Acknowledgments: Some of the slides are fully or partially obtained from other sources. A reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.
Program Analysis

- How could we analyze a program (with source code) and look for problems?
- How accurate would our analysis be without executing the code?
- If we execute the code, what input values should we use to test/analyze the code?
- What if we don’t have the source code?

When I suggest using static code analysis to reduce the number of errors

https://www.viva64.com
Symbolic Execution
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

[Chowdhury’15]
Motivation

• Writing and maintaining tests is tedious and error-prone

• Idea: Automated Test Generation
  • Generate regression test suite
  • Execute all reachable statements
  • Catch any assertion violations
Approach

- Dynamic Symbolic Execution
  - Stores program state concretely and symbolically
  - Solves constraints to guide execution at branch points
  - Explores all execution paths of the unit tested

- Example of Hybrid Analysis
  - Collaboratively combines dynamic and static analysis
Execution Paths of a Program

- Program can be seen as binary tree with possibly infinite depth
  - Called **Computation Tree**

- Each node represents the execution of a conditional statement

- Each edge represents the execution of a sequence of non-conditional statements

- Each path in the tree represents an equivalence class of inputs

\[\text{[Naik’18]}\]
Example of Computation Tree

```c
void test_me(int x, int y) {
    if (2*y == x) {
        if (x <= y+10)
            print("OK");
        else {
            print("something bad");
            ERROR;
        }
    } else
        print("OK");
}
```
Existing Approach I

• Random Testing:
  • Generate random inputs
  • Execute the program on those (concrete) inputs

• Problem:
  • Probability of reaching error could be astronomically small

Probability of ERROR:

\[
\frac{1}{2^{32}} \approx 0.000000023\%
\]

```c
void test_me(int x) {
    if (x == 94389) {
        ERROR;
    }
}
```
Existing Approach II

• Symbolic Execution
  • Use symbolic values for inputs
  • Execute program symbolically on symbolic input values
  • Collect symbolic path constraints
  • Use theorem prover to check if a branch can be taken

• Problem:
  • Does not scale for large programs

```c
void test_me(int x) {
  if (x*3 == 15) {
    if (x % 5 == 0)
      print("OK");
    else {
      print("something bad");
      ERROR;
    }
  } else
    print("OK");
}
```
Existing Approach II

- Symbolic Execution
  - Use symbolic values for inputs
  - Execute program symbolically on symbolic input values
  - Collect symbolic path constraints
  - Use theorem prover to check if a branch can be taken

- Problem:
  - Does not scale for large programs

```c
void test_me(int x) {
    // c = product of two large primes
    if (pow(2, x) % c == 17) {
        print("something bad");
        ERROR;
    } else
        print("OK");
}
```

Symbolic execution will say both branches are reachable: False Positive
Combined Approach

- Dynamic Symbolic Execution (DSE)
  - Start with random input values
  - Keep track of both concrete values and symbolic constraints
  - Use concrete values to simplify symbolic constraints
  - Incomplete theorem-prover

