Empir Software Eng (2007) 12:143–160
DOI 10.1007/s10664-006-9007-3

A study of effectiveness of dynamic slicing in locating real faults

Xiangyu Zhang · Neelam Gupta · Rajiv Gupta

Published online: 29 July 2006

Editor: Guiliano Antoniol

Abstract Dynamic slicing algorithms have been considered to aid in debugging for many years. However, as far as we know, no detailed studies on evaluating the benefits of using dynamic slicing for locating real faults present in programs have been carried out. In this paper we study the effectiveness of fault location using dynamic slicing for a set of real bugs reported in some widely used software programs. Our results show that of the 19 faults studied, 12 faults were captured by data slices, 7 required the use of full slices, and none of them required the use of relevant slices. Moreover, it was observed that dynamic slicing considerably reduced the subset of program statements that needed to be examined to locate faulty statements. Interestingly, we observed that all of the memory bugs in the faulty versions were captured by data slices. The dynamic slices that captured faulty code included 0.45 to 63.18% of statements that were executed at least once.

Keywords Dynamic program slicing · Fault location · Data slicing · Full slicing · Exploring slices

1 Introduction

The concept of program slicing was first introduced by Weiser (1982). He introduced program slicing as a debugging aid and gave the first *static slicing* algorithm. The static slice of a reference to a variable at a program point is the set of program statements that *can influence* the value of the variable at the given program point under some execution. Therefore every

X. Zhang · N. Gupta · R. Gupta (✉)
Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA
e-mail: gupta@cs.arizona.edu

X. Zhang
e-mail: xyzhang@cs.arizona.edu

N. Gupta
e-mail: ngupta@cs.arizona.edu

statement in the program from which there is a chain of static data and/or control dependences leading to the variable reference belongs to the static slice of the variable reference. Let us consider a program containing faulty statements. Given a program point at which the value of a variable is output, if the static slice of this variable reference contains one or more faulty statements then under some executions incorrect results may be produced at the output statement. By studying the static slice of the output, a programmer may be able to detect the faulty statements. However, since static slices can be large due to the use of conservative dependence analysis, effort required to locate the faulty statement is expected to be large.

To improve the effectiveness of slicing in program debugging, Korel and Laski proposed the idea of *dynamic slicing* (Korel and Laski, 1988). The dynamic slice of a variable at an execution point includes all those executed statements which *actually effected* the value of the variable at that point during a specific execution. In other words, a statement belongs to the dynamic slice of a variable reference at an execution point if there is a chain of *dynamic* data and/or control dependences from the statement to the variable reference. By studying the dynamic slice of a variable we are in a better position to determine the actual cause which led to the variable having an erroneous value under the specific execution being debugged. Since dynamic slices contain a significantly smaller subset of statements belonging to corresponding static slices, they are more suitable for debugging. Results of our study of dynamic slices reported in Zhang and Gupta (2004) show that the number of distinct statements executed at least once during a program execution were 2.46 to 56.08 times more than the number of statements in a dynamic slice. However, the above results were based on computing large number of slices for correct versions of programs. In this paper we will study the effectiveness of dynamic slices for locating faults in program versions with real bugs.

While in the above discussion we have considered dynamic slices that are computed by considering both data and control dependences, previous works have considered three variants of dynamic slices. Different dynamic slicing algorithms use different notions of what they consider as *influencing* the value of a variable for a given program execution. These three variants include:

- *Data slicing.* Statements that directly or indirectly influence the computation of faulty output value through chains of *dynamic data dependences* are included in data slices (Zhang et al., 2003).
- *Full slicing.* Statements that directly or indirectly influence the computation of faulty output value through chains of *dynamic data and/or control dependences* are included in full slices (Korel and Laski, 1988).
- *Relevant slicing.* While relevant slices also consider data and control dependences, in addition, they include predicates that actually did not affect the output but could have affected it had they been evaluated differently, direct data dependences of these predicates, and chains of dynamic data and control dependences of these direct data dependences (Agrawal et al., 1993b; Gyimothy et al., 1999).

While dynamic slicing has long been considered useful for debugging (Agrawal and Horgan, 1990; Korel and Laski, 1988; Agrawal et al., 1993a), experimental studies evaluating the effectiveness of slicing have not been carried out. The main goal of this paper is to experimentally evaluate the three dynamic slicing algorithms. The effectiveness of a given slicing algorithm in fault location is determined by two factors: *How often is the faulty statement present in the slice?* and *How big is the slice, i.e. how many statements are included in the slice?* We present a comparative evaluation of data and full dynamic slicing

algorithms with respect to these two criteria. The following relationship holds among the various slices: Static Slice ⊇ Relevant Slice ⊇ Full Slice ⊇ Data Slice. For the class of errors being considered, although, the faulty statement that causes an erroneous output to be produced is guaranteed to be present in the static slice and the relevant slice of the erroneous output, it may or may not be captured by the data slice and full slice.

We carried out experiments with a set of *real* faulty program versions of some widely used programs. The key results of our experimental study are as follows:

- *Applicability*. Our results show that dynamic slicing was found to be applicable in all faults studied. For 15 faults, the dynamic slice considered was the backward dynamic slice of an erroneous output. For four faults the program did not produce any output. In these cases we were able to capture the faults in the forward dynamic slice of the minimal failure inducing input (Hildebrandt and Zeller, 2000). In our study dynamic slicing we found that 12 faults were captured by data slices and 7 faults required the use of full slices. Interestingly, we observed that all of the *memory bugs* in the faulty versions which cause programs to crash due to segmentation fault were captured by the *data slices*.
- *Effectiveness*. It was observed that dynamic slicing considerably reduced the subset of program statements that needed to be examined to locate faulty statements. The dynamic slices that captured faulty code included 0.45 to 63.18% of statements that were executed at least once. These statements represented only a fraction of the total code (0.04 to 8.52%) in the programs.
- *Relevant Slicing*. Although in general faulty code may not be captured by full slices and use of relevant slicing may be required, we observed that for this set of real bugs we did not require the use of relevant slicing.
- *Exploring Slices*. Having computed the fault candidate set in form of a dynamic slice, this is next presented to the programmer who must examine it to locate faulty statements. We found that if the programmer examines statements starting from the statement that computed the erroneous value going backwards in the order of their appearance, 1.35 to 78.12% of the statements in the dynamic slices was examined before the faulty statements were located.

