Execution Suppression: An Automated Iterative Technique for Locating Memory Errors

DENNIS JEFFREY, VIJAY NAGARAJAN, and RAJIV GUPTA
The University of California, Riverside
and
NEELAM GUPTA

By studying the behavior of several programs that crash due to memory errors, we observed that locating the errors can be challenging because significant propagation of corrupt memory values can occur prior to the point of the crash. In this paper, we present an automated approach for locating memory errors in the presence of memory corruption propagation. Our approach leverages the information revealed by a program crash: when a crash occurs, this reveals a subset of the memory corruption that exists in the execution. By suppressing (nullifying) the effect of this known corruption during execution, the crash is avoided and any remaining (hidden) corruption may then be exposed by subsequent crashes. The newly-exposed corruption can then be suppressed in turn. By iterating this process until no further crashes occur, the first point of memory corruption – and the likely root cause of the program failure – can be identified. However, this iterative approach may terminate prematurely, since programs may not crash even when memory corruption is present during execution. To address this, we show how crashes can be exposed in an execution by manipulating the relative ordering of particular variables within memory. By revealing crashes through this variable re-ordering, the effectiveness and applicability of the execution suppression approach can be improved. We describe a set of experiments illustrating the effectiveness of our approach in consistently and precisely identifying the first points of memory corruption in executions that fail due to memory errors. We also discuss a baseline software implementation of execution suppression that incurs an average overhead of 7.2x, and describe how to reduce this overhead to 1.8x through hardware support.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

General Terms: Algorithms, Experimentation, Performance, Reliability, Verification

Additional Key Words and Phrases: execution suppression, fault localization, hardware support, memory corruption propagation, memory errors, variable re-ordering

---

Authors' addresses: D. Jeffrey, Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043; e-mail: dennisjeffrey@google.com. V. Nagarajan, School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh, EH8 9AB; e-mail: vijay.nagarajan@ed.ac.uk. R. Gupta, The University of California at Riverside, 900 University Ave., Riverside, CA 92521; e-mail: gupta@cs.ucr.edu. N. Gupta, Independent; e-mail: guptajneelam@gmail.com.
This research is supported by NSF grants CNS-0751961, CNS-0751949, CNS-0810906, and CCF-0753470 to UC Riverside.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0164-0925/20YY/0500-0001 $5.00

1. INTRODUCTION

Software programming is a complicated and error-prone activity. It is commonly known that programs do not always behave as intended, and this is often caused by the presence of *errors* in program source code. When an error is traversed during execution of a program, this can cause an *infection*, in which the executing program state differs from what is intended by the programmer. Sometimes, infected executions can result in a *failure*, which is an externally-observable, abnormal program behavior such as a crash or the outputting of incorrect values. Unfortunately, due to the prevalence of software systems in use around the world, program errors have the potential to cause widespread disaster and even loss of life. As a result, the task of locating, understanding, and fixing errors in programs is of the utmost importance. This task is known as *software debugging*. Debugging is an important and necessary phase of software development that promotes more robust and reliable software.

Memory errors represent an important class of software error that causes mishandling of memory during program execution. One example of mishandling of memory occurs when a program attempts to read from or write to an incorrect memory location. Such memory errors often manifest themselves in the form of a program crash. Some common types of memory errors are the following.

—*Buffer overflows*: This occurs when memory locations are accessed that are outside of proper buffer boundaries. Such overflows can cause unexpected corruption of program data that can eventually cause a crash. *Stack smashing* is one type of problem that can arise due to a buffer overflow, which corrupts the return address of a function on the call stack.

—*Uninitialized reads*: This occurs when the value contained in a memory location is loaded before any proper value has been stored into that location. This can lead to unexpected program behavior due to an arbitrary value being loaded.

—*NULL dereferences*: This occurs when the pointer used to access a memory location is unexpectedly NULL. A NULL dereference may occur due to an uninitialized read or to prematurely assigning the value NULL to a pointer variable.

—*Dangling pointers*: A dangling pointer is one that does not point to a valid object of the appropriate type in memory. One cause is when a memory object is explicitly deallocated, while a pointer to that object retains its original address value. Subsequent uses of this dangling pointer can lead to unexpected program behavior.

—*Double frees*: This occurs when a call to function `free`() is performed using a deallocated address that has already been previously freed. Such an error may lead to a program abort.

In general, a memory error manifests itself by undergoing three specific events at one or more execution points during program execution.

(1) *Traversal of the error.* This is when the portion of code ultimately responsible for a program failure is executed.

(2) *First point of memory corruption.* When an execution has become infected due to traversal of an error, a first point of memory corruption may then occur,

at which point memory is mishandled in some way. This can in turn cause memory to be mishandled at subsequent execution points as well (i.e., memory corruption may *propagate* during execution).

(3) *Failure*. This is the point at which a developer can actually observe that an execution has become infected. Not all infections, however, may lead to failures. One type of failure that often results from memory corruption is a program crash, though not all memory corruption may lead to a crash, and there are other types of failures such as producing incorrect output.

Once a failure occurs during execution, a developer must find the location of the error so that it can be eliminated. However, in general the error may not be at the same point at which the failure occurs. The infected portion of execution, separating the traversal of the error from the actual failure, may be very long. We conducted a study involving 11 programs that can crash due to memory errors, to study how memory corruption can propagate during an infected execution (described in detail in Section 2). We found that not only can the length of an infected execution be very large, but there can also be a significant amount of memory corruption propagation during the infected execution that can effectively conceal the location of the true error. Further complicating matters is the possibility that different inputs traversing the same error can lead to different failures at different execution points. The goal of our work is to describe an automated technique that will account for these issues and significantly reduce the burden placed on developers for locating memory errors.

Our approach for assisting developers in locating memory errors is based on two key ideas: *execution suppression* and *variable re-ordering*. We use our *execution suppression* technique to identify the first point of memory corruption in an infected execution that fails due to a crash caused by a memory error. We assume that the first point of memory corruption is either at, or very close to, the memory error. Suppression is the idea of *omitting one or more instructions during program execution*. We use the concept of suppression iteratively to gradually isolate the first point of memory corruption. When a crash occurs, this reveals that the memory location(s) accessed at the point of the crash is corrupt. In essence, each crash reveals a subset of the memory corruption in an infected execution. This subset of known memory corruption, and everything else in the infected execution directly or indirectly dependent upon it, is then suppressed during re-execution of the program by simply omitting the effect of the associated instructions during execution. This effectively causes only the subset of the original execution to be re-executed, that does not involve or depend upon the identified memory corruption. Note that this guarantees that the original crash, and any crashes that might have occurred due to the suppressed instructions, will be avoided in the re-execution. This is because the effect of all instructions directly or indirectly dependent upon any suppressed instruction will be omitted during re-execution. At the end of the re-execution, if the result is that no other crashes occur, then the last suppressed point of memory corruption is likely to be the first point of memory corruption in the execution. On the other hand, if another crash does occur, then this reveals that additional memory corruption remains in the execution and so the process should be repeated to isolate the first point of memory corruption.

The idea of execution suppression fundamentally relies on the assumption that if memory corruption exists in an execution, then a program crash will occur. However, this assumption does not always hold in practice because memory corruption will not always lead to a crash. Our second key idea, *variable re-ordering*, is a technique that can sometimes expose crashes due to memory corruption in an execution that does not otherwise result in a crash. The intuition is as follows: even though the relative ordering of variables in memory should not affect the correctness of a program, in the presence of memory errors, the relative ordering can in fact affect where and when crashes might occur. We describe an approach that systematically tries different variable orderings in memory to attempt to expose crashes due to memory corruption. By combining execution suppression with variable re-ordering, the effectiveness and applicability of our approach is greatly improved.

Our approach is designed to be iterative so that the first point of memory corruption can be identified even in executions with significant propagation of memory corruption. Our approach is also fully automated, which makes it useful for quickly determining whether different inputs exhibiting different program failures are due to the same error. Moreover, our approach is general and can be applied to any memory errors that involve corrupted memory and can result in a program crash.

To implement our approach, we describe a software implementation of execution suppression (not including variable re-ordering) that results in an average overhead of 7.2x. We then show how this can be reduced to an average overhead of 2.7x by making use of hardware support in Itanium processors that was originally intended for deferred exception handling. Finally, we show how to reduce the average overhead to 1.8x by extending the cache, main memory, and data bus with an extra bit for each word. Our performance studies were conducted on the SESC simulator targeting the MIPS instruction set.

We describe experimental results of running our approach using a set of 11 real programs with known memory errors to show that the first point of memory corruption can be precisely identified by our approach in all benchmark programs. In these programs, the first point of memory corruption is always either at, or very close to, the actual memory error.

The contributions of this paper are as follows.

—A new, automated approach for locating memory errors in the presence of memory corruption propagation that incorporates the ideas of *execution suppression* and *variable re-ordering*.

—A detailed study of memory corruption using a set of 11 real programs with known memory errors to motivate the development of our approach.

—Discussion and overhead comparison of different software and hardware implementations of execution suppression.

—An experimental evaluation of our approach using 11 real programs containing memory errors and a set of crashing and non-crashing inputs to those programs.

The rest of this paper is organized as follows. In the next section, we describe a detailed study of memory corruption we conducted to motivate the development of our approach. Section 3 describes our approach in detail. Software and hardware issues for implementing execution suppression, along with a performance comparison

Table I. Memory error programs analyzed in our memory corruption study.

Program Name	# Lines of Code	Error Type	Error Location	Program Description
gzip-1.2.4	6.3 K	GO	gzip.c: 828	file compression
man-1.5h1	10.8 K	GO	man.c: 979	display manual pages
bc-1.06	10.7 K	HO	storage.c: 176	arbitrary precision calculator
pine-4.44	211.9 K	HO	bldaddr.c: 7270	Internet news and e-mail
mutt-1.4.2.1	65.9 K	HO	utf7.c: 152	e-mail client
ncompress-4.2.4	1.4 K	SO	compress42.c: 886	file compression
polymorph-0.4.0	1.1 K	SO	polymorph.c: 191	filename converter
xv-3.10a	69.2 K	SO	xvbmp.c: 165	image manipulation
tar-1.13.25	28.4 K	ND	incremen.c: 180	archiving utility
tidy-34132	35.9 K	ND	parser.c: 854	HTML quality enhancer
cvs-1.11.4	104.1 K	DF	near server.c: 992	versioning system

of several different implementations, are presented in Section 4. An experimental study illustrating the effectiveness of our approach is given in Section 5. Section 6 discusses related work, and our conclusions are summarized in Section 7.

