Automated Testcase Generation for Numerical Support Functions in Embedded Systems

Johann Schumann[1] and Stefan-Alexander Schneider[2]

[1] SGT, Inc./ NASA Ames, Moffett Field, CA 94035, `Johann.M.Schumann@nasa.gov`
[2] Schneider System Consulting, München, Germany, `sahschneider@gmx.de`

Abstract. We present a tool for the automatic generation of test stimuli for small numerical support functions, e.g., code for trigonometric functions, quaternions, filters, or table lookup. Our tool is based on KLEE to produce a set of test stimuli for full path coverage. We use a method of iterative deepening over abstractions to deal with floating-point values. During actual testing the stimuli exercise the code against a reference implementation. We illustrate our approach with results of experiments with low-level trigonometric functions, interpolation routines, and mathematical support functions from an open source UAS autopilot.

1 Introduction

Modern aircraft, spacecraft, or cars contain a large amount of software that is required to function properly for safe system operation and to accomplish the mission. It is estimated that a modern mid-size car is running more than 100 millions lines of code [1] on potentially more than 100 individual processing units. With the increase of software size and complexity, model-based approaches have found their way into safety-relevant applications in the aerospace and automotive domain. Although extensive analyses can be performed on the model level, a large percentage of the overall development cost for safety-critical software is spent on Verification and Validation (V&V) of the actual code and has become a huge challenge for system integrators and subsystem vendors.

Several prominent standards have been developed that require testing with a specific coverage metric depending on the safety-criticality of the code. For example, *ISO 26262 Road Vehicles* [2] requires testing according to MC/DC (Modified Condition Decision Coverage) for code belonging to Automotive Safety Integrity Level (ASIL) D. For levels A and B, only statement coverage is "highly recommended". Similarly, DO 178-C [3] defines levels A–E, where level A concerns the most critical software that has to be tested to 100% MC/DC coverage.

The application software, in particular, when generated using a model-based tool, requires a large number of low level support routines, which typically include advanced floating point operations (like trigonometric functions, matrices, vectors, or quaternions) as well as support functions for the auto-generated code (e.g., table look-up, interpolation, filters, or integrators). Many embedded system use the Netlib[1] mathematical library, or parts thereof like FDLIBM.[2] Also, John Hauser's SoftFloat[3] is being widely used. Most underlying algorithms, approximations, and tables are based on well-known algorithms [4]. Often such routines are part of the compiler or operating system package. Therefore, they are assumed to be given and correct and their proper testing tends to be ignored.

Because testing of such routines is essential, but manual test case generation is cumbersome and time consuming, we have developed a tool for the automatic generation of test stimuli for small numerical subroutines. In the following, we will first give a description of testcase generation using symbolic execution with Klee. We then describe our tool architecture and discuss iterative deepening of abstractions. To illustrate advantages and limitations of our tool, we present results of experiments on trigonometric subroutines, table lookup, and a set of low-level mathematical support functions for an open source autopilot.

2 Automatic Testcase Generation

The input to our tool is a support function $o = f(x_1, \ldots, x_m)$ implemented[4] in C or C++. The tool generates test stimuli, i.e., a set of vectors with concrete values $\langle x_1^i, \ldots, x_m^i \rangle$ that, when given as parameters to f, will fully cover the code of f. For testing, we use the test stimuli to exercise f, compare the calculated result o against a reference implementation and measure the code coverage according to the required coverage metric using an external tool.

Since we test against a reference implementation and do not use the output of our tool as an oracle, soundness of stimulus generation tool is not required. Tool unsoundness, however, can lead to an increased number of unnecessary test stimuli, decreasing testing performance.

Due to the requirement of handling floating-point values, the testcase generation has to be incomplete in general. Our tool architecture uses iterative deepening over abstractions to accomplish a reasonably complete set of test stimuli. We obtain the actual coverage by using an external trusted tool.