The rest of the paper is organized as follows. In Section 2 we give describe data and full dynamic slicing. In Section 3 we give the overview of our slicing tool which can be easily adapted to compute different types of slices. Section 4 presents the results of our experiments. Related work is presented in Section 5 and the conclusions are given in Section 6.

2 Dynamic Slicing Algorithms

In this section we illustrate the strengths and weaknesses of two dynamic slicing algorithms considered in this paper using examples. We also briefly describe the dynamic information that must be captured in order to compute the dynamic slices.

2.1 Data Slicing

Let us consider the execution of the program on an input that reveals the fault by producing an erroneous output value. Further let us assume that the presence of the faulty statement does not alter the execution control flow, i.e. the set of statements executed for this input are

the same whether or not the fault is present. Under these conditions, the erroneous output must have been produced by a fault in the form of a computational error in one of statements whose computed value is related to the output value through a chain of dynamic data dependences. The data slice of the erroneous output value includes all statements that are visited by starting from the output value and then taking the transitive closure over dynamic data dependences. Thus, in the above situation, the faulty statement will be present in the data slice of the erroneous output. Because a dynamic data slice can be small and easy to understand, the faulty statement is easier to locate by examining the data slice.

The example in Fig. 1 illustrates data slicing. The program on the left hand side of the figure is a faulty version of the program in which statement 13 contains an error (as indicated in the figure, $z = x - y + 1$ should be replaced by $z = x - y$). For a test input the correct and erroneous output values are shown in the figure. As we can see, this error does not alter the control flow up to the point the program generates an erroneous output value. The computation of the data slice of the erroneous output value of z yields the set of statements $\{5, 6, 13, 14\}$. Apart from the read and output statements, we have statement 13 in this data slice which is the faulty statement.

The computation of a data slice requires the identification of *dynamic data dependences* at runtime. In presence of arrays and pointers we must maintain relevant information to detect dynamic data dependences. An execution of a statement at runtime is uniquely identified by the identity of statement and the execution instance of that statement (because a statement may be executed multiple times at runtime). A dynamic data dependence exists

1. read (a);	
2. read (n);	
3. i=0;	$(Faulty\ Statement)$
4. while (i<n) {	13. $z = x - y + 1$
5. read (x);	should be 13. $z = x - y$
6. read (y);	
7. a=a/x;	
8. b=x;	
9. if (a>1)	Input: $a = 2;\ n = 1;\ x = -1;\ y = 1;$
10. b=a-4;	Erroneous output: $z = -1;$
11. if (b>0)	Correct output: $z = -2;$
12. z=x+y;	
else	
13. z=x-y+1;	
14. output (z);	$Data\ Slice = \{5, 6, 13, 14\}$
15. i=i+1;	
}	

Fig. 1 Example of data slice

from an execution instance of a statement that defines a value and an execution instance of a statement that later uses that value. For each address, we must remember the execution instance of the statement that last wrote to that address. Later when the value is used by an execution instance of a statement, we can establish the dynamic data dependence between the relevant execution instances of two statements.

2.2 Full Slicing

Let us consider another example in which the faulty statement is not captured by the data slice but it is captured by the full slice. Figure 2 shows a faulty program where there is a mistake in statement 10 as shown. When this faulty program is executed on the given input, incorrect output value is produced at statement 14. The program outputs the value 4 at 14 while the correct output value is 0. The faulty statement 10 is not in set $\{5, 6, 12, 14\}$ which is the data slice of z at 14. This is because the fault does not affect value of z at 14 through a chain of dynamic data dependences. Instead fault in statement 10 affects the outcome of predicate at 11 changing the direction of the branch and thus causing statement 12 to be executed instead of statement 13. The value of z thus computed is altered. The data slice of z at 14 contains statement 12 which is executed by mistake but it does not contain the faulty statement 10.

Full slices correctly handle the above situation by considering control dependences. A statement s is true (false) control dependent upon a predicate p if and only if p's true (false) outcome determines whether s will be executed. Full slices are computed by taking the transitive closure over both dynamic data and control dependence edges starting from the

1. read (a);
2. read (n);
3. i=0;
4. while (i<n) {
5. **read (x);**
6. **read (y);**
7. **a=a/x;**
8. b=x;
9. **if (a>1)**
10. **b=a-3;**
11. **if (b>0)**
12. **z=x+y;**
else
13. z=x-y;
14. output (z);
15. i=i+1;
}
	$(Faulty\ Statement)$
	10. $b = a - 3$
	should be $\rightarrow$ 10. $b = a - 4$
	
	Input: $a = 8;\ n = 1;\ x = 2;\ y = 2;$
	Erroneous output: $z = 4;$
	Correct output: $z = 0;$
	
	$Full\ Slice = \{1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 14\}$
	$10 \notin Data\ Slice = \{5, 6, 12, 14\}$

Fig. 2 Example of full slice vs. data slice

output value. In the above example, when both types of dependences are considered, statement 10 is included in the full slice. This is because statement 12 is control dependent upon predicate 11 which is data dependent upon statement 10.

To compute full slices, in addition to detecting dynamic data dependences, we must also detect *dynamic control dependences*. While a statement can be statically control dependent upon multiple predicates, at runtime, each execution instance of a statement is dynamically control dependent upon a single predicate. The predicate on which the execution of a statement is control dependent is found as follows. First let us assume that there are no recursive procedures. Given an execution of a statement s, prior to its execution, the most recently executed predicate p on which s is statically control dependent is found. The execution of s is dynamically control dependent upon this execution of p. Timestamps can be associated with execution instances of statements in order to evaluate the above condition. A second condition is needed in presence of recursion. For a given execution of statement s to be dynamically control dependent upon an execution of a predicate p, the execution instances of both must correspond to the same function invocation.

3 Our Slicing Tool

We have developed a dynamic slicing tool which was used to conduct the experiments described in the next section. Our tool executes `gcc` compiler generated binaries for Intel $\times 86$ and computes dynamic slices based upon forward computation algorithms described in the preceding section. Even though our tool works on binary level, the slices can be mapped back to source code level using the debugging information generated by `gcc`.

Figure 3 shows the main components of the tool. The *static analysis* component of our tool computes static control dependence (CD) and potential dependence (PD) information required during full and relevant slice computations from the binary. The static analysis was implemented using the *Diablo* (http://www.elis.ugent.be/diablo/) retargetable link-time binary rewriting framework as this framework already has the capability of constructing the control flow graph from $\times 86$ binary.