2. MEMORY CORRUPTION STUDY

2.1 Subject Programs and Results

We conducted a study of memory corruption using 11 real programs containing known memory errors to motivate the development of our approach for automatically locating the first point of memory corruption in an execution. The programs used in our study were obtained from [Lu et al. 2005; Narayanasamy et al. 2005; Zhou et al. 2004] and are described in Table I. The first column in the table shows the program name and version number. The second column shows the number of lines of code (in thousands). In the third column, the following abbreviations are used to indicate the memory error type: global buffer overflow (GO); heap buffer overflow (HO); stack buffer overflow (SO); NULL dereference (ND); and double free (DF). The fourth column shows the file name and line number of the location of the memory error. The right-most column gives a brief description of the program. We selected these subject programs because they have been used as benchmarks in prior research and they contain a variety of types of memory errors.

The goal of our study was to understand the nature of memory errors to motivate the development of an effective approach for automatically locating them. For each memory error, we identified the following: the *location of the error* (already known beforehand); the *first point of memory corruption*; and the *failure point* (this was always the point of execution termination, which was either a crash, or an observed wrong output). To be able to specify the first point of memory corruption, we used the following definition for memory corruption.

Definition 2.1.1. **Memory corruption** occurs during the execution of a program when either an incorrect memory location is accessed (read or written when it should not have been), or an incorrect memory address value is assigned to a (pointer) variable.

The above definition for memory corruption captures the act of mishandling memory addresses as well as the propagation of corrupt memory address values. Note that memory corruption can propagate through other non-address values that may become infected due to memory corruption in an execution. However, infected

Table II. Results for each analyzed input for each subject program. In the column heading for "Distance", the following abbreviations are used: error traversal (ERROR), first point of memory corruption (CORRUPT), and point of execution termination (END).

Program Name	Input Type	Distance: [ERROR→CORRUPT] + [CORRUPT→END]	
		# Static Dependence Edges	# Executed Instr. Instances
gzip-1.2.4	No Crash	0 + 1	0 + 41,168
	Crash Point 1	0 + 1	0 + 36,902
man-1.5h1	Crash Point 1	1 + 8	296 + 14,239,521
bc-1.06	No Crash	1 + 0	8 + 33,567
	Crash Point 1	1 + 1	8 + 5,004
pine-4.44	No Crash	5 + 20	1,103 + 1,390,483
	Crash Point 1	5 + 14	1,103 + 10,165
mutt-1.4.2.1	No Crash	0 + 9	0 + 140,750
	Crash Point 1	0 + 8	0 + 5,697
ncompress-4.2.4	No Crash	0 + 1	0 + 7,318
	Crash Point 1	0 + 2	0 + 11,616
	Crash Point 2	0 + 1	0 + 19,637
polymorph-0.4.0	No Crash	1 + 2	4,294 + 99,723
	Crash Point 1	1 + 2	4,321 + 99,762
	Crash Point 2	1 + 1	4,354 + 113,083
xv-3.10a	No Crash	1 + 2	122 + 185,818
	Crash Point 1	1 + 1	124 + 158,640
tar-1.13.25	Crash Point 1	0 + 1	0 + 210,505
tidy-34132	Crash Point 1	0 + 2	0 + 57
cvs-1.11.4	Crash Point 1	1 + 0	5,164 + 0

non-address values are not considered to be "memory corruption" according to our definition, since they cannot directly cause a crash unless they are later used to compute an incorrect memory address. Essentially, we define "memory corruption" to be a mishandling of memory that can directly cause a program crash.

The data we collected for each subject program in our study is reported in Table II. For each of the subject programs, we created a variety of different inputs that we knew traversed the error and triggered memory corruption. We then observed whether or not a crash occurred, and if so, at which program statement the crash occurred. For each distinct execution outcome (either no crash, or a crash at a particular statement), we selected one representative input associated with that outcome (listed in column 2) and studied it in detail. From the execution of each input, we measured the *distance* from the traversal of the error until the first point of memory corruption, and from the first point of memory corruption until execution termination. This distance was measured first in terms of *static dependence edges* (column 3), showing the extent to which memory corruption can propagate during execution. The static dependence edges were identified by looking at the program code statically, following the chain of data and control dependencies observed in the source code. The distance was also measured in terms of *dynamic instruction instances in the execution* (column 4), indicating the length of the program execution between these particular execution points.

For example, for program `gzip` that has a global buffer overflow in a call to `strcpy`(), we were able to create one memory-corruption-inducing input that did not crash, and another input that caused a crash at one program point. For both inputs, the static dependence and dynamic instruction instance distances from the error traversal to the first point of memory corruption is 0. This indicates that in this program, the traversal of the error occurs precisely at the first point of memory corruption. On the other hand, both inputs have a static dependence distance of 1

from the first point of memory corruption until the point of execution termination, and a corresponding dynamic instruction instance distance of 41,168 instructions (the non-crashing input) and 36,902 instructions (the crashing input).

2.2 Key Observations

From the results presented in Table II, it can be seen that static dependence distances from the point of error traversal until the point of execution termination are usually more than 1, and sometimes considerably more than 1 (e.g., programs `man`, `pine`, and `mutt`). Even when static dependence distances are relatively small, the instruction instance distances can be quite large (e.g., programs `polymorph` and `xv`). Thus, the first observation we make from our study concerns these total distances.

Observation 2.2.1. (**Total distances**) The total distance, both in terms of static dependence edges as well as dynamically-executed instruction instances, between the point of error traversal and the point of execution termination, can be large.

The above observation suggests that in crashing executions, the memory error may be difficult to manually locate from the point of the crash. Traversal of the error may have occurred much earlier in time than the point of the crash. There may also be a large degree of memory corruption propagation during execution. Thus, an automated approach to help developers isolate the first point of memory corruption can be of great help in locating memory errors.

An interesting result pertaining to static dependence distance occurs for the non-crashing input of program `bc`. In this case, the dependence distance from the first point of memory corruption until execution termination is 0, but this is because there happens to be no memory corruption propagation during this execution. In the execution, a buffer overflow causes an unexpected write to another memory location, but this memory location is associated with a variable that is never accessed during the rest of the execution.

The second important observation we make from the results of our study deal with the types of inputs we analyzed. All analyzed inputs triggered memory corruption, but they often had different execution outcomes.

Observation 2.2.2. (**Inputs triggering memory corruption**) Different inputs triggering memory corruption may lead to crashes at different program locations, or they may result in no crash at all.

If different inputs lead to crashes at different program locations, this can be misleading and may cause a developer to suspect that the inputs reveal multiple distinct errors when in fact they may all be due to the same error. An automated approach to help locate memory errors can help a developer to quickly group crashing inputs according to their associated errors. One useful application of this capability would be to allow developers to prioritize the fixing of errors by determining which errors are associated with the most undesirable crashing inputs.

Inputs that trigger memory corruption but do not result in any crash may conceal the fact that a memory error exists. Even if wrong output is produced, a developer may not be able to easily tell whether the wrong output is due to a memory error or to a non-memory error. In order to improve software reliability, it would be

desirable for an input that triggers memory corruption to result in a crash. This guarantees that the memory error will be revealed and encourages the developer to address the problem.

The inputs we created for program `man` represent an interesting case from among the programs with buffer overflows. This is the only program with a buffer overflow in which we could *only* create inputs that trigger memory corruption and then crash. For this program, it turns out that the error is such that if at least one memory location gets corrupted, then a crash will happen. Thus, we could not create any inputs for this program that triggered corruption but did not crash.

A third observation we make concerns the relative distance from error traversal to the first point of memory corruption, compared to the distance from the first point of memory corruption to the point of execution termination.

Observation 2.2.3. (**Relative distances**) Across all programs in our study, the relative distance from error traversal to the first point of memory corruption, is generally *considerably less* than the distance from the first point of memory corruption to the point of execution termination.

In all programs except `pine`, the static dependence distance from the root cause to the first point of memory corruption is always 0 or 1. As a result, an automated approach to isolate memory errors can still be very effective if it seeks to only isolate the first point of memory corruption. From the first point of memory corruption, a developer should only need to exert minimal effort to find the actual error.

Together, the three main observations resulting from our study of memory errors and memory corruption motivate our approach that is based on the ideas of execution suppression and variable re-ordering. In the next section, we describe in detail our approach for isolating the first point of memory corruption in an execution that fails due to a memory error.

3. APPROACH

Given a failing execution that involves memory corruption due to a memory error, our approach uses the key ideas of *execution suppression* and *variable re-ordering* to automatically locate the first point of memory corruption. This can help a developer to quickly locate the error itself. We first describe the execution suppression technique. Then, we describe the variable re-ordering technique that can be used to improve the effectiveness and applicability of execution suppression. Finally, we describe the complete approach that incorporates both techniques.

3.1 The Execution Suppression Component

The notion of execution suppression involves *omitting one or more statements (instructions) during program execution.* This idea can be used iteratively to reveal the first point of memory corruption in an execution. In general, when there are multiple instances of memory corruption that exist in an execution, then eventually one of these instances may cause a crash. If we suppress (omit execution of) the associated statements directly causing this crash, as well as any other statements directly or indirectly dependent upon these suppressed statements, this would avoid the crash and allow execution to proceed further. This provides the remaining memory corruption opportunity to cause other crashes. This, in turn, reveals more of the

memory corruption, until finally the first point of memory corruption is revealed. The first point of corruption is assumed to be identified when no further crashes occur, because suppressing the first point of memory corruption will ensure that the program will not produce any further crashes. This is the essence of the execution suppression approach. We stress that execution suppression is a *dynamic* approach that involves runtime tracking of memory location accesses (as opposed to a more static approach that might rely on compiler analysis, which we do not consider).

Care must be taken when implementing execution suppression, since the naive approach of simply omitting arbitrary program statements can lead to an inconsistent program state in which crashes may result due to the suppression itself. To address this problem, whenever one statement is suppressed in an execution, then all subsequent statements that would be influenced by the suppressed statement are themselves suppressed. This guarantees that all executed statements are those which do not depend on any suppressed statement. In this way, any subsequent crashes that may occur must be due to an error in the program, and not to the suppression itself.

