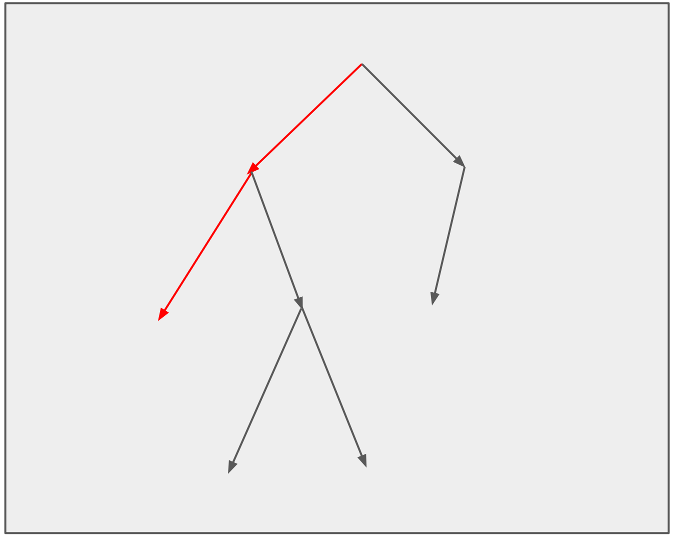
Incorrect Logic and Symbolic Execution

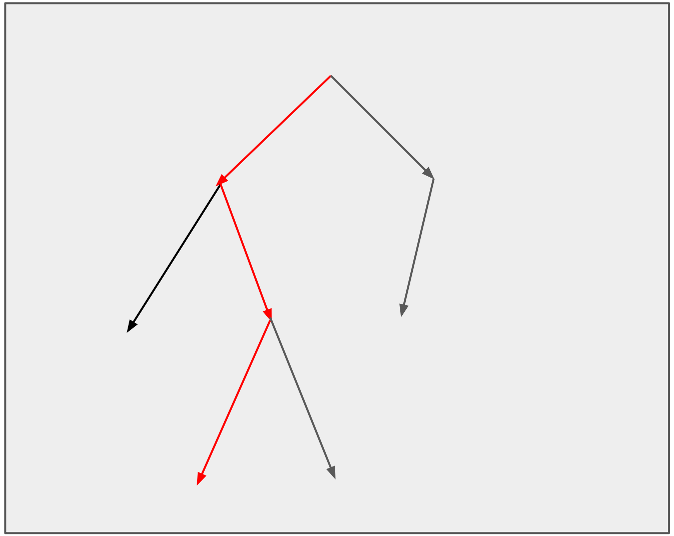
Compositional Symbolic Execution

Compositional Dynamic Test Generation(POPL'07), based on DART("SMART = Scalable DART")

- Compute procedure summary and reused them in the verification of caller;
    - If there are two procedure F and G where F calls G in its body; The total execution paths is |F|*|G|; but using procedure summary can reduce it to |F| + |G|.
    - Exponential to linear
- Keep the feature of dynamic symbolic execution(mention the symbolic assignment and concrete value assignment)
    - Which means it will be fallback(concretization) to concrete value and lose completeness, like DART.
- Top-down instead of bottom-up, demand-driven
    - Bottom-up will generate full procedure summary for lower level procedure, although some of paths are not reachable, too expensive.
    - Bottom-up loses the calling context, which can help the the concretization. E.g. it is hard to sample "x == hash(y)", but probably the one calling context has constraint "x == 10; y == 5" just satisfies this condition. I think this situation is rare.