The *dynamic profiling* component of our system which is based upon the *Valgrind* memory debugger and profiler (http://valgrind.org/) accepts the same `gcc` generated binary, instruments it by calling the *slicing instrumenter*, and executes the instrumented code with the support of the *slicing runtime*. The slicing instrumenter and slicing runtime were developed by us to enable collection of dynamic information and computation of dynamic slices. Valgrind's kernel is a dynamic instrumenter which takes the binary and before executing any new (never instrumented) basic blocks it calls the instrumentation function,

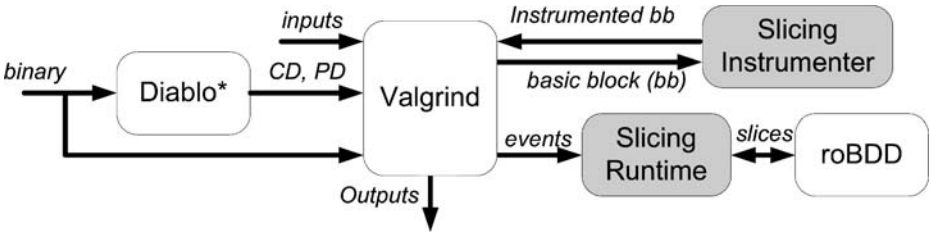

Fig. 3 Slicing infrastructure

which is provided by the slicing instrumenter. The instrumentation function instruments the provided basic block and returns the new basic block to the Valgrind kernel. The kernel executes the instrumented basic block instead of the original one. The instrumented basic block is copied to a new code space and thus it can be reused without calling the instrumenter again. The instrumentation is dynamic in the sense that the user can enforce the expiration of any instrumented basic block such that the original basic block has to be instrumented again (i.e., instrumentation can be turned on and off as desired). Thus, we can easily turn off/on the slicing instrumentation for sake of time performance or for certain code, e.g., library code. The slicing runtime essentially consists of a set of call back functions for certain events (e.g., entering functions, accessing memory, binary operations, predicates etc.). The CD and PD information computed by the static analysis component is represented based on the virtual addresses which can be understood by Valgrind.

Now let us briefly discuss the algorithms used for computing dynamic slices. Two types of methods for computing backward dynamic slices have been proposed: *backward computation* methods (Agrawal and Horgan, 1990; Zhang et al., 2003); and *forward computation* methods (Beszedes et al., 2001; Zhang et al., 2004). In backward computation methods the program dependences that are exercised during a program execution are captured and saved in the form of a dynamic dependence graph. Dynamic slices are constructed upon user's requests by backward traversal of the dynamic dependence graph. Although this approach allows computation of *all* dynamic slices of all variables at all execution points, a problem with this method is its space cost. In *forward computation* methods (Beszedes et al., 2001; Zhang et al., 2004) latest backward dynamic slices of all program variables are computed and maintained as sets of statements as the program executes. Advantage of this approach is that the space cost is no longer proportional to the length of execution but rather proportional to the number of variables times the number of statements in the program. Our system incorporates implementations of both forward and backward computation methods.

As mentioned above, the forward computation algorithms maintain the latest dynamic slice for each variable/location. These dynamic slices are stored in *reduced ordered Binary Decision Diagram* (roBDD) (Lin-Nielsen) component of our system. Earlier work (Zhang et al., 2004) identifies three characteristics of dynamic slices: same dynamic slices tend to *reappear* from time to time during execution, different slices tend to *share statements*, and *clusters of statements* located near each other in the program often appear in a dynamic slice. These characteristics resulted in the observation that roBDD representation of sparse sets was suitable for storing dynamic slices as it was both space and time efficient. The roBDD benefits us in the following respects. Each unique slice is presented by unique integer number in roBDD, which implies that if and only if two slices are identical, they are represented by the same integer number. The whole set of statements in the slice can be recovered from roBDD using that number. This is critical to our design because now for each variable (memory location) we only need to store one integer. Use of roBDD achieves space efficiency because roBDD is capable of removing duplicate, overlapping, and clustered sets which are exactly the characteristics of slices. Using roBDD also provides time efficiency because roBDD implementations of set operations are very efficient. More details about why and how we use roBDD can be found in (Zhang et al., 2004).

We also implemented a simple debugging interface which provides limited capabilities including setting breakpoints, continuing execution, stopping after certain steps of execution, slicing on a register, slicing on a memory location, and slicing on the latest instance of a predicate.

4 Experimental Evaluation

In this section we present results of experiments that we conducted. For these experiments we collected faulty versions of commonly used programs. Unlike our previous study (Zhang et al., 2005) of dynamic slicing algorithms that used faulty versions of programs created by injecting faults in them, this study uses real programs with real bugs that were reported by users of these programs. We carried out two main experiments. The first experiment involves a study of the data and full slices of these programs. This experiment enabled the comparison of data and full dynamic slicing in terms of their applicability (i.e., their ability to capture faults) and effectiveness (i.e., their sizes). The second experiment shows how the computed dynamic slices may be explored by the programmer to locate faults.

The faulty versions of the programs along with the descriptions of the faults are given in Table 1. The source from which the faulty version was obtained is also given. As we can see, these programs are widely used. In addition we would like to note that the first nine faults (i.e., faults in programs `grep 2.5` through `make 3.80`) cause the programs to produce wrong outputs while the last ten faults (i.e., faults in programs `gzip-1.2.4` through `mc-4.5.55`) contain memory bugs lead to a segmentation error. Memory bugs essentially cause memory corruption problems and when a corrupted memory location is accessed, the program crashes with a segmentation fault error message.

