Random Testing of C Compilers Based on Test Program Generation by Equivalence Transformation

Kazuhiro NAKAMURA† and Nagisa ISHIURA
School of Science and Technology, Kwansei Gakuin University
2-1 Gakuen, Sanda, Hyogo 669-1337, Japan

***Abstract*—This paper presents a new method of generating test programs for random testing of C compilers based on equivalence transformation on C programs. Although equivalence transformation on programs is a promising way of generating new test programs without undefined behavior, existing methods needed valid test programs used as seeds, and they applied only addition/deletion of statements to/from unreachable portion of the seed test programs. On the other hand, our method generates a vast variety of test programs starting from a trivial initial program by repeatedly applying equivalence transformation, which works on live codes. Furthermore, our method can control the resulting operand values of the subexpressions in test programs, which contributes to a higher bug detection ability. A random test systems, Orange4, based on our method has detected bugs in the latest development versions of GCC-6.0.0 and LLVM-3.9.**

I. Introduction

Compilers are infrastructure tools for developing software in all the fields, including enterprise systems, embedded systems, mission critical systems, etc. Although lexical and syntax analysis modules in mature compilers such as GCC and LLVM have rarely malfunctions, optimization modules, which perform aggressive and sophisticated transformation on intermediate representation, are vulnerable to new bugs, and actually many bugs are still being reported to their bug databases [1], [2]. Thus testing of compilers to secure their reliability is a critical issue.

Although compilers are intensively tested using test suites, huge sets of test programs such as [3], [4], [5], [6], it is theoretically impossible to validate compilers completely with a finite set of programs. Randomly generated test programs are used as complements of the test suites in an attempt to detect potential deep bugs.

Csmith [7] is one of the most successful random test system for C compilers, which reported 79 bugs in GCCs and 202 bugs in LLVMs. It covers broad range of syntax of the C language, including arrays, structs/unions, conditional and loop statements, function calls, etc. However, in order to avoid generating test programs with undefined behavior (such as zero division), it places some conservative restrictions on the syntax of test programs.

On the other hand, a precomputation-based method, as employed in Orange3 [9], generates more aggressive test programs by avoiding undefined behavior based on the precise precomputation of the code behavior during program construction. Orange3 has reported 8 bugs and 5 bugs in the latest versions of GCCs and LLVMs, respectively, which can not be detected by Csmith. However, it is difficult to handle complex syntax by this method. Orange3 can only generate programs with sequence of assign statements, and its extension is limited to *for* loops [10].

Proteus [13] and Athena [14] are recent successful random test systems based on equivalence transformation of test programs. They generate a set of test programs from an existing test program. However, they need the seed test programs which must be valid and complex enough. Moreover, transformation is limited to deletion or addition of code fragments which are not on the execution paths of the programs.

It is also an issue that randomly generated expressions tend to evaluate to specific values. This means that code optimization for arithmetic expressions is tested on limited operand values.

To address these issues, this paper proposes a new test generation method for C compiler random testing. It can generate valid and complex test programs by repeatedly applying equivalence transformation on test programs starting with a trivial but valid seed program. The transformation in our method is more sophisticated than those of Proteus [13] and Athena [14] in the sense that it works on live codes as well as dead codes. While it can generate aggressive test programs as the precomputation based method, it can handle more complex syntax. Besides, it can control the distribution of the values of operands of every subexpression in the programs, which contributes to a higher bug detection ability.

A random test systems, Orange4, based on the proposed method has detected bugs in the latest development versions of GCC-6.0.0 and LLVM-3.9. It has also detected a bug in GCC-4.5.0 by a test program which cannot be generated by Orange3.

II. Random Testing of Compilers

A. Approaches

Random testing of compilers refers to testing compilers by automatically generated random programs. Each program is compiled by the compiler under test, executed, and checked if the result is correct. The major two challenges in compiler random testing are how to tell the correct behavior of the randomly generated programs and how to avoid generating programs with undefined behavior (zero division, signed overflow, out of bounds array access, etc). There are three major approaches to address these issues.