For the testcase generation, we use Klee,[5] which is a symbolic execution engine based upon the LLVM framework.[6] It exhaustively explores all paths of the code; variables of interest (in our case, x_i) are treated as symbolic values, and each path is represented by a path constraint. For example, the code fragment `if (x<0||x>10) A; else B;` produces three distinct path constraints: $\langle [x < 0] : A \rangle$ means that A can be reached by making the first condition true; similarly for $\langle [x > 10] : A \rangle$, the second condition must be true. Finally, the path constraint $\langle [\neg(x < 0) \wedge \neg(x > 10)] : B \rangle$ reaches B. Solving each path constraint leads to a set of test stimuli, for example, $\{-1, 11, 5\}$. Here, 3 test cases are needed for full

[1] `http://netlib.org`
[2] `http://www.netlib.org/fdlibm`
[3] `http://jhauser.us/arithmetic/SoftFloat.html`
[4] This code can also include calls to initialize objects or data structures.
[5] `klee.llvm.org` or [5]
[6] `http://www.llvm.org`

path coverage; statement coverage only requires two stimuli, e.g., $\{-1, 5\}$. KLEE uses the powerful STP[7] solver to find solutions for the path constraints. However, KLEE only provides very little support for floating point numbers; in most cases, KLEE silently instantiates the variable with a random value. KLEE-FP [6] has been designed to reason about equivalence of floating point numbers and is not suitable for this task. Yet, we chose to use KLEE, because it can handle the full C/C++ syntax and provides support for bitwise operations, which is essential for our purposes.

Our tool architecture and process is depicted in Figure 1. Starting with code under test P, which implements the function $o = f(\cdot)$ in one or more syntactic procedures, and an initial set of parameters $d = 0$, a parameterized abstraction is generated and applied to P. In this abstraction, all variables of type `float` or `double` are converted to integers. Each floating point constant c is represented as $sign(c)\lfloor\min(maxint, |c| \times 10^d)\rfloor$. We chose a base of 10 because then the abstraction can be done on the source code by simply moving decimal points. Embedded function calls to other low-level routines (e.g., sqrt, sin) are abstracted by simple Taylor series or table lookup. Since most of the results of floating point operations in P do not show up in equality comparisons in conditional statements, our abstraction is often successful by using this fixed-point abstraction with d decimal places. Additional abstraction parameters define, how often P is invoked during each test—an important step for testing reentrant functions like filters. The abstracted code P_A is processed by KLEE, which returns a set of (abstracted) test stimuli T_A. They might cover all paths in P_A or only a subset if KLEE timed out. We translate T_A into actual test stimuli and use them to exercise P; coverage is measured on the original code P. If we are not satisfied with the results, the parameters controlling the abstraction are incremented and the iterative deepening loop starts again.

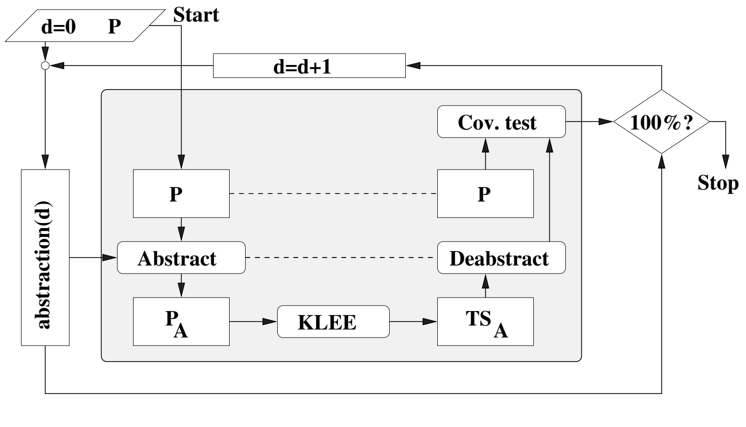

Fig. 1. Tool architecture

[7] `http://people.csail.mit.edu/vganesh/STP_files/stp.html`

3 Experiments

In this section, we describe selected experiments with this tool and discuss findings, advantages, and limitations of our approach.

Trigonometric Functions. Functions to calculate trigonometric functions are often considered part of the operating system or compiler. However, for small embedded systems, such functions must be provided externally and must be tested accordingly. As an example, consider a standard implementation (e.g., [8]) of the trigonometric function `double sin(double x)`. In a first step,[8] the input x is broken down into its components (exponent, mantissa, and sign) according to IEEE 754 [7] using a C union and bit-fields. After handling cases for infinity, NaN, and very small argument values, x is normalized to $[0 \ldots \pi/2]$, and the quadrant is determined. Finally, the function value is approximated by a 7th order polynomial. A complex algorithm for multiplication without loss of accuracy is used (see [8], [9]). Multiple macro definitions are used to handle machine-dependent issues. Although there are no loops in this code, there is a substantially complex control flow with 10 nested if-then-elses and one switch statement with 4 cases and an empty default label. Such a code structure makes a manual development of test cases hard.

