Introducing Loop Statements in Random Testing of C Compilers Based on Expected Value Calculation

Kazuhiro Nakamura [1] Nagisa Ishiura [1]

[1] School of Science and Technology, Kwansei Gakuin University, Sanda, Hyogo, Japan

Abstract— This paper presents a method of reinforcing random testing of C compilers by introducing loop statements. While random testing based on precomputation of expected values is powerful in detecting bugs in C compilers, loop statements have not been handled, due to difficulties in avoiding undefined behavior. In this paper, an extended method to eliminate undefined behavior in loop bodies is proposed, where arrays of precomputed constants are used to modify problematic operands during loop iterations. A random test system based on the proposed method has uncovered a new bug in the latest version of LLVM which can not be detected by the existing methods.

I. Introduction

Since compilers are infrastructure tools for software development, extremely high reliability is required to them. Although compilers go through extensive testing using test suites of enormous volumes, there are cases where potential bugs survive. Random testing, which tests compilers by randomly generated programs, is the last resort to hunt for such bugs.

Various methods of random testing have been proposed so far, which are classified into two categories. One is based on differential testing [1]. It can generate a broad class test programs, but various restrictions are posed on test programs to guarantee the validity of their dynamic behavior. Csmith [2] is a representative of this approach. The other category is based on precomputation of expected correct behavior of test programs. While it is easier to avoid generating invalid test programs, it can cover limited range of C syntax. Quest [3] and Orange3 [4] belong to this category.

Csmith detected more than 325 new bugs in GCC and LLVM and contributed to the improvement of these compilers. However, there exist bugs which are difficult to find by differential testing. Orange3, which targets arithmetic optimization, has reported such bugs in the latest versions of GCC and LLVM. Thus the two approaches are complement and there still remain rooms for reinforcing testing methods in the both approaches.

This paper attempts to extend the ability of the second category by introducing loop statements in random programs, while Quest tests calling conventions and Orange3 only arithmetic optimization. The major difficulty of avoiding dynamic undefined behavior during loop iterations is solved by extending the code modification method in orange3 [4]. By this method, a new bug has been revealed in the latest version of LLVM.

```
int x0 = 5;
int x1 = 3;

t0 = x0 >> x1;
t1 = x0 / t0;
```

⇒

```
int x0 = 5;
int x1 = 3;
int k0 = 1;

t0 = x0 >> x1;
t1 = x0 / (t0 + k0);
```

Fig. 1. Undefined behavior elimination in Orange3.

II. Random testing of C compilers

There are two major challenges in random testing of compilers; how to judge the correctness of the behavior of randomly generated programs, and how to randomly generate only valid programs.

Csmith [2] solves the first problem by differential testing [1] (comparison of the results from different compilers). In order to eliminate invalid dynamic behavior in test programs, it generates pessimistic codes. For example, array subscripts are always guarded by modulo operators to fit their values within valid subscript ranges, which may degrade the testing capabilities.

On the other hand, Orange3 [4] overcomes the challenges by precomputing the exact behavior of the generated programs. If undefined behavior is detected in a program, it is eliminated by modifying the program. For example, in Fig. 1, zero division is detected in the last line of the left side code, for `t0` evaluates to zero, then the expression is modified to make the deviser non-zero. The precomputation based approach would produce better test programs but can be applicable to limited classed of C syntax. Orange3 can generate programs only consisting of a sequence of assign statements.

III. Introducing loop statements

This paper presents a method of introducing loop statements in random testing of C compilers based on expected value calculation.

Fig. 2 shows an example of a test program generated by the proposed method. Loops with limited bounds are considered. The loops may be nested. The variables may be assigned lexically only once but may be updated multiple times within loops. Array variables may appear on the right hand sides of assign statements but not on the left hand sides.

During test program generation, the correct value of each variable is computed at every assign statement, and the operands for every operation are tested whether they trigger undefined behavior or not. If undefined behavior is detected, the subexpression in question is modified. Undefined behavior in loops are handles as follows:

(1) The value of every subexpression is tracked for every iteration.

