Symbolic Execution

Suman Jana

Acknowledgement: Baishakhi Ray (Uva), Omar Chowdhury (Purdue), Saswat Anand (GA Tech), Rupak Majumdar (UCLA), Koushik Sen (UCB)
What is the goal?

static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams, uint8_t *signature, Uint16 signatureLen)
{
    OSStatus err;
    ...
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    // code omitted for brevity...
    err = sslRawVerify(ctx, ctx->peerPubKey, dataToSign, dataToSignLen, signature, signatureLen);
    if (err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
            "returned %d\n", (int)err);
        goto fail;
    }
    fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
Testing

• Testing approaches are in general manual
• Time consuming process
• Error-prone
• Incomplete
• Depends on the quality of the test cases or inputs
• Provides little in terms of coverage
Can we do better in terms of test generation? Can we somehow make it automatic?

Yes, we can.
Background: SAT

Given a propositional formula in CNF, find if there exists an assignment to Boolean variables that makes the formula true:

\[ \varphi = \omega_1 \land \omega_2 \land \omega_3 \]

\[ \omega_1 = (b \lor c) \]

\[ \omega_2 = (\neg a \lor \neg d) \]

\[ \omega_3 = (\neg b \lor d) \]

\[ A = \{a=0, b=1, c=0, d=1\} \]
Background: SMT (Satisfiability Modulo Theory)

- An SMT instance is a generalization of a **Boolean SAT** instance.
- Various sets of variables are replaced by **predicates** from a variety of underlying theories.

**Input:** a **first-order** formula \( \varphi \) over background theory (Arithmetic, Arrays, Bit-vectors, Algebraic Datatypes)

**Output:** is \( \varphi \) satisfiable?
  - does \( \varphi \) have a model?
  - Is there a refutation of \( \varphi = \text{proof of } \neg \varphi \)?
Background: SMT

\[ b + 2 = c \quad \text{and} \quad f(\text{read}(\text{write}(a,b,3), c-2)) \neq f(c-b+1) \]
Example SMT Solving

\[ b + 2 = c \text{ and } f(\text{read}(\text{write}(a, b, 3), c - 2)) \neq f(c - b + 1) \]

[Substituting \( c \) by \( b + 2 \)]

\[ b + 2 = c \text{ and } f(\text{read}(\text{write}(a, b, 3), b + 2 - 2)) \neq f(b + 2 - b + 1) \]

[Arithmetic simplification]

\[ b + 2 = c \text{ and } f(\text{read}(\text{write}(a, b, 3), b)) \neq f(3) \]

[Applying array theory axiom]

\[ \forall a, i, v: \text{read}(\text{write}(a, i, v), i) = v \]

\[ b + 2 = c \text{ and } f(3) \neq f(3) \textbf{ [NOT SATISFIABLE]} \]

\[ \text{read} : \text{array} \times \text{index} \rightarrow \text{element} \]

\[ \text{write} : \text{array} \times \text{index} \times \text{element} \rightarrow \text{array} \]
Program Validation Approaches

Cost (programmer effort, time, expertise) vs. Confidence

- Verification
- Extended Static Analysis
- Symbolic Execution
- Concolic Execution & White-box Fuzzing (dynamic)
- Ad-hoc testing (dynamic)
Automatic Test Generation
Symbolic & Concolic Execution

How do we automatically generate test inputs that induce the program to go in different paths?

Intuition:

◦ Divide the whole possible input space of the program into equivalent classes of input.

◦ For each equivalence class, all inputs in that equivalence class will induce the same program path.

◦ Test one input from each equivalence class.
Void func(int x, int y) {
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            ERROR
    }
}

int main() {
    int x = sym_input();
    int y = sym_input();
    func(x, y);
    return 0;
}
Symbolic Execution

Execute the program with symbolic valued inputs (Goal: good path coverage)

Represents equivalence class of inputs with first order logic formulas (path constraints)

One path constraint abstractly represents all inputs that induces the program execution to go down a specific path

Solve the path constraint to obtain one representative input that exercises the program to go down that specific path

Symbolic execution implementations: KLEE, Java PathFinder, etc.
More details on Symbolic Execution

Instead of concrete state, the program maintains **symbolic states**, each of which maps variables to symbolic values.

**Path condition** is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its **execution tree**, in which some paths are feasible and some are infeasible.
Symbolic Execution

How does symbolic execution work?

Void func(int x, int y){
  int z = 2 * y;
  if(z == x){
    if (x > y + 10)
      ERROR
  }
}

int main(){
  int x = sym_input();
  int y = sym_input();
  func(x, y);
  return 0;
}

Note: Require inputs to be marked as symbol
Symbolic Execution

How does symbolic execution work?