To illustrate the functionality and usefulness of execution suppression, consider the sample code presented in Figure 1. In this snippet of code, there exists a copy-paste error at line 4 (the programmer copies lines 1 and 2, pastes into lines 3 and 4, and then 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 into 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 an infected memory location, which can lead to further infection at other memory locations (lines 9, 10, and 11). Essentially, the error at line 4 immediately causes memory corruption that propagates through multiple locations until eventually a program crash may occur (potentially at lines 12, 13, and 15).

Suppose the code in Figure 1 is exercised on some input. This is represented pictorially in Figure 2 (A). Initially, pointer $q2$ is corrupted at line 4 since it points to an incorrect memory location. That memory location is then infected at line 8, where the value previously stored at that location is mistakenly overwritten. Then, the definition at location a (line 9) is infected since it uses the infected value from location $*p2/*q2$. The definition for location b (line 10) is similarly infected. This further results in infection of location c (line 11). Now, suppose that the program crashes at line 12 due to infected array index c accessing an illegal address outside the bounds of array $intArray$. This is a buffer overflow failure. When the buffer overflow failure is observed at line 12, identifying the root cause at line 4 is not obvious since in practice we do not know the first point of memory corruption and how the corruption might propagate during infection.

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 the crash. This is depicted in Fig. 2 (B). To do this, we notice at line 12 that the value at location c is used, along with the base address for variable $intArray$, to compute the effective memory address to

```
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
```

Fig. 1. Example code to illustrate the functionality and usefulness of execution suppression.

access. Since either of these used locations could be infected, we identify the last definitions of both (though for *intArray*, this is not shown in the figure). Location c is last defined at line 11. We then re-execute the program on the same input, but during execution we suppress the definition of the base address for *intArray* (not shown in the figure) as well as the definition of c at line 11 (by *not* performing the store to location c). Accordingly, execution of any subsequent statements directly or indirectly influenced by these definitions are also suppressed. In our example, only lines 11 and 12 are suppressed when the program is re-executed. However, suppose that now the execution reaches line 13 and another crash occurs. This is possible since infected 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 at line 4, but its location is not obvious at this point.

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 Figure 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 (similarly for the last definition of the base address for *structArray*, not shown in the figure). Note that line 8 is the appropriate last definition of $*p2$, since pointer $q2$ actually refers to the same location as $p2$. In our example, during execution we therefore suppress lines 8, 9, 10, 11, 12, and 13. Note that lines 14 and 15 are not suppressed since the infected 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 $p2$

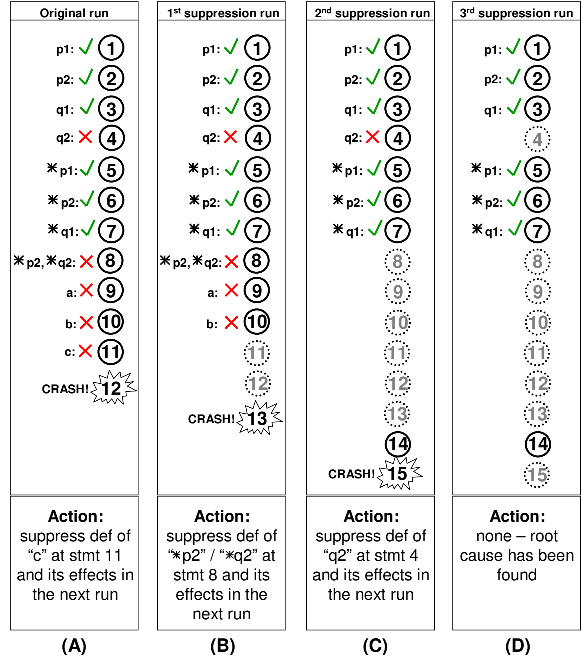

Fig. 2. The original run and 3 execution suppression runs for the code in Figure 1. Solid circles are executed statements, and dotted circles are suppressed statements. Statements defining a memory location are annotated with information showing whether the location is infected/corrupt (x) or not infected/corrupt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

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 (last defined at line 4).

Finally, we re-execute the program a third time as shown in Figure 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 all memory corruption has been suppressed during execution. In other words, only the statements involving un-infected memory locations are exercised. As a result, we conclude that the most-recently identified statement for suppression – line 4 – contains the error, because it is the root cause of all the memory corruption that led to the program crashes. Overall, our example shows how execution suppression gradually isolates the first point of memory corruption by suppressing program crashes to iteratively reveal more memory corruption. This continues until no further crashes occur when the first point of memory corruption is revealed and suppressed.

The algorithm for our execution suppression approach is shown in Figure 3. The approach requires as input a program and an associated test case for which a program crash occurs due to memory corruption. The approach iteratively searches for points of memory corruption in the execution and suppresses the effects of these points until the first point of memory corruption is found (which is assumed to be

input:
Program P and test case t causing a crash due to a memory bug.
output:
Stmt(s) identified as the first point of memory corruption in execution of t on P.
algorithm ExecutionSuppressionApproach
begin
1: S_{def} := "undefined";
2: *Initial_Suppression_Points* := {};
3: **while** a program crash C occurs during execution of t on P **do**
4: S_{crash} := the stmt instance directly causing crash C;
5: *Loc* := accessed memory location(s) causing crash C at stmt instance S_{crash};
6: S_{def} := the stmt instance(s) originally defining the value(s) in *Loc* prior to its use at S_{crash};
7: **if** S_{def} does not exist **then**
8 S_{def} := the stmt instance(s) originally defining the address(es) of *Loc* prior to its use at S_{crash};
endif
9: **add** S_{def} to set *Initial_Suppression_Points*;
10: **re-execute** t on P while suppressing (nullifying) the effects of
(1) all stmt instances in *Initial_Suppression_Points*;
(2) all stmt instances directly/indirectly influenced by some stmt instance in *Initial_Suppression_Points*
endwhile
11: **report** the program statement(s) associated with the latest S_{def};
end ExecutionSuppressionApproach

Fig. 3. Execution suppression algorithm to identify the first point of memory corruption in an execution that crashes due to a memory error.

at or near to the memory error). The approach iterates as long as a program crash occurs. It is assumed that if memory corruption exists in an execution, then a crash will occur. Thus, when no further crashes occur, the most recent point(s) of suppression is assumed to be the first point of memory corruption in the execution.

The main loop comprising our approach is shown in lines 3–10 in Figure 3. This loop iterates as long as a crash occurs. On each iteration, the corrupted/infected memory location and the associated statement instance causing the crash are identified (lines 4 and 5). In some cases, such as crashing array accesses that involve both a base address as well as an index value, there may be more than one location that could be corrupted/infected; all such locations are considered. The statement instance that defined this corrupted memory location is identified (line 6). However, an accessed memory location may have no prior definition in cases where the address of the accessed location itself is incorrect. In this case, the statement instance that defined the incorrect address is identified instead (lines 7 and 8). Again, in the case where more than one location is identified at line 5, then the same number of associated statement instances will be identified in line 6 or 8. Lines 6 through 8 essentially give preference to the possibility of an incorrect value in a memory location as opposed to the possibility of the memory location itself being incorrect. However, this approach is effective and worked well in our experiments. This is

because if the memory location itself is incorrect, then it is unlikely for there to be a prior definition to that location (so line 8 is highly likely to be executed in this case). When looking for the last definition, the *original* definition of the infected value is identified (bypassing any copies that may occur, for instance, by passing values through function calls). This is because the original definition of an infected value is the one that is ultimately responsible for the crash caused by that infected value. Once the definition statement(s) is identified, it is then added to the set of "initial suppression points", the execution points at which suppression should be initiated upon program re-execution (line 9). The key step of our approach is then performed (line 10), in which the program is re-executed using the same input. During execution, the direct and indirect effects of all statement instances in the set of initial suppression points are suppressed. The effect of this suppression is that the previously-occurring crash will be avoided, since the memory corruption directly causing it will have been suppressed. Thus, either a new crash will occur in the execution – in which case the loop iterates again – or else no crash will occur and the last-identified initial suppression point is identified as the likely first point of memory corruption in the execution (line 11). On rare occasions, more than one likely first point of memory corruption may be outputted by our approach at line 11, since lines 6 and 8 may identify more than one statement instance. In these situations, a developer may have to manually analyze a few statements to determine which one is the true first point of corruption. However, this situation did not arise in our experiments, as our approach always outputted a single identified statement in our benchmark programs.

We now illustrate our execution suppression algorithm using the example depicted in Figure 4. This figure shows a crashing execution associated with a memory error in the `pine` program. In the figure, the root cause (error), the first point of memory corruption, and the point of the crash, are highlighted. Solid arrows between statements represent propagation of incorrect values during the crashing execution: thin arrows from the point of the root cause until the first point of memory corruption represent propagation of non-address incorrect values; thick arrows from the first point of memory corruption until the crash show propagation of memory corruption. Arrows with dotted lines represent control flow. To follow the path leading up to the failure, start at the point of the root cause and follow the arrows until the point of the crash is reached.