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```
An Illustrative Example

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

### Concrete Execution
- **x = 22**
- **y = 7**

### Symbolic Execution
- **x = x₀**
- **y = y₀**

### Path Condition
- Symbolic state
- Concrete state

[Naik’18]
### An Illustrative Example

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

#### Concrete Execution
- **x**: 22
- **y**: 7
- **z**: 14

#### Symbolic Execution
- **x**: $x_0$
- **y**: $y_0$
- **z**: $2*y_0$
An Illustrative Example

```
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution
- **x** = 22
- **y** = 7
- **z** = 14

Symbolic Execution
- **x** = **x₀**
- **y** = **y₀**
- **z** = **2*y₀**

Path Condition
- 2*<sup>**y₀**</sup> != **x₀**
An Illustrative Example

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 22</td>
<td>x = x₀</td>
<td>2*y₀ ≠ x₀</td>
</tr>
<tr>
<td>y = 7</td>
<td>y = y₀</td>
<td></td>
</tr>
<tr>
<td>z = 14</td>
<td>z = 2*y₀</td>
<td></td>
</tr>
</tbody>
</table>

Solve: 2*y₀ == x₀

Solution: x₀ = 2, y₀ = 1
An Illustrative Example

int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
An Illustrative Example

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution

<table>
<thead>
<tr>
<th>Concrete state</th>
<th>Symbolic state</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 2</td>
<td>x = x₀</td>
<td>2*y₀ == x₀</td>
</tr>
<tr>
<td>y = 1</td>
<td>y = y₀</td>
<td></td>
</tr>
<tr>
<td>z = 2</td>
<td>z = 2*y₀</td>
<td></td>
</tr>
</tbody>
</table>
An Illustrative Example

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 2</td>
<td>x = x₀</td>
<td>2∗y₀ == x₀</td>
</tr>
<tr>
<td>y = 1</td>
<td>y = y₀</td>
<td>x₀ &lt;= y₀+10</td>
</tr>
<tr>
<td>z = 2</td>
<td>z = 2∗y₀</td>
<td></td>
</tr>
</tbody>
</table>
An Illustrative Example

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution:
- **concrete state**
  - `x = 2`
  - `y = 1`
  - `z = 2`

Symbolic Execution:
- **symbolic state**
  - `x = x_0`
  - `y = y_0`
  - `z = 2*y_0`

Path Condition:
- `2*y_0 == x_0`
- `x_0 <= y_0+10`

**Solve:** `(2*y_0 == x_0) and (x_0 > y_0+10)`

**Solution:** `x_0 = 30, y_0 = 15`
An Illustrative Example

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution:
- `x = 30`
- `y = 15`

Symbolic Execution:
- `x = x_0`
- `y = y_0`

Path Condition:
- Check if `x > y + 10`
An Illustrative Example

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>concrete state</strong></td>
<td><strong>symbolic state</strong></td>
</tr>
<tr>
<td>$x = 30$</td>
<td>$x = x_0$</td>
</tr>
<tr>
<td>$y = 15$</td>
<td>$y = y_0$</td>
</tr>
<tr>
<td>$z = 30$</td>
<td>$z = 2*y_0$</td>
</tr>
</tbody>
</table>

Symbolic Execution:
- $x = x_0$
- $y = y_0$
- $z = 2*y_0$

Path Condition:
- ERROR
An Illustrative Example

```c
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>concrete state</strong></td>
<td><strong>symbolic state</strong></td>
<td></td>
</tr>
<tr>
<td>x = 30</td>
<td>x = x&lt;sub&gt;0&lt;/sub&gt;</td>
<td>2*y&lt;sub&gt;0&lt;/sub&gt; == x&lt;sub&gt;0&lt;/sub&gt;</td>
</tr>
<tr>
<td>y = 15</td>
<td>y = y&lt;sub&gt;0&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>z = 30</td>
<td>z = 2*y&lt;sub&gt;0&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An Illustrative Example

```
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution

- **Concrete State**
  - \(x = 30\)
  - \(y = 15\)
  - \(z = 30\)

- **Symbolic State**
  - \(x = x_0\)
  - \(y = y_0\)
  - \(z = 2y_0\)

- **Path Condition**
  - \(2y_0 \geq x_0\)
  - \(x_0 > y_0 + 10\)

Program Error
A More Complex Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y + 10)
            ERROR;
}
```

**Concrete Execution**

- concrete state
  - \( x = 22 \)
  - \( y = 7 \)

**Symbolic Execution**

- symbolic state
  - \( x = x_0 \)
  - \( y = y_0 \)

**Path Condition**
A More Complex Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution

- **Concrete state**
  - `x = 22`
  - `y = 7`
  - `z = 601...129`

Symbolic Execution

- **Symbolic state**
  - `x = x_0`
  - `y = y_0`
  - `z = secure_hash(y_0)`

Path condition
A More Complex Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 22 )</td>
<td>( x = x_0 )</td>
<td>secure_hash(( y_0 )) \neq x_0</td>
</tr>
<tr>
<td>( y = 7 )</td>
<td>( y = y_0 )</td>
<td></td>
</tr>
<tr>
<td>( z = 601...129 )</td>
<td>( z = secure_hash(y_0) )</td>
<td></td>
</tr>
</tbody>
</table>