Path constraints represent equivalence classes of inputs
SMT Queries

Counterexample queries (generate a test case)

Branch queries (whether a branch is valid)

\[ \text{Path Constraints} = \{C_1, C_2, ..., C_n\}; \]

If \( K \) then

Use queries to determine validity of a branch

else path is impossible: \( C_1 \land C_2 \land ... \land C_n \land \neg K \) is UNSAT

then path is impossible: \( C_1 \land C_2 \land ... \land C_n \land K \) is UNSAT

else

then path is impossible: \( C_1 \land C_2 \land ... \land C_n \land \neg K \) is UNSAT

else path is impossible: \( C_1 \land C_2 \land ... \land C_n \land K \) is UNSAT
Optimizing SMT Queries

Expression rewriting
- Simple arithmetic simplifications ($x \times 0 = 0$)
- Strength reduction ($x \times 2^n = x \ll n$)
- Linear simplification ($2 \times x - x = x$)

Constraint set simplification
- $x < 10 \land x = 5 \implies x = 5$

Implied Value Concretization
- $x + 1 = 10 \implies x = 9$

Constraint Independence
- $i < j \land j < 20 \land k > 0 \land i = 20 \implies i < j \land i < 20 \land i = 20$
Optimizing SMT Queries (contd.)

Counter-example Cache
- $i < 10 \&\& i = 10$ (no solution)
- $i < 10 \&\& j = 8$ (satisfiable, with variable assignments $i \rightarrow 5, j \rightarrow 8$)

Superset of unsatisfiable constraints
- $\{i < 10, i = 10, j = 12\}$ (unsatisfiable)

Subset of satisfiable constraints
- $i \rightarrow 5, j \rightarrow 8$, satisfies $i < 10$

Superset of satisfiable constraints
- Same variable assignments might work
It is possible to extend symbolic execution to help find bugs. How does Symbolic Execution Find bugs?

Write a dedicated checker for each kind of bug (e.g., buffer overflow, integer overflow, integer underflow) and give us concrete input values that will trigger the bug. Even though we only fork in branches assume current PC is 0.

Divide by zero example --- \( y = \frac{x}{z} \) where \( x \) and \( z \) are symbolic variables and assume current PC is \( f \).

One branch in which \( z = 0 \) and another where \( z \neq 0 \).

We will get two paths with the following constraints:

\[
\begin{align*}
z &= 0 \\
\Rightarrow & \quad z \neq 0
\end{align*}
\]

Solving the constraint \( z = 0 \) will give us concrete input values that will trigger the divide by zero error.
Classic Symbolic Execution ---

Practical Issues

Loops and recursions --- infinite execution tree

Path explosion --- exponentially many paths

Heap modeling --- symbolic data structures and pointers

SMT solver limitations --- dealing with complex path constraints

Environment modeling --- dealing with native/system/library calls/file operations/network events

Coverage Problem --- may not reach deep into the execution tree, specially when encountering loops.
Solution: Concolic Execution

Concolic = Concrete + Symbolic

Combining Classical Testing with Automatic Program Analysis

Also called dynamic symbolic execution

The intention is to visit deep into the program execution tree
Program is simultaneously executed with concrete and symbolic inputs
Start off the execution with a random input
Specially useful in cases of remote procedure call

Concolic execution implementations: SAGE (Microsoft), CREST
Concolic Execution Steps

- Generate a random seed input to start execution
- Concretely execute the program with the random seed input and collect the path constraint
- Example: \( a \land b \land c \)
- In the next iteration, negate the last conjunct to obtain the constraint \( a \land b \land \neg c \)
- Solve it to get input to the path which matches all the branch decisions except the last one

