Symbolic Execution

Joe Hendrix
Galois, Inc
SMT Summer School 2015
Galois, Inc
We solve hard research problems for clients.
Symbolic Execution

- ... is a technique for mapping code into logic.
- ... is widely used in both verification and testing.
- ... can be applied to many programming languages and logics.
- ... is a easily explained idea that is over 40 years old
  - **Symbolic Execution and Program Testing**, King, CACM 1976
- ... still an active area of research.

https://github.com/saswatanand/symexbib
How is **Symbolic** execution different from standard execution?

- In most program languages, execution operates by evaluating expressions in the language into **values** such as (fixed precision) integers, Booleans, objects, etc.

- **Symbolic** execution lifts evaluates expressions into **formulas** that denote the value the expression will take relative to some **variables**.

- Primitive operations are lifted to generate new formulas.

- Differences arise when dealing with **conditionals** and/or **loops**.
What **problems** can symbolic simulation solve?

- **Program Safety/Preimage** computation
  - Can we find bugs and/or prove their absence? e.g., find an input that results in an invalid memory access.
  - Can we find inputs that lead to a desired state? e.g., solve control or planning problems.

- **Program Equivalence** Checking
  - Given a program and a functional specification, can we prove that the program implements the specification?
Verification Strategy

1. Use forward symbolic simulation to unroll implementations, and generate terms that precisely describe results.

2. Show equivalence of two terms through rewriting, and SAT/SMT solvers.
Example: Verification of Elliptic Curve Cryptography

- In 2010-2012, at Galois, we verified an implementation of ECDSA over the same curve in Java.
  - The code is much faster than BouncyCastle’s ECC implementation, and only slightly slower than OpenSSL, which has full access to the hardware.
  - The Java implementation has been verified to be functionally equivalent to a Cryptol specification.
- During verification, we found an overflow bug in the modular division routine that was triggered on a very small set of inputs.
Modular division bug

- We had code for computing modular reduction.
  \[ z = a \mod (2^{384} - 2^{128} - 2^{96} + 2^{32} - 1) \]
- \( a \) is a 768-bit number encoded as an array.
- The result \( z \) is a 384-bit number encoded in an array.
- We wrote the specification in Cryptol.
Modular division bug

- We had code for computing modular reduction.
  \[ z = a \mod (2^{384} - 2^{128} - 2^{96} + 2^{32} - 1) \]
- \(a\) is a 768-bit number encoded as an array.
- The result \(z\) is a 384-bit number encoded in an array.
- We wrote the specification in Cryptol.

There was an overflow bug here.
Modular division bug

NISTCurveFactory.java (line 964):

d = (z[ 0] & LONG_MASK) + of;
z[ 0] = (int) d; d >>= 32;
d = (z[ 1] & LONG_MASK) - of;
z[ 1] = (int) d; d >>= 32;
d += (z[ 2] & LONG_MASK);
Modular division bug

NISTCurveFactory.java (line 964):

```java
    d = (z[0] & LONG_MASK) + of;
    z[0] = (int) d; d >>= 32;
    d += (z[1] & LONG_MASK) - of;
    z[1] = (int) d; d >>= 32;
    d += (z[2] & LONG_MASK);
```
Modular division bug

Bug only occurs when this addition overflows.

Previous code guaranteed that $0 < of < 5$

```java
    d = (z[ 0] & LONG_MASK) + of;
    z[ 0] = (int) d; d >>= 32;
    d += (z[ 1] & LONG_MASK) - of;
    z[ 1] = (int) d; d >>= 32;
    d += (z[ 2] & LONG_MASK);
```
Modular division bug

abc found bug in 20 seconds.
Testing found bug after 2 hours (8 billion field reductions).
OpenSSL modular division

Later, we learned of a similar bug in OpenSSL:

In 2007, Harry Reimann discovered a bug in \texttt{BN\_nist\_mod\_384}, a function used for field division in OpenSSL’s implementation of the NIST P-384 elliptic curve.

The bug was found by code inspection is vary rare: we were unable to reproduce after many billions of test vectors.

The was no known exploit at the time. It was found the day before release of OpenSSL0.9.8g; fix committed 6 months later.
Exploiting ECDH in OpenSSL

In 2012, Brumley, Barbosa, Page, and Vercauteren published a paper showing an adaptive attack that allowed full key recovery by triggering the bug.