Table 1 Faults used in the study

Program	Bug description	Source
grep 2.5	Using -i -o together produces wrong output	http://savannah.gnu.org
grep 2.5.1	(a) Using -F -w together produces wrong output	http://savannah.gnu.org
	(b) Using -o -n together produces wrong output	http://comments.gmane.org/gmane.comp.gnu.grep.bugs/
	(c) "echo dor̂e | grep dor̂e" finds no match	http://comments.gmane.org/gmane.comp.gnu.grep.bugs/
flex 2.5.31	(a) Some variable is not defined with option -l, which fails the compilation of xfree86	http://soureforge.net
	(b) String "]]" is not allowed in user's code	http://soureforge.net
	(c) The generated code contains extra #endif	http://soureforge.net
make 3.80	(a) Backslashes in dependency names are not removed	http://savannah.gnu.org
	(b) Fail to recognize the updated file status while there are multiple target in the pattern rule	http://savannah.gnu.org
gzip-1.2.4	1,024 byte long filename overflows into global variable	AccMon (Zhou et al. 2004)
ncompress-4.2.4	1,024 byte long filename corrupts stack return address	AccMon (Zhou et al. 2004)
polymorph-0.4.0	2,048 byte long filename corrupts stack return address	AccMon (Zhou et al. 2004)
tar-1.13.25	Wrong loop bounds lead to heap object overflow	AccMon (Zhou et al. 2004)
bc-1.06	Misuse of bounds variable corrupts heap objects	AccMon (Zhou et al. 2004)
tidy-34132	Memory corruption problem	AccMon (Zhou et al. 2004)
mutt-1.4.2.1i	Heap buffer bound miscalculation	http://www.securiteam.com/
pine-4.44	(a) Missing end quote corrupts stack	http://www.xatrix.com/
	(b) Special characters corrupt heap buffer	http://www.securityfocus.com/
mc-4.5.55	Uninitialized string corrupts stack	http://www.securityfocus.com/

4.1 Data Slices vs. Full Slices

4.1.1 Applicability and Effectiveness of Data and Full Dynamic Slicing

Our first experimented involved computing the dynamic data slices and dynamic full slices for the failed runs that exercise the faults. Before we compute dynamic slices we must identify a value in the failed run on which to perform dynamic data/full slicing. We encountered three kinds of situations in these faults which were handled as follows:

- For programs that produced an incorrect output value, backward dynamic slicing was performed starting at the first incorrect output value produced during the failed run.
- For the programs that crashed, the value which when referenced caused the crash served as the basis for computing the backward dynamic slice.
- For the four faults in `grep`, it was not possible to perform backward dynamic slicing. When these four faults were exercised the program did not crash but rather it produced incorrect output. However, this incorrect output essentially was *no output*. Since no output was produced, we did not have a value on which to base backward slicing computation. To handle these situations we found the minimal failure inducing input (Hildebrandt and Zeller, 2000) which is the part of the input that triggered the failure. The faulty code was then captured in the *forward* dynamic slice of the failure inducing input.

The results of dynamic slicing are shown in Table 2. The column *In* indicates whether the faulty code was captured by the data slice (DS), in this case it is also captured by the full slice, or whether it is only captured by the full slice (FS). As we can see, out of the 19 faults considered, 12 faults were captured by dynamic data slices, and the remaining 7 faults were captured only by dynamic full slices. We would like to mention that in case of faults in `pine` and `mc`, where the faults are captured by the dynamic data slices, we were unable to

Table 2 Data slices and full slices

Program	LOC	Exec (LOC%)	DS (Exec%)	FS (Exec%)	In	Min (LOC%)
grep 2.5	8,581	1,157 (13.48%)	67 (5.79%)	731 (63.18%)	FS	731 (8.52%)
grep 2.5.1 (a)	8,587	509 (5.93%)	15 (2.95%)	32 (6.29%)	FS	32 (0.37%)
grep 2.5.1 (b)	8,587	1,123 (13.08%)	90 (8.02%)	599 (53.34%)	FS	599 (6.98%)
grep 2.5.1 (c)	8,587	1,338 (15.58%)	6 (0.45%)	12 (0.90%)	DS	6 (0.07%)
flex 2.5.31 (a)	26,754	1,871 (6.99%)	159 (8.59%)	695 (37.15%)	FS	695 (2.60%)
flex 2.5.31 (b)	26,754	2,198 (8.22%)	89 (4.05%)	272 (12.37%)	FS	272 (1.07%)
flex 2.5.31 (c)	26,754	2,053 (7.67%)	24 (1.17%)	50 (2.44%)	DS	24 (0.09%)
make 3.80 (a)	29,978	2,277 (7.60%)	388 (17.04%)	981 (43.08%)	FS	981 (3.27%)
make 3.80 (b)	29,978	2,740 (9.14%)	588 (21.46%)	1290 (47.08%)	FS	1290 (4.30%)
gzip-1.2.4	8,164	118 (1.45%)	14 (11.86%)	34 (28.81%)	DS	14 (0.17%)
ncompress-4.2.4	1,923	59 (3.07%)	13 (22.03%)	18 (30.51%)	DS	13 (0.68%)
polymorph-0.4.0	716	45 (6.29%)	17 (37.78%)	21 (46.67%)	DS	17 (2.38%)
tar-1.13.25	25,854	445 (1.72%)	44 (9.89%)	105 (23.60%)	DS	44 (0.17%)
bc-1.06	8,288	636 (7.67%)	76 (11.95%)	204 (32.07%)	DS	76 (0.92%)
tidy-34132	31,132	1,519 (4.88%)	148 (9.74%)	554 (36.47%)	DS	148 (0.48%)
mutt-1.4.2.1	71,774	2,551 (3.55%)	242 (9.49%)	1052 (41.24%)	DS	242 (0.34%)
pine-4.44 (a)	253,832	3,930 (1.55%)	102 (2.60%)	–	DS	102 (0.04%)
pine-4.44 (b)	253,832	8,956 (3.53%)	605 (6.76%)	–	DS	605 (0.24%)
mc-4.5.55	66,944	3,154 (4.71%)	48 (1.52%)	–	DS	48 (0.07%)

compute the sizes of the full dynamic slices. For `pine`, the version of `diablo` used in our system was not able to handle the compiled binary because it is very large (over 30 MB) and thus control dependence analysis could not be performed. For `mc`, we ran out of shadow space used by `valgrind` for computing full slices. However, we were able to compute dynamic data slices for these programs. Since faults were captured by the dynamic data slices, the relevant results for these programs are being reported.

