 1024 characters, then the destination buffer $ifname$ will be overflowed, thereby corrupting certain global memory locations. Suppose the faulty program is executed and the specified file name $iname$ is actually 1107 characters long. When line 1719 is exercised, an attempt is made to free global variable env. However, this results in a program crash due to a corrupted value in env resulting from the overflow at line 1009.

When running our approach to isolate the root cause of the program crash, it is observed that the accessed memory address env that causes the crash, is last defined within the call to "strcpy" at line 1009 due to the buffer overflow at that point. The program is then re-executed on the same input while suppressing this particular definition into location env within the "strcpy" function call, plus any direct or indirect uses of that corrupted location (including the call to "free" at line 1719). This prevents the original program crash from occurring. However, in general a buffer overflow may corrupt many memory locations. In this particular example, a 1107-character string (not including the trailing NULL) is written into a 1024-character buffer, and so there are nearly 100 memory locations that could have been corrupted. Any of these other corrupted memory locations may lead to other program crashes which we would further suppress during subsequent program re-executions. This process may continue until eventually the entire buffer overflow is suppressed, at which point no failure related to the overflow will occur. At that point, the root cause at line

```
       ...
974:   if (dir_name != NULL)
975:       free (dir_name);      // potential double free
       ...
983:   if (dir_len > 0 && dir[dir_len - 1] == '/')
985:   {
           ...
989:       return;
990:   }
       ...
992:   dir_name = xmalloc (
           strlen (server_temp_dir) + dir_len + 40);
```

Figure 6. Double free bug in `cvs-1.11.4`.

1009 would be reported. In this case, the fix would be to either check the size of string $iname$ prior to performing the "strcpy" at line 1009, or else to use a safer string copy function such as "strncpy."

Double Free Example. Program `cvs-1.11.4` contains a known bug in which a double free can occur. The relevant code from file *server.c* is shown in Fig. 6. At lines 974 – 975 in this code, a pointer dir_name is freed if it is non-NULL. The pointer is then subsequently re-assigned to point to a new allocated memory region at line 992. However, in-between these two program points, there is a conditional check at lines 983 – 990 which can return from the current function. In the event that pointer dir_name is freed at line 975 and then the condition at line 983 subsequently evaluates to $true$, the function will return before dir_name is re-assigned to another memory chunk. In other words, dir_name will still contain the old memory address that was previously freed. If the given function is invoked again, the condition at line 974 will be $true$ since dir_name still contains the non-NULL, previously-freed address. This will lead to a double-free error at line 975, in which case the program will abort (crash).

This particular bug is interesting because the root cause is that of a missing statement. Here, the problem is that dir_name may contain a stale value which could be subsequently used, when in fact that stale value should be "cleared" by being set to NULL. Essentially, at the point of the double-free at line 975, the pointer dir_name is corrupted because it contains stale data. When our approach is run using an execution of this program that exhibits the double free, it is observed that the corrupted data in location dir_name used at line 975 is last defined in the call to "xmalloc" at line 992. Upon re-execution, when this assignment to dir_name is suppressed along with all direct and indirect uses of the pointer (including the uses in the call to "free" at line 975), then the double-free failure is avoided. As a result, the root cause identified by our approach is the definition of the pointer in question at line 992. Although the actual root cause is a missing statement, our approach identifies the closest match from among the statements actually present. This is because the statement identified by our approach is an assignment statement to pointer dir_name, which is the same kind of statement that is missing from the program and is the root cause of the bug. The assigned value to pointer dir_name at line 992 actually becomes corrupted at the point at which the value becomes stale and the program fails to overwrite that stale value. Given line 992 as identified by our approach and information about the observed failure that originally occurred at line 975, a developer can then figure out the missing statement that will correct the bug.

As demonstrated by the above examples, our general approach can be effective at isolating root causes for different kinds of memory bugs. Due to space limitations, we can only discuss the above few examples in detail. However, we have also successfully applied our approach to a variety of other memory bugs identified in real software. The next section provides implementation details and then summarizes the results of our experience in applying our approach to a set of benchmark programs.

3. Experimental study

3.1. Implementation

Our approach is implemented based on the high-level system design shown in Fig. 7. Our implementation consists of three main components: the *Valgrind Core*; the *Suppression Execution Tool*; and the *Memory Bug Root Cause Isolator*.