- Ephemeral keys provide a mitigation.
- Several Linux distributions were still unpatched.
- The authors call for formal verification:

  We suggest that the effort required to adopt a development strategy capable of supporting formal verification is both warranted, and an increasingly important area for future work.
Introduction to Symbolic Execution

- Consider the function below:

```c
int max(int x, int y) {
    int r;
    if (x >= y) r = x;
    else r = y;
    return r;
}
```

- In execution, the expressions x and y are assigned concrete values, e.g.:

```plaintext
{x → 3, y → 4}
```
Simple example

Consider the function below:

```c
int max(int x, int y) {
    int r;
    if (x >= y) r = x;
    else r = y;
    return r;
}
```

In **symbolic** execution, the expressions \( x \) and \( y \) may be assigned formulas, including variables:

\[
\{ x \rightarrow u, \ y \rightarrow v \}
\]
Simple example

- Consider the function below:

```c
int max(int x, int y) {
    int r;
    if (x >= y) r = x;
    else r = y;
    return r;
}
```

- In *symbolic* execution, the expressions may be assigned other expressions, including variables:

  ```
  \{ x \rightarrow u, \ y \rightarrow v \}
  ```

  To evaluate an expression, one just generates a formula denoting the result:

  ```
  u \geq v
  ```
Simple example

Consider the function below:

```c
int max(int x, int y) {
    int r;
    if (x >= y) r = x;
    else r = y;
    return r;
}
```

In **symbolic** execution, the expressions, including variables:

```c
{x → u, y → v}
```

However, this condition may be either true or false depending on the value of **u** and **v**.
Basic Idea: **Split** Execution into Multiple Paths

- Instead of choosing, we can just split execution into multiple paths.
- To do this, we first associate an extra predicate, called the **path condition**, to the state of the program.
- We can then extend the path condition when encountering symbolic branches.

\[
\begin{align*}
&T, \quad \{ x \to u, y \to v \} \\
&u \geq v, \quad \{ x \to u, y \to v \} \\
&\neg(u \geq v), \quad \{ x \to u, y \to v \}
\end{align*}
\]
Loop Example

```c
int amax(int c, int[] x) {
    int max = -1;
    for (int i=0; i < c; ++i) {
        if (max < x[i])
            max = x[i];
        else
            ;
    }
    return max;
}
```

- `amax(u, x)`, will run forever (or at least a long time) if `u` is symbolic.
- Even if `c` is bound to a constant, the total number of paths is exponential to `c`. 
The non-termination and path explosion problems are major challenges when applying symbolic execution to real programs.

I’ll describe a couple techniques to address this:

- **Concolic Execution**
- **Path Merging**

There are also hybrid techniques that integrate the two, and model-checking techniques such as interpolation to address this.
Approach 1. Concolic Execution

- This is a approach to fuzzing that integrates concrete and symbolic introduction.

- First introduced in 2005:

  
  DART: Directed Automated Random Testing
  Godefroid, Klarlund, Sen, PLDI 2005

- There are many tools for doing this: SAGE (MSR), PEX (MSR), KLEE (Stanford), Jalangi (Berkeley), jFuzz (MIT), Mayhem (CMU/Forall Secure), FuzzBall (Berkeley)
Uses of Concolic Execution on Real Code

- SAGE (MSR) is running on “hundreds of machines”, 1/3 of all fuzzing bugs found due to SAGE

  500 Machine-Years of Software Model Checking and SMT Solving
  Patrice Godefroid
  SEFM’2014

- MergePoint (CMU) was run on a 100 node cluster analyzing over 33 thousand Linux Binaries

  - Found 11,687 bugs in 4,379 distinct applications.