In this example execution, the memory error (root cause of the failure) occurs in file `bldaddr.c`, line 7270. At this statement, a size value is estimated to be too small because it does not account for the possibility of special characters in an input string. This infected size value propagates through several statements until it is used at line 7126 to allocate a heap buffer. We consider this pointer variable assignment to be the first point of memory corruption because the buffer is allocated based on an incorrect size. A pointer to this buffer is then passed through several function calls until function `rfc822_cat` in file `rfc822.c` is executed. Within this function, data is written into the buffer and the buffer is overflowed. Control then eventually reaches file `bldaddr.c`, line 7134, where the pointer to the infected buffer is returned to file `mailindx.c`, line 4502. The pointer is finally passed to file `fs_unix.c`, line

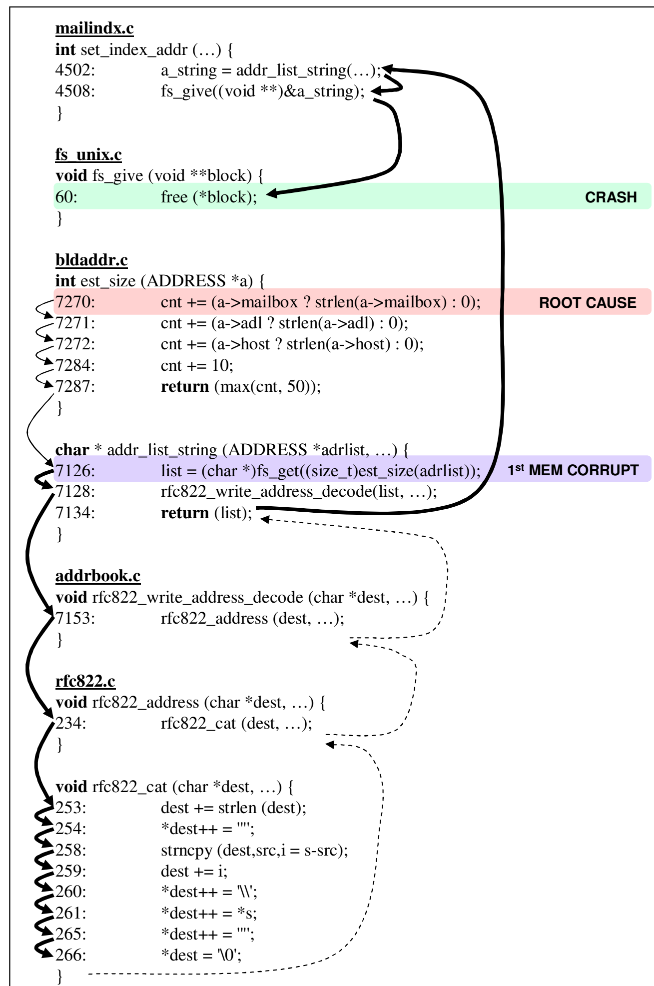

Fig. 4. Selection of statements from the `pine` program illustrating how traversal of a memory error can lead to memory corruption in an execution and ultimately trigger a crash. Thin solid arrows between statements represent propagation of incorrect non-address values. Thick solid arrows represent memory corruption propagation. Dotted arrows represent flow of control.

60, where a call to `free`() occurs that finally results in a program crash. The crash occurs because of the earlier buffer overflow corrupting an important value that is needed by the `free`() function.

It turns out that for this execution, there are 5 static dependence edges (1,103 dynamic instruction instances) from the point of error traversal until the first point of memory corruption, and an additional 14 static dependence edges (10,165 dynamic instruction instances) from the first point of memory corruption until the point of the crash. Thus, there is considerable propagation of incorrect values and corrupt memory locations in this crashing execution. Further, 19 static dependence edges and over 11,000 dynamic instruction instances separate the root cause from

the crash. This illustrates the potential for memory errors to have complicated effects on program execution, and demonstrates why memory errors can be difficult to locate.

When running our execution suppression approach on this program execution to isolate the first point of memory corruption, it is determined that the program crash in function `free`() is caused by an unexpected software abort, due to accessing a single particular memory location. The original definition of this location happens to be the definition of variable *list* at line 7126 of file `bldaddr.c`. This is actually the first point of memory corruption, since the buffer is allocated to be too small at this point. As a result, the execution suppression approach re-executes the program while suppressing this definition and all of its effects. In this case, all memory corruption is avoided (since all statements influenced by the too-small buffer are suppressed) and therefore no program crash occurs. The approach then reports the correct first point of memory corruption in this execution. Thus, even though `pine` has a large degree of memory corruption propagation in this case, our approach is able to bypass many memory corruption dependence edges when isolating the first point of memory corruption. Whether or not this is possible in general depends upon which corrupted memory locations directly cause a crash in a given execution. In the case of `pine`, we were lucky in that the first corrupted memory location directly led to the first crash, so our approach identified the ideal result very quickly. However, as we will show in our experimental study, our approach may sometimes require several iterations to identify the first point of memory corruption.

3.2 The Variable Re-ordering Component

The execution suppression approach relies on the fundamental assumption that memory corruption in an execution will cause a program crash. This assumption may not hold in cases where corrupted memory is never accessed, or in cases where it may be accessed in such a way that no crash happens to occur. Evidence that this can happen was seen in our memory corruption study (Section 2) in which we were able to create inputs in certain cases that trigger memory corruption but do not result in a crash. As a result, this exposes two limitations of execution suppression. First, the approach is not applicable to executions that do not originally crash, even though they may traverse an error and cause memory corruption. Second, the approach may terminate prematurely in cases where no crash occurs during a suppression execution even though memory corruption still exists in the execution. Premature termination would mean that the approach would identify some point of memory corruption along the path from the error to the point of the failure, but it would not be the *first* point of memory corruption. To address these flaws, we propose the idea of *variable re-ordering* to expose crashes due to memory corruption where crashes may not otherwise occur. Variable re-ordering involves altering the relative ordering of variable locations in memory, prior to using them during execution. The idea is based on the observation that memory errors often lead to unexpected reading or writing of memory locations (other variables) at execution points when those variables should not have been accessed. Depending upon which variables are unexpectedly accessed, a crash may or may not occur. For instance, overflowing a buffer may cause other program variables to be unexpectedly overwritten. As another example, writing to the location pointed to by an infected

input:
  Program execution E that does not result in any crash.
  (Optional): set of known corrupt memory locations $Corrupt$.
output:
  A variable ordering O that causes execution E to result in a crash, or else
  NULL if no such O can be found.
algorithm VariableReordering
begin
1: $V_{accessed}$:= set of global, stack, and heap variables accessed during execution E;
2: $Spaces$:= {global, stack, heap};
3: $Spaces$:= **sort** $Spaces$ so those associated with at least one address
  in $Corrupt$ (if specified) are ordered first; break ties in increasing
  order of # of associated variables in $V_{accessed}$;
4: **for each** type of memory space *space* $\in$ $Spaces$ taken in sorted order **do**
5:   $Var_Orderings$:= set of distinct variable orderings involving variables
    in $V_{accessed}$ that are associated with memory space *space*;
6:   **for each** variable ordering $O \in Var_Orderings$ **do**
7:     **if** variable ordering O applied to execution E causes E to crash
      then report O;
   endfor
  endfor
8: **report** NULL;
end VariableReordering

Fig. 5. General variable re-ordering algorithm to expose crashes in an execution that triggers memory corruption.

pointer variable may cause some other variable to be unexpectedly overwritten.

Our general approach for variable re-ordering to expose crashes is presented in Figure 5. The approach is essentially a search algorithm that tries different variable orderings to try to find one that leads to a crash. The input to our algorithm is a non-crashing execution. A second, optional input is a set of memory locations previously known to be corrupt; this information may be available if the execution suppression approach was previously run to identify a subset of the corrupted memory, prior to invoking the variable re-ordering algorithm. This optional information can be useful for prioritizing the variables to re-order, to try to expose crashes more quickly. The first step of the approach is to identify the set of all variables accessed at some point during the execution (line 1). These variables may be in global space, on the stack, or in the heap. Only the accessed variables need to be considered as candidates for re-ordering, since non-accessed variables in the execution cannot lead to any crashes. Next, the *global*, *stack*, and *heap* memory spaces to consider are ordered so that if any of these spaces involve known corrupted memory, they are ordered first; ties are broken by ordering the spaces in increasing order of the number of accessed variables associated with each space (lines 2–3). Considering the memory spaces in this order (loop in lines 4–7) increases the chances that a crash will be exposed quickly. This is because memory spaces already known to involve memory corruption might be more likely to cause crashes if the variables within these spaces are re-ordered. For the variables in each memory space, the

set of distinct variable orderings to try are found (line 5). Heuristics may need to be used here to limit the number of orderings in cases where there may be a large number of potential orderings to try. In the next paragraph, we describe how we did this for our experiments. Next, for each variable ordering, that ordering is applied to the given execution to see if a crash occurs; if so, then the particular ordering causing the crash is reported (lines 6–7). If no crashes occur after trying all variable orderings, then the value NULL is returned (line 8) to indicate that no ordering was found.

In practice, there are likely to be many possible variable orderings for a particular execution. For example, for program `xv` used as one of our experimental benchmarks, it turns out that one of the executions we analyzed involved accessing over 400 distinct global variables. To blindly try all possible permutations of these global variables in memory would take a very long time. Instead, we followed a heuristic that significantly limits the number of variable orderings to try, while still likely exposing crashes through variable re-ordering when it is possible to do so. This heuristic is based on the observation that crashes are usually caused by infection of address-related variables (either pointer variables, or variables that are used to compute addresses, such as array index variables). Moreover, these variables are most likely to become infected when they are placed immediately after buffers, because potential buffer overflows may unexpectedly overwrite these variables. Thus, our heuristic considers only those accessed variables that are either associated with buffers, or else are used to compute addresses. Further, these variables are re-ordered only to ensure that different address-related variables are placed immediately after different buffers (no need to try all possible permutations of the variables). In our experiments, this allowed us to significantly reduce the number of accessed variables to consider for re-ordering, and drastically reduced the total number of variable re-orderings that needed to be performed. Moreover, we should note that each program execution that performs variable re-ordering can account for multiple (*buffer*, *address-variable*) pairs. For example, if there are 5 distinct buffers and 5 distinct address-variables under consideration, then there are 25 (*buffer*, *address-variable*) pairs of interest; however, we actually only need a total of 5 executions to account for all of these, since 5 different (*buffer*, *address-variable*) pairs can be accounted for on each execution.

For each of the global, stack, and heap spaces, we use different techniques to re-order the associated variables. In global space, considered variables are simply rearranged in global memory prior to program execution. One way to implement this would be to modify a compiler to alter the layout of globals in memory. However, we followed a different approach in which we used the Valgrind dynamic binary translation framework [Nethercote and Seward 2007] to simulate this. We allocated our own memory space for global variables and then mapped each global variable to a corresponding (specially-ordered) location in our own memory space. Then we ensured than any subsequent accesses to global variables operated on our own allocated space. In stack space, considered local variables at the start of each function call are rearranged on the call stack by instrumenting within Valgrind to modify the order in which they are pushed onto the call stack; references to the local variables within the function call are then adjusted accordingly. For func-

```
compress42.c
void comprexx (char **fileptr) {
882:        int fdin;
883:        int fdout;
884:        char tempname[MAXPATHLEN];
886:        strcpy (tempname,*fileptr);            STACK OVERFLOW
}
```

Fig. 6. Selection of statements from the `ncompress` program to illustrate variable re-ordering.

tion calls that involve at least one accessed stack buffer, we ensure that one of the attempted variable orderings involves placing the function call return address immediately after a stack buffer (to encourage stack smashing when it is possible). Also, re-ordering of local variables may involve moving them to global space, which is semantically correct as long as a particular function is known to have at most one activation record on the call stack at any given time. Finally, the problem of variable re-ordering in heap space is more challenging because these variables are dynamically allocated and deallocated during execution. For simplicity, we chose to handle only the heap variables specially by associating a special "magic value" that is located adjacent to each heap variable. At the time of variable deallocation or execution termination, this magic value is checked to see if it has been overwritten; if so, a program abort (crash) is produced indicating which program instruction overwrote that magic value. Thus, we do not actually re-order heap variables in memory.