Figure 7. High-level system design for the implementation of our approach. The bi-directional arrows represent interactions between the system components.

Valgrind Core. The *Valgrind* infrastructure [18] provides a synthetic CPU in software and allows for dynamic binary instrumentation of an executable program. Valgrind includes a set of tools that perform certain profiling and debugging tasks, but new tools can be added to the infrastructure to perform customized instrumentation tasks.

Suppression Execution Tool. We created the Suppression Execution Tool as a new tool for Valgrind. The tool takes as input an executable program with a test case causing a memory failure, and a set of (possibly empty) instruction instances whose effects should be suppressed during

execution (called "suppression points"). The tool then performs the execution while simultaneously carrying out two tasks required by our approach: tracing and suppression. The tracing is required to see which memory locations are accessed and when. The suppression is required to nullify the effects of the instructions during execution that involve corrupted memory, when searching for the root cause of a memory bug.

For tracing during a given execution, the tool records a trace of the memory locations accessed (loaded from and stored to) during the execution. To do this, the tool instruments each non-suppressed load and store instruction to record the current program counter, its associated instance number, the type of instruction (i.e., load or store), and the address of the accessed memory location. This information makes it possible to identify which accessed memory location directly caused a memory failure, and which instruction instance last defined that memory location. The identified instruction instance can then be specified as one of the suppression points for the next execution (i.e., for the next invocation of the Suppression Execution Tool).

For suppression during a given execution, the Suppression Execution Tool performs "suppression information flow tracking" at all instructions, as well as "actual suppression" at the appropriate load and store instructions. To do the suppression information flow tracking, we associate every memory location and register with a *shadow location* that contains information about whether or not the associated location needs to have its effects suppressed. Initially, all shadow locations are marked as "not suppressed." At an instruction, if at least one of the used memory locations or registers is marked as "suppressed," then any defined memory locations or registers are also marked as "suppressed." On the other hand, if none of the used locations are marked as suppressed, then any defined locations are marked as "not suppressed." Tracking this information during execution ensures that any instructions which directly or indirectly use a suppressed location can have their effects suppressed as well. Memory locations are initially marked as suppressed when they are used in an instruction instance that is specified as a suppression point.

Besides tracking suppression information, "actual suppression" is performed at memory load and store instructions. At a store instruction instance that uses a suppressed location, the effect of the store is suppressed by *not* writing to the destination location. The effect is as if the store never occurred and the destination location retains whatever value was originally contained there. Similarly, for a load instruction instance that uses a suppressed location, the load is suppressed by *not* reading from the source location. The effect is as if the load never occurred and the destination register being loaded into retains whatever arbitrary data was originally contained there. This arbitrary data will never be used since anything affected by it will be suppressed as well. Note that under this approach, any program output affected by known corrupted/suppressed memory will also be suppressed (avoided). However, this is fine because the feedback for our algorithm is any observable failure involving previously-unknown corrupted memory, such as a new program crash or a corrupted output value that was previously not known to be corrupted.

There are a few special considerations to make when suppressing. If a corrupted location is used in a conditional check, then we must decide how the control-flow of the execution should be affected when we "suppress" it. In our current implementation, we chose the simple solution of merely bypassing the entire conditional structure involving the corrupted location. For example, we would bypass an entire "if/else" structure or an entire loop if the associated condition used a corrupted location. This simple solution seemed to work well in the benchmark programs we studied. An alternative solution may be to force the program to take a particular conditional outcome, perhaps based on profiling data taken from other test executions. Special consideration must also be made for corruption of the return address of a function call. This must be specially handled because we cannot avoid the function return (it would likely not be helpful to simply terminate execution), and we cannot simply jump to an arbitrary address upon function return. Instead, we use profiling data from the current and other test executions to cause the function to return to a known, valid address. Finally, corrupted input to system calls must also be handled. To do this, we simply refrain from making system calls when they involve at least one corrupted input value. The same approach can be used to handle library calls if desired, although we have not currently implemented this.

Memory Bug Root Cause Isolator. This is the main driver module for our approach that manages the suppression re-executions and identifies the suppression points. Given a faulty program and test case causing a memory-related failure in the program, this module first invokes the Suppression Execution Tool using an empty set of suppression points to record memory access tracing information from the test case execution. From this, a first suppression point is identified which is passed as input to a second invocation of the Suppression Execution Tool. If another program failure occurs, then another suppression point is identified and another re-execution is performed. Eventually, no memory-related failure will occur and the latest identified suppression point is reported as the likely root cause.