Symbolic Execution

Solve: \( secure_hash(y_0) = x_0 \)

Don’t know how to solve! Stuck?
A More Complex Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
    
Not stuck! Use concrete state: replace \( y_0 \) by 7
```
A More Complex Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y + 10)
            ERROR;
}
```

**Concrete Execution**
- `x = 22`
- `y = 7`
- `z = 601...129`

**Symbolic Execution**
- `x = x_0`
- `y = y_0`
- `z = secure_hash(y_0)`

**Path Condition**
- `secure_hash(y_0) != x_0`

**Solve:**
- `601...129 == x_0`

**Solution:**
- `x_0 = 601...129, y_0 = 7`
A More Complex Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution

- **Concrete State**
  - `x = 601...129`
  - `y = 7`

Symbolic Execution

- **Symbolic State**
  - `x = x_0`
  - `y = y_0`

- **Path Condition**
A More Complex Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution

- **Concrete state**
  - \( x = 601\ldots129 \)
  - \( y = 7 \)
  - \( z = 601\ldots129 \)

Symbolic Execution

- **Symbolic state**
  - \( x = x_0 \)
  - \( y = y_0 \)
  - \( z = \text{secure_hash}(y_0) \)

Path condition

[Naik’18]
A More Complex Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x) {
        if (x > y + 10)
            ERROR;
    }
}
```

Concrete Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 601...129</td>
<td>x = x_0</td>
<td>secure_hash(y_0) == x_0</td>
</tr>
<tr>
<td>y = 7</td>
<td>y = y_0</td>
<td></td>
</tr>
<tr>
<td>z = 601...129</td>
<td>z = secure_hash(y_0)</td>
<td></td>
</tr>
</tbody>
</table>

[Naik’18]
A More Complex Example

```c
int foo(int v) {
    return secure_hash(v);
}
void test_me(int x, int y) {
    int z = foo(y);
    if (z == x) {
        if (x > y+10)
            ERROR;
    }
}
```

Concrete Execution
- **concrete state**: $x = 601...129$, $y = 7$, $z = secure_hash(y) = 601...129$
- **symbolic state**: $x = x_0$, $y = y_0$, $z = secure_hash(y_0)$
- **path condition**: $secure_hash(y_0) == x_0$ \(\Rightarrow x_0 > y_0 + 10\)

Program Error

[Naik’18]
Example Application

- DSE tests the below program starting with input \( x = 1 \). What is the input and constraint \((C_1 \land C_2 \land C_3)\) solved in each run of DSE? Use depth-first search and leave trailing constraints blank if unused.

```c
int test_me(int x) {
    int[] A = {5, 7, 9};
    int i = 0;
    while (i < 3) {
        if (A[i] == x)
            break;
        i++;
    }
    return i;
}
```

<table>
<thead>
<tr>
<th>Run</th>
<th>x</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5 != x0</td>
<td>7 != x0</td>
<td>9 == x0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Application

- DSE tests the below program starting with input \( x = 1 \). What is the input and constraint \((C_1 \land C_2 \land C_3)\) solved in each run of DSE? Use depth-first search and leave trailing constraints blank if unused.

<table>
<thead>
<tr>
<th>Run</th>
<th>x</th>
<th>( C_1 )</th>
<th>( C_2 )</th>
<th>( C_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>( 5 \neq x_0 )</td>
<td>( 7 \neq x_0 )</td>
<td>( 9 = x_0 )</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>( 5 \neq x_0 )</td>
<td>( 7 = x_0 )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>( 5 = x_0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
int test_me(int x) {
    int[] A = { 5, 7, 9 };
    int i = 0;
    while (i < 3) {
        if (A[i] == x)
            break;
        i++;
    }
    return i;
}
```
A Third Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    if (x != y)
        if (foo(x) == foo(y))
            ERROR;
}
```

**Concrete Execution**
- `x = 22`
- `y = 7`

**Symbolic Execution**
- `x = x_0`
- `y = y_0`

[Naik’18]
A Third Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    if (x != y)
        if (foo(x) == foo(y))
            ERROR;
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>concrete state</strong></td>
<td><strong>symbolic state</strong></td>
</tr>
<tr>
<td>x = 22</td>
<td>x = x₀</td>
</tr>
<tr>
<td>y = 7</td>
<td>y = y₀</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₀ != y₀</td>
</tr>
</tbody>
</table>

[Naik’18]
A Third Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    if (x != y)
        if (foo(x) == foo(y))
            ERROR;
}
```