With our tool, we generated a total of 44 test stimuli in less than 10s CPU time on an Intel Macbook Pro. This set of stimuli also contain NaN and Inf, which are encoded according to IEEE 754 by specific settings of mantissa and exponent bits. Several iterations of abstractions resulted in $d = 7$. Due to technical restrictions of KLEE, it also was necessary to pass two 32bit integer values instead of one 64bit double to the function.

When executing the generated test stimuli, two interesting observations could be made: (1) a comparison of the calculated values against the standard Mac OSX implementation revealed that, while the error between this code and the reference was in general between 10^{-11} and 10^{-18}, two test stimuli caused errors that were larger than 3×10^{-6}, which might give raise to some concern. (2) a detailed analysis of the results with the industry-standard testing and coverage tool LDRA[9] revealed that this piece of code, which is actually a part of a commercial distribution, contains dead code. The empty default label in the switch statement can never be reached due to the range of the argument. Thus no test set can produce 100% MC/DC coverage, a fact which makes one wonder if that routine was ever tested according to that metric.

Interpolation Table. One of the most common block types in model-based systems like Simulink is the table lookup or 1-D interpolation block. Given an input u, it calculates an approximation of $f(u)$, whereby values of $f(x)$ for monotonically increasing values of x are given statically as a table (see Figure 2A for a code sketch). We have analyzed a generic version of an 1-D table lookup, which is somewhat similar to Mathworks' `rt_look.c`.[10] After checking for boundary cases, a binary search is used to find the appropriate indices into the table.

[8] See suppl. material `ti.arc.nasa.gov/profiles/schumann/publications/nfm2014`

[9] `http://ldra.com`

[10] `rtw_demos/rt_look.c` is found in Mathworks' distribution of RealTime Workshop.

We used our tool to generate test stimuli for two relevant scenarios: (1) given a concrete lookup table $\langle x, f(x)\rangle_{1..len}$, find values for u such that all paths are covered. E.g., for $x = \langle -2, 0, 3, 5, 8\rangle$, and f the identity function, the following six test cases for u are generated in less than 0.1s: $u \in \{-2147483648, -1, 0, 2, 4, 6, 8\}$. Here, $d = 1$ was sufficient to obtain full coverage. In general, the necessary value of d depends on the minimal difference $\Delta = x_{i+1} - x_i$. In the abstracted program Δ must be at least 2 in order to trigger the divide-and-conquer algorithm. This requires that $d \geq \log_{10}\lceil \min_i(x_{i+1} - x_i)\rceil$. In scenario (2), given the desired length len of the interpolation table, triples $(\langle x, f(x)\rangle, u)$ with $length(x) = len$ are generated such that the code is fully covered. Note that the values of x must be increasing monotonically. Therefore, the additional constraint $x_1 < x_2 \ldots$ must be specified in the test driver. Figure 2B shows, for different values of len, the number C_0 of all generated stimuli and the number C of stimuli that obey our constraint and can be used as proper test stimuli.

```
double lookup(double *x, double *f, int len, double u){
if (u <= x[0]) return f[0];      // outside the table (left)
else if (u >= x[len-1]) return f[len-1]; // outside (right)
else
  for(;;){    // do binary search
    _assert( (x[bot] < u) && (u < x[top]));
    ind = (bot + top)/2;     // find middle
    if (...)
        top = ...; bot = ...
    else
        return f[ind];
    } }
```

len	C_0	C	t[s]
5	15	11	0.3
10	32	9	1.2
20	66	19	4.6
30	96	29	10.2
100	330	231	130

Fig. 2. A: code sketch of interpolation routine, B: number of generated test stimuli

All results have been obtained with the assertion `_assert` (Fig. 2A) turned off. When activated, it is textually replaced by a conditional statement that aborts the execution if the condition is not met. Interestingly, KLEE could still find a full coverage test set. This indicates that there exist stimuli u, which, for a given table x, cause the abortion of the execution. In an embedded system, such a behavior could have disastrous consequences. A closer look at the code reveals that the actual binary search loop is correct, but the assertion in `rt_look.c` is wrong (R2014a and earlier).

