Test Harnesses for Pointer-based C Programs

Jakob Selberg

October 2020

Table of Contents

Goals

- Why test from witnesses: supply an accessible interface to verification results
- My contribution: extending CPAchecker's existing test generation functionality to pointer based programs.

Input and output of CPAchecker

input

- C Program
- Specification e.g. any call to function verifier_error() unreachable

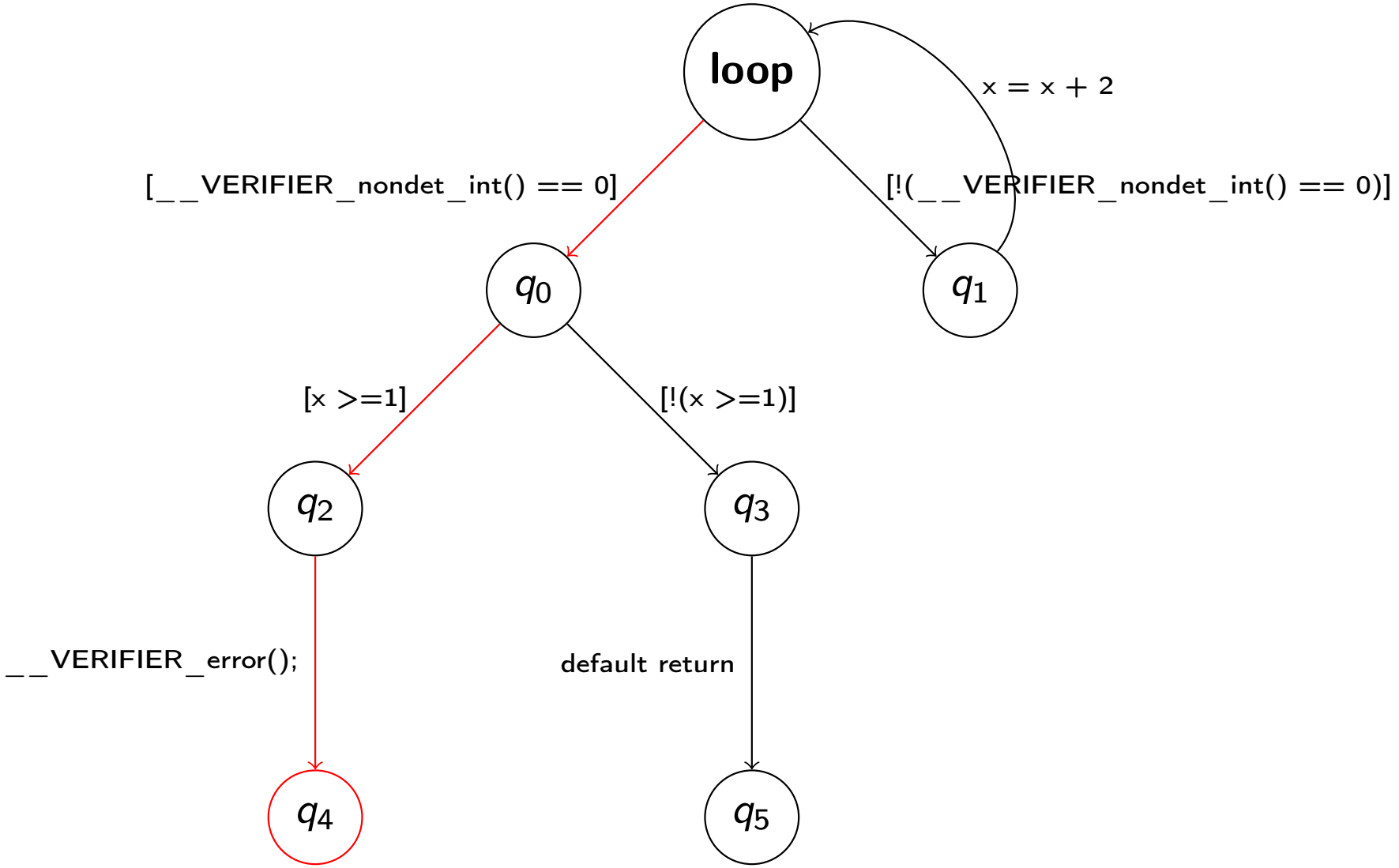
output:

- if violation of spec was found: violation witness

Example of a simple C Program

```
1  extern void __VERIFIER_error(void) ;
2  extern int __VERIFIER_nondet_int(void) ;
3  extern void __VERIFIER_int_sink(int i);
4  int main () {
5      unsigned int x = 1 ;
6      __VERIFIER_int_sink(x);
7      while ( __VERIFIER_nondet_int() ) {
8      x = x + 2;
9      }
10     if ( x >= 1 ) __VERIFIER_error() ;
11 }
```

Violation Witness



Input and output of test programs

- Input function: __VERIFIER_nondet_int() : unimplemented function, return values non-deterministic
- Output function: __VERIFIER_int_sink(x): unimplemented function, no return value, takes parameters

Drawbacks of violation witnesses

Working with a violation witness:

- can often contain an unreachable violation
- can be very complex
- developer has to learn to read them
- cant use any familiar tools for investigation

Solution: Tests from witnesses

Idea

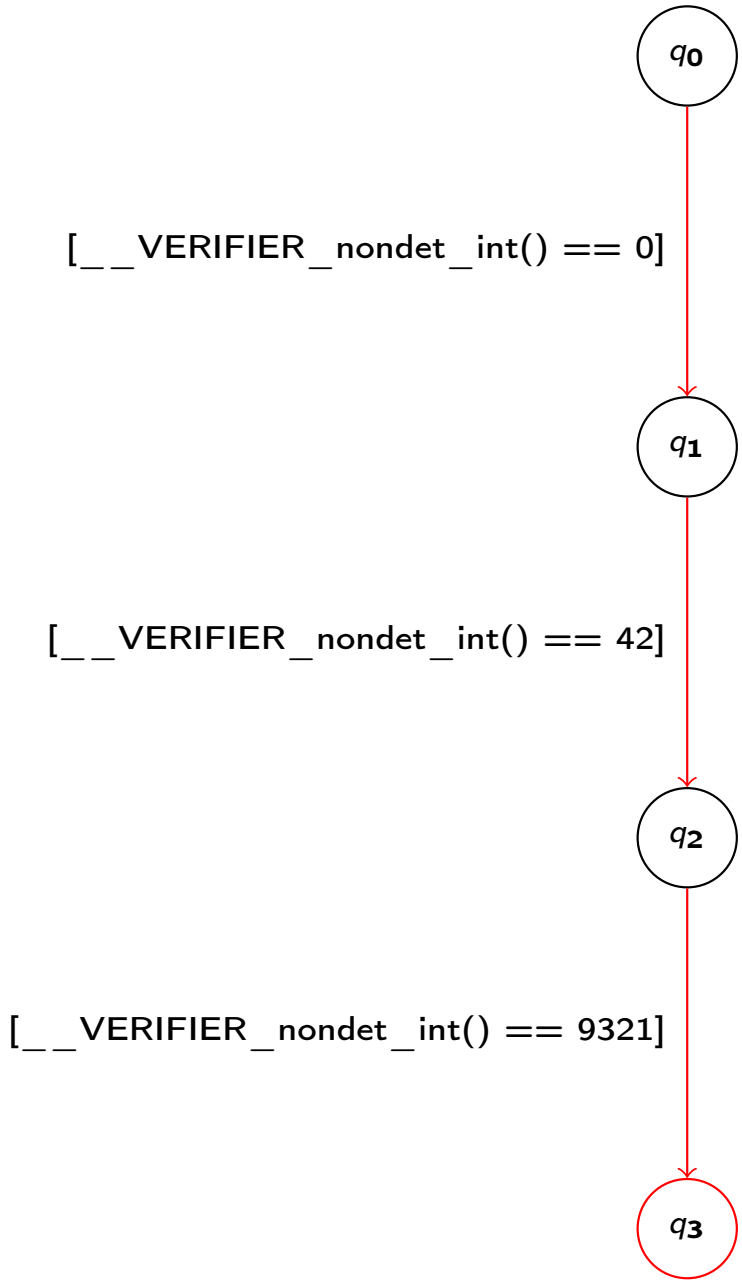
- error path in witness holds assumptions about input values
- let's implement the input function to return exactly those values in that order
- compiling the original program with that implementation should always lead to the witnessed violation