### Concrete Execution
- **concrete state**
  - `x = 22`
  - `y = 7`

### Symbolic Execution
- **symbolic state**
  - `x = x_0`
  - `y = y_0`
- **path condition**
  - `x_0 != y_0`
  - `secure_hash(x_0) != secure_hash(y_0)`

### Solve
- `x_0 != y_0` and
- `secure_hash(x_0) == secure_hash(y_0)`

**Use concrete state:** replace `y_0` by 7.

[Naik’18]
A Third Example

```c
int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    if (x != y)
        if (foo(x) == foo(y))
            ERROR;
}
```

### Concrete Execution
- **Concrete state**
  - \( x = 22 \)
  - \( y = 7 \)

### Symbolic Execution
- **Symbolic state**
  - \( x = x_0 \)
  - \( y = y_0 \)

**Path condition**
- \( x_0 \neq y_0 \)
- \( \text{secure_hash}(x_0) \neq \text{secure_hash}(y_0) \)

---

**Solve:** \( x_0 \neq 7 \) and
- \( \text{secure_hash}(x_0) = 601...129 \)

**Use concrete state:** replace \( x_0 \) by 22.

[Naik’18]
A Third Example

int foo(int v) {
    return secure_hash(v);
}

void test_me(int x, int y) {
    if (x != y)
    if (foo(x) == foo(y))
        ERROR;
}

False negative!

Concrete Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 22</td>
<td>x = x_0</td>
</tr>
<tr>
<td>y = 7</td>
<td>y = y_0</td>
</tr>
</tbody>
</table>

Symbolic state

path condition

x_0 != y_0
secure_hash(x_0) != secure_hash(y_0)

Solve: 22 != 7 and 438...861 == 601...129
Unsatisfiable!
Another Example: Testing Data Structures

- Random Test Driver:
  - random value for x
  - random memory graph reachable from p

- Probability of reaching ERROR is extremely low

```c
typedef struct cell {
    int data;
    struct cell *next;
} cell;

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
```
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

Concrete Execution

symbolic state

x = 236
p = NULL

path condition

symbolic state

x = x₀
p = p₀

concrete state

x = 236
p = NULL
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

Concrete Execution

symbolic state

concrete state

x = 236
p = NULL

x = x_0
p = p_0

path condition

x_0 > 0

[Naik’18]
typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

Concrete Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 236</td>
<td>x = x₀</td>
<td>x₀ &gt; 0</td>
</tr>
<tr>
<td>p = NULL</td>
<td>p = p₀</td>
<td>p₀ == NULL</td>
</tr>
</tbody>
</table>

Symbolic Execution

Solve: x₀ > 0 and p₀ != NULL

Solution: x₀ = 236, p₀ → 634 NULL
typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

Concrete Execution

Symbolic Execution

Concrete state
x = 236
p = 634 NULL

Symbolic state
x = x_0
p = p_0
p->data = v_0
p->next = n_0

path condition
x_0 > 0

Naik’18
typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
**Data-Structure Example**

### Concrete Execution
- **x** = 236
- **p** → 634 | NULL

### Symbolic Execution
- **x** = **x₀**
- **p** = **p₀**
- **p**→data = **v₀**
- **p**→next = **n₀**

### Path Condition
- **x₀** > 0
- **p₀** != NULL
- 2*<sup>x₀</sup>+1 != **v₀**

```c
typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
```
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

int x = 236;
int p = foo(x); // Concrete state
p->data = 634;
p->next = NULL;