ArduPilot. ArduPilot[11] is an open source project aiming to provide high quality code for a simple autopilot for small fixed wing or rotorcraft UAVs, RC cars, or model boats. Ardupilot is implemented in C++ and runs on the Arduino platform.[12] Its mathematical libraries contain numerous functions dealing with trigonometric functions (via table lookup), vectors, matrices, quaternions, and filters. We used our tool to generate test stimuli for a number of those functions, leveraging the fact that KLEE can work on C++ code with templates. Although the code for each function is short and usually does not contain any loops, the presence of (nested) conditional statements makes our tool convenient for the task of testcase generation. In our experiments, we generated between 2 and more than a hundred test stimuli (e.g., 116 for a function, which determines if a

[11] `http://code.google.com/ardupilot-mega`

[12] `http://arduino.cc`

point is inside or outside a closed polygon with 7 edges).

4 Conclusions and Future Work

We have presented a tool for the automatic generation of test stimuli for small numeric support functions. Based upon KLEE, it uses iterative deepening over abstractions to deal with floating point operations. Because in practically all examples we analyzed so far, the results of floating point operations in P do not show up in equality comparisons, our abstraction is often successful in producing a sufficient set of test stimuli. Although our tool has been able to conveniently and automatically generate test stimuli for a number of small, but often "tricky" numerical support routines, our approach still has several shortcomings. For example, configuration parameters and #define macros or template parameters (e.g., length of a filter buffer) currently cannot be treated symbolically and thus cannot be varied by our tool. Furthermore, preparation and abstraction of the code has not been fully automated yet, and support for writing test drivers and test scripts with symbolic variables is still very primitive. Obviously, scalability is an issue with larger programs, or programs, which contain nested loops (e.g., matrix operations). There, the restriction to MC/DC coverage to substantially reduce number of explored paths and generated stimuli and an abstraction for loops or the ability to modify KLEE's behavior on generating path conditions should be investigated.

References

1. Charette, R.: This car runs on code (2009) `http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code`
2. Intl. standard ISO 26262 Road Vehicles – functional safety 1st ed (2011)
3. RTCA: DO-178C: Software considerations in airborne systems and equipment certification (2011)
4. Hart, J.F., Cheney, E.W., Lawson, C.L., Maehly, H.J., Mesztenyi, C.K., Rice, J.R., Thacher, J.H.G., Witzgall, C.: Computer Approximations. SIAM Series in Applied Mathematics. John Wiley and Sons (1968)
5. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX Symp on Operating Systems Design and Implementation, OSDI. (2008) 209–224
6. Collingbourne, P., Cadar, D, Kelly, P.: Symbolic Crosschecking of Floating-Point and SIMD Code. In: EuroSys (2011).
7. IEEE standard 754 for floating-point arithmetic (2008)
8. Overton, M.L.: Numerical computing with IEEE floating point arithmetic - including one theorem, one rule of thumb, and 101 exercises. SIAM (2001)
9. Huckle, T., Schneider, S.A.: Numerische Methoden: Eine Einführung für Informatiker, Naturwissenschaftler, Ingenieure und Mathematiker. Springer (2006)
10. Giannakopoulou, D., Bushnell, D.H., Schumann, J., Erzberger, H., Heere, K.: Formal testing for separation assurance. Annals of Mathematics and Artificial Intelligence **63** (2011) 5–30

Automated Testcase Generation for Numerical Support Functions in Embedded Systems (Supplemental Material)

Johann Schumann[1] and Stefan-Alexander Schneider[2]

[1] SGT, Inc./ NASA Ames, Moffett Field, CA 94035, Johann.M.Schumann@nasa.gov
[2] Schneider System Consulting, München, Germany, sahschneider@gmx.de

Abstract. This document contains supplemental material only.

	Methods	**ASIL**			
		A	B	C	D
1a	Statement Coverage	++	++	+	+
1b	Branch Coverage	+	++	++	++
1c	MC/DC Coverage	+	+	+	++

Table 1. Different code coverage metrics according to ISO *26262 Road vehicles – Functional safety*, 6–9, Table12. The symbol ++ indicates highly recommended and + recommended for the identified Automotive Safety Integrity Level (ASIL).