Now let us see how dynamic slicing reduces the amount of code the programmer has to examine to locate faulty code. In Table 2, *LOC* is the lines of code in each program, *Exec* represents the lines of code that are actually executed during the failed run (i.e., the remaining lines of code are not executed during the failed run)—the number in parenthesis is the value of *Exec* expressed as a percentage of *LOC*. *DS* and *FS* give the lines of code that are not only executed but also belong to the dynamic data slices and full slices, respectively—the numbers in parenthesis are the values of *DS* and *FS* expressed as a percentage of *Exec*. Finally, *Min* is the number of lines of code in the smallest of *Exec*, *DS*, and *FS* that actually captures the faulty code—the number in parenthesis is the value of *Min* expressed as a percentage of *LOC*. In other words, *Min* is the fault candidate set that must be examined by the programmer to locate faulty code. From the data in Table 2 we can make several observations.

First we notice that the lines of code in *Exec* is a small percentage ranging from 1.45 to 15.58% of the total lines of code *LOC* in the program. Since *Exec* is a small percentage of *LOC*, even this rudimentary dynamic information is quite effective in reducing the size of the fault candidate set presented to the programmer for examination. Second, we observe that the sizes of dynamic data and full slices are significantly smaller than *Exec*. The sizes of *DS* range from 0.45 to 37.78% of the sizes of *Exec* and the sizes of *FS* range from 0.90 to 63.18% of the sizes of *Exec*. We also observe that sizes of dynamic data slices are significantly smaller than sizes of dynamic full slices in most of the cases. Finally, the *Min* column we present the size of the fault candidate set that is of significance for the programmer. We observe that the lines of code in *Min* is a very small percentage ranging from 0.04 to 8.52% of *LOC* the total lines of code in the program. Thus we conclude that dynamic information offers significant reductions in the size of the fault candidate set.

4.1.2 Memory Bugs

One key issue is when to use dynamic data slices and when to use full dynamic slices. We observe that for all faults that are memory bugs dynamic data slices captured the faulty code. It is easy to identify that the program has been effected by a memory bug when it crashes with a segmentation fault error. In such situations the user can use dynamic data slicing instead of full dynamic slicing. Through further analysis that we next describe, we determined the reason due to which dynamic data slices are so effective for programs with memory bugs that cause program to crash with a segmentation fault. In other words, even though data slicing is not effective in capturing faulty statement in general, it is very effective for memory related bugs. Since the data slices can be significantly smaller than the full slices (e.g., `tar`, `bc`, etc.) and therefore using data slices for memory related bugs it quite advantageous.

The reason why data slices are so effective for memory bugs is that the program crash is caused due to the presence of an *unexpected dynamic data dependence* between the point at which memory is corrupted and the later point at which the corrupted value is used. In fact the memory corruption typically corrupts a pointer and its use causes a crash because it dereferences the pointer. Dynamic data slice captures all appropriate dynamic data dependences

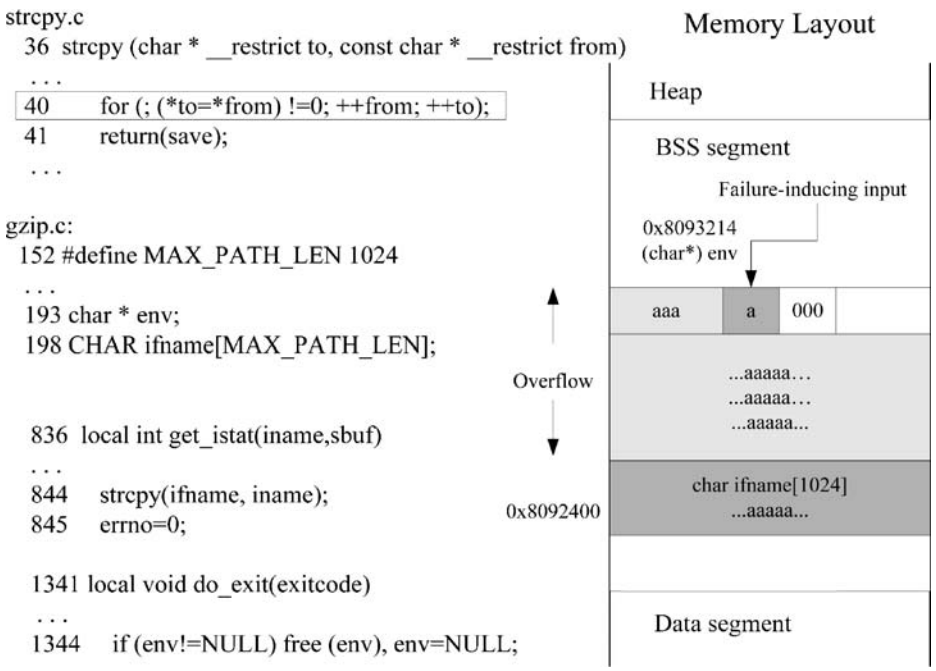

Fig. 4 Buffer overflow bug.

including the unexpected dynamic data dependence and therefore it is able to capture faulty code. To illustrate the above, let us consider the case of `gzip` which contains a buffer overflow problem. In Fig. 4, on the left hand side we show the relevant code segment for the problem. The problem happens in the `strcpy` statement at line 844. Variable `ifname` is a global array defined at line 198. The size of the array is defined as 1,024. Before the `strcpy` statement at line 844, there is no check on the length of the string `iname`. If the length of string `iname` is longer than 1,024, then the buffer overflows. If the length of string `iname` is larger than 3,604, the value of `env` is changed due to buffer overflow. This is because according to the memory layout shown in Fig. 4, the difference between `env` and `ifname` is 3,604 bytes. Later when at line 1,344 *free(env)* is executed, the program crashes due to presence of an illegal memory address in `env`. When dynamic data slice is computed for this illegal address, the faulty statement in `strcpy` is captured in the dynamic data slice.

4.1.3 Relevant Slicing

It has been observed (Agrawal et al., 1993b; Gyimothy et al., 1999) that in some situations faulty statements are not captured by full slices. Consider the following faulty version of a program. Let us consider the situation in which statement $y = 0$; is erroneous and it causes the predicate $y > 0$ to evaluate to false instead of being true. False evaluation of the predicate causes the execution of the assignment to x inside the if-statement to be bypassed leading to incorrect value of x to be output. Since the statement inside the if-statement is not

executed there is not dependence between the output statement and the faulty statement $y = 0;$. In other words, the dynamic full slice does not capture the faulty statement.

```
x = 1;
y = 0;
if y > 0 then
    x = 2;
endif
output(x);
```

In general, the basic reason is that some statements which should have been executed did not get executed due to the fault. To handle the above situation a new form of dependence needs to be introduced between certain predicate outcomes and uses. Given a use u, let us define a *potentially depends* set $PD(u)$ such that the set contains members of the form that specify predicates and their outcomes (i.e., p^T or p^F). If p^T (p^F) is present in $PD(u)$, it means that if prior to the execution of u predicate p was executed, and its outcome was T (F), then while no definition corresponding to u was encountered, it could have been encountered if p had evaluated to F (T). For the above example this means that $(y > 0)^F \in PD(output(x))$ because when the outcome of predicate $y > 0$ is F, no definition of x is encountered after execution of $y > 0$ while if $y > 0$ had evaluated to T the definition $x = 2$ would have been encountered. The potentially depends property is a static property of u which is precomputed and later used at runtime to compute relevant slices.