The heuristics proposed in this section to make variable re-ordering practical were done with our experimental benchmark programs in mind. These programs mostly involve buffer overflow errors, and so we designed our heuristics to be effective in the presence of buffer overflows. However, for other kinds of memory errors, it may turn out that the proposed heuristics may be less effective at exposing crashes. For instance, an uninitialized read may cause an arbitrary memory address to be accessed, and this memory address may not be directly associated with any address-related variables or buffers (which are the focus of our heurisitics). As a result, the proposed set of heuristics may have to be refined or altered to be effective in the presence of other kinds of memory errors besides buffer overflows. However, the basic technique of altering the layout of variables in memory is a general idea that we believe can be made practical and effective for a variety of memory errors.

Figure 6 shows a selection of statements from the `ncompress` program analyzed in our study of memory errors in Section 2. In this program, the stack buffer *tempname* in function `comprexx` is allocated with a fixed size. Thus, the `strcpy`() at line 886 can overflow this buffer if *fileptr*, which points to an input string of arbitrary length, is too long. For this program, one of the inputs analyzed in our memory error study caused the buffer to be overflowed, but resulted in no crash. This is because, on the call stack, the local integer variables *fdin* and *fdout* were positioned in memory directly after the buffer *tempname*, but before the function

call return address. In the non-crashing input, it turned out that the overflow was relatively small and only infected the values of the two local integer variables, whose infected values were not subsequently used in a way that could result in a program crash. However, through variable re-ordering, one of the considered alternative orderings is one in which the function call return address is placed immediately after the overflowed buffer on the call stack. Under this variable ordering, the same buffer overflow now corrupts the function call return address, leading to a crash.

3.3 The Complete Approach

Figure 7 shows our complete approach incorporating both the execution suppression technique and the variable re-ordering technique. Note that the required input to this approach is simply a program and corresponding input that causes memory corruption. Because of variable re-ordering, it is no longer necessary for the program input to initially result in a crash. Given a program execution involving memory corruption (line 1), the execution suppression technique is executed (line 3). Execution suppression might terminate immediately if the initial execution does not end in a crash. Otherwise, the execution suppression technique will proceed until an execution results in no crash. In the case that the identified statement is not the true first point of memory corruption, there is a chance that variable re-ordering will expose a new crash. Thus, the variable re-ordering technique is initiated to see whether any further crashes can be found to expose more memory corruption (line 4). If a variable ordering causing a crash is found, then the modified execution with appropriate variable ordering to cause the crash is identified (lines 5–6). This crashing execution is then passed once again to the execution suppression technique to resume isolating the first point of memory corruption (back at line 3). The approach iterates until finally there are no crashes in the execution suppression technique and the variable re-ordering technique cannot find any further crashes. At this point, the most recent statement identified by the execution suppression technique is reported as the likely first point of memory corruption (line 8). As described earlier for execution suppression, the output of our approach may include more than one statement in certain cases. However, this situation did not arise in our experiments.

There are several observations to make about our approach. First, the idea of execution suppression is effective because it works in the general case when memory corruption propagation occurs in a distributed fashion – with each infected memory location potentially influencing multiple other memory locations – rather than in a straight-line fashion. Even though suppressing some corruption may avoid one failure, there is a chance that any remaining corruption would still lead to subsequent failures. The approach can also be effective when multiple independent memory errors exist in a program simultaneously. On each iteration of our approach, the algorithm gets closer to identifying the first point of memory corruption for *some* memory error (the approach is not sensitive to *which* memory error). Once a first point of memory corruption is found, the associated error can be fixed and then the approach can be run again on the modified program to identify any remaining memory errors. Also, the technique of variable re-ordering is a general technique for exposing crashes that may work not only for buffer overflows, but also for other types of memory corruption as well. Other techniques – such as setting and check-

```
input:
    Program P and test case t causing memory corruption due to a memory error.
output:
    Stmt(s) identified as the first point of memory corruption in the execution
      of t on P.
algorithm IsolateFirstPointOfMemoryCorruption
begin
1:  E := the execution of test case t on program P;
2:  do
3:      Identified_Statement := run execution suppression approach using E;
4:      Variable_Ordering := run variable re-ordering approach using the most
          recent suppression execution performed in line 3 above;
5:      if (Variable_Ordering != NULL) then
6:          E := the new crashing execution using Variable_Ordering computed
              in line 4 above;
7:  while (Variable_Ordering != NULL);
8:  report Identified_Statement;
end IsolateFirstPointOfMemoryCorruption
```

Fig. 7. Complete approach to isolate the first point of memory corruption in an execution.

ing “magic values” at the boundaries of particular variables – are primarily useful for only certain kinds of memory corruption such as buffer overflows.

Finally, our approach can be used to isolate root causes for a variety of memory errors. This is because the approach views memory errors in terms of accesses to corrupted memory locations during program execution, and this trait is shared by most memory errors. For example, consider a dangling pointer to a deallocated memory location. Suppose this memory location is later re-allocated and used, but in the meantime (due to the dangling pointer), an unexpected write occurs to this memory location, causing a crash once the infected value at that location is accessed. In general, this type of error can be particularly tricky to find, especially since the offending write can be associated with a completely different type than the type associated with the access that causes the crash. However, in this situation our approach can immediately identify the last instruction performing the offending write. This is because our implementation (described in the next section) considers instructions and memory accesses at the binary level, and so the associated types of memory locations are not considered. As a result, our approach is able to quickly identify the offending write due to the dangling pointer in this case. On the other hand, our approach assumes that crashes are caused by memory corruption; our approach is not designed to handle memory errors and crashes that do not involve memory corruption. An important class of such memory errors is the *memory leak*, in which allocated memory is not deallocated when it is no longer needed. This can eventually cause a crash when the program runs out of available memory. However, memory leak errors do not generally involve memory corruption, so our approach is likely to be ineffective for locating the root causes of memory leaks.

4. IMPLEMENTATION OF EXECUTION SUPPRESSION

We now describe some details for implementing execution suppression in our approach (not including variable re-ordering). First, we present a general implementation that is described at a conceptual level. Then, we discuss a software-only implementation that makes use of the Valgrind dynamic binary translation framework [Nethercote and Seward 2007]. Next, we discuss how we can take advantage of some hardware support available in Itanium processors (originally meant for deferred exception handling) to reduce overhead. Finally, we describe a hardware-intensive implementation of execution suppression, using hardware similar to that in *dynamic information flow tracking* (DIFT) approaches [Dalton et al. 2007; Venkataramani et al. 2008], to further reduce overhead. We then compare the overheads for these different implementations. All implementations consider program executions at the binary instruction level.

4.1 General Implementation

Figure 8 shows the conceptual steps involved in any implementation of execution suppression. The first main step is to identify the instruction instance that is responsible for defining a memory location that directly causes a crash. To enable this, we maintain a global counter representing a dynamic instruction count (variable *global_cnt*). Every register and memory word is associated with a count value, which represents the instruction that last defined it. Thus, for every instruction that defines a memory word or a register, we increment *global_cnt* and store it in the counter associated with the defined location (line 1). To allow us to pinpoint the instruction instance responsible for a crash, we store the instruction counts associated with the two sources (lines 2–3). Thus, if a crash occurs at the current instruction, we can easily identify the instruction instance responsible for defining the memory location causing the crash.

Once we have identified the instruction instance directly causing the crash, we re-execute the program and begin suppression starting at the appropriate instruction instance (determined by the condition at line 4). The fact that we are in suppression mode is represented by a global *suppress* flag (the flag is set in line 5). In addition to setting this flag, we also mark the target of the current instruction as *corrupt* to indicate that it is corrupt/infected (line 6). Once the *suppress* flag is set, each executed instruction from this point follows the suppression semantics if either source location is marked as *corrupt* (line 9), otherwise it follows the regular semantics (lines 11–12). If the suppression semantics are followed due to at least one of the sources being marked *corrupt*, we simply flow the *corruption* bit by setting the *target* (defined memory location) of the instruction to be *corrupt*. If the regular semantics are followed, the instruction executes normally, and the *target* (defined memory location) of the instruction is marked as *not corrupt* since it was computed using only non-corrupt values (this step is necessary in case the target location was previously marked as *corrupt* in the execution). Note that the description in Figure 8 shows the steps for all data transfer and arithmetic instructions. In a control transfer function, we skip both branches if the predicate is marked corrupt.

```
Global variables:
 - global_cnt: the current dynamic instruction count
 - suppress_cnt1, suppress_cnt2: potential suppression points in the event of a crash
 - suppress: flag that indicates if in suppression mode, initially false
Variables associated with every register and memory word:
 - cnt: global count value associated with the instruction last defining this
     register/memory word
 - corrupt: a boolean that is true if register/memory word is corrupt, false otherwise

Case:  target = src1 op src2       // target, src1, src2  can be register or memory word

1.  target.cnt = ++global_cnt      // update target to current instruction count value
2.  suppress_cnt1 = src1.cnt       // note suppression points in case program crashes here
3.  suppress_cnt2 = src2.cnt

4.  if (global_cnt == suppress_cnt1  or global_cnt == suppression_cnt2)
5.    suppress = true              // initiate suppression mode
6.    target.corrupt = true        // mark this first corrupted location
7.  if (suppress)
8.     if (src1.corrupt or src2.corrupt)
9.        target.corrupt = true    // suppression semantics
10.    else
11.       target = src1 op src2    // regular semantics
12.       target.corrupt = false
```

Fig. 8. General implementation of execution suppression.

4.2 Software Implementation

Our approach is implemented fully in software based on the high-level system design shown in Figure 9. Our design consists of three main components: the *Valgrind Core*, the *Suppression Execution Tool*, and the *Memory Bug Root Cause Isolator*.

(*Valgrind Core*) The *Valgrind* infrastructure [Nethercote and Seward 2007] 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 an associated input, 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 execution suppression. The tracing is required to see which memory locations are accessed and when. The execution suppression is required to nullify the effects of the instructions during execution that involve infected memory, when searching for the first point of memory corruption.