Why not from the first?
void testme (int x, int y)
{
    z = 2*y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
Concolic execution example

```c
void testme (int x, int y) {
    z = 2*y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution
symbolic state

Concrete
state

Symbolic
state

path condition

x = 22, y = 7

x = a, y = b

x = a, y = b
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
void testme(int x, int y) {
    z = 2*y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```c
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
    x = 22, y = 7, z = 14
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>concrete state</td>
<td>symbolic state</td>
<td>2*b != a</td>
</tr>
<tr>
<td>x = a, y = b,</td>
<td>x = a, y = b,</td>
<td></td>
</tr>
<tr>
<td>z = 2*b</td>
<td>z = 2*b</td>
<td></td>
</tr>
</tbody>
</table>
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
void testme (int x, int y) {
    z = 2 * y;
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

Concrete Execution
    concrete state
    x = 2, y = 1,
    z = 2

Symbolic Execution
    symbolic state
    x = a, y = b,
    z = 2*b

path condition
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
void testme (int x, int y) {
    z = 2 * y;
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
void testme (int x, int y) {
    z = 2 * y;
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}

Concrete Execution

Symbolic Execution

Concrete state

Symbolic state

Path condition

2 * b == a
a > b + 10

Program Error

x = 30, y = 15
z = 30
x = a, y = b

x = 30, y = 15
z = 30
Limitations

Path Space of a Large Program is Huge
- Path Explosion Problem
Limitations

Path Space of a Large Program is Huge
  - Path Explosion Problem

Explored by Concolic Testing

Entire Computation Tree
Limitations:
a comparative view

Concolic: Broad, shallow

Random: Narrow, deep
Limitations: Example

Example ( ) {
1: state = 0;
2: while(1) {
3:  s = input();
4:  c = input();
5:  if(c==':' && state==0)
  state=1;
6:  else if(c=='\n' && state==1)
  state=2;
7:  else if (s[0]=='I' &&
       s[1]=='C' &&
       s[2]=='S' &&
       s[3]=='E' &&
       state==2) {
       COVER_ME;;
    }
}
}
Limitations: Example

Example ( ) {
1: state = 0;
2: while(1) {
3:   s = input();
4:   c = input();
5:   if(c=='.' && state==0)
       state=1;
6:   else if(c=='\n' && state==1)
       state=2;
7:   else if (s[0]=='I' &&
       s[1]=='C' &&
       s[2]=='S' &&
       s[3]=='E' &&
       state==2) {
       COVER_ME;
   }
}
}

- **Pure random testing** can get to state = 2
  But difficult to get ‘ICSE’ as a Sequence

  Probability $1/(2^8)^6 \approx 3 \times 10^{-15}$

- Conversely, **concolic testing** can generate ‘ICSE’ but explores many paths to get to state = 2
Hybrid concolic testing

Interleave Random Testing and Concolic Testing to increase coverage

while (not required coverage) {
  while (not saturation) 
  perform random testing;
  Checkpoint;
  while (not increase in coverage) 
  perform concolic testing;
  Restore;
}

while (not required coverage) {
  while (not saturation) 
  perform random testing;
  Checkpoint;
  while (not increase in coverage) 
  perform concolic testing;
  Restore;
}
Hybrid Concolic Testing

while (not required coverage) {
  while (not saturation)
    perform random testing;
  Checkpoint;
  while (not increase in coverage)
    perform concolic testing;
  Restore;
}

Interleave Random Testing and Concolic Testing to increase coverage

Deep, broad search
Hybrid Search
Hybrid Concolic Testing

Example ( ) {
  1: state = 0;
  2: while(1) {
    3:   s = input();
    4:   c = input();
    5:   if(c==':' && state==0)
            state=1;
    6:   else if(c=='\n' && state==1)
            state=2;
    7:   else if (s[0]==‘I’ &&
              s[1]==‘C’ &&
              s[2]==‘S’ &&
              s[3]==‘E’ &&
              state==2) {
              COVER_ME:;
            }
  }
}

Random Phase

- ‘$’, ‘&’, ‘-’, ‘6’, ‘:’, ‘%’, ‘^’, ‘\n’, ‘x’, ‘~’ ...
- Saturates after many (~10000) iterations
- In less than 1 second
- COVER_ME is not reached
Hybrid Concolic Testing

Example ( ) {
1: state = 0;
2: while(1) {
3:   s = input();
4:   c = input();
5:   if(c=='.' && state==0)
       state=1;
6:   else if(c=='\n' && state==1)
      state=2;
7:   else if (s[0]=='I' &&
             s[1]=='C' &&
             s[2]=='S' &&
             s[3]=='E' &&
             state==2) {
       COVER_ME:;
    }
}
}

Random Phase
- ‘$’, ‘&’, ‘-’, ‘6’, ‘.’, ‘%’, ‘^’, ‘\n’, ‘x’, ‘~’ ...
  - Saturates after many (~10000) iterations
  - In less than 1 second
  - COVER_ME is not reached

Concolic Phase
  - Reaches COVER_ME
Hybrid Concolic Testing

- 4x more coverage than random
- 2x more coverage than concolic
Summary

Concolic Testing

Random Testing

Hybrid Concolic Testing
Further reading

Symbolic execution and program testing - James King

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs - Cadar et. al.

Symbolic Execution for Software Testing: Three Decades Later - Cadar and Sen

DART: Directed Automated Random Testing - Godefroid et. al.

CUTE: A Concolic Unit Testing Engine for C - Sen et. al.