

Random Code Repairs

Adam Fuchs and Khoo Yit Phang

Introduction

- Pre- and post-conditions are used to help ensure program correctness
- Buffer overflows can be prevented by making an assertion that the array index is within the bounds of the allocated memory for the array
- Symbolic execution and/or abstract interpretation can be used to check assertions statically
- What do you do if the program doesn't match its specifications?

Employ lots of monkeys!

It was the best of times, it was the blurst of times.

- Random program changes can mutate the program to match specifications
- Other semantics are not always preserved
- Anecdotally, even an incorrect change can show the programmer where he went wrong

Minmax Example

```
int main(void)
{ int input1 ;
  int input2 ;
  int input3 ;
  int least ;
  int most ;

  {
  least = input1;
  most = input1;
  __ASSUME(input1 >= 0, input2 >= 0, input3 >= 0);
  if (most < input2) {
    most = input2;
  }
  if (most < input3) {
    most = input3;
  }
  if (least > input2) {
    most = input2;
  }
  if (least > input3) {
    least = input3;
  }
  __ASSERT(least <= input1, least <= input2, least <= input3, most >= input1, most >= input2,
           most >= input3);
  return (0);
}
```

Random Repair Algorithm

1. Run program through a symbolic executor
2. For each assertion, check to see if the program state at that line is consistent with the assertion
3. If assertions are consistent, then end
4. Else, if this candidate is more consistent than previous, remember it
5. Apply a random modification to the original program, and go to step 1; abort after time/iteration limit
6. Show the best candidate

Random Repaired Minmax Candidate

```
int __randomrepair__52__main(void)
```

```
{ int input1 ;  
  int input2 ;  
  int input3 ;  
  int least ;  
  int most ;
```

```
{  
  least = input1;  
  most = input1;  
  __ASSUME(input1 >= 0, input2 >= 0, input3 >= 0);
```

```
  if (most < input2) {  
    most = input2;
```

```
  }  
  if (most < input3) {  
    most = input3;
```

```
  }  
  if (least > input2) {  
    most = most;
```

```
  }  
  if (least > input3) {  
    least = input3;
```

```
  }  
  __ASSERT(least <= input1, least <= input2, least <= input3, most >= input1, most >= input2,  
           most >= input3);
```

```
  return (0);
```

```
}
```

More consistent (most is now correct), but still not correct

Random Repaired Minmax Candidate

```
int __randomrepair__52__main(void)
```

```
{ int input1 ;  
  int input2 ;  
  int input3 ;  
  int least ;  
  int most ;
```

```
{  
  least = input1;  
  most = input1;  
  __ASSUME(input1 >= 0, input2 >= 0, input3 >= 0);
```

```
  if (most < input2) {  
    most = input2;
```

```
  }
```

```
  if (most < input3) {  
    most = input3;
```

```
  }
```

```
  if (least > input2) {  
    input3 = input2;
```

```
  }
```

```
  if (least > input3) {  
    least = input3;
```

```
  }
```

```
  __ASSERT(least <= input1, least <= input2, least <= input3, most >= input1, most >= input2,  
           most >= input3);
```

```
  return (0);
```

```
}
```

Consistent, but not the
desired semantics.

Random Repaired Strcopy Candidate

Correct!

Almost always right.

```
int main(void)
{ char src[10];
  char dst[10];
  int i;

  {
  __ASSUME(1);
  i = 0;
  while (i < 10 && (int )src[i] != 0) {
    __ASSERT(i < 10);
    dst[i] = src[i];
    i ++;
  }
  __ASSERT(i < 10);
  dst[i] = (char )'\000';
  return (0);
}
}
```

```
int __randomrepair__6__main(void)
{ char src[10];
  char dst[10];
  int i;

  {
  __ASSUME(1);
  i = 0;
  while (i + 1 < 10 && (int )src[i] != 0)
  {
    __ASSERT(i < 10);
    dst[i] = src[i];
    i ++;
  }
  __ASSERT(i < 10);
  dst[i] = (char )'\000';
  return (0);
}
}
```

Classes of Random Transformations

Implemented

- Add one to an integer expression
- Flip a boolean expression
- Substitute one variable for another

Other Options

- Subtract one to an integer expression
- Swap indexes on a 2-d array (i.e. $a[i][j] \rightarrow a[j][i]$)
- Swap integer constants with each other, or possibly with integer variables
- Re-order operations (e.g. $i+j/k \rightarrow (i+j)/k$)
- Promote integer division to ceiling of double division
- Promoting integers to doubles

Random Transformations with CIL

- Must add scope and type tracking to avoid introducing errors when substituting variables
- Must discriminate between bool and int for +1 and ! transformations
 - Both show up as int types within CIL
 - Discrimination is based on usage
- Transformation is done with a single configurable visitor in two passes:
 - First, a count of each of the possible changes is done so that we can guarantee flat distribution of a fixed number of changes
 - In the second pass, all changes are enacted

Inconsistency Measure

- Gulwani & Jovic (2007)
- $\mathcal{M}(\phi, \phi')$ where ϕ in DNF, ϕ' in CNF

$$\mathcal{M}(\phi, \bigwedge_{i=1}^m C_i) = \sum_{i=1}^m \frac{1}{m} \times \mathcal{M}(\phi, C_i)$$

$$\mathcal{M}(\bigvee_{j=1}^k D_j, C_i) = \sum_{j=1}^k \frac{1}{k} \times \mathcal{M}(D_j, C_i)$$

$$\mathcal{M}(D_j, C_i) = 0 \text{ if } D \Rightarrow C, 1 \text{ otherwise}$$

Scalability?

- Current implementation is completely random: may repeat repairs
- Repair strategy: Given n possible patches, 90% probability of finding a repair by trial k :
 - $k_{.90} = -1/(\log(n-1) - \log(n)) \approx 2n$
- Program size: Given number of identifier I and use sites S , number of patches n :
 - $n = I * S$
- Repair size: Layering m patches:
 - $N(n, m) = n^m$

Limitations

- Inter-procedural repairs not yet supported
- With the current algorithm, randomly finding a solution when the number of edits required is high is improbable
- Static evaluation of some program-assertion combinations is intractable, or even undecidable
- CIL strips some contextual information, so "source level" bugs are better fixed with different tools
 - e.g. `if(a =! b)` instead of `if(a != b)`
 - e.g. `for(i = 0; i < 10; i++); {...}`

Future Work

- Guide the random search with heuristics or probabilistic criteria
- Don't repeat the same change
- Estimate scalability of search on large programs, or programs that require many changes
- Partition program between assume and assert statements
- Improve sensitivity to C standards (e.g. don't substitute const variables)
- Expand the classes of random program changes
- Extend to support inter-procedural evaluation and modification

Acknowledgements

Thanks to Martin Ma (kkma@cs) for providing the C symbolic executor, and helping to adapt it to our needs

Thanks to Octavian Udrea (udrea@umiacs) for helping with the inconsistency measure, and for providing test cases from Pistachio