Solve: \( x_0 > 0 \) and \( p_0 \neq \text{NULL} \) and \( 2x_0 + 1 = v_0 \)

Solution: \( x_0 = 1,\ p_0 \rightarrow 3\ \text{NULL} \)
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

Concrete Execution

<table>
<thead>
<tr>
<th>concrete state</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
</tr>
<tr>
<td>p = p₀</td>
</tr>
<tr>
<td>p-&gt;data = v₀</td>
</tr>
<tr>
<td>p-&gt;next = n₀</td>
</tr>
</tbody>
</table>

Symbolic Execution

<table>
<thead>
<tr>
<th>symbolic state</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = x₀</td>
</tr>
<tr>
<td>p = p₀</td>
</tr>
<tr>
<td>p-&gt;data = v₀</td>
</tr>
<tr>
<td>p-&gt;next = n₀</td>
</tr>
</tbody>
</table>

path condition

| x₀ > 0         |
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

Concrete Execution

Symbolic Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>x = x₀</td>
<td>x₀ &gt; 0</td>
</tr>
<tr>
<td>p = p₀</td>
<td>p₀ != NULL</td>
<td></td>
</tr>
<tr>
<td>p-&gt;data = v₀</td>
<td>p-&gt;data = v₀</td>
<td>2*x₀ + 1 == v₀</td>
</tr>
<tr>
<td>p-&gt;next = n₀</td>
<td>p-&gt;next = n₀</td>
<td></td>
</tr>
</tbody>
</table>
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

Concrete Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>x = x₀</td>
</tr>
<tr>
<td>p = p₀</td>
<td>p = p₀</td>
</tr>
<tr>
<td>p-&gt;data = v₀</td>
<td>p-&gt;data = v₀</td>
</tr>
<tr>
<td>p-&gt;next = n₀</td>
<td>p-&gt;next = n₀</td>
</tr>
</tbody>
</table>

Symbolic Execution

<table>
<thead>
<tr>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₀ &gt; 0</td>
</tr>
<tr>
<td>p₀ != NULL</td>
</tr>
<tr>
<td>2*x₀ + 1 == v₀</td>
</tr>
<tr>
<td>n₀ != p₀</td>
</tr>
</tbody>
</table>

Solve: x₀ > 0 and p₀ != NULL and 2*x₀ + 1 == v₀ and n₀ == p₀

Solution: x₀ = 1, p₀ = 3

[Naik’18]
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

Concrete Execution

| x = 1 |
| p    |

Symbolic Execution

| x = x_0 |
| p = p_0 |
| p->data = v_0 |
| p->next = n_0 |

path condition

[Naik’18]
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

Concrete Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>Symbolic Execution</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>x = x₀</td>
<td>x₀ &gt; 0</td>
</tr>
<tr>
<td>p → 3</td>
<td>p = p₀</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p-&gt;data = v₀</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p-&gt;next = n₀</td>
<td></td>
</tr>
</tbody>
</table>
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

Concrete Execution
symbolic state
x = x_0
p = p_0
p->data = v_0
p->next = n_0

path condition
x_0 > 0
p_0 != NULL
2*x_0+1 == v_0

symbolic state
x = 1
p = 3
Data-Structure Example

typedef struct cell {
    int data;
    struct cell *next;
} cell;

int foo(int v) { return 2*v + 1; }

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}

Program Error

Concrete Execution

- concrete state:
  - \( x = 1 \)
- symbolic state:
  - \( x = x_0 \)
  - \( p = p_0 \)
  - \( p->data = v_0 \)
  - \( p->next = n_0 \)

Symbolic Execution

- path condition:
  - \( x_0 > 0 \)
  - \( p_0 != \) NULL
  - \( 2*x_0 + 1 == v_0 \)
  - \( n_0 != p_0 \)
Approach in a Nutshell

- Generate concrete inputs, each taking different program path
- On each input, execute program both concretely and symbolically
- Both cooperate with each other:
  - Concrete execution guides symbolic execution
    - Enables it to overcome incompleteness of theorem prover
  - Symbolic execution guides generation of concrete inputs
    - Increases program code coverage

[Naik’18]
Realistic Implementations

- KLEE: LLVM (C family of languages)
- PEX: .NET Framework
- jCUTE: Java
- Jalangi: Javascript
- SAGE and S2E: binaries (x86, ARM, ...)