DART

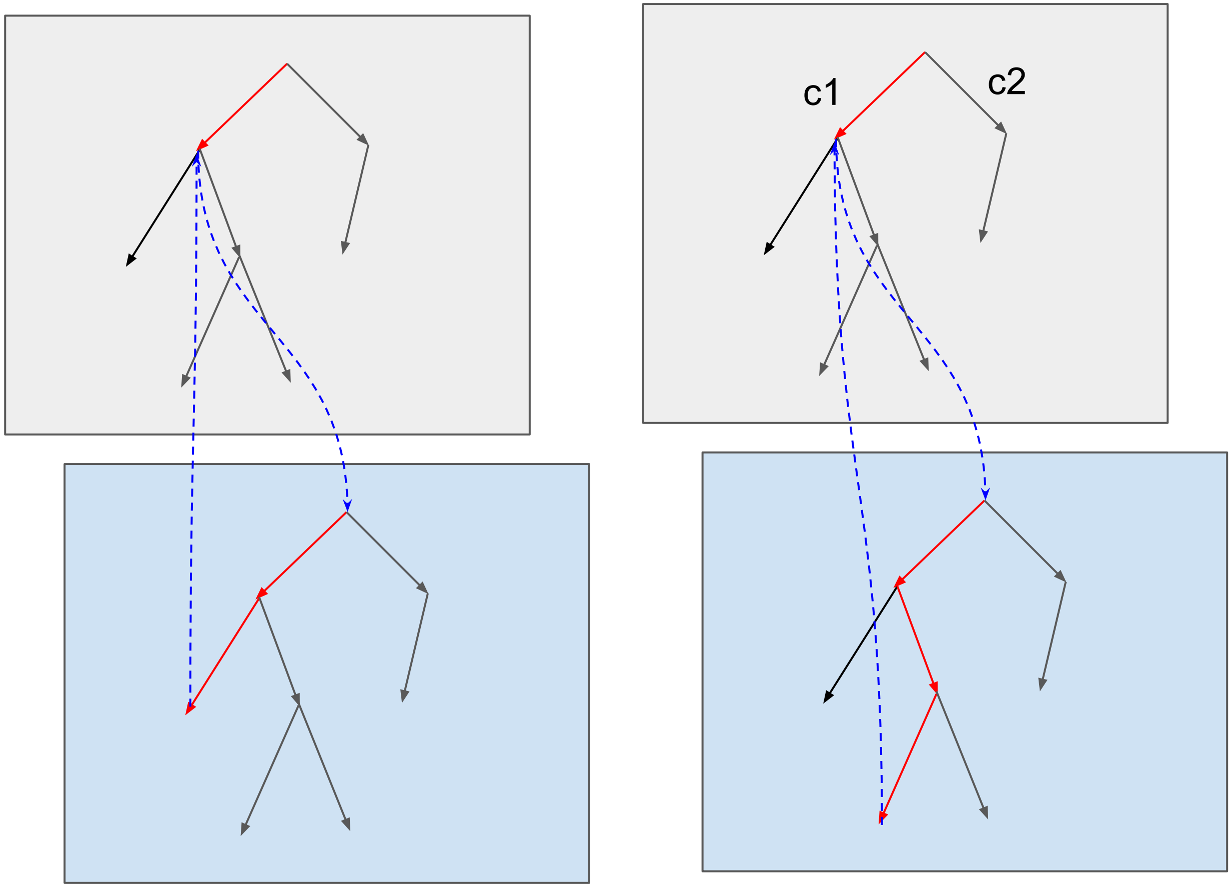




Keep the both symbolic execution(the path) and concrete execution(real input)

1. Sample a random real input, get the corresponding execution path(in red), compute the path constraint
2. Use solve to compute a real input which not satisfy the old path constraint to force the algorithm explore a new path
3. If path constraint is out of theory, use concrete value for instead(but the result with labeled as incompelete)

SMART



1. Explore all possible paths in callee then continue explore in caller, same as DART.
2. May not explore all paths in callee(as c1 may conflict with some paths)
3. Build procedure summary for callee for reusing
4. Refine summary(explore more paths) if need, e.g when path start from c2.

Consideration

1. Keep a very long execution path.
2. Implicitly shows that keep calling context can prune the paths in the exploration in callee. Kind of like MSS, by top-down verification, infer specification on client side, then can simplify the verification in library code.
3. The reverse case can explore. The calling context is the precondition of the callee, on the other hand, there could be a “post calling context”: e.g. if f(x) == 0 then error(), Keep “ret == 0” in the exploration of f can also prunes the paths.

Compositional May-Must Program Analysis

Compositional May-Must Program Analysis: Unleashing the Power of Alternation(POPL’10)

- May is over-approximation; must is under-approximation
- Compute both over-approximation and under-approximation for bug finding
- over-approximation helps pruning the paths in under-approximation, verse vice.

Examples

```
void  main(int j)
{
1: int i = 0;
2: x = foo(i,j);
3: if (x == 1)
4:   assert(false);
}
```

```
int foo(int i,j)
{
11: if (j > 0)
12:   return bar(i)+1;
13: else
14:   return i+1;
}
```

Figure 3. Example of must summary benefiting from not-may summary.

