```
def ex7(inpt):
        x = 0
        n = 10  # N This is a forall.
        y = 10  # N This is a forall.
        prob = inpt[0] # This is a forall.
        while(n > 0):
                d = bernoulli.rvs(size=1, p=prob)[0]
                if(d):
                        x = x + y # x is probabilistic here
                n = n-1
        return x

# Expected Value of "x"
# E[x] = prob * n * y
assert((P(x - (prob * n * y)) <= 0) <= 0.6)
```

Fig. 1. Program listing for finding probabilistic `assert` violation.

```
(0.0000001 < prob_sym),
(1 <= y_sym),
(prob_sym <= 1),
(10 <= n_sym)
```

Fig. 2. Constraints on `Path-1` where control flow is in the `while` loop.

```
(0.0000001 < prob_sym),
(1 <= y_sym),
(prob_sym <= 1),
n_sym == 0 # Exit from Loop
```

Fig. 3. Constraints on `Path-2` where control flow exits the `while` loop.

```
double prob;
int x, n, y;

// forall variables
klee_make_symbolic(&y, sizeof(y), "y_sym");
klee_make_symbolic(&n, sizeof(n), "n_sym");
make_pse_symbolic(&prob, sizeof(prob), "prob_sym", 0, 1);

// Some restrictions on the values.
klee_assume(y >= 0);
klee_assume(10 <= n);

// Work around for silent concretization of prob.
prob = 0.5;

// Prob Sym Variables.
make_pse_symbolic(&x, sizeof(x), "x_pse_sym", 0, 2147483647);

std::default_random_engine generator;
std::bernoulli_distribution bernoulli_rvs(prob);

// This is the "inner" loop which affects the final value of "x".
while (n > 0)
{
        int d = bernoulli_rvs(generator);
        if (d)
                x = x + y;
        n = n - 1;
}

return 0;
```

Fig. 4. KLEE Transformation of the program for testing the assert.

```
long double prob;
int x, n, y, n_start, y_start;
long long unsigned int win = 0, loop_run = 0, prob_runs = 0;

scanf("%d", &n);
scanf("%d", &y);
scanf("%Lf", &prob);

n_start = n;
y_start = y;

while (termCount-)
{
        x = 0;
        int n_loop = n; // maintain "n" for re-runs of loop.

        // program execution
        std::default_random_engine generator;
        std::bernoulli_distribution bernoulli_rvs(prob);

        // "n" gets changed, we need to maintain the loop.
        while (n_loop > 0)
        {
                int d = bernoulli_rvs(generator);
                if (d)
                {
                        // This doesn't favours the "win" condition.
                        x = x + y;
                }
                n_loop -= 1;

                // Iterations of the inner loop.
                // How many times the probabilistic loop gets executed?
                prob_runs++;
        }

        // Sample for probability. This is the assert condition.
        if ((double)x - (prob * n * y) <= 0)
                win++;

        // No. of times the program gets executed.
        loop_run++;
}
```

Fig. 5. Testing `Translate()` of the program for testing the `assert`.

Table 1. Comparing prob(p) value Vs assert status for low values of prob(p) shown for limited iterations.

prob-value	Program Runs	Loop Runs	Wins	P((x - (prob * n * y)) <= 0)	n	y
0.0000001298	100000	5600000	100000	1.000000	56	21
0.0000001596	100000	4000000	100000	1.000000	40	17
0.0000003384	100000	3600000	100000	1.000000	36	28
0.0000039147	100000	2800000	100000	1.000000	28	18
0.0000077294	100000	2300000	100000	1.000000	23	24
0.0000153588	100000	2700000	100000	1.000000	27	16
0.0000306176	100000	2100000	100000	1.000000	21	15
0.0000611352	100000	2200000	100000	1.000000	22	14
0.0001221703	100000	2000000	100000	1.000000	20	11
0.0002442406	100000	2600000	100000	1.000000	26	10
0.0004883812	100000	1900000	100000	1.000000	19	13
0.0009766624	100000	1800000	100000	1.000000	18	9
0.0019532248	100000	2500000	100000	1.000000	25	12
0.0039063496	100000	2400000	100000	1.000000	24	8
0.0078125992	**100000**	**1700000**	**0**	**0.000000**	**17**	**7**
0.0156250984	100000	1600000	0	0.000000	16	5
0.0312500969	100000	1400000	0	0.000000	14	6
0.0625000937	100000	1300000	0	0.000000	13	4
0.2500000750	100000	1100000	0	0.000000	11	2
0.5000000500	100000	1000000	0	0.000000	10	1

```
(0.0075243180 < prob_sym),
(1 <= y_sym),
(prob_sym <= 0.0078125992),
(10 <= n_sym)
```

Fig. 6. Narrowing constraints on Path-1 where control flow is in the while loop.

Table 2. Comparing prob(p) value to cases Vs assert status for low values of prob(p) on narrowing

prob-value	Program Runs	Loop Runs	Wins	P((x - (prob * n * y)) <= 0)	n	y
0.0068589250	100000	2500000	100000	1.000000	25	12
0.0068665544	100000	1600000	100000	1.000000	16	5
0.0068970719	100000	1400000	100000	1.000000	14	6
0.0069581071	100000	1300000	100000	1.000000	13	4
0.0070801774	100000	1200000	100000	1.000000	12	3
0.0073243180	100000	1100000	100000	1.000000	11	2
0.0078125992	**100000**	**1700000**	**0**	**0.000000**	**17**	**7**
0.0156250984	100000	1600000	0	0.000000	16	5

Table 3. Comparing prob(p) value to cases Vs assert status for low values of prob(p) on further narrowing

prob-value	Program Runs	Loop Runs	Wins	P((x - (prob * n * y)) <= 0)	n	y
0.0075603531	100000	1300000	100000	1.000000	13	4
0.0075963883	100000	1200000	100000	1.000000	12	3
0.0076684586	100000	1100000	100000	1.000000	11	2
0.0077243207	100000	2100000	0	0.000000	21	24
0.0077243234	100000	2800000	0	0.000000	28	16
0.0078125992	**100000**	**1700000**	**0**	**0.000000**	**17**	**7**