[Naik’18]
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 --- \( y = x / z \) where \( x \) and \( z \) are symbolic variables and assume current PC (i.e. path constraint) is \( f \)
  - Even though we only fork in branches we will now fork in the division operator
  - One branch in which \( z = 0 \) and another where \( z \neq 0 \)
  - We will get two paths with the following constraints:
    - \( z = 0 \) \&\& \( f \), \( z \neq 0 \) \&\& \( f \)
    - Solving the constraint \( z = 0 \) \&\& \( f \) will give us concrete input values that will trigger the divide by zero error.
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: \( y = x / z \) where \( x \) and \( z \) are symbolic variables, and assume current PC (i.e. path constraint) is \( f \).
  - Even though we only fork in branches, we will now fork in the division operator.
  - One branch in which \( z = 0 \) and another where \( z \neq 0 \).
  - We will get two paths with the following constraints:
    - \( z = 0 \) \&\& \( f \), \( z \neq 0 \) \&\& \( f \)
  - Solving the constraint \( z = 0 \) \&\& \( f \) 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
EXE vs. KLEE and the UC-KLEE
EXE: Automatically Generating Inputs of Death,
David Dill, Vijay Ganesh, Cristian Cadar, Dawson Engler,
Peter Pawlowski, CCS’06
What if you could find all the bugs in your code, automatically?
EXE: EXecution generated Executions

• The Idea:
  • Code can automatically generate its own (potentially highly complex) test cases
EXE: EXecution generated Executions

- The Algorithm
  - Symbolic execution
    +
  - Constraint solving

[Leibowitz’13]
EXE: EXecution generated Executions

• As program runs
  • Executes each feasible path, tracking all constraints

• A path terminates upon
  • exit() crash
  • failed ‘assert’ error detection

• When a path terminates
  • Calls STP to solve the path’s constraints for concrete values
EXE: EXecution generated Executions

- Identifies all input values causing these errors
  - Null or Out-of-bounds memory reference
  - Overflow
  - Division or modulo by 0
- Identifies all input values causing assert invalidation
Example

```c
int main(void) {
    unsigned int i, t, a[4] = { 1, 3, 5, 2 }; 
    
    if (i >= 4)
        exit(0); 
    char *p = (char *)a + i * 4; 
    *p = *p - 1 
    t = a[*p]; 
    t = t / a[i]; 
    if (t == 2)
        assert(i == 1); 
    else 
        assert(i == 3); 
    return 0; 
}
```

[Leibowitz’13]
int main(void) {  
    unsigned int i, t, a[4] = { 1, 3, 5, 2 };  
    make_symbolic(&i);  
    if (i >= 4)  
        exit(0);  
    char *p = (char *)a + i * 4;  
    *p = *p - 1  
    t = a[*p];  
    t = t / a[i];  
    if (t == 2)  
        assert(i == 1);  
    else  
        assert(i == 3);  
    return 0;  
}  

Marks the 4 bytes associated with 32-bit variable ‘i’ as symbolic
Compiling...

**Example C Code**

```c
int main(void) {
    unsigned int i, t, a[4] = {1, 3, 5, 2};
    make_symbolic(&i);
    if (i >= 4)
        exit(0);
    char *p = (char *)a + i * 4;
    *p = *p - 1;
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
    return 0;
}
```

**EXE Compiler**

**Example.out**

**Executable**

Inserts checks around every assignment, expression & branch, to determine if its operands are concrete or symbolic.