3.2. Programs, results, and discussion

We selected a variety of real programs with known memory bugs to study the effectiveness of our approach in iso-

lating the root causes. These subject programs are taken from [15, 16, 25] and are described in Table 1. We selected only single-threaded programs for our analysis, and chose programs representing a variety of memory bugs that involve corrupted memory locations. We did not select any programs with memory leak bugs since this type of memory bug is not currently handled by our approach. Lines of code were measured using the `SLOCCount` tool [10].

Program Name	Lines of Code	Bug Type	Root Cause Location
gzip-1.2.4	6304	GO	gzip.c, line 1009
ncompress-4.2.4	1436	SS	compress42.c, line 886
polymorph-0.4.0	1061	GO/SS	polymorph.c, line 118
tar-1.13.25	28366	ND	incremen.c, line 180
bc-1.06	10704	HO	storage.c, line 176
tidy-34132	35883	ND	parser.c, line 856
man-1.5h1	10750	GO	man.c, line 979
cvs-1.11.4	104086	DF	near server.c, line 992

Table 1. Overview of benchmark programs containing real memory bugs. In the "Bug Type" column, the following abbreviations are used: *global overflow* **(GO),** *heap overflow* **(HO),** *stack smash* **(SS),** *NULL dereference* **(ND), and** *double free* **(DF).**

In our experiments, we considered an "observable failure" of a program execution to be a program crash. The results of running our approach on the subject programs are given in Table 2. For each subject program, this table shows: the total number of program executions required, including the initial execution with no suppression ("# Exec. Req."); the source code statement identified by our approach as the likely root cause ("Identified Statement"); and the dependence-edge distance from the identified statement to the actual root cause ("Dep. Dist. to Root Cause"), which is 0 when the identified statement is precisely the root cause.

Besides programs `gzip` and `cvs` that were previously discussed in detail in Section 2.3, programs `ncompress`, `polymorph`, `tar`, and `tidy` all result in our approach precisely identifying the root cause of the memory bugs.

Programs `ncompress` and `polymorph` involve unchecked calls to "strcpy" that can overflow stack and global memory buffers, respectively. For `ncompress`, a segmentation fault does not occur until a later call to function "perror." The memory location directly involved in the crash turns out to be last defined in the faulty "strcpy." When this definition and its subsequent effects are suppressed during execution, program control reaches the return point of the current function call and then crashes again. This is due to the stack overflow corrupting the return address of the function call. When this corruption is suppressed and the function is able to return to a known valid address, then one more segmentation fault occurs upon function return due to another corrupted stack location resulting from the same faulty "strcpy." On the next suppression execution, no memory failures are observed. The faulty "strcpy" is then successfully identified by our approach as the root cause. For `polymorph`, a program crash occurs within the faulty "strcpy" itself, due to an attempted write to an unmapped memory region. When this store is suppressed, program control continues and eventually crashes again with a segmentation fault in a call to "strlen." The accessed memory location causing this crash is again last defined in the faulty "strcpy." Upon further suppression, the execution is able to complete without any memory failures, and the faulty "strcpy" is identified as the root cause.

Program	# Exec. Req.	Identified Statement	Dep. Dist. to Root Cause
gzip-1.2.4	2	gzip.c, line 1009	0
ncompress-4.2.4	4	compress42.c, line 886	0
polymorph-0.4.0	3	polymorph.c, line 118	0
tar-1.13.25	2	incremen.c, line 180	0
bc-1.06	2	storage.c, line 177	1
tidy-34132	2	parser.c, line 856	0
man-1.5h1	3	manfile.c, line 243	2
cvs-1.11.4	2	server.c, line 992	0

Table 2. Experimental results when searching for root causes using our approach.

For `tar` and `tidy`, segmentation faults occur due to NULL pointer dereferences. In both cases, the last definitions of the accessed memory locations containing NULL are identified as suppression points. This causes both executions to run to completion without exhibiting any failures. The statements causing the pointers to be NULL are then identified as the root causes of the bugs.