The first one is based on differential testing [8], where test programs are compiled by different compilers and the execution results are compared. Undefined behavior is avoided by imposing restrictions on the syntax for generating test programs. For example, every divide operation is guarded as `(b!=0)?(a/b):(a)`, or every array access as `x =`

† Currently with SCSK Corporation, Toyosu Front, 3-2-20, Toyosu, Koto-ku, Tokyo 135-8110, Japan.

```
01: #include <stdio.h>
02: #define OK()   printf("@OK@\n")
03: #define NG(fmt,val)  printf("@NG@ (test = " fmt ")\n",val)
04:
05: static const volatile float x0 = -9.0F;
    ...
15: signed int t3 = -93467547;
16:
17: int main (void)
18: {
19:     static const double x3 = -7461802.0;
        ...
29:     signed long t4 = 285128L;
30:
31:     t0 = (x10+((x16<(x9/x13))+x4));
32:     t1 = ((x6*(x17<<((((signed int)x12)+k22)>>x21)))&&x19);
33:     t2 = ((t0/(x12<=(x12>=x17)))&((signed int)x12));
34:     t3 = (((((signed int)x3)+k25)>>(x15+x13))>(x2&((signed int)x0)));
35:     t4 = (x10|(x2-((x19==x12)>>x27)));
36:
37:     if (t0 == 2284795846113651LL) { OK(); } else { NG("%lld", t0); }
        ...
41:     if (t4 == -1L) { OK(); } else { NG("%ld", t4); }
42:
43:     return 0;
44: }
```

Fig. 1. Example of Orange3 test program.

`a[(i+j)%n]` where `n` is the size of array `a`. Csmith [7] is based on this approach.

Another approach is precomputation based program generation. Correct behavior of test programs is computed beforehand according to the language standard. Undefined behavior is eliminated during program construction based on the precomputed values of subexpressions. For example, `a/(x+y)` is rewritten, when and only when `x+y` evaluates to zero, into `a/(x+y+c)` where `c` is a non-zero variable. Orange3 [9] is based on this approach. In general, this approach can generate more aggressive test programs than the first one, but the range of the syntax it can handle is much smaller. An example of the test programs generated by Orange3 is shown in Fig. 1. It consists only of assign statements with arithmetic expressions. Although there has been an attempt to extend Orange3 to handle loops [10], it would be very difficult to detect and avoid undefined behavior in further complex programs.

The third rather new approach is based on equivalence transformation on test programs. Given a valid test program, it generates other test programs by applying transformation that do not change the output of the program. Proteus [13] and Athena [14] are based on this approach. Variety of transformation would be possible, but the two systems use only limited one; they delete or add code fragments which are never executed in the programs, identified by code coverage evaluation.

B. Polarization of Expression and Operand Values

In the C language, result of comparison (such as <=) is either 0 or 1 of the `signed int` type. Then randomly generated arithmetic expressions with comparison operators tend to evaluate to small values. TABLE I summarizes the distribution of the resulting values of the expressions in test programs generated by Orange3. MIN and MAX stand for the minimum and the maximum values of the type. We can see that 0 account for a large percentage. As a consequence, the operands of the all operations should have the same distribution. Many arithmetic optimization rules are triggered by particular operand values, but only part of them are well tested in this situation.

C. Generation of Expressions by Derivation of Syntax Trees

Nakahashi and Yura proposed a derivation based method of generating random arithmetic expressions without undefined behavior [11], [12]. In this method, a target value for an expression is first determined, as `36` in Fig. 2. Then, an operation and operands are determined so that the expression evaluates to the target value. In the figure, `36` is expanded to `9<<2`. A

TABLE I. DISTRIBUTION OF RESULTING VALUES OF EXPRESSIONS IN ORANGE3 TEST PROGRAMS.

type	MIN∼−1	0	1	2∼MAX
signed int	8.9%	42.1%	23.9%	25.1%
unsigned int		31.7%	3.3%	65.0%
signed long	10.9%	16.2%	1.5%	71.4%
unsigned long		27.0%	2.3%	70.7%
signed long long	11.9%	15.9%	1.1%	71.1%
unsigned long long		29.9%	3.5%	66.6%

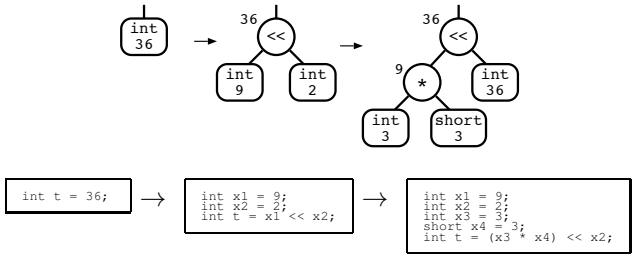


Fig. 2. Generation of expression by derivation.

value may be replaced by a variable which holds the value. This process is recursively repeated until the expression of a desired size is obtained. During derivation, values are chosen so that they do not trigger undefined behavior. Furthermore, the values are chosen arbitrarily as long as they do not trigger undefined behavior.