**UnSigned Int A[4] = {1,3,5,2}**

**If (i >= 4)**
Compiling...

example.c

```c
int main(void) {
    unsigned int i, t, a[4] = {1, 3, 5, 2};

    make_symbolic(&i);
    if (i >= 4)
        exit(0);
    char *p = (char *)a + i * 4;
    *p = *p - 1;
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
    return 0;
}
```

EXE compiler

Executable

example.out

Inserts checks around every assignment, expression & branch, to determine if its operands are concrete or symbolic

If any operand is symbolic, the operation is not performed, but is added as a constraint for the current path

[Leibowitz’13]
Compiling...

```
int main(void) {
    unsigned int i, t, a[4] = { 1, 3, 5, 2 }
    make_symbolic(&i);
    if (i >= 4)
        exit(0);
    char *p = (char *)a + i * 4;
    *p = *p - 1
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
    return 0;
}
```

Inserts code to fork program execution when it reaches a symbolic branch point, so that it can explore each possibility

**Example:**

- **if (i >= 4)**
- **(i < 4)**

```
EXE compiler
```

```
example.out
```

Executable

Spring 1398
Ce 874 - Symbolic Execution

[Leibowitz’13]
Compiling...

```
int main(void) {
    unsigned int i, t, a[4] = { 1, 3, 5, 2 }

    make_symbolic(&i);
    if (i >= 4)
        exit(0);
    char *p = (char *)a + i * 4;
    *p = *p - 1
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
    return 0;
}
```

 Inserts code to fork program execution when it reaches a symbolic branch point, so that it can explore each possibility

 For each branch constraint, queries STP for existence of at least one solution for the current path. If not – stops executing path
Inserts code for checking if a symbolic expression could have any possible value that could cause errors
Compiling...

```c
int main(void) {
    unsigned int i, t, a[4] = {1, 3, 5, 2};
    make_symbolic(&i);
    if (i >= 4)
        exit(0);
    char *p = (char *)a + i * 4;
    *p = *p - 1;
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
    return 0;
}
```

Inserts code for checking if a symbolic expression could have any possible value that could cause errors

If the check passes – the path has been verified as safe under all possible input values

EXE compiler

```c
EXE compiler
```

`example.out` Executable

3
Running…

```c
int main(void) {
    unsigned int i, t, a[4] = { 1, 3, 5, 2 };
    make_symbolic(&i);
    if (i >= 4)
        exit(0);
    char *p = (char *)a + i * 4;
    *p = *p - 1;
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
    return 0;
}
```

e.g. $i = 8$

EXE generates a test case

$4 \leq i$

[Leibowitz’13]
Running...

```c
int main(void) {
    unsigned int i, t, a[4] = {1, 3, 5, 2};
    make_symbolic(&i);
    if (i >= 4)
        exit(0);
    char *p = (char *)a + i * 4;
    *p = *p - 1
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
    return 0;
}
```

```
e.g.   i = 2
p → a[2] = 5
a[2] = 5 – 1 = 4
t = a[4]
Out of bounds
EXE generates a test case
```

Spring 1398  Ce 874 - Symbolic Execution  [Leibowitz’13]
Running…

```c
int main(void) {
    unsigned int i, t, a[4] = {1, 3, 5, 2};
    make_symbolic(&i);
    if (i >= 4)
        exit(0);
    char *p = (char *)a + i * 4;
    *p = *p - 1
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
    return 0;
}
```

0\leq i \leq 4, \ i \neq 2

- e.g.  i = 0
- p \rightarrow a[0] = 1
- a[0] = 1 - 1 = 0
- t = a[0]
- t = t / 0

Division by 0

EXE generates a test case

[Leibowitz’13]
Running…

```c
int main(void) {
    unsigned int i, t, a[4] = { 1, 3, 5, 2 };
    make_symbolic(&i);
    if (i >= 4)
        exit(0);
    char *p = (char*)a + i * 4;
    *p = *p - 1;
    t = a[*p];
    t = t / a[i];
    if (t == 2)
        assert(i == 1);
    else
        assert(i == 3);
    return 0;
}
```

0 ≤ i ≤ 4 , i ≠ 2 , i ≠ 0

- i = 3
  - p → a[3]
  - a[3] = 1
  - t = a[1]
  - t ≠ 2
  - EXE determines neither ‘assert’ fails
- i = 1
  - p → a[1]
  - a[1] = 2
  - t = a[2]
  - t = 2