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

Fig. 9. High-level system design for the software implementation of our approach. The bidirectional arrows represent interactions between the system components.

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 directly or indirectly influenced by 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 dependent upon it will be suppressed as well.

There are a few special considerations to make when suppressing. If an infected location is used in a conditional check, then we suppress the entire conditional structure involving the infected location. For example, we would suppress an entire "if/else" structure or an entire loop if the associated condition uses an infected location at some point during execution. This solution is simple and worked well in the benchmark programs we studied. In order to determine where a conditional structure ends, we relied on static analysis of the program structure at source code level, and mapped this information back to binary level as needed; Valgrind provides facilities for mapping source code statements to their associated binary instructions, as long as the instrumented program is compiled with debugging information. Special consideration must also be made for infection of the return address of a function call. This must be specially handled because we cannot suppress the function return, 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, infected input to system calls must also be handled. To do this, we simply refrain from making system calls when they involve at least one infected input value. The same approach can be used to handle library calls if desired, although we have not currently implemented this. We observe that suppressing system and/or library calls when they involve at least one infected input value is an approximation. A more precise approach would be to only suppress those instructions within the function calls that actually depend upon the infected inputs. However, for system calls in particular, we cannot use Valgrind to instrument instructions within a system call itself, since Valgrind actually executes system calls on the real CPU. Thus, our approximation of omitting system calls entirely when they involve at least one infected input, was done for implementation reasons. However, this approximation was still able to lead to good results in our experiments.

(*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 its associated input, 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 crash occurs, then another suppression point is identified and another re-execution is performed. Eventually, no crash will occur and the latest identified suppression point(s) is reported.

4.3 Using Hardware Support for Deferred Exception Handling

This implementation is motivated by the similarity of our general execution suppression approach to the hardware support provided in Itanium processors for deferred exception handling. Itanium processors allow instructions to be executed speculatively. However, speculative instructions can cause exceptions, which should be reported only when any *non-speculative* instruction uses any speculatively-produced values. To implement this, whenever a *speculative* instruction causes an exception, the target (defined register) of the instruction is tagged with a special value, known as NaT ("not a thing"). The flow of the NaT has to be propagated as the program executes. Finally, when a non-speculative instruction uses any of the values tagged as NaT, an exception must be reported. To implement this, Itanium processors associate a NaT bit with every register; the hardware automatically propagates NaT bits for register-based instructions.

We observe that the NaT bit is analogous to the *corrupt* bit in our general implementation, and the propagation semantics of the NaT bit is identical to the propagation semantics of our *corrupt* bit. Thus, we take advantage of the NaT bits and the propagation hardware for implementing execution suppression. This essentially means that lines 7–12 in our general implementation (Figure 8), are now automatically taken care of by the hardware. This results in significantly reduced overhead for the execution suppression approach. However, since there are no NaT bits associated with memory words in the Itanium processor, we still require software implementation for the propagation of *corrupt* bits for memory instructions.

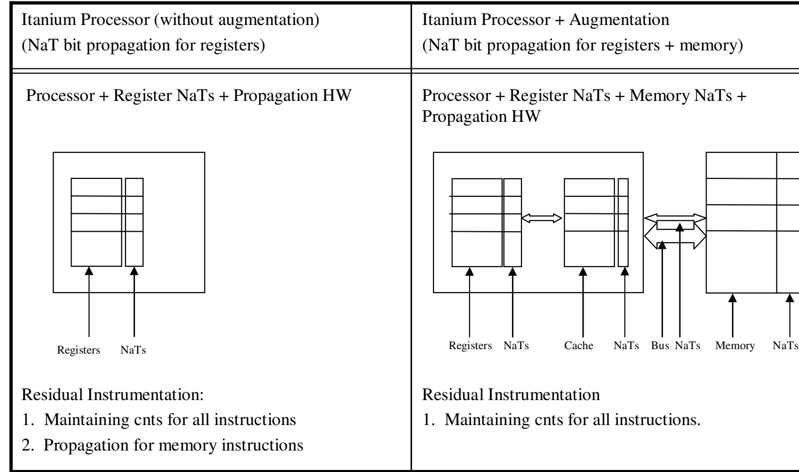

Fig. 10. Itanium processor with and without augmentation.

4.4 Memory Augmentation Support

In this approach, we augment the Itanium processor with additional support so that propagation of *NaT* bits is handled for memory instructions as well. To enable this, we associate *NaT* bits with every memory word; this results in the addition of a *NaT* bit for every memory word in main memory, as well as for every word in caches and in the external data bus. This further reduces the execution overhead. Figure 10 shows the architecture for the Itanium processor both with and without this augmentation.

4.5 Overhead Comparison

We compared the overheads of the execution suppression approach for each of our implementations. While our actual software implementation using the Valgrind infrastructure is targeted toward x86 machines, our other implementations were implemented in a simulator. We used the SESC simulator targeting the MIPS instruction set. Our simulations targeted an in-order processor with a 16-KB 4-way L1 cache and a 512-KB 2-way L2 cache, with a memory latency of 250 cycles. We implemented support for handling deferred exceptions into the simulator, in addition to supporting NaT bits for both register and memory locations. To obtain a fair comparison between all approaches, we ran the software implementation in the simulator as well (the regular Valgrind-based implementation incurs an overhead of around 50x – 100x, but this is because Valgrind is designed for ease of writing complex instrumentation tools, and not for optimizing runtime performance). We performed instrumentation by modifying `gcc-4.1` to generate the instrumentation code.

Figure 11 shows the overhead for all three approaches normalized to original execution time, for one run of each of the programs analyzed in our memory corruption study in Section 2. The right-most entry in the graph shows the average

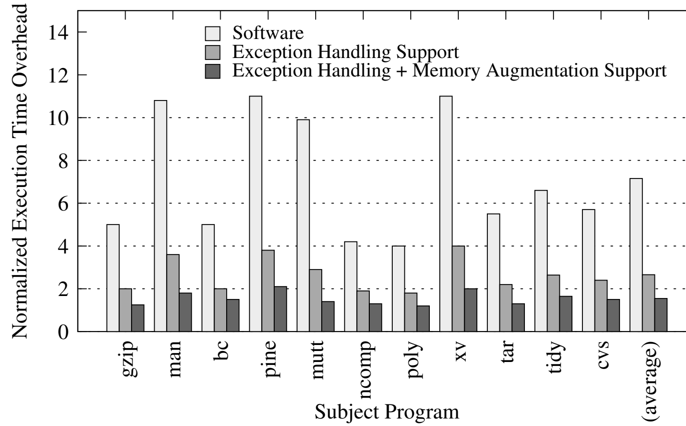

Fig. 11. Execution time overheads for different execution suppression implementations.

results. As expected, the software-only overhead is highest, with 7.2x overhead on average. Using the deferred exception hardware support available in the Itanium processor, we can significantly reduce this overhead to 2.7x on average. Finally, using the memory augmentation approach, we can further reduce this overhead to an average of 1.8x.

5. EXPERIMENTS

For our experiments, we used the same set of memory error programs and inputs described previously in Tables I and II in Section 2. First, we considered every crashing input and executed the basic execution suppression approach *without variable re-ordering*. The results are shown in Table III. For each crashing input, the table shows the total number of program executions required by the execution suppression approach to isolate the first point of memory corruption ("# Exec. Req."). Also, the statement identified by the approach is reported ("Identified Statement"), along with the dependence distances from the identified statement to the first point of memory corruption ("1st Corrupt"), and from the identified statement to the actual memory error ("Error"). For example, in the crashing execution for program `man`, a total of 2 program executions are required by the execution suppression approach: the first is the original crashing execution, and the second is a suppression re-execution that resulted in no further crashes. A statement was identified that happened to be 1 dependence edge away from the first point of memory corruption, and 2 edges away from the error. Although the first point of memory corruption was missed in this case, the execution suppression approach was able to precisely identify the first point of memory corruption in the executions for all other inputs (even for programs `ncompress` and `polymorph` where crashes occurred at two different program points). As was observed previously in our memory corruption study and as is shown in Table III, these identified statements were either at, or relatively close to, the memory errors.

Table III. **Suppression-only results**: experimental results when running our approach with only execution suppression (no variable re-ordering), using different crashing inputs on our subject programs.

Program Name	Input Type	# Exec. Req.	Identified Statement	Dep. Distance To... 1st Corrupt	Dep. Distance To... Error
gzip-1.2.4	Crash Point 1	2	gzip.c: 828	0	0
man-1.5h1	Crash Point 1	2	manfile.c: 243	1	2
bc-1.06	Crash Point 1	2	storage.c: 177	0	1
pine-4.44	Crash Point 1	2	bldaddr.c: 7126	0	5
mutt-1.4.2.1	Crash Point 1	3	utf7.c: 192	0	1
ncompress-4.2.4	Crash Point 1	2	compress42.c: 886	0	0
	Crash Point 2	4	compress42.c: 886	0	0
polymorph-0.4.0	Crash Point 1	2	polymorph.c: 198	0	1
	Crash Point 2	3	polymorph.c: 193	0	1
xv-3.10a	Crash Point 1	4	xvbmp.c: 167	0	2
tar-1.13.25	Crash Point 1	2	incremen.c: 180	0	0
tidy-34132	Crash Point 1	2	parser.c: 854	0	0
cvs-1.11.4	Crash Point 1	2	server.c: 992	0	0

Program `man` was the only case in which our execution suppression approach was not able to precisely identify the first point of memory corruption. In this case, the approach terminated prematurely because no crash occurred even though memory corruption was still present in the execution. As a result, we tried using our complete approach that includes variable re-ordering to see if the results for `man` could be improved. Indeed, as is shown in Table IV, variable re-ordering allows us to precisely identify the first point of memory corruption in the execution for `man`. This is because variable re-ordering exposes one additional crash in the execution that was not originally observed when running the suppression-only approach. In this case, forcing an array index variable to be located in memory directly after an overflowed buffer causes a new crash to occur, revealing some additional corrupted memory that allows the approach to precisely find the first point of memory corruption.

Another important feature of variable re-ordering is that it enables our execution suppression approach to be applicable to other inputs that trigger memory corruption but do not initially result in a program crash. As a result, Table IV also shows the results of running our complete approach using the seven non-crashing inputs analyzed earlier in our memory corruption study in Section 2. Without variable re-ordering, we would not have been able to apply our execution suppression approach to these non-crashing inputs.