In an earlier study (Zhang et al., 2005) we reported that when faults are present in predicate statements, full slices are sometimes inadequate and therefore one must use dynamic relevant slices. In this earlier study faults were artificially injected in predicates and studied. In contrast, the results reported in this paper are based upon some real bugs reported by users. We observed that for these real bugs relevant slicing was not needed even though some of these bugs did influence the outcomes of predicates during the failed run. To understand why relevant slices were needed in the earlier study but not in this new study we further studied the nature of bugs in the programs. In the earlier study based upon Siemens suite we noticed that many bugs were injected by changing the predicates and even shortening the predicates by eliminating part of the condition. As a result the situation of the type illustrated earlier where code that should have been executed is bypassed arose requiring the need for relevant slicing. On the other hand, when we studied the incorrect evaluations of predicates in real bugs we noticed a very different behavior. In most of the cases incorrect evaluation of predicates was present in programs with buffer overflow bugs. Here incorrect evaluation of a loop predicate caused the loop body to be executed too many times leading to buffer overflow and memory corruption which caused the program to crash. In other words, the incorrect evaluation of predicates did not cause execution of code to be bypassed and hence the need for using relevant slicing did not arise.

4.2 Exploring Dynamic Slices

A dynamic slice provides a fault candidate set that the programmer must examine to identify the faulty statement. Therefore smaller the set of statements that the user has to examine the better it is. Even though dynamic slices produce fault candidate sets that are

Table 3 Exploring dynamic slices

Program	Slice type	Slice size (*SS*)	Explored SS (*ESS*)	*EDD*
grep 2.5	FS	731	86 (11.76%)	9
grep 2.5.1 (a)	FS	32	25 (78.12%)	8
grep 2.5.1 (b)	FS	599	157 (26.21%)	11
grep 2.5.1 (c)	FS	12	6 (50.00%)	3
flex 2.5.31 (a)	FS	695	13 (1.87%)	5
flex 2.5.31 (b)	FS	272	109 (40.07%)	31
flex 2.5.31 (c)	FS	50	3 (6.00%)	2
make 3.80 (a)	FS	981	187 (19.06%)	21
make 3.80 (b)	FS	1,290	53 (4.11%)	19
gzip-1.2.4	DS	14	2 (14.28%)	2
ncompress-4.2.4	DS	13	1 (7.69%)	1
polymorph-0.4.0	DS	17	4 (23.53%)	3
tar-1.13.25	DS	44	5 (11.36%)	4
bc-1.06	DS	76	4 (5.26%)	3
tidy-34132	DS	148	2 (1.35%)	2
mutt-1.4.2.1	DS	242	17 (7.02%)	4
pine-4.44 (a)	DS	102	3 (2.94%)	3
pine-4.44 (b)	DS	605	38 (6.28%)	18
mc-4.5.55	DS	48	2 (4.17%)	2

small in comparison to the set of executed statements, it can still be quite a lot of work to examine all of the statements in these slices. Therefore we considered a strategy in which the statements in the dynamic slice are *ordered* and the programmer examines the statements in that order. Once the faulty code is encountered by the programmer, the fault is located and the programmer need not examine rest of the dynamic slice. In other words, the programmer need not always explore the entire dynamic slice. The strategy we used orders the statements according to the *dependence distance* between them and the statement at which error was observed. More precisely, the dependence distance of a statement in the dynamic slice is the length of the minimum length chain of dependences starting from the statement and ending at the statement at which error was observed.

The results of this experiment are discussed next. In Table 3 the column *Slice Type* indicates the kind of slice that was explored in this experiment. As we can see, we explored the dynamic data slices (*DS*) for programs with memory bugs and dynamic full slices (*FS*) for other programs. Based upon the observations of the preceding sections this choice is most appropriate. The column *Slice Size* (*SS*) gives the size of the dynamic slice being explored and *Explored Slice Size (ESS)* gives the size of portion of the slice that was explored before the faulty code was encountered. The size of *ESS* as a percentage of *SS* is also given in parenthesis. As we can see, the lines of code in the dynamic slice that were explored as a percentage of the total lines of code in the dynamic slice ranges from 1.35 to 78.12%. In seven out of 11 cases this number is in single digits. Thus, using our proposed strategy, in practice, a programmer has to examine far fewer statements. Finally the maximum dependence distance up to which the dynamic slice was explored (*EDD*) is given. As we can see this dependence distance was found to be small for programs where dynamic full slices were used and for programs with memory bugs this distance was mostly less than five.

Table 4 Dynamic dependence graph size and execution times

Program	DDG size (KB)	Slicing time (s)	DDG time (s)
grep 2.5	760	0.04	35.5
grep 2.5.1(a)	794	0.04	29.2
grep 2.5.1(b)	333	0.02	4.4
grep 2.5.1(c)	968	0.06	20.1
flex-3.51(a)	196,131	4.39	135.5
flex-3.5.31(b)	202,441	3.14	138.9
flex-3.5.31(c)	199,170	6.71	130.2
make 3.80(a)	17,409	0.24	28.0
make 3.80(b)	15,801	1.74	34.6
gzip-1.2.4	164	0.01	1.2
ncompress	211	0.03	1.1
polymorph	173	0.03	0.4
tar	420	0.01	10.9
bc	1,404	0.15	6.7
tidy	92,872	0.53	17.5
mutt-1.4.2.1	74,358	4.34	284.1
pine-4.44 (a)	44,108	6.16	63.5
pine-4.44 (b)	70,266	4.33	68.4
mc-4.5.55	208,849	0.6	120.7

4.3 Cost of Dynamic Slicing

The cost of dynamic slicing consists of two main components: the space cost which is the memory needed to store the dynamic dependence graph (DDG) required for computing the dynamic slices; and the execution time cost which includes the time to collect the runtime information and build the dynamic dependence graph and the time to perform dynamic slicing. The above costs for the faults studied are given in Table 4. The size of the dynamic dependence graph is given in column *DDG Size*. The size of the graph depends upon the length of the failing program run. As we can see the size varies from 173 KB to nearly 209 MB. The time spent on building the dynamic dependence graph, given by column *DDG Time*, ranges from 0.4 to 284.1 s. As we can see, the time is typically proportional to the length of the run, i.e. the size of the DDG. The slicing times are given in column *Slicing Time* and they range from 0.01 to 6.71 s.