For program `bc`, our approach identifies a statement that is one dependence edge away from the root cause of the bug. In this case, a heap buffer overflow occurs due to an incorrect variable used in a loop condition. This causes corruption of the loop control variable that is used as an index into the overflowed buffer. However, a failure is not observed until the program crashes later on in a call to "malloc." The suppression point identified in this case happens to be associated with the statement at which the buffer overflow occurs. Upon suppression, the program executes without exhibiting any memory failures. As a result, the statement associated with the buffer overflow is identified as the likely root cause. However, this is actually one dependence edge away from the actual root cause, which is the surrounding loop condition that is ultimately responsible for the buffer overflow. The reason our approach is not able to precisely identify the root cause here is because the remaining corruption remains "hidden," since the program does not exhibit any failures even while this corruption remains.

For program `man`, an erroneous condition within a loop can prevent control from breaking out of the loop when necessary, resulting in a global buffer overflow. Unlike for most of the other subject programs, memory corruption actually

propagates quite a bit during execution before a failure is observed in this case. A segmentation fault first occurs in a location far removed from the original root cause, through multiple dependences and in a completely separate source file. The corresponding identified suppression point is also in the separate file. Upon suppression, program control proceeds until another segmentation fault occurs due to another corrupted memory location that is still present. Another suppression point is identified, and finally execution is able to terminate without exhibiting any failures. However, in this case the identified root cause is actually two dependence edges away from the actual root cause, since additional corrupted memory locations remained even though no memory failure was observed during the latest suppression execution. Given the high degree of memory corruption propagation that occurs in this program, however, our approach is still quite effective because it identifies a statement only two dependence edges away from the root cause.

As seen in Table 2, each benchmark program required between 2 and 4 total executions to isolate the root cause. Although instrumentation when executing within Valgrind causes the program to slow down considerably as compared to executing normally outside of Valgrind, each program execution within Valgrind in our experimental study never took more than a few seconds. As a result, our approach was able to isolate the root cause of the memory bug for each of our benchmark programs in time on the order of seconds. We believe these timing results to be reasonable given a debugging context.

Overall, the results of our experimental study suggest that our approach can be quite effective for precisely identifying the root causes of a variety of memory bugs. This is true in relatively simple cases when a failure may be directly linked to the root cause via a dependence edge, and also in more complex cases where significant memory corruption propagation may occur prior to an observed program failure.

4. Related work

Fault Detection. Detecting the presence of bugs in software is a topic that has been extensively studied. *Eclat* [19] infers an operational model of correct program behavior and identifies inputs that violate this model. *ESC/Java* [6] identifies certain programming errors at compile-time using an annotation language. *Check 'n' Crash* [4] derives error conditions statically and then attempts to generate test cases to dynamically verify the existence of errors. *Daikon* [5] and *DIDUCE* [8] automatically extract program invariants and monitor for violations during execution. *AccMon* [25] also uses an invariant-based approach that identifies program instructions that typically access different memory locations. Other bug-finding approaches are designed for specific kinds of bugs. *CP-Miner* [14] searches for copy-paste bugs in large-scale software systems. *EXPLODE* [22] identifies data integrity bugs in storage systems. Other tools such as *Valgrind* [18], *Purify* [9], and *CCured* [17] identify particular kinds of memory-related bugs.

Fault Localization. As illustrated by the approach proposed in this paper, the task of locating bugs in program source code is performed once a bug has been exposed. *Fault localization* is the process of automatically narrowing or guiding the search for program bugs to help developers identify erroneous statements more quickly. The following techniques have been proposed to help isolate bugs in programs. *Static Slicing* [21] identifies a subset of program statements that may influence the value of a variable at a particular program location. The related concepts of *Dynamic Slicing* [1, 13, 24] and *Relevant Slicing* [2, 7] have also been studied. In general, slicing identifies *all* statements that influence or are influenced by a variable at a program point. This set of statements can potentially represent many chains of dependences in a program. Thus, slicing may identify a relatively large set of statements, only one of which may be a root cause (and slicing alone provides no way to distinguish between the identified statements). Our approach, on the other hand, seeks to pinpoint precisely the single root cause of a memory bug. Our approach can effectively identify the appropriate dependence chain containing the root cause, and then isolate the point along that chain at which the root cause is located.