1. Assume bar has an over-approximation(Hoare Logic) statement: {i = 0}bar(i){ret != 0};
2. Then error in f can be reduced to prove the under-approximation statement of foo:
   [i = 0]foo(i,j)[ret == 1]
3. There are two branches on line 11, we can choose either one in the incorrectness logic, but we do not know to choose which one. However, {i = 0}bar(i){ret != 0} can help us to jump the then branch.

Example

```
void main(int i,j)
{
0: int x;
1: if (i > 2 && j > 2) {
2:    x = foo(i,j);
3:    if (x < 0)
4:       assert(false);
   }
}
```

```
int foo(int i,j)
{
10: int r,y;
11: y = bar(i);
12: if (y > 10)
13:    r = g(j);
14: else
15:    r = y;
16: return r;
}
```

```
int g(int j)
{
20: if (j > 0)
21:    return j;
22: else
23:    return -j;
}
```

Figure 4. Example of not-may summary benefiting from must summary.

Assume:
[i>5]bar(i)[ret > 20];
[i<5]bar(i)[ret < 5];
{true}bar(i){ret >= 0}

Try prove the main is correct, which can be reduced to {i>2 /\ j > 2}foo(i,j){ret >= 0}

The else branch in foo is easy to verify;

The [i>5]bar(i)[ret > 20] shows the then branch is reachable, which is very hard to prove in over-approximation.
I don’t think it makes sense

Consideration

1. Correctness logic can help to prune and guild the incorrectness proof, as there are too many possible choices in incorrectness logic.
2. The ok and error cases should be considered, thus there are 4 specifications of a procedure f.
    a. {p} f {error:...}: from any possible state in p, the error happens.
    b. {p} f {ok: q}: from any possible state in p, q holds.
    c. [p] f [error:...]: exists a state in p, cause an error
    d. [p] f [ok: q]: exists a state in p, make q happen.

{p} f {error:...} can refine pre-condition: [True] y = f(x); ....; [error: e1] if we known {p} f {error: e2}, then the assumption can be refined as [not p] y = f(x); ...; [error: e1]

The usage of {p} f {ok: q} is shown before.

Uninterpt in Symbolic Execution

Higher-Order Test Generation(PLDI’2021)

- Keep complicate function(like hash) in the path constraint
- Use “higher order solver”(although not exists) to solve the path constraint:

  Forall hash, exists x, …

  Where using forall function thus is not in FOL.

- Implementation: sampling the complicate function to reduce it back to FOL.

Example

EXAMPLE 3. Consider the function

```
int bar(int x, int y) {  // x,y are inputs
  if ((x == hash(y)) AND (y == hash(x))) {
    ... // error
  }
  . ..
}
```

Normal dynamic symbolic execution will random set x = 1 and y = 2, then hash(1) = 13 and hash(2) = 64; and try to use constraint (x == 13)&&(y ==64) to find a new path reach the error, although it is wrong.

Higher order constraint keep hash: forall hash, exists x y, (x == hash(y))&&(y == hash(x)), which is unsat.

Predicate transformation

Incorrectness separation logic uses forward transformation, which is very similar with symbolic execution.

- Divided the program to path, and do forward transformation from true.
- Merge branches by:

$$\textsc{Choice}\quad \frac{\vdash [p]\, \mathbb{C}_i\, [\epsilon : q] \qquad \text{for some } i \in \{1,2\}}{\vdash [p]\ \mathbb{C}_1 + \mathbb{C}_2\ [\epsilon : q]}$$

$$\textsc{Disj}\quad \frac{\vdash [p_1]\ \mathbb{C}\ [\epsilon : q_1] \qquad \vdash [p_2]\ \mathbb{C}\ [\epsilon : q_2]}{\vdash [p_1 \vee p_2]\ \mathbb{C}\ [\epsilon : q_1 \vee q_2]}$$

Dish rule is approximated, cannot always generate the tight statement.

Infer over openssl

I tried their incorrectness mode but can not find bug mentioned in their paper.

False positive:

- Complicate function which over the symbol bound, then infer will not generate summary for it.
- Customer data structure(https://github.com/openssl/openssl/blob/OpenSSL_1_0_1g/crypto/stack/stack.c). e.g. A stack implemented by a variable size memory block, and an integer recording its length, the insert function reallocate the memory when overflow.
    - The summary inference of “insert” failed
    - Infer does not found the memory size is bound by one integer

The code base is large, I am still try to analyse it...