5 Related Work

Dynamic slicing was introduced as an aid to debugging by Korel and Laski 1988. Although the idea seems very promising, it has not been used in practice. There is a practical reason for this. The problem of the high cost of computing dynamic slices had not been addressed till recently. In recent work (Zhang and Gupta, 2004; Zhang et al., 2004), we developed practical implementations of dynamic slicing for both backward computation (Zhang and Gupta, 2004) and forward computation (Zhang et al., 2004) algorithms have been developed. We demonstrated that dynamic slices of program runs that were 67 million to 140 million instructions in length, on an average, took 1.92 to 36.25 s to compute (Zhang and Gupta, 2004).

Dynamic slicing has been studied as an aid to debugging by many researchers (Agrawal et al., 1993b; Kamkar, 1993; Korel and Rilling, 1997; Agrawal et al., 1995; Pan and Spafford, 1992). Agrawal et al. (1995) proposed subtracting a single correct execution trace from a single failed execution trace. Pan and Spafford (1992) presented a family of heuristics for fault localization using dynamic slicing. Compared to these previous works, we are the first one to compare the effectiveness of dynamic slicing algorithms in fault location.

General studies of dynamic slice sizes have been conducted. For example, in our work in Zhang and Gupta (2004), it was shown that the number of distinct statements executed at least once during a program execution were 2.46 to 56.08 times more than the number of statements in the dynamic slice. However, these results are based upon computing dynamic slices of randomly selected values computed by correct versions of programs. In another study (Zhang et al., 2005) we computed dynamic slices based upon failed runs of faulty versions of programs. These faults had been injected into the programs. In contrast the study presented in this paper consider a set of real faults reported by users of widely used programs. Some of the observations of this study based upon real faults are different from those of the previous study of injected faults. The differences and the explanation for these differences are as follows:

- First, in our current study, for the faults in `grep`, no output was produced and hence instead of backward dynamic slices we had to make use of forward dynamic slices. Similar situation did not arise for the Siemens suite (Hutchins et al., 1994; http://www.cse.unl.edu/~galileo/sir) programs used in the earlier study (Zhang et al., 2005).
- Second, in the earlier study the need for using relevant slicing arose while in our current study data and full dynamic slices were able to capture all faults. As explained earlier, in our current study that contains many memory bugs, most of the situations where predicates evaluated incorrectly, the incorrect evaluation did not cause bypassing of the code but rather execution of the code that should have been bypassed. Therefore, relevant slicing was not needed.

We would also like to point out that, while examining the statements in a dynamic slice, the relevance of dependence distance from the erroneous output has long been considered useful (Antoniol et al., 1997; Ball and Eick, 1994; Krinke, 2004). Therefore in tools for visualizing dynamic slices, ways have been explored to communicate to the programmer the dependence distance information. For example, in Krinke (2004), Krinke uses different shades of gray to highlight the statements in the dynamic slice. In particular, the darker the shade, the smaller is the dependence distance. In this paper, through experiments, we have validated the merit of using dependence distance information while exploring dynamic slices.

A lot of interesting research other than dynamic slicing have been carried on in fault location. Zeller has presented a series techniques (Hildebrandt and Zeller, 2000; Zeller, 2002; Cleve and Andreas Zeller, 2005) from isolating the critical input to isolating cost-effect chains in both space and time. The basic idea is to find the specific part of the *input/program state* which is critical to the program failure by minimizing the difference between the *input/program state* leading to a passing run and that leading to a failing run. We believe our technique can be combined with Zeller's technique in many aspects, for instance, the isolated *causes* are perfect slicing criteria starting from which dynamic slicing may produce a much smaller fault candidate set than from the failure point. Renieris and Reiss (2003) presented a technique that selects the single passing run that most resembles to the failing run and reports the difference between these two runs. Jones (2004) presented a

technique that uses software visualization to assist fault location. Their technique provides a ranking of each statement according to its ratio of failing tests to correct tests.

6 Conclusions

The development of dynamic slicing was motivated by the problem of locating the faulty code when an execution of a program fails. There has been a significant amount of research on developing algorithms for computing different types of dynamic slices. The contribution of this paper is to present an experimental evaluation of effectiveness of dynamic slices for the benefit of using them to locate a faulty statement in a program. In particular, this is the first study based upon real faults reported by users of widely used programs. From our experiments we found that data slices were found to be very effective for memory related faults and for remaining faults full slicing was adequate. None of the faults required the use of relevant slicing. Finally, we found that even if the slice size is large, the user may have to examine only a subset of statements in the slice before encountering the faulty statement.

Acknowledgments This work is supported by grants from Microsoft, IBM, and NSF grants CNS-0614707, CCF-0541382, CCF-0324969, and EIA-0080123 to the University of Arizona. We would like to thank the reviewers for their suggestions that encouraged us to do even more thorough job of revising the original submission.