Predicate Switching [23] attempts to isolate erroneous code by identifying "critical" predicates whose outcomes can be altered during a failing run to cause it to become successful. However, critical predicates may not be found in all cases. Moreover, once a critical predicate is found, then it may still be difficult to pinpoint the root cause of a bug based upon the critical predicate. A critical predicate may be used to identify a subset of statements that might be likely to contain the root cause, but like for slicing, this set of statements may not uniquely identify the root cause. Our approach does not rely on searching for critical predicates, and seeks to identify a single statement that is highly likely to be a root cause.

In *Delta Debugging* [3], failure-inducing input is identified that allows for the computation of cause-effect chains for failures, which can in turn be linked to faulty code. This involves substituting state (the values of variables) between passing and failing runs. A related *Value Replacement* idea was proposed [11] that attempts to replace the values used at certain statement instances with alternate sets of values; if any value replacement causes a failing run to become successful, then the statement associated with the value replacement may be erroneous. The *Nearest Neighbor* approach [20] compares the spectra for two similar executions (one successful and one failing) to identify the most sus-

picious parts of a program. *Tarantula* [12] is a statistical approach that ranks program statements according to suspiciousness values determined by how many failing versus passing tests exercise each statement. In general, these approaches analyze the information from multiple test cases to try to prioritize the program statements according to their likelihood of being faulty. Our approach, in contrast, uses only a single failing test case and tries to isolate the root cause of a memory bug by repeatedly suppressing more and more instruction instances in the test case execution until the failure is avoided.

5. Conclusions and future work

We have presented a general approach for automatically identifying the root causes of memory-related bugs by suppressing the effects of corrupted memory locations during a faulty program execution. Our approach is designed to work for any memory errors that involve memory corruption, particularly those in which memory corruption propagates arbitrarily far during execution until an observable memory-related failure occurs. This includes – but is not limited to – common memory errors such as buffer overflows, stack smashes, uninitialized reads, and double frees. We presented an empirical study in which our approach was able to isolate the root causes of memory bugs found in several real software systems. In the future, we hope to enhance our approach to handle other kinds of memory bugs such as memory leaks. We also hope to conduct a more extensive empirical study to better understand the benefits of our approach on software systems containing memory bugs.

Acknowledgements. We would like to thank the anonymous reviewers for their valuable feedback. This research is supported by NSF grants CNS-0751961, CNS-0751949, CNS-0810906, and CCF-0753470 to UC Riverside.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[2] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. *International Conference on Software Maintenance*, pages 348–357, September 1993.

[3] H. Cleve and A. Zeller. Locating causes of program failures. *Intl. Conf. on Software Eng.*, pages 342–351, May 2005.

[4] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. *International Conference on Software Engineering*, pages 422–431, May 2005.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[6] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[7] T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. *Foundations of Software Engineering*, pages 303–321, September 1999.

[8] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. *International Conference on Software Engineering*, pages 291–301, May 2002.

[9] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. *Proceedings of the USENIX Winter Technical Conference*, pages 125–136, 1992.

[10] `http://www.dwheeler.com/sloccount`.

[11] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. *International Symposium on Software Testing and Analysis*, July 2008.

[12] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *Intl. Conf. on Automated Soft. Eng.*, pages 273–282, November 2005.

[13] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. on Soft. Eng.*, 32(3):176–192, March 2006.

[15] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. BugBench: Benchmarks for evaluating bug detection tools. *Workshop on the Evaluation of Software Defect Detection Tools (co-located with PLDI)*, June 2005.

[16] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. *International Symposium on Computer Architecture*, pages 284–295, June 2005.

[17] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. *Symposium on Principles of Programming Languages*, pages 128–139, January 2002.

[18] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Conf. on Prog. Lang. Design and Impl.*, pages 89–100, June 2007.

[19] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. *Object-Oriented Programming, 19th European Conf.*, pages 504–527, July 2005.

[20] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. *International Conference on Automated Software Engineering*, pages 30–39, October 2003.

[21] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[22] J. Yang, C. Sar, and D. R. Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. *Operating Sys. Design and Impl.*, pages 131–146, Nov. 2006.

[23] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. *International Conference on Software Engineering*, pages 272–281, May 2006.

[24] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 169–180, 2006.

[25] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. P. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. *Intl. Symp. on Microarchitecture*, pg. 269–280, Dec. 2004.