TABLE I
EXPERIMENTAL RESULTS (COMPARISON WITH ORANGE3 [4].)

compiler (target)	time [h]	Orange3 [4]			proposed method		
		#test	#error	(#loop)	#test	#error	(#loop)
GCC-4.5.0 (x86_64-pc-linux)	20.0	105,175	0	(0)	50,678	4	(1)
GCC-4.9.0 (x86_64-pc-linux)	200.0	795,971	3	(0)	423,782	3	(0)
LLVM-3.3 (x86_64-pc-linux)	2.0	9,531	4	(0)	3,711	6	(0)
LLVM-3.6 (x86_64-pc-linux)	0.1	594	65	(0)	337	25	(13)

Tested options: `-O0`, `-O3`, `-Os` CPU: Core i7-870 2.93GHz

```
01: #include <stdio.h>
02: #define OK() printf("@OK@\n")
03: #define NG() printf("@NG@\n")
04:
05: int x0[3][2] = {{0,-1},{-2,-3},{-4,-5}};
06: int t0 = 3;
07: long x2 = 8;
08:
09: int main (void)
10: {
11:   int t1 = -2;
12:   short x1 = 3;
13:   int t2 = 1;
14:
15:   for ( int i = 0; i < 3; i++ ) {
16:     t0 = x2 * x1;
17:     for ( int j = 0; j < 2; j++ ) {
18:       t1 = x0[i][j] + x1;
19:       t2 = x1 - t1;
20:     }
21:   }
22:
23:   if (t0 == 24) { OK(); } else { NG(); }
24:   if (t1 == -2) { OK(); } else { NG(); }
25:   if (t2 == 5) { OK(); } else { NG(); }
26:
27:   return 0;
28: }
```

Fig. 2. Example of a test program generated by proposed method.

```
int x0 = 5;
int x1[3] = {1,2,3};
for ( i=0; i<3; i++ ) {
  t0 = x0 >> x1[i];
  t1 = x0 / t0;
}
```

⇒

```
int x0 = 5;
int x1[3] = {1,2,3};
int k0[3] = {0,0,1};
for ( i=0; i<3; i++ ) {
  t0 = x0 >> x1[i];
  t1 = x0 / (t0+k0[i]);
}
```

Fig. 3. Undefined behavior elimination in proposed method.

(2) If any one of them incurs undefined behavior, the subexpression is modified by inserting an add operation with the other operand being an array variable. The subscripts of the array variable are the loop iteration variables. The array variable is initialized so that it evaluates to some arbitrary value to avoid undefined behavior in the iteration in question and zeros for the other iterations.

Fig. 3 shows an example. In the initial code on the left side, the values of variable `t0` are 2, 1, and 0 for `i` = 0, 1, and 2, respectively, which causes zero devision on subexpression `x0/t0` when `i` is 2. The code is modified into the one in the right, where array variable `k0[i]` is added to `t0` which is initialized as `{0,0,1}` so that the deviser will be nonzero in the last iteration.

IV. EXPERIMENTAL RESULT

An extension of Orange3 has been implemented based on the proposed method. Table I compares the results of the test run by Orange3 [4] and the proposed method. Column "#test" shows the number of the random programs generated to test the compiler, "#error" the number of the programs that detected errors. The error programs were minimized for analysis. "#loop" indicates the number of programs that still contain loops after minimization. Fig. 4 is one of the error programs for LLVM-3.6, which was reported to LLVM Bugzilla [1].

```
01: int a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p;
02: int main (void)
03: {
04:   volatile int q=0, r=0, s=0, t=0, u=0;
05:   for (i = 0; i < 1; i++) {
06:     volatile int v = 1;
07:     a = m + 1;
08:     b = n;
09:     c = 1 / v;
10:     d = r + o;
11:     e = t;
12:     f = p + t;
13:     g = a + 1;
14:     h = g + q;
15:     j = s;
16:     k = r + l;
17:   }
18:   int w = g | (int)(16777219.0L + u);
19:   if (w != 16777219) { __builtin_abort(); }
20:   return 0;
21: }
```

Fig. 4. Error program for LLVM-3.6.

V. CONCLUSION

This paper has presented a method of introducing loops in random testing of C compilers based on expected value calculation. By the new method, new error cases which can not be generated by the previous methods were found.

We are now working on further reinforcing the expected value based method by incorporating conditional statements, function calls, etc.

REFERENCES

[1] W. M. McKeeman: "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107 (Dec. 1998).

[2] X. Yang, Y. Chen, E. Eide, and J. Regehr: "Finding and understanding bugs in C compilers," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 283–294 (June 2011).

[3] C. Lindig: "Find a compiler bug in 5 minutes," in *Proc. ACM International Symposium on Automated Analysis-Driven Debugging*, pp. 3–12 (Sept. 2005).

[4] E. Nagai, A. Hashimoto, and N. Ishiura: "Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers," in *Proc. SASIMI 2013*, pp. 88–93 (Oct. 2013).

[1] http://www.llvm.org/bugs/show_bug.cgi?id=21160