References

Agrawal H, Horgan J (1990) Dynamic program slicing. ACM SIGPLAN conference on programming language design and implementation, pp 246–256

Agrawal H, DeMillo R, Spafford E (1993a) Debugging with dynamic slicing and backtracking. Software Practice and Experience 23(6):589–616

Agrawal H, Horgan EW Jr, London SA (1993b) Incremental regression testing. IEEE conference on software maintenance, Montreal, Canada, pp 348–357

Agrawal H, Horgan J, London S, Wong W (1995) Fault localization using execution slices and dataflow tests. 6th IEEE international symposium on software reliability engineering, 143–151

Antoniol G, Fiutem R, Lutteri G, Tonella P, Zanfei, S, Merlo E (1997, October) Program understanding and maintenance with the CANTO environment. International conference on software maintenance, pp 72–, Bari, Italy

Ball T, Eick SG (1994, October) Visualizing program slices. IEEE Symposium on Visual Languages, pp 288–295, St. Louis, Missouri

Beszedes A, Gergely T, Szabo ZM, Csirik J, Gyimothy T (2001, March) Dynamic slicing method for maintenance of large C programs. 5th European conference on software maintenance and reengineering, pp 105–113

Cleve H, Andreas Zeller (2005) Locating causes of program failures. 27th international conference on software engineering, pp 342–351

Gyimothy T, Beszedes A, Forgacs I (1999) An efficient relevant slicing method for debugging. 7th European software engineering conference and 7th ACM SIGSOFT international symposium on foundations of software engineering. Toulouse, France, pp 303–321

Hildebrandt R, Zeller A (2000) Simplifying failure-inducing input. International symposium on software testing and analysis, pp 135–145

Hutchins M, Foster H, Goradia T, Ostrand T (1994) Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. 16th international conference on software engineering, pp 191–200

Jones JA (2004) Fault localization using visualization of test information. 26th international conference on software engineering, pp 54 –56

Kamkar M (1993) Interprocedural dynamic slicing with applications to debugging and testing. Ph.D. thesis, Linkoping University

Korel B, Laski J (1988) Dynamic program slicing. Inf Process Lett 29(3):155–163
Korel B, Rilling J (1997) Application of dynamic slicing in program debugging. 3rd international workshop on automatic debugging. Linkoping, Sweden, pp 43–58
Krinke J (2004) Visualization of program dependence and slices. International conference on software maintenance, pp 168–177
Lin-Nielsen J BuDDy, A binary decision diagram package. Department of Information Technology, Technical University of Denmark, http://www.itu.dk/research/buddy/.
Pan H, Spafford EH (1992) Heuristics for automatic localization of software faults. Technical Report SERC-TR-116-P, Purdue University
Renieris M, Reiss S (2003) Fault localization with nearest neighbor queries. IEEE international conference on automated software engineering, pp 30–39
Weiser M (1982) Program slicing. IEEE Trans Softw Eng SE-10(4):352–357
Zeller A (2002) Isolating cause-effect chains from computer programs. 10th ACM SIGSOFT symposium on foundations of software engineering. Charleston, South Carolina, pp 1–10
Zhang X, Gupta R (2004, June) Cost effective dynamic program slicing. ACM SIGPLAN conference on programming language design and implementation, pp 94–106
Zhang X, Gupta R, Zhang Y (2003, May) Precise dynamic slicing algorithms. IEEE/ACM international conference on software engineering, Portland, Oregon, pp 319–329
Zhang X, Gupta R, Zhang Y (2004) Effective forward computation of dynamic slices using reduced ordered binary decision diagrams. IEEE international conference on software engineering. Edinburgh, UK, pp 502–511
Zhang X, He H, Gupta N, Gupta R (2005, September) Experimental evaluation of using dynamic slices for fault location. SIGSOFT-SIGPLAN sixth international symposium on automated and analysis-driven debugging. Moterey, California, pp 33–42
Zhou P, Liu W, Fei L, Lu S, Qin F, Zhou Y, Midkiff SP, Torrellas J (2004) AccMon: automatically detecting memory-related bugs via program counter-based invariants. 37th annual international symposium on microarchitecture, pp 269–280

Xiangyu Zhang expects to complete his PhD degree in computer science from the University of Arizona in August 2006. He will join Purdue University in the Fall of 2006 as an Assitant Professor in the Computer Science Department. He obtained his BS and MS degrees in the University of Science and Technology of China. His research interests include automatic debugging, software reliability, computer security, and program profiling. He has published papers in many prestigious conferences and journals such as PLDI, TOPLAS, TACO, FSE, ICSE, ASE, MICRO, and HPCA.

Neelam Gupta is an Assistant Professor of Computer Science at the University of Arizona. Her research areas include software testing, dynamic program analysis, and automated debugging. Her research has been funded by NSF, Microsoft, IBM, and Arizona Center for Information Science and Technology (ACIST). Neelam is a member of ACM and IEEE. She served as a co-chair of the program committee of Fourth International Workshop on Dynamic Analysis (WODA 2006), and is serving as a co-chair of the program committee of Third International Workshop on Software Quality Assurance (SOQUA 2006). She has also served or is serving on the program committees of IEEE International Conference on Automated Software Engineering (ASE 2006, ASE 2003), International Computer Software and Applications Conference (COMPSAC 2006), International Workshop on Dynamic Analysis (WODA 2004, 2005), International Workshop on Security, Privacy, and Trust for Pervasive Applications (SPTPA 2006), and Workshop on Software Quality (SOQUA 2005).

Rajiv Gupta is a Professor of Computer Science at The University of Arizona. His areas of research interest include (Embedded Systems) Compiler and Architectural Support for Optimization of Performance, Power, & Memory, and providing Security; (Software Engineering) Software Tools for Profiling, Slicing, and Debugging; and (Program Analysis) Static, Dynamic, and Profile-based. He has published over 190 articles in refereed conferences and journals, he holds 8 US patents, and has supervised 14 PhD dissertations. Papers coauthored by him have been selected for: inclusion in 20 Years of PLDI (1979-1999), distinguished paper award in ICSE 2003, most original paper award in ICPP 2003, and outstanding paper award in ICECCS 1996. Rajiv is a member of the Technical Advisory Group (TAG) on Networking and Information Technology by the US President's Council of Advisors on Science and Technology (PCAST). Rajiv received the National Science Foundation's Presidential Young Investigator Award in 1991 and served as an IEEE Distinguished Visitor for the period of 2000-2002. He served as the Program Chair for PLDI'03, HPCA'03, and LCTES'05 conferences and Co-General Chair for CGO'05 conference. Rajiv has served in over 75 program committees including those of the PLDI, POPL, CGO, CC, MICRO, HPCA, IEEE MICRO Top Picks, ISPASS, LCTES, CASES, HiPEAC, ICS, PACT, PEPM, PASTE, and ICCL. He serves as an Associate Editor for ACM Transactions on Architecture and Code Optimization, Parallel Computing journal, Journal of Embedded Computing, and Computer Languages, Systems and Structures journal. Rajiv is a member of the ACM and a senior member of IEEE.