Symbolic and Concolic Execution of Programs

Information Security, CS 526

Omar Chowdhury
Reading for this lecture

- 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
- A Few Billion Lines of Code Later Using Static Analysis to Find Bugs in the Real World - Engler et. al.
- DART: Directed Automated Random Testing - Godefroid et. al.
- CUTE: A Concolic Unit Testing Engine for C - Sen et. al.
What is the goal?

```c
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
    uint8_t *signature, UInt16 signatureLen)
{
    OSSstatus err;
    ...
    if ((err = SSLHashSHA1.update(hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    // code ommitted for brevity...
    err = sslRawVerify(ctx,
        ctx->peerPubKey, /* plaintext */
        dataToSign, /* plaintext length */
        dataToSignLen,
        signature,
        signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify 
            "returned %d\n", (int)err);
        goto fail;
    }
    fail:
    SSLFreeBuffer(&signedHahes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```
Testing

- Majority of the testing approaches are manual
- Time consuming process
- Error-prone
- Incomplete
- Depends on the quality of the test cases or inputs
- Provides little in terms of coverage
Obvious Questions?

Can we do better in terms of test generation or how 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:

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

\[ \omega_1 = (b \lor c) \]
\[ \omega_2 = (\neg a \lor \neg d) \]
\[ \omega_3 = (\neg b \lor d) \]

\[ A = \{ a=0, b=1, c=0, d=1 \} \]
Background: SMT

SMT: Satisfiability Modulo Theories

Input: a first-order formula $\varphi$ over background theory

Output: is $\varphi$ satisfiable?
  - does $\varphi$ have a model?
  - Is there a refutation of $\varphi = \text{proof of } \neg \varphi$?

For most SMT solvers: $\varphi$ is a ground formula
  - Background theories: Arithmetic, Arrays, Bit-vectors, Algebraic Datatypes
  - Most SMT solvers support simple first-order sorts
Background: SMT

- $b + 2 = c$ and $f(read(write(a, b, 3), c - 2)) \neq f(c - b + 1)$

Diagram:
- Arithmetic
- Array Theory
- Uninterpreted Function
Example SMT Solving

• $b + 2 = c$ and $f(\text{read(\text{write}(a,b,3), c-2)}) \neq f(c-b+1)$
  [Substituting $c$ by $b+2$]
• $b + 2 = c$ and $f(\text{read(\text{write}(a,b,3), b+2-2)}) \neq f(b+2-b+1)$
  [Arithmetic simplification]
• $b + 2 = c$ and $f(\text{read(\text{write}(a,b,3), b)}) \neq f(3)$
  [Applying array theory axiom–
  forall a,i,v:read(\text{write}(a,i,v), i) = v]
• $b+2 = c$ and $f(3) \neq f(3)$ [NOT SATISFIABLE]
Program Validation Approaches

- Cost (programmer effort, time, expertise)
- Confidence

- Ad-hoc testing
- Fuzzing
- Concolic Execution & White-box
- Symbolic Execution
- Extended Static Analysis
- Static Analysis
- Verification
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.
Symbolic Execution — History

• **1976**: A system to generate test data and symbolically execute programs (Lori Clarke)
• **1976**: Symbolic execution and program testing (James King)
• **2005–present**: practical symbolic execution
  • Using SMT solvers
  • Heuristics to control exponential explosion
  • Heap modeling and reasoning about pointers
  • Environment modeling
  • Dealing with solver limitations
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 – Description

• 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 represent 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
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 (contd.)

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 symbolic
Symbolic Execution (contd.)

How does symbolic execution work?

Path constraints represent equivalence classes of inputs

\[ x = a = 0 \]
\[ y = b = 1 \]
\[ x = a = 5 \]
\[ y = b = 4 \]
\[ x = a = 2 \]
\[ y = b = 3 \]

\[ x = a = 2 \]
\[ y = b = 1 \]
\[ x = a = 4 \]
\[ y = b = 2 \]
\[ x = a = -6 \]
\[ y = b = -3 \]

\[ x = a = 40 \]
\[ y = b = 20 \]
\[ x = a = 30 \]
\[ y = b = 15 \]
\[ x = a = 48 \]
\[ y = b = 24 \]
SMT Queries

- Counterexample queries (generate a test case)
- Branch queries (whether a branch is valid)

Path Constraints = \( \{C_1, C_2, ..., C_n\} \); SAT

Use queries to determine validity of a branch

- then
- 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
Optimizing SMT Queries

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

• Constraint set simplification
  • $x < 10 && x = 5 \quad \rightarrow \quad x = 5$

• Implied Value Concretization
  • $x + 1 = 10 \quad \rightarrow \quad x = 9$

• Constraint Independence
  • $i<j && i < 20 && k > 0$; new constraint $i = 20$
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
How does Symbolic Execution Find bugs?

• It is possible to extend symbolic execution to help us catch bugs

• **How**: Dedicated checkers
  
  • **Divide by zero example**: Consider the case where $x$ and $z$ are symbolic variables and assume current PC is $f$
  
  • Even though we only fork in branches we will now fork in the division operator. Branches we will now fork in:

  - One branch in which $z = 0$
  - Another where $z \neq 0$

  We will get two paths with the following constraints:

  - $z = 0 \land f$
  - $z \neq 0 \land f$

  Solving the constraint $z = 0 \land f$ will give us concrete input values that will trigger the divide by zero error.

Write a dedicated checker for each kind of bug (e.g., buffer overflow, integer overflow, integer underflow).
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
Classic Symbolic Execution —
Practical Issues (possible solutions)

• **Infinite execution tree**
  - Finitize paths by limiting the PC size (bounded verification)
  - Use loop invariants (verification)

• **Path explosion**
  - Select next branch at random
  - Select next branch based on coverage
  - Interleave symbolic execution with random testing

• **Heap modeling**
  - Segmented address space via the theory of arrays (Klee)
  - Lazy concretization (JPF)
  - Concolic lazy concretization (CUTE)
Classic Symbolic Execution – Practical Issues (possible solutions)

• **SMT solver limitations**
  - On-the-fly expression simplification
  - Incremental solving
  - Solution caching
  - Counterexample caching
  - Substituting concrete values for symbolic in complex PCs (CUTE)

• **Environment modeling**
  - Partial state concretization
  - Manual models of the environment (Klee)
Symbolic execution may not reach deep into the execution tree. Specially when encountering loops.
Solution: Concolic Execution

• Concolic = Concrete + Symbolic
• Sometimes 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 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?
Concolic Execution

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

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

Random seed $x = 2, y = 1$

Path constraint

```
2b != a
2b == a
2b == a &&
   a <= b + 10
2b == a &&
   a > b + 10
ERROR
```

$2b = a$ and $a <= b + 10$

$2b = a$ and $a > b + 10$

$2b = a$

Random seed

$x = a = 30$

$y = b = 15$

ERROR
Acknowledgement

Some of the content are derived from the slides of Endadul Haque, Emina Torlak, Nikolaj Bjørner, Bruno Dutertre, and Leonardo de Moura