Listing 1.1. Memory format for IEEE 752 double floating point numbers and data structure for bitwise access.

```
union ieee_bits {
  double d;  // double
  struct {   // mantissa+exponent
    unsigned int m2:32;    // mantissa2
    unsigned int m1:20;    // mantissa1
    unsigned int exp:11;   // exponent
    unsigned int sign:1;
    } b;
  struct {
    unsigned int i1;      // 32 bit package
    unsigned int i2;
        } i; };
```

Listing 1.2. The pseudo-code for the argument qualification and polynomial approximation of the numerical support function $(double)\sin((double)\,x)$.

```
double sin (double x) {

md.d = x;                 // break the argument up into parts
xexp = (int) md.b.exp;
                          // handle boundary conditions
if (xexp == IEEE_MAX) {        // x is Not-a-Number or infinity
   if (md.b.m1 || md.b.m2)
      return x;           //  x is Not-a-Number
   else
      return NaN;         // 0.0/0.0
   } else if (xexp == 0) // x is infinity or denormalized
      return x;
   else if (xexp <= (IEEE_BIAS - IEEE_MANT - 2))
      return x - x*x;          // x is very small
   else if (xexp <= (IEEE_BIAS - IEEE_MANT/4))
      return x - x^3/6;    // x is small
   }
if (x < 0) {
      set sign to 1; // negative
      x = - x;
   }
// map to range of argument to x <= pi/2
if (xexp < IEEE_BIAS) { // 2^52/4 < x < 1
   skip;                // x already < pi/2
else if (xexp <= (IEEE_BIAS + IEEE_MANT)) {
     xm = b * 2_pi_hi + 1/2c
     xn.d = xm + mag52
     bot2 = xn.b.m2 & 3u
     // split xm into top 26 and bottom 26 bits
     split(a1, a2, xm);
     exactmul2(x3,x4, xm,a1,a2, pi2_hi,pi2_hi_hi,pi2_hi_lo);
     exactmul2(x5,x6, xm,a1,a2, pi2_lo,pi2_lo_hi,pi2_lo_lo);
     x = ((((x - x3) - x4) - x5)- x6) - xm*pi2_lo2;
            // reduce to 0 <= x <= pi/2

     switch(bot2) {
       case 0: if (x < 0.0) { x = -x; sign ^= 1; } break;
       case 1: if (x < 0.0) { x = pi2_hi + x; } else { x = pi2_hi - x; } break;
       case 2: if (x < 0.0) { x = -x; } else { sign ^= 1; } break;
       case 3: sign ^= 1; if (x < 0.0) {x = pi2_hi + x;} else {x = pi2_hi - x;}break;
       default: ;
       }
   } else { // 2^53 <= x
     return LOSS;
   }
x = x* _2_pi_hi;// map to range 0 <= x <= 1
if (x > X_EPS) {
   x2 = x*x;
   x = POLYNOM(x2); // Horner 7th degree
   } else {
   x = x * pi2_hi;
   }
if (sign) x = -x;
return x;
}
```

$P[.]$	
$P[0]$	0.15707963267948963959e1
$P[1]$	-0.64596409750621907082
$P[2]$	0.79692626245618008 06e-1
$P[3]$	-0.468175413106023168e-2
$P[4]$	0.16044116846982831e-3
$P[5]$	-0.359880911703133e-5
$P[6]$	0.5688203332688e-7
$P[7]$	-0.64462136749e-9

Table 2. Coefficients for the 7-the degree polynomial used within the sin algorithm. The polynomial is evaluated using a Horner schema.

Listing 1.3. Simplified Pseudo-Code for Table Lookup.

```
double lookup(double *x, double *f,
  int len, double u){

if (u <= x[0]) // outside the table (left)
  return f[0];
else if (u >= x[len-1]) // outside (right)
  return f[len-1];
else
  for(;;){    //do binary search
    _assert( (x[bot] < u) && (u < x[top]));
    ind = (bot + top)/2;
    if (...
        top = ...
        bot = ...
    else
        return f[ind];
    ...
    }
}
```

Listing 1.4. KLEE test driver for 1D table lookup

```
main(){
  const static int x[] = {-2,-1,0,1,2};
  const static int F[] = {0,1,2,3,4};
  const int len = 5;
  int u, val;

  klee_make_symbolic(&u, sizeof(u), "u");
  val = lookup(x,F,len,u);
}
```