  Enhancing Symbolic Execution with Veritesting
  Avgerinos, Rebert, Cha, and Brumley
  ICSE 2014
Concolic Execution

- In concolic execution, one associates a concrete value $v_x$ to each symbolic formula.

\[ (T, \{ x \rightarrow (u,4), y \rightarrow (v,3) \}) \]

- When encountering a branch ($x \geq y$), choose the branch satisfied by the current assignment, but add the symbolic predicate to the path condition:

\[ (u \geq v, \{ x \rightarrow (u,4), y \rightarrow (v,3) \}) \]
Concolic Execution

- Eventually this should terminate, and there should be a conjunction of path constraints:
  \[ p_1 \land p_2 \land \ldots \land p_k \]

- To explore alternative paths, just ask the solver to find new model satisfying different path constraints:
  \[
  \begin{align*}
  M &\models \neg p_1 \\
  M &\models p_1 \land \neg p_2 \\
  &\vdots \\
  M &\models p_1 \land \ldots \land \neg p_k
  \end{align*}
  \]

- This process can be repeated to enumerate paths through the program.

- This iteration process may run forever, but is great for automated testing.
Approach 2. Merge Execution Paths

- Instead of a set of states and path constraint pairs, we can maintain an execution tree.

- By maintaining a tree, we can **merge** execution paths when divergent paths rejoin.
Loop Example

1. int amax(int c, int[] x) {
2.   int max = -1;
3.   for (int i=0; i < c; ++i) {
4.     if (max < x[i])
5.       max = x[i];
6.     else
7.       ;
8.   }
9.   return max;
10. }

Loop Example

1. int amax(int c, int[] x) {
2.     int max = -1;
3.     for (int i = 0; i < c; ++i) {
4.         if (max < x[i])
5.             max = x[i];
6.     } else
7.         ;
8. }}
9. return max;
10. }
Loop Example

1. int amax(int c, int[] x) {
2.    int max = -1;
3.    for (int i=0; i < c; ++i) {
4.      if (max < x[i])
5.        max = x[i];
6.      else
7.        ;
8.    }
9.    return max;
10. }

{ pc = 3, max = -1, i = 0, c = 4, x = a }
Loop Example

1. int amax(int c, int[] x) {
2.     int max = -1;
3.     for (int i=0; i < c; ++i) {
4.         if (max < x[i])
5.             max = x[i];
6.         else
7.             ;
8.     }
9.     return max;
10. }

{ pc = 4, max = -1, i = 0, c = 4, x = a }
Loop Example

1. int amax(int c, int[] x) {
2.   int max = -1;
3.   for (int i=0; i < c; ++i) {
4.     if (max < x[i])
5.       max = x[i];
6.   } else
7.     ;
8. }
9. ite(-1 < rd(a, 0),
10.   { pc = 5, max = -1, i = 0, c = 4, x = a },
11.   { pc = 7, max = -1, i = 0, c = 4, x = a })
Loop Example

1. int amax(int c, int[] x) {
2.     int max = -1;
3.     for (int i=0; i < c; ++i) {
4.       if (max < x[i])
5.         max = x[i];
6.     } else
7.     ;
8. }

ite(-1 < rd(a, 0),
9.     { pc = 8, max = rd(a,0), i = 0, c = 4, x = a },
10.    { pc = 7, max = -1, i = 0, c = 4, x = a })
1. int amax(int c, int[] x) {
2.     int max = -1;
3.     for (int i=0; i < c; ++i) {
4.         if (max < x[i])
5.             max = x[i];
6.         else
7.             ;
8.     }
9.     return max;
10. }

ite(-1 < rd(a, 0),
    { pc = 8, max = rd(a,0), i = 0, c = 4, x = a },
    { pc = 8, max = -1, i = 0, c = 4, x = a },
int amax(int c, int[] x) {
    int max = -1;
    for (int i=0; i < c; ++i) {
        if (max < x[i]) {
            max = x[i];
        }
        else {
            ;
        }
    }
    return max;
}

Loop Example

{ pc = 8,
  max = ite(-1 < rd(a, 0), rd(a,0), -1), 
  i = 0, 
  c = 4, 
  x = a },
1. int amax(int c, int[] x) {
2.   int max = -1;
3.   for (int i=0; i < c; ++i) {
4.     if (max < x[i])
5.       max = x[i];
6.     else
7.       ;
8.   }
9.   return max;
10. }

Loop Example:
{ pc = 3,
  max = ite(-1 < rd(a, 0), rd(a,0), -1),
  i = 1,
  c = 4,
  x = a },
Loop Example

```c
1. int amax(int c, int[] x) {
2.   int max = -1;
3.   for (int i=0; i < c; ++i) {
4.     if (max < x[i])
5.       max = x[i];
6.   }
7.   return max;
}
```

```c
ite(ite(-1 < rd(a, 0), rd(a,0), -1) < rd(a, 0),
    { pc = 5,
      max = ite(-1 < rd(a, 0), rd(a,0), -1),
      i = 1, c = 4, x = a },
    { pc = 7,
      max = ite(-1 < rd(a, 0), rd(a,0), -1),
      i = 1, c = 4, x = a })
```
Shared Terms

- Without sharing the size of the terms denoting expressions will grow exponentially on many problems.