The format of Table IV is the same as Table III, except the column "# Exec. Req." has been replaced by two columns: "# Crash Exposed", indicating how many additional crashes were found through the use of variable re-ordering before our approach terminated (1 in all cases except for `gzip`); and "# Var. Order Ex.", indicating the maximum number of variable re-ordering *executions* required in order to find the exposed crash (in the case of `gzip`, the number of re-ordering executions in order to discover that no crash could be exposed). In the figure, the number of required variable re-ordering executions is not listed for programs `bc`, `pine`, and `mutt`. This is because these three benchmarks involve heap-buffer overflows, and as was described previously in section 3.2, we handle heap buffers differently and do not perform variable re-ordering with heap variables.

It turns out that for all of the non-crashing inputs except `gzip`, our approach was able to find a particular variable re-ordering that exposed a crash in the execution.

Table IV. **Complete approach (execution suppression with variable re-ordering) results**: experimental results when running our complete approach, using different inputs on our subject programs.

Program Name	Input Type	# Crash Exposed	# Var. Order Ex.	Identified Statement	Dep. Distance To...	
					1st Corrupt	Error
gzip-1.2.4	No Crash	0	15	—	—	—
man-1.5h1	Crash Pt. 1	1	18	man.c: 977	0	1
bc-1.06	No Crash	1	—	storage.c: 177	0	1
pine-4.44	No Crash	1	—	bldaddr.c: 7126	0	5
mutt-1.4.2.1	No Crash	1	—	utf7.c: 192	0	1
ncompress-4.2.4	No Crash	1	5	compress42.c:886	0	0
polymorph-0.4.0	No Crash	1	6	polymorph.c:198	0	1
xv-3.10a	No Crash	1	135	xvbmp.c: 168	0	2

In all of these cases, this made the execution suppression approach applicable, which in turn resulted in the first point of memory corruption being precisely identified without the need to expose any more crashes through variable re-ordering. For `gzip`, the initial input did not crash because a global buffer was overflowed by 1 position, erroneously writing the value NULL into another global variable that happened to already have value NULL. However, through manual inspection we discovered that none of the other global variables were accessed in the rest of the execution in such a way that they could have resulted in a crash if they had been infected. As a result, it would not have been possible for our variable re-ordering technique to force a crash to happen in this particular case.

In order to expose a crash through variable re-ordering, the maximum number of program executions required to achieve this for each benchmark ranged from 5 executions for program `ncompress`, to 135 executions for program `xv`. The reason `xv` requires so many distinct program executions in this case, is because the particular execution under consideration happens to access 202 different global buffers and 67 different address-related variables (much higher than in all the other benchmarks). It therefore takes quite a few executions in this case to group these variables in different ways to try to expose crashes.

Across all of our benchmark programs, the actual time required to execute each program given our suppression and variable re-ordering implementations never took more than a few seconds per execution. When considering the number of executions required for variable re-ordering, this translated to total runtime ranging from several seconds (for `ncompress` and `polymorph`), to several minutes (`xv` was the worst case that required about 5 minutes to run). We believe this timing is reasonable in a debugging context. However, it is clear that the scalability of our approach on larger benchmarks will be determined by the variable re-ordering component. This is because variable re-ordering can potentially require many more program executions to try to expose new crashes, as compared to the number of suppression executions. It remains to be seen how variable re-ordering can scale to programs with larger sets of accessed variables and buffers. The set of heuristics used in the current work to make variable re-ordering feasible may need to be refined to make the technique more practical and useful when more variables and buffers are present.

6. RELATED WORK

Locating Memory Errors. We first proposed the idea of execution suppression for locating memory errors in [Jeffrey et al. 2008b]. In that work, the basic execution suppression approach was described, and some experimental results were given that highlighted the effectiveness of execution suppression in locating memory errors that were associated with a set of crashing program executions. The current work extends that paper and is much more comprehensive. In the current work, we consider a superset of the subject programs analyzed in the prior work, including a few new programs with even more complicated memory corruption propagation patterns. We studied the memory errors in our subject programs in detail, and gained some valuable insights into the nature of memory errors and memory corruption propagation. These observations motivated not only our original execution suppression idea, but also the new variable re-ordering idea that significantly improves the effectiveness and applicability of execution suppression. In the current work we also describe and compare different software and hardware implementations of execution suppression.

The execution suppression approach attempts to isolate only the first point of memory corruption in a crashing execution, which is likely to be very close to the memory error. Other approaches for locating code involving memory errors can potentially report a large set of program statements that must be examined by hand until the error is found. For example, *static slicing* [Weiser 1984] identifies a subset of program statements that *may* influence the value of a variable at a program location. *Dynamic slicing* [Agrawal and Horgan 1990; Korel and Laski 1988; Zhang et al. 2006b] finds the statements that actually *do* influence a variable value in a particular execution. The related concept of *Relevant Slicing* [Agrawal et al. 1993; Gyimothy et al. 1999] has also been studied. In general, program slicing identifies a set of statements that can potentially represent many chains of dependencies in a program, from which it may take considerable time to find the error.

Valgrind [Nethercote and Seward 2007] and *Purify* [Hastings and Joyce 1992] can be used to detect memory errors, but are restrictive in that they look for particular kinds of memory errors. In one sense, our approach is more general because it can be used to locate any errors involving corrupted memory. On the other hand, *Valgrind* and *Purify* can detect some errors that may not lead to crashes, such as memory leaks. *CCured* [Necula et al. 2002] is an approach for verifying type-safety of pointers both statically and during runtime, which can be used to find potential memory errors. However, their approach requires modifications to program source code, which is not required by our approach.

There has been a variety of work on techniques for detecting buffer overflows. For example, *Write Integrity Testing* (WIT) [Akritidis et al. 2008] uses a combination of static analysis and runtime instrumentation to ensure that instructions do not write to unintended storage locations, and control does not transfer to unintended targets; the average space and runtime overhead of their approach is around 10%. Ruwase and Lam's *CRED* tool [Ruwase and Lam 2004] performs bounds-checking in order to detect buffer overflow attacks, incurring an overhead of 1% to 130%. While these techniques incur relatively low overhead, the main difference compared

to our approach is that the bounds-checking approaches are designed specifically to capture out-of-bounds memory accesses. On the other hand, buffer overflows are only one type of memory error that can be located using execution suppression. Other errors that may not involve out-of-bounds memory accesses, such as double frees and uninitialized reads, can also be located by our approach.

Tolerating Memory Errors. Although our current work focuses on locating memory errors, there has been significant research on tolerating the effects of memory errors. These techniques can be useful especially for deployed software, in which it is often vital to avoid program failures. The Ph.D. dissertation of Michael Bond [Bond 2008] describes two techniques for tolerating memory leak errors. The first technique, *Melt* [Bond and McKinley 2008], identifies stale objects that a program is not accessing, stores these stale objects to disk, and activates these objects only if a program subsequently accesses them. The second technique, *leak pruning* [Bond and McKinley 2009], predicts leaked objects based on data structure usage patterns, and then reclaims these objects at runtime; an error is thrown if any reclaimed object is later accessed. Other recent work on detecting memory leaks has resulted in the development of *Hound* [Novark et al. 2009], a runtime system that helps identify the sources of memory leaks and bloat in `C` and `C++` applications, that results in no false positives.

There has been recent work on protecting against heap-based memory errors to improve program reliability. *DieHard* [Berger and Zorn 2006] provides memory safety with high probability by randomizing the location of objects in a large heap and by replicating execution. The goal of DieHard is the opposite of variable re-ordering: whereas DieHard spaces out heap blocks to *reduce* the likelihood of crashes, variable re-ordering changes the layout of variables in memory in order to *expose* new crashes. *Archipelago* [Lvin et al. 2008] allocates heap objects far apart in virtual address space to combat buffer overflows, and protects against dangling pointer errors by preserving freed objects after they are freed. *Exterminator* [Novark et al. 2007] pinpoints heap-based memory errors and derives runtime patches to avoid them in the current and subsequent executions. Unlike these approaches that are targeted toward heap-based memory errors, our current work targets a more general class of memory errors that involve corrupted memory, which may involve memory other than the heap. Also, our work seeks to locate errors after they cause a failure, rather than to tolerate their effects during runtime.

While our current work attempts to isolate memory corruption in an execution, the *Samurai* system [Pattabiraman et al. 2008] provides safeguards against corruption of critical data through a memory model called *critical memory.* Their system uses replication and forward error correction to ensure that non-critical updates do not corrupt critical data. However, the system requires that critical memory be explicitly identified by a programmer.

The idea of *data diversity* has been proposed [Ammann and Knight 1988] to achieve fault tolerance. Data diversity observes that when a program fails due to certain input, in some cases the program failure can be avoided if the program input is changed in some minor way (according to the program specification). Further, if multiple copies of the same program are executed in parallel on slightly-different input sets, then a voting scheme can be applied to select the program output that

is most likely to be correct. The variable re-ordering component of our approach is related to the concept of data diversity, because re-ordering involves making minor changes to the relative ordering of variables in memory, prior to executing the program. However, variable re-ordering only changes the layout of variables in memory, not the actual input values to the program.

Tallam et. al. [Tallam et al. 2008] described a technique for avoiding program failures that uses the concept of safe execution perturbations. The key idea is to use a checkpointing/logging mechanism to capture a program execution into an event log. When a failure occurs, the system searches for a perturbation of the execution that will avoid the failure, by altering the event log and then replying the execution. If a failure-avoiding perturbation is found, it is recorded as a dynamic patch that can be applied to future executions to prevent a repeat of the failure.

General Fault Localization. A variety of other approaches for locating errors in software have been proposed that are not specifically designed to handle memory errors. These approaches do not explicitly account for the propagation of corrupted memory that can result due to memory errors.

Predicate Switching [Zhang et al. 2006a] 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 error based upon the critical predicate. A critical predicate may be used to identify a subset of statements that might be likely to contain the error, but like for slicing, this set of statements may not uniquely identify the error. Our approach seeks to identify a single statement that is highly likely to be at, or near to, the error.