Listing 1.5. Code fragment for monotonicity constraint

```
if(x[0] < x[1] && x[1] < x[2] && ..)
  r=lookup(..);
else
  r=-1;
```

n	x_n	E_{ref}
1	0.0488281250	1.9e-11
2	0.048828145354091221	1.9e-11
3	590295810358705651712.0	N/A
4	137438954176.75451660156250	0
5	131072.02110725082457065582	1.1e-16
6	1.021675541361788	0
7	2048.0	5.6e-17
8	2097152.28180931787937879562	1.1e-16
9	1048576.04403630522638559934	1.1e-16
10	512.011448051895968092	0
11	8192.0103308404868585058	1.1e-16
12	8192.0195236411855148617	0
13	2097152.13230490731075406075	0
14	68719477974.53822326660156250	0
15	68719484038.4492340878906250	1.1e-16
16	1.022142187527940	0
17	8192.0189697450514358934	0
18	137438958099.9297790527343750	3.5e-18
19	32.070940668450703	1.1e-16
20	-0.0488281250	1.9e-11
21	-0.048828146286775316	1.9e-11
22	-590295810358705651712.0	N/A
23	-1.017512665761998	1.1e-16
24	-4096.026914338559436146	1.1e-16
25	-2199023747489.17871093750	1.1e-16
26	-137438977261.382751464843750	0
27	-2097152.39780779741704463959	2.2e-16
28	-2097152.28940287604928016663	1.1e-16
29	-2097152.37323756702244281769	0
30	-524288.0	0
31	-137438963969.7497253417968750	3.4e-06
32	-68719489512.50517272949218750	3.8e-06
33	-1.01257808057709	0
34	-1.017452079403846	1.1e-16
35	-33554434.43163714557886123657	0
36	-32.0541842897661127	1.1e-16
37	-524288.05263395013753324747	5.6e-17
38	-1.0749030579782	1.1e-16
39	0.044409	0
40	0.0	0
41	0.0	0
42	inf	NaN
43	NaN	NaN
44	NaN	NaN

Table 3. Generated Testcases for the *sin* support routine with $d = 7$. n is the testcase number, x_n the test stimulus, and error $E_ref = \sqrt{(\sin x_n - \hat{\sin} x_n)^2}$ with respect to reference implementation $\hat{\sin}$ on a Macbook Pro (Mac OSX 10.6.8).

Given $\boldsymbol{x} = \langle x_0, x_1, ..., x_{n-1} \rangle$ monotonically increasing and $\boldsymbol{F} = \langle f(x_0), \ldots, f(x_{n-1}) \rangle$
1. if u is outside the range of $\boldsymbol{x}$, return $f(x_0)$ or $f(x_{n-1})$,   2. otherwise, determine the index $0 \leq i < n$ such that $x_i \leq u \leq x_{i+1}$, and   3. calculate the table lookup value as $(f(x_i) + f(x_{i+1}))/2$ or by linear interpolation.

Table 4. High-level description of 1D table lookup

#	u	$\hat{x}$
1	0	-2147483648, -2147483647, -2113929216, 0, 1
2	4098	4, 6, 7, 4096, 4099
3	264	64,265,291,8481,131337
4	10	-1610612732, -1610612730, 10, 536870920, 536870922
5	-2147483639	-2147483648, -2147483646, -2147483639, -1610612728, -1610612727
6	-2147483643	-2147483648, -2147483646, -2147483645, -2147483644, -2130706427
7	-2147479552	-2147483648, -2147483646, -2147483645, -2147479552, -2147479551
8	-2147481472	-2147483136, -2147481471, -2147481464, -1610602368, 262273
9	-2147483647	-2147483648, -2147483647, 0, 4, 6
10	6	0, 1, 2, 3, 6
11	0	0, 1, 4, 6, 7

Table 5. Generated stimuli for $n = 5$. All paths of the code are fully covered by this test set.

n	C_0	C	t[s]
5	15	11	0.3
10	32	9	1.2
20	66	19	4.6
30	96	29	10.2
100	330	231	130

Table 6. The number of test cases generated for different lengths of the lookup table. C_0 is the number of test cases generated by KLEE, C is the number of valid test cases (i.e., vectors with increasing values), and run-time t for their generation.