- Fortunately, SAT/SMT solvers either support shared terms directly, or they can be supported by introducing auxiliary variables.

```plaintext
let x0 = rd(a,0)
  x1 = ite(-1 < x0, x0, -1)
in ite(x1 < x1,
  { pc = 5, max = x1, i = 1, c = 4, x = a },
  { pc = 7, max = x1, i = 1, c = 4, x = a })
```
Complications to Symbolic Execution

- Exploring infeasible paths
- Supporting complex datatypes in the language such as the C “Heap”
- Handling compiled program representations
  - LLVM, JVM: Control Flow Graphs
  - Machine Code: Control Flow Discovery
- Composition: Function Specifications
How to Avoid Infeasible Paths

- Recall that the symbolic simulator can potentially split at each execution branch.
  - Want to avoid exploring infeasible branches.

- Use simplification to reduce formulas to values where possible.

- Concolic execution avoids this by maintaining a concrete assignment, that serves as a witness the path is feasible.
  - In the verification case, one could accomplish the same feature by calling the SAT solver at each branch.
  - An online connection to the SMT solver is needed for decent performance, but this affects the SMT solver.
Reasoning about the **Heap**

- As a first approximation, the heap can be viewed as just a map from addresses to bytes.

  \[
  \text{read}(m, a) \quad \text{write}(m, a, v)
  \]

- Would like to ensure that if you write a constant \( c \) to address \( a \), write to other locations distinct from \( a \), then you get to \( c \) back.

- Early concolic simulators would not support symbolic addresses.
  - It is easy to support symbolic writes, but this results in everything becoming symbolic.
  - To manage this, one can maintain an over approximation of what a symbolic address can manage or query a solver (McVeto, BAP, BitBlaze, Mayhem, SAW)
The LLVM and JVM representations either explicitly or implicitly reduce programs to control flow graphs.

```plaintext
if (pred) {
    // something
} else {
    // else
} // rest
...  
br pred lbl0 lbl1
lbl0:
    // something
jump lbl2
lbl1:
    // else
jump lbl2
lbl2:
    // rest
```
Post-Dominators

- We say that a block $y$ post-dominates a block $x$, if all executions out of $x$ eventually reach $y$.

$$\text{postdom}(x)$$

- The immediate post-dominator of $x$ is the “earliest” post-dominator of $x$.

$$y = \text{imm\_postdom}(x) \iff y \in \text{postdom}(x) \land (\forall z \in \text{postdom}(x)) \ y \not\in \text{postdom}(z)$$
Post-dominators

1. int amax(int c, int[] x) {
2.    int max = -1;
3.    for (int i=0; i < c; ++i) {
4.      if (max < x[i])
5.        max = x[i];
6.      else
7.        ;
8.    }
9.    return max;
10. }
Post-dominator Tree
Symbolically Simulating Machine Code

- Machine code poses special challenges:
  - The set of **code locations** is not known statically, but must be discovered.
  - **Function boundaries** and **type signatures** must also be discovered.
  - There may be indirect jumps, such as **jump tables**, within functions.
  - There are also implicit calls, such as **tail calls**.