In *Delta Debugging*, failure-inducing input is identified [Zeller and Hildebrandt 2002] that allows for the computation of cause-effect chains for failures [Zeller 2002], which can in turn be linked to faulty code [Cleve and Zeller 2005]. This approach involves substituting state (the values of variables) between passing and failing runs. A related *Value Replacement* idea was proposed [Jeffrey et al. 2008a] that attempts to replace the values used at certain statement instances with alternative 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 [Renieris and Reiss 2003] compares the spectra for two similar executions (one successful and one failing) to identify the most suspicious parts of a program. *Tarantula* [Jones et al. 2002] 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 identify likely faulty statements in a program. Our approach, in contrast, considers only a single failing execution and tries to isolate the first point of memory corruption by repeatedly revealing and suppressing additional memory corruption in the execution.

Exposing errors. Our variable re-ordering idea can be viewed as a technique for exposing errors in software (with a specific focus on causing a program to crash). The problem of exposing software errors has been extensively studied. For example, *Eclat* [Pacheco and Ernst 2005] infers an operational model of correct program

behavior and identifies inputs that violate this model. *ESC/Java* [Flanagan et al. 2002] identifies certain programming errors at compile-time using an annotation language. *Check 'n' Crash* [Csallner and Smaragdakis 2005] derives error conditions statically and then attempts to generate test cases to dynamically verify the existence of errors. *Daikon* [Ernst et al. 2001] and *DIDUCE* [Hangal and Lam 2002] automatically extract program invariants and monitor for violations during execution. Other error-exposing approaches are designed for specific kinds of errors. *CP-Miner* [Li et al. 2006] searches for copy-paste errors in large-scale software, and *EXPLODE* [Yang et al. 2006] identifies data integrity errors in storage systems.

Hardware Support for Debugging. There has been some work on providing hardware support for techniques to aid in debugging memory errors. *AccMon* [Zhou et al. 2004] describes hardware support for an invariant-based approach that identifies program instructions that typically access different memory locations. *HeapMon* [Shetty et al. 2006] takes advantage of extra cores to improve the efficiency of error monitoring for heap memory errors. *PathExpander* [Lu et al. 2006] provides support to increase the path coverage of dynamic error-detection tools by executing non-taken paths in a sandbox environment. This allows for error detection in paths that would have otherwise not been analyzed. *FlexiTaint* [Venkataramani et al. 2008] and *Raksha* [Dalton et al. 2007] are recent works describing hardware support for *dynamic information flow tracking* (DIFT). In our work, we also evaluate hardware support for the execution suppression approach, which uses ideas similar to DIFT. Recent work [Chen et al. 2008] describes how deferred exception handling in the Itanium processor can be used to perform DIFT efficiently.

7. CONCLUSIONS

This paper presented an automated approach for assisting developers in locating memory errors in software. The approach iteratively identifies and suppresses the effects of known corrupted memory locations in a crashing execution, until the first point of memory corruption in the execution is identified. This point is likely to be at, or near to, the error. We also showed how the idea of variable re-ordering can be used to expose crashes due to memory corruption in cases where crashes may not otherwise occur. By combining the ideas of execution suppression and variable re-ordering, our approach is both general and highly effective.

To motivate the development of our approach, we presented a detailed study of 11 real-world benchmark programs containing known memory errors that involve varying degrees of memory corruption propagation. We conducted an experimental analysis of our approach using the 11 benchmark programs. In all cases, our approach was able to precisely identify the first point of memory corruption in an execution, and this point was always either at, or very close to, the memory error. Finally, we discussed software and hardware implementation issues for execution suppression (not including variable re-ordering), and we compared the overheads of several different implementations. We were able to reduce the average overhead of a software-only implementation from 7.2x to 1.8x by adding hardware support.

REFERENCES

AGRAWAL, H. AND HORGAN, J. R. 1990. Dynamic program slicing. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 246–256.

AGRAWAL, H., HORGAN, J. R., KRAUSER, E. W., AND LONDON, S. 1993. Incremental regression testing. *IEEE International Conference on Software Maintenance*, 348–357.

AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. 2008. Preventing memory error exploits with WIT. *2008 IEEE Symposium on Security and Privacy*, 263–277.

AMMANN, P. AND KNIGHT, J. 1988. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers 37,* 4 (April), 418–425.

BERGER, E. D. AND ZORN, B. G. 2006. DieHard: Probabilistic memory safety for unsafe languages. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 158–168.

BOND, M. 2008. Diagnosing and tolerating bugs in deployed systems. *PhD Thesis, The University of Texas at Austin*.

BOND, M. AND MCKINLEY, K. 2008. Tolerating memory leaks. *23rd Annual International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 109–126.

BOND, M. AND MCKINLEY, K. 2009. Leak pruning. *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 277–288.

CHEN, H., WU, X., YUAN, L., ZANG, B., YEW, P., AND CHONG, F. T. 2008. From speculation to security: Practical and efficient information flow tracking using speculative hardware. *International Symposium on Computer Architecture*.

CLEVE, H. AND ZELLER, A. 2005. Locating causes of program failures. *27th International Conference on Software Engineering*, 342–351.

CSALLNER, C. AND SMARAGDAKIS, Y. 2005. Check 'n' Crash: Combining static checking and testing. *Proc. 27th ACM/IEEE International Conference on Software Engineering*, 422–431.

DALTON, M., KANNAN, H., AND KOZYRAKIS, C. 2007. Raksha: a flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. 482–493.

ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering 27,* 2 (February), 99–123.

FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for java. *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 234–245.

GYIMOTHY, T., BESZEDES, A., AND FORGACS, I. 1999. An efficient relevant slicing method for debugging. *ACM/SIGSOFT Foundations of Software Engineering*, 303–321.

HANGAL, S. AND LAM, M. S. 2002. Tracking down software bugs using automatic anomaly detection. *Proc. of the 24th International Conf. on Software Engineering*, 291–301.

HASTINGS, R. AND JOYCE, B. 1992. Purify: Fast detection of memory leaks and access errors. *Proceedings of the USENIX Winter Technical Conference*, 125–136.

JEFFREY, D., GUPTA, N., AND GUPTA, R. 2008a. Fault localization using value replacement. *International Symposium on Software Testing and Analysis*, 167–178.

JEFFREY, D., GUPTA, N., AND GUPTA, R. 2008b. Identifying the root causes of memory bugs using corrupted memory location suppression. *IEEE International Conference on Software Maintenance*, 356–365.

JONES, J. A., HARROLD, M. J., AND STASKO, J. 2002. Visualization of test information to assist fault localization. *Proc. of the 24th International Conf. on Software Engineering*, 467–477.

KOREL, B. AND LASKI, J. 1988. Dynamic program slicing. *Information Processing Letters 29,* 3 (October), 155–163.

LI, Z., LU, S., MYAGMAR, S., AND ZHOU, Y. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. on Software Engineering 32,* 3 (March), 176–192.

LU, S., LI, Z., QIN, F., TAN, L., ZHOU, P., AND ZHOU, Y. 2005. BugBench: Benchmarks for evaluating bug detection tools. *Workshop on the Evaluation of Software Defect Detection Tools Co-located with PLDI*.

LU, S., ZHOU, P., LIU, W., ZHOU, Y., AND TORRELLAS, J. 2006. PathExpander: Architectural support for increasing the path coverage of dynamic bug detection. *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 38–52.

LVIN, V. B., NOVARK, G., BERGER, E. D., AND ZORN, B. G. 2008. Archipelago: Trading address space for reliability and security. *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 115–124.

NARAYANASAMY, S., POKAM, G., AND CALDER, B. 2005. BugNet: Continuously recording program execution for deterministic replay debugging. *Proceedings of the 32nd annual international symposium on Computer Architecture*, 284–295.

NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2002. CCured: type-safe retrofitting of legacy code. *Symposium on Principles of Programming Languages*, 128–139.

NETHERCOTE, N. AND SEWARD, J. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 89–100.

NOVARK, G., BERGER, E. D., AND ZORN, B. G. 2007. Exterminator: Automatically correcting memory errors with high probability. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1–11.

NOVARK, G., BERGER, E. D., AND ZORN, B. G. 2009. Efficiently and precisely locating memory leaks and bloat. *Conference on Programming Language Design and Implementation*, 397–407.

PACHECO, C. AND ERNST, M. D. 2005. Eclat: Automatic generation and classification of test inputs. *Object-Oriented Programming, 19th European Conference*, 504–527.

PATTABIRAMAN, K., GROVER, V., AND ZORN, B. 2008. Samurai: Protecting critical data in unsafe languages. *EuroSys '08*, 219–232.

RENIERIS, M. AND REISS, S. 2003. Fault localization with nearest neighbor queries. *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, 30–39.

RUWASE, O. AND LAM, M. 2004. A practical dynamic buffer overflow detector. *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 159–169.

SHETTY, R., KHARBUTLI, M., SOLIHIN, Y., AND PRVULOVIC, M. 2006. HeapMon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM Journal of Research and Development 50,* 2/3 (March), 261–275.

TALLAM, S., TIAN, C., GUPTA, R., AND ZHANG, X. 2008. Avoiding program failures through safe execution perturbations. *32nd Annual IEEE International Computer Software and Applications Conference*, 152–159.

VENKATARAMANI, G., DOUDALIS, I., SOLIHIN, Y., AND PRVULOVIC, M. 2008. FlexiTaint: A programmable accelerator for dynamic taint propagation. *Proceedings of the 14th IEEE International Symposium on High-Performance Computer Architecture*.

WEISER, M. 1984. Program slicing. *IEEE Trans. on Software Engineering 10,* 4 (July), 352–357.

YANG, J., SAR, C., AND ENGLER, D. R. 2006. EXPLODE: A lightweight, general system for finding serious storage system errors. *7th Symposium on Operating Systems Design and Implementation*, 131–146.

ZELLER, A. 2002. Isolating cause-effect chains from computer programs. *10th International Symposium on the Foundations of Software Engineering*, 1–10.

ZELLER, A. AND HILDEBRANDT, R. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28,* 2 (February), 183–200.

ZHANG, X., GUPTA, N., AND GUPTA, R. 2006a. Locating faults through automated predicate switching. *Proc. of the 28th International Conference on Software Engineering*, 272–281.

ZHANG, X., GUPTA, N., AND GUPTA, R. 2006b. Pruning dynamic slices with confidence. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 169–180.

ZHOU, P., LIU, W., FEI, L., LU, S., QIN, F., ZHOU, Y., MIDKIFF, S. P., AND TORRELLAS, J. 2004. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. *37th Annual International Symposium on Microarchitecture*, 269–280.

Received MONTH YEAR; revised MONTH YEAR; accepted MONTH YEAR