III. RANDOM PROGRAM GENERATION BASED ON EQUIVALENCE TRANSFORMATION

A. Overview

In this section, a method of generating random programs for compiler testing by equivalence transformation on programs is presented. In this method, a complex valid program is generated starting from a trivial program by repetitive application of equivalence transformation.

The trivial program used as the initial seed in our method is shown in Fig. 3. It is a valid test program whose correct behavior is to exit without printing anything. An example of the programs generated by our method is shown is Fig. 4. Lines 5–69 in the program define variables, lines 71–112 do computation, and lines 114–128 check if the computation is correct. It contains nested *for* and *if* statements.

B. Transformation Rules

By "equivalence transformation," we refer to transformation that does not change the behavior of programs in terms of their output. This paper introduces the following six transformation rules.

(1) Adding a variable declaration

A variable declaration statement is added to the program under generation. The type, the scope (global, static, or local), and the modifier (none, const, volatile, or const volatile) are randomly determined. The initial value of the variable is also chosen randomly from the domain of its type. Fig. 5 (1) illustrates how this rule is applied to the initial program.

(2) Adding a constant assignment

An assign statement is added, whose left-hand side is a declared variable and right-hand side is a constant. Fig. 5 (2) shows an example of this rule application. We impose a restriction that each variable may be assigned only once. This is to make the minimization procedure in III-D easier. Since

```
#include <stdio.h>
#define NG() printf("@NG@\n")

int main (void) {
    return 0;
}
```

Fig. 3. Initial seed program.

```
001: #include <stdio.h>
002: #define OK()  printf("@OK@\n")
003: #define NG(test,fmt,val)  printf("@NG@ ("test" = " fmt ")\n",val)
004:
005: static signed long long x40 = 5473828800LL;
     ...
037: static unsigned short t18 = 9U;
038:
039: int main (void)
040: {
041:     volatile unsigned int x12 = 43794490U;
         ...
068:     signed long x103 = 143L;
069:     signed int i;
070:
071:     for( i = ((signed int)(x40/((signed long)x36)));
072:          i > ((signed int)(x41-x42));
073:          i -= ((signed int)(x43-((signed long)x42)))) {
074:         ...
085:         if( ((unsigned long)(((unsigned char)x68)/x69)) ) {
086:             t7 = ((unsigned long)(x70>>((signed int)x52)));
087:             t8 = ((unsigned char)(x71>>((unsigned long long)t2)));
088:         }
089:         else {
090:             t9 = ((unsigned long)(x72%((signed long long)x34)));
091:             t10 = ((signed char)(x75%((signed long long)x56)));
092:             t11 = (((signed char)x21)*x78);
093:             t12 = ((unsigned int)(x81&x82));
094:         }
             ...
112:     }
113:
114:     if (t1 == 3863U) { OK(); } else { NG("t1", "%hu", t1); }
         ...
128:     if (t17 == 2765) { OK(); } else { NG("t17", "%hd", t17); }
129:
130:     return 0;
131: }
```

Fig. 4. Test program generated by the proposed method.

recent compilers adopt the SSA (static single assignment) based internal representation, we consider this restriction does not seriously degrade the bug detection ability.

(3) Adding an *if* statement

An *if* statement is introduced into the test program. As shown in Fig. 6, existing list of statements are partitioned into four segments, then an *if* statement, with the second and the third segments in its *then* and *else* parts, is inserted between the first and the fourth segments. The *else* part may be omitted (in which case, the statement list is partitioned into three segments). At this point, the condition of the *if* statement is a constant, a zero or a non-zero random value. Nested *if* statements can be generated, as shown in Fig. 5 (3).

(4) Adding a *for* statement

A *for* statement is added in the similar way as in (3), as shown in Fig. 5 (4). Its loop control must be of the format

`for (`*var*=*expression*$_1$`;` *var*<*expression*$_2$`;` *var*+=*expression*$_3$`)`

where *var* is a variable separate from the ones declared in (1). Each of *expression*$_i$ is a constant at this point.

An important restriction here is that the values of the loop controlling expressions are chosen so that the loop will iterate *at most once*. This drastically eases generation of valid test programs without undefined behavior. Since the loop count is unknown to compilers when the loop controlling expressions contain volatile variables, optimization related to the loop will still be performed[1].

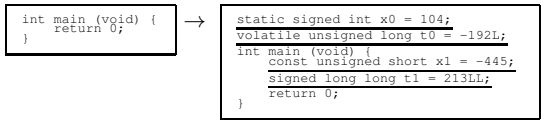
(5) Adding a verification statement

A statement to check the result is inserted, just before the return statement, as shown in Fig. 5 (5).