- There are many projects aimed at discovering this information:
  - BitBlaze, BAP, McVeto, McSema, SecondWrite, RevGen, …

- Concolic execution is still quite feasible.
Function Specifications

- One of the nice things about symbolic execution is that it can succeed with minimal specifications.
  - You just execute code forward.
- Nevertheless, specifications can still be useful:
  - To address symbolic non-termination.
  - To simplify resulting formulas.
Loop Example

1. int amax(int c, int[] x) {
2.     int max = -1;
3.     for (int i=0; i < c; ++i) {
4.         if (max < x[i])
5.             max = x[i];
6.     } else
7.         ;
8. }  
9.     return max;
10. }
Loop Example

1. int amax(int c, int[] x) {
2.     int max = -1;
3.     for (int i=0; i < c; ++i) {
4.         if (max < x[i])
5.             max = x[i];
6.         else
7.             ;
8.     }
9.     return max;
10. }

1 Iteration

```c
1. int amax(int c, int x[]) {
2.    int max = -1;
3.    for (int i=0; i < c; ++i) {
4.      if (max < x[i])
5.        max = x[i];
6.      else
7.        ;
8.    }
9.    return max;
10. }
```

```plaintext
let x0 = rd(a, 0)
x1 = ite(-1 < x0, x0, -1)
in { pc = 3, max = x1, i1 = 1, c = 4, x = a }
```
2 Iterations

1. int amax(int c, int[] x) {
    int max = -1;
    for (int i=0; i < c; ++i) {
        if (max < x[i])
            max = x[i];
        else
            ;
    }
    return max;
}

let x0 = rd(a,0)
    x1 = ite(-1 < x0, x0, -1)
    x2 = rd(a,1)
    x3 = ite(x1 < x2, x1, -1)
in { pc = 3, max = x3, i1 = 2, c = 4, x = a}
3 Iterations

1. int amax(int c, int[] x) {
   let x0 = rd(a,0)
   x1 = ite(-1 < x0, x0, -1)
   x2 = rd(a,1)
   x3 = ite(x1 < x2, x1, x2)
   x4 = rd(a,2)
   x5 = ite(x3 < x4, x3, x3)
   in { pc = 3, max = x5, i1 = 3, c = 4, x = a}
4 Iterations

1. int amax(int c, int[] x) {
   int max = -1;
   for (int i=0; i < c; ++i) {
      if (max < x[i])
         max = x[i];
      else
         ;
   }
   return max;
}

let x0 = rd(a,0)
let x1 = ite(-1 < x0, x0, -1)
let x2 = rd(a,1)
let x3 = ite(x1 < x2, x2, x1)
let x4 = rd(a,2)
let x5 = ite(x3 < x4, x4, x3)
let x6 = rd(a,3)
let x7 = ite(x5 < x6, x6, x5)
in { pc = 10, max = x7, i1 = 4, c = 4, x = a}
Specification of $\text{amax}$

- The formula has a lot of Boolean structure.
  - SAT solver may be inefficient when dealing with this.
- A specification of $\text{amax}$ would be that $\text{amax}$ returns a value that is equal to one of the inputs, and greater than or equal to all of inputs.

\[
\begin{align*}
    r &= rd(a, 0) \lor r = rd(a, 1) \lor r = rd(a, 2) \lor r = rd(a, 3) \\
    r \geq rd(a, 0) & \land r \geq rd(a, 1) \land r \geq rd(a, 2) \land r \geq rd(a, 3)
\end{align*}
\]
Use in Elliptic Curve Verification

- **Cryptographic Protocols**
  - **ECDSA**: Digital Signatures
  - **ECDH**: Key Agreement

- **One Way Functions**
  - Scalar Multiplication: $R = s \cdot P$
  - Twin Multiplication: $R = s \cdot P + t \cdot Q$

- **Point Operations**
  - Addition: $R = P + Q$
  - Subtraction: $R = P - Q$
  - Doubling: $R = 2 \cdot P$

- **Field Operations**
  - Multiplication
  - Squaring
  - Division
  - Doubling
Compositional Verification

- Once a specification is defined, it can be used to simplify later methods.
  ```c
  void ec_double(JacobianPoint r) {
    ...
    field_add(t4, r.x, t4);
    field_mul(t5, t4, t5);
    field_mul3(t4, t5);
    ...
  }
  ```

- Rather than execute code for field_add, simulator simply replaces value at t4 with an application of Cryptol specification.
Summary

- Symbolic execution is widely used technique for mapping code into logic.
- Tools tend to be language-specific, but techniques often generalize.
- Research challenges include developing ways to verify non-terminating programs, deal with concurrency, and improve scalability.
- Engineering challenges include modeling the language in a way amenable to simulation, and finding good mappings between the programming language and solver language.