CIS 700 - Course Project

Symbolic Execution With Hash Functions

Michael Henehan
Aalok Thakkar

Higher-Order Test Generation

Patrice Godefroid
Microsoft Research
pg@microsoft.com

Abstract

Symbolic reasoning about large programs is bound to be imprecise. How to deal with this imprecision is a fundamental problem in program analysis. Imprecision forces approximation. Traditional static program verification builds "may" over-approximations of the program behaviors to check universal "for-all-paths" properties, while automatic test generation requires "must" under-approximations to check existential "for-some-path" properties.

In this paper, we introduce a new approach to test generation where tests are derived from *validity proofs* of first-order logic formulas, rather than *satisfying assignments* of quantifier-free first-order logic formulas as usual. Two key ingredients of this *higher-order test generation* are to (1) represent complex/unknown pro-

Symbolic reasoning about large programs is bound to be imprecise. If perfect bit-precise symbolic reasoning was possible, static program analysis would detect standard programming errors without reporting false alarms. How to deal with this imprecision is a fundamental problem in program analysis. Traditional static program verification builds "may" over-approximations of the program behaviors in order to prove correctness, but at the cost of reporting false alarms. Dually, automatic test generation requires "must" under-approximations in order to drive program executions and find bugs without reporting false alarms, but at the cost of possibly missing bugs.

Most of the program analysis literature discusses program verification for universal properties. Yet, except for static type systems

Example 1:

```
int obscure(int x, int y) {
   if (x == hash(y)) return -1;      // error
   return 0;                            // ok
}
```

Example 2:

```
int obscure(int x, int y) {
   if (hash(x) == hash(y)+1) return -1;     // error
   return 0;                                // ok
}
```

Example 3:

```
int foo(int x, int y) {
   if (x == hash(y)) {
      ...
      if (y == 10) return -1; // error
   }
   ...
}
```

Sound Concretization

A path constraint pc_w is sound if every input assignment satisfying pc_w defines a program execution following path w.

Whenever a symbolic expression e is concretized during symbolic execution, for all symbolic variables x_i occurring in e, a new concretization constraint $x_i = I_i$ is added to the path constraint. This implies that the value of each such symbolic variable x_i is fixed to a constant equal to the corresponding current input value I_i below in the path constraint.

Challenge

requires conservatively estimating **all possible inputs and outputs** of **all individual instructions** and **all unknown/library/operating-system functions** used by the program under test

So what about hash?

Higher-order Test Generation

1. Uninterpreted functions are used to represent unknown functions or instructions during symbolic execution;
2. New test inputs are derived from validity proofs of first-order logic formulas with uninterpreted functions;
3. Concrete input-output value pairs need be recorded as uninterpreted function samples that are used when generating new concrete test inputs.

Idea

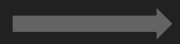
- Instead of using uninterpreted functions to represent unknown functions, replace that unknown function with a known function of similar behavior

The Modification

- Take specified function (i.e. hashing function) and, instead of executing the function on symbolic input, skip execution and return symbolic data
- Define a replacement function to accomplish this
- Have KLEE replace every instance of a hashing function with the defined replacement function

Example

```c
/*
 * Compute the md5 hash of the given data, return a string representation of
 * the output
 */
char *computeDigest(char *data, int dataLengthBytes) {
  int i, j;
  unsigned char *digestBuffer =
    (unsigned char *) malloc(MD5_DIGEST_LENGTH * sizeof(unsigned char));

  char *digest = (char *) malloc(32);

  // Comput the MD5 hash
  struct MD5Context c;
  __md5_Init(&c);
  __md5_Update(&c, (const unsigned char *)data, (unsigned int)dataLengthBytes);
  __md5_Final(digestBuffer, &c);

  // Convert to string format
  for (i = 0, j= 0; i < 32; i+=2, j += sizeof(unsigned char)) {
    sprintf(&digest[i], "%02x", digestBuffer[j]);
  }

  free(digestBuffer);

  return digest;
}
```



```c
char *replacement(void) {
  char *rtn =
    (char *) malloc(32);
  memset(rtn, '\0', 32);

  klee_make_symbolic(rtn, 32, "rtn");

  return rtn;
}
```

Demo

- Two programs with the same functionality
- Two versions of KLEE
- What advantages/disadvantages does our tool have over the original?

Statistics: Original KLEE on Code with Hashing

- Elapsed Time: 16 seconds
- Explored Paths: 13
- Total instructions: 152821
- Completed Paths: 13

- Did not reach the target path (in 16 seconds)

Statistics: Original KLEE on Code without Hashing

- Elapsed Time: < 1 second
- Explored Paths: 6
- Total instructions: 5958
- Completed Paths: 6

- Reached the target path

Statistics: Modified KLEE on Code with Hashing

- Elapsed Time: < 1 second
- Explored Paths: 198
- Total instructions: 27329
- Completed Paths: 198

- Reached the target path

Statistics: Modified KLEE on Code without Hashing

- Elapsed Time: < 1 second
- Explored Paths: 6
- Total instructions: 5965
- Completed Paths: 6

- Reached the target path

Limitations

- Lose some context (i.e. in the example, hashing output no longer depends on the input)
  - False Positives (i.e. if hashing output was correct, but input was not)
  - False Negatives (i.e. if hashing output was not correct, but input was)

Advantages

- Chose not to interpret functions that:
  - Don't have side effects we want to test
  - May lead to path explosion (due to lot of loops, etc.)

Future Work and Extensions

Fully implement Godefroid's paper

Reason modulo domain specific theories

Increase scalability by using compositional techniques, pruning redundant paths, and heuristics search.