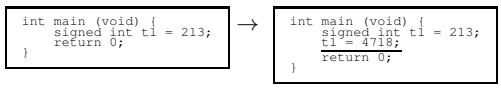
(6) Expanding a constant into an expression

Each constant created by rules (2) through (4) is expanded into complex arithmetic expressions by the method in [11], [12], as shown in Fig. 5 (6).

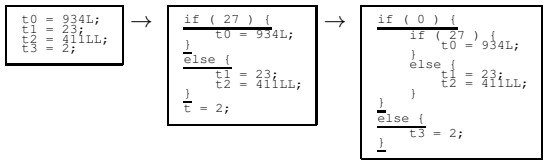
[1]Optimization dependent on the loop iteration count can not be tested by the program generated by this rule, though.



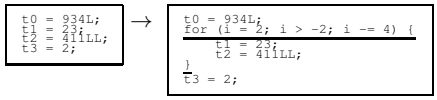
(1) Adding variable declarations.



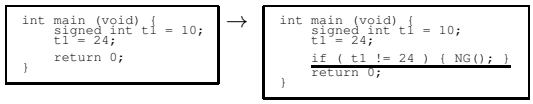
(2) Adding constant assignments.



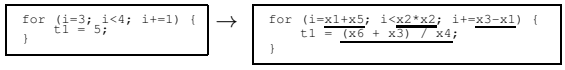
(3) Adding *if* statements.



(4) Adding *for* statements.



(5) Adding verification statements.



(6) Expanding constants into expressions.

Fig. 5. Examples of rule application.

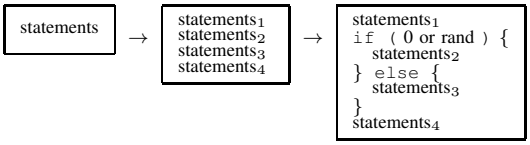


Fig. 6. Insertion of an *if* statement.

C. Control of the Operand Values

Our method can arbitrarily determine the values of expressions and operands of subexpressions. This can be used to enhance bug detection ability of test programs.

At each subexpression derivation step described in II-C, one of the operand values can be chosen arbitrarily. Here, we can use random values with non-uniform distribution. Especially, we can put stress on boundary values. This can be done by raising the probabilities of choosing the minimum and the maximum values of the type.

D. Minimization of Error Programs

Once an error is detected by a test program, the cause of the error must be tracked down to fix possible bugs. For this purpose, minimization of error programs is an essential process where the error program is reduced to a program as small as possible and yet presents the same error.

The minimization procedure in our method is similar to that of Orange3 [9]. It repeats (1) replacement of expressions by constants, (2) minimization of expressions, (3) deletion of *if*

TABLE II. EXPERIMENTAL RESULT.

Compiler (target)	Orange3		Orange4 (uniform)		Orange4 (non-uniform)	
	#test	#error	#test	#error	#test	#error
GCC-4.8.4 (x86_64-pc-linux)	57,051	0	29,871	0	30,187	16
GCC-5.2.1 (x86_64-pc-linux)	49,402	0	30,208	0	26,659	16
GCC-6.0.0 (x86_64-pc-linux)	37,417	0	24,762	0	21,905	2
LLVM-3.6 (x86_64-pc-linux)	54,972	1	32,778	0	29,467	4
LLVM-3.8 (x86_64-pc-linux)	44,685	1	51,985	0	24,258	0

time: 72 [h], #op per program: 1000, CPU: Intel Core i7–4930K 3.40GHz, RAM: 15.6GB
non-uniform distribution: MIN 10%, −1 10%, MAX 10%, rand 70%

```
#define INT_MIN (-2147483647 - 1)

int main(void)
{
  int  x0 = INT_MIN;
  long x1 = 0L;
  int  x2 = 0;
  int  t  = (0 || (INT_MIN - (int) (x0 - x1)));
  if ( t != 0 ) { x2 = t; __builtin_abort(); }
  return 0;
}
```

(a) Error program that detected a bug in GCC-6.0.0.

```
int x =  0;
int y =  -1;

int main (void)
{
   int a = 0x7FFFFFFF + x;
   int b = 0x7FFFFFFF + y;
   int t = (unsigned int) a >= (unsigned int) b;
   if ( t != 1 ) { __builtin_abort(); }
   return 0;
}
```

(b) Error program that detected a bug in LLVM-3.9.

Fig. 7. Error programs detected by Orange4.

and *for* statements, and (4) deletion of variables and constants, until any of the transformation eliminates the error.