Advantages of tests over witnesses

Advantages:

- successful test generation rules out spurious violations
- can be analyzed and debugged with the tools and methods familiar to developers

Existing implementation of test harness generation




```
1  unsigned int __VERIFIER_nondet_int_index__ = 0;
2  int __VERIFIER_nondet_int() {
3     switch (__VERIFIER_nondet_int_index__) {
4       case 0: retval = 0; break;
5       case 1: retval = 42; break;
6       case 2: retval = 9321; break;
7     }
8     ++__VERIFIER_nondet_int_index__;
9     return retval;
10 }
```

Limitations of current implementation

Limitations of this approach:

- error-path assumptions must only include types that can be assigned literal values

```
1 |     int* i = __VERIFIER_nondet_pointer();
2 |     int* j = __VERIFIER_nondet_pointer();
3 |     if(i == j) {
4 |         __VERIFIER_error();
5 |     }
```

- cannot handle violations dependent on undefined or unspecified behavior. E.g. depending on the order of evaluation of function parameters.

```
1 |     int i;
2 |     foo(setIToZero(), setIToOne());
3 |     if(i == 1) {
4 |         __VERIFIER_error();
5 |     }
```

We addressed the first limitation and extended the functionality to handle pointer types as well

C Types

literally assigned: int, char, etc.
int a = 5;
char c = 'A';

not literally assigned: derived types such as pointer to, array of
int* i = malloc(sizeof(int));
int** q = &i;
explicit value cannot be known at compile time

Problem

if we can't know the explicit value of an expression, we can't implement it as a return value of our switch-case function

Handling pointer types

Central assumption

Only those memory locations can be given as input, that have previously been given as output. I.e. those that have been parameters to calls to output functions.

Naive approach

A CPA with:

- explicit tracking of all pointers
- partition pointers into equivalence classes
- collection of all memory locations given as output
- merge pointer equivalence classes at equivalence assumptions

Example of naive implementation state information

```
1  #include <stdlib.h>
2
3  extern void output(int* p);
4  extern int* input(void);
5
6  int main() {
7    int *p = malloc(sizeof(int));
8    output(p);
9    int *q = input();
10   if (p == q) {
11     __VERIFIER_error();
12   }
13   return 0;
14 }
```

```
1
2
3
4
5
6
7  (p,m1)
8  (p,m1),[m1]
9  (p,m1)(q,i1,[m1]),[m1]
10 (p,q,i1,m1), [m1]
11
12
13
14
```

Problems with naive approach

- must track every single addressable memory location (variables, struct fields), lots of useless information
- High computational and memory cost
- duplicates existing, more sophisticated functionality, e.g. PredicateCPA
- PredicateCPA is a powerful analysis, much more likely to be improved/maintained in the future

PredicateCPA-based implementation

CPA that collects memory locations given to output functions, and leverages state information from the PredicateCPA to resolve aliasing information

PredicateCPA

Implements predicate abstraction analysis. Its state consists of an abstraction formula and a concrete *path formula*, we can use the path formula to infer equality of pointer type variables.