IV. EXPERIMENTAL RESULT

Orange4, a random test system based on the proposed method, has been implemented in Perl 5, which runs on Ubuntu Linux, Mac OS X, Cygwin on Windows, etc. Five versions of GCC and LLVM were tested for 72 hours with options `-O0`, `-O3`, and `-Os`. The target size of the programs in terms of the number of operators was set to 1,000.

TABLE II summarizes the result. Columns "Orange4 (uniform)" and "Orange4 (non-uniform)" show the result by Orange4 with operand values of uniform distribution and non-uniform distribution, respectively. Columns "#test" show the numbers of the programs tested, and "#error" the numbers of the programs that detected errors. In the non-uniform distribution mode, the minimum values are chosen at the probability of 10%, the maximum values at 10%, -1 at 10%, and the other values at 70%. The proposed method (Orange4) with non-uniform distribution succeeded in detecting much more errors than Orange3.

Fig. 7 (a) and (b) are the minimized error programs that detected bugs in the latest development versions of GCC-6.0.0 (20151112 experimental) and LLVM-3.9 (trunk 259289), respectively. When compiled with `-O2` option, the both programs ended up executing `__builtin_abort()` since the generated code computed wrong values for variable `t`. We had reported the errors to Bugzillas of GCC ([1] #68528) and LLVM ([2] #26407), respectively, and the both bugs were fixed. Orange4 also detected an error program with nested for and if statements, which can not be generated by Orange3, on GCC-4.5 (The program is omitted for space limitation).

V. CONCLUSION

This paper has proposed a new method for C compiler random testing based on equivalence transformation on test programs. A test system, Orange4, based on the proposed method detected bugs in the latest versions of GCC and LLVM.

The current version of Orange4 only supports *for* and *if* statements. We are now working on extending Orange4 to generate arrays, structs, function calls etc., to enhance bug detection abilities.

Acknowledgment: Authors would like to thank Mr. M. Nakahashi (currently with Shin Maywa Soft Technologies, Ltd.), Mr. S. Yura (currently with the Ministry of Defence), Mr. A. Hashimoto (currently with Nomura Research Institute, Ltd.), and Ms. K. Fujiwara (currently with NTT Communications Corp.), who were with Ishiura Laboratory of Kwansei Gakuin University, for their help developing Orange4. We would also like to thank all the members of the Ishiura Laboratory for their discussion on this research. This work has been partly supported by JSPS Kakenhi Grant #25330073.

REFERENCES

[1] https://gcc.gnu.org/bugzilla/ (accessed 2016-04-23).

[2] https://llvm.org/bugs/ (accessed 2016-04-23).

[3] Plum Hall, Inc.: The Plum Hall Validation Suite for C, http://www.plumhall.com/stec.html (accessed 2016-04-23).

[4] Free Software Foundation, Inc.: Installing GCC: Testing, http://gcc.gnu.org/install/test.html (accessed 2016-04-23).

[5] T. Fukumoto, K. Morimoto, and N. Ishiura: "Accelerating regression test of compilers by test program merging," in *Proc. SASIMI 2012*, pp. 42–47 (Mar. 2012).

[6] http://github.com/ishiura-compiler/CF3 (accessed 2016-04-23).

[7] X. Yang, Y. Chen, E. Eide, and J. Regehr: "Finding and understanding bugs in C compilers," in *Proc. ACM PLDI 2011*, pp. 283–294 (June 2011).

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

[9] E. Nagai, A. Hashimoto and N. Ishiura: "Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions," *IPSJ Trans. SLDM*, vol. 7, pp. 91–100 (Aug. 2014).

[10] K. Nakamura and N. Ishiura: "Introducing loop statements in random testing of C compilers based on expected value calculation," in *Proc. SASIMI 2015*, pp. 226–227 (Mar. 2015).

[11] M. Nakahashi: "Random testing of C compilers based on derivation of syntax trees of arithmetic expressions" (in Japanese), Master Thesis, Graduate School of Science and Technology, Kwansei Gakuin University (Mar. 2014).

[12] S. Yura, M. Nakahashi, and N. Ishiura: "Random testing for C compilers based on derivation of syntax trees of arithmetic expressions" (in Japanese), in *Proc. 2015 IEICE General Convention*, AS-1-4 (Mar. 2015).

[13] V. Le, C. Sun, and Z. Su: "Randomized stress-testing of link-time optimizers," in *Proc. 2015 International Symposium on Software Testing and Analysis*, pp. 327–337 (July 2015).

[14] V. Le, C. Sun, and Z. Su: "Finding deep compiler bugs via guided stochastic program mutation," in *Proc. ACM International Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 386–399 (Oct. 2015).