Example C Program

```
1  extern void __output(int* p);
2  extern int* __input(void);
3
4  int main() {
5    int *k = malloc(sizeof(int));
6    __output(k);
7    int *p = malloc(sizeof(int));
8    __output(p);
9    int *q = __input();
10   int *r = __input();
11   if (k == r) {
12           if (p == q) {
13                   __VERIFIER_error();
14           }
15   }
16   return 0;
17 }
```

Path formula of the program

$$(p_3 = r_3) \wedge (p_3 = q_3) \wedge (q_3 = \_\_input_1) \wedge (r_3 = \_\_input_0) \wedge ....$$

Using the pathFormula

To use this information we do two things:

- Conjunct it with the information we have gathered about memory locations given as outputs
- use SMT-solver to resolve formulas that equate each return value of our input functions to the n-th memory location given as output, for n

function *"index"*: denotes the index of a memory location in the array of all memory locations given as output.
index(q@1) = 0 means that q@1 was the first memory location given as output.

Example of state information, and index formula

path formula conjunction with index function:

$$(p_3 = r_3) \wedge (p_3 = q_3) \wedge (q_3 = \_\_input_1) \wedge (r_3 = \_\_input_0) \wedge (index(p_3) = 0) \wedge (index(q_3) = 1)$$

resolve for index returned by nth call to input function: e.g. resolve $index(input@1)$

Frame Title

Program Statement	HarnessCPA	PredicateCPA
extern void foo(int* c);	$\{\}$	$\{\}$
extern *int bar();	$\{\}$	$\{\}$
int* p = malloc(sizeof(int))	$\{\}$	$\{\}$
foo(p);	$\varphi(p_0) = 0$	$\{\}$
int* q = malloc(sizeof(int));	$\varphi(p_0) = 0$	$\{\}$
foo(q);	$\varphi(p_0) = 0 \wedge \varphi(q_0) = 1$	$\{\}$
int* r = bar()	$\varphi(p_0) = 0 \wedge \varphi(q_0) = 1$	$\{r_0 = bar_0\}$
if(r==q) error();	$\varphi(p_0) = 0 \wedge \varphi(q_0) = 1$	$\{r_0 = bar_0, r_0 = q_0\}$

Implementing the input functions

Analogous to the implementation for normal types:

- use a switch-case statement over n for the n-th call
- store all pointers in an array
- retrieve them from the position we learned from the resolution of the index function

Program Example 1

```
1  int main() {
2  int i = 5;
3  int arr[1];
4  int* p;
5  foo(arr);
6  p = bar();
7
8  if(p == arr) {
9    __VERIFIER_error();
10 }
11 return 0;
12 }
```

Partial Test Harness for Program Example 1

```
1  void* HARNESS_externPointersArray [1];
2  unsigned long int bar_ret_counter = 0;
3  int *bar(){
4    int * retval;
5    switch(bar_ret_counter) {
6          case 0: retval = (int *) HARNESS_externPointersArray
               [0]; break;
7    }
8    ++bar_ret_counter;
9    return retval;
10 }
```

Program Example 2

```
1   foo(a);
2   foo(b);
3   foo(c);
4   foo(d);
5   p = bar();
6   q = bar();
7   r = bar();
8   s = bar();
9
10  if (p == d) {
11    if(q == a) {
12      if(r == c) {
13        if(s == b) {
14          __VERIFIER_error();
15        }
16      }
17    }
18  }
```

Partial Test Harness for Program Example 2

```
1  void* HARNESS_externPointersArray [4];
2  unsigned long int bar_ret_counter = 0;
3  int *bar(){
4    int * retval;
5    switch(bar_ret_counter) {
6          case 0: retval = (int *) HARNESS_externPointersArray
               [3]; break;
7          case 1: retval = (int *) HARNESS_externPointersArray
               [0]; break;
8          case 2: retval = (int *) HARNESS_externPointersArray
               [2]; break;
9          case 3: retval = (int *) HARNESS_externPointersArray
               [1]; break;
10   }
11   ++bar_ret_counter;
12   return retval;
13 }
```

Evaluation and Conclusion

- Tested on svcomp-19 programs: unfortunately no improvement
- Future work: find out why test generation fails so often
- If likely to be useful, handle some undefined/unspecified behavior