2 valid test cases
Optimizations

1. Caching constraints to avoid calling STP
   • Goal – avoid calling STP when possible
   • Results of queries and constraint solutions are cached
   • Cache is managed by a server process
   • Naïve implementation – significant overhead

---

[Leibowitz’13]
1. Caching constraints to avoid calling STP
   • Goal – avoid calling STP when possible
   • Results of queries and constraint solutions are cached
   • Cache is managed by a server process
   • Naïve implementation – significant overhead
Optimizations

1. Caching constraints to avoid calling STP
   - Goal – avoid calling STP when possible
   - Results of queries and constraint solutions are cached
   - Cache is managed by a server process
   - Naïve implementation – significant overhead

[Leibowitz’13]
Optimizations

2. Constraint Independence
   • Breaking constraints into multiple, independent, subsets
   • Discard irrelevant constraints
   • Small cost for computing independent subsets
   • May yield additional cache hits
Optimizations

2. Constraint Independence

```cpp
if (A[i] > A[i+1]) {
    ...
}
if (B[j] + B[j-1] == B[j+1]) {
    ...
}
```

4 possible paths

2 consecutive independent branches

- \((A[i] \leq A[i+1]) \land (B[j] + B[j-1] \neq B[j+1])\)
Optimizations

2. Constraint Independence

1. $A[i] \leq A[i+1]$
   - 1st “if”
   - 2nd “if”
   - no optimization

   - 2nd “if”

(A[i] ≤ A[i+1])
&&
(B[j] + B[j-1] ≠ B[j+1])

(A[i] ≤ A[i+1])
&&
(B[j] + B[j-1] = B[j+1])

(A[i] > A[i+1])
&&
(B[j] + B[j-1] ≠ B[j+1])

(A[i] > A[i+1])
&&
(B[j] + B[j-1] = B[j+1])

[Leibowitz’13]
2. Constraint Independence

with optimization

1. $A[i] \leq A[i+1]$

3. $(B[j] + B[j-1] \neq B[j+1])$

with optimization

$(B[j] + B[j-1] \neq B[j+1])$
$(B[j] + B[j-1] = B[j+1])$

[Leibowitz’13]
Optimizations

2. Constraint Independence

\[ 2(2^n - 1) \text{ queries to STP} \]

\[ 2n \text{ queries to STP} \]

[Leibowitz’13]
3. Search Heuristics – “Best First Search” & DFS

- By default, EXE uses DFS when forking for picking which branch to follow first
- Problem – Loops bounded by symbolic variables
- Solution
  - Each forked process calls search server, and blocks
  - Server picks process blocked on line of code which has run the fewer number of times
  - Picked process and children are run with DFS
Optimizations

• Experimental Performance
  • Used to find bugs in
    • 2 packet filters (FreeBSD & Linux)
    • DHCP server (udhcppd)
    • Perl compatible regular expressions library (pcre)
    • XML parser library (expat)
  • Ran EXE without optimizations, with each optimization separately, and with all optimizations
Optimizations

- Experimental Performance
  - Positive
    - With both caching & independence – Faster by 7%-20%
    - Cache hit rate jumps sharply with independence
    - Cost of independence – near zero
    - Best First Search gets (almost) full coverage more than twice as fast than DFS
    - Coverage with BFS compared to random testing: 92% against 57%

[Leibowitz’13]
Optimizations

• Experimental Performance
  • Interesting
    • Actual growth of number of paths is much smaller than potentially exponential growth
    • EXE is able to handle relatively complex code
  • Negative
    • Cache lookup has significant overhead, as conversion of queries to string is dominant
    • STP by far remains highest cost (as expected)

[Leibowitz’13]
Advantages

• Automation – “competition” is manual and random testing
• Coverage - can test any executable code path and (given enough time) exhaust them all
• Generation of actual attacks and exploits
• No false positives
Limitations

- Optimizations – far from perfect implementation
- Benchmarks – hand-picked, small-scaled
- Single threaded – each path is explored independently from others
- Code doesn’t interact with it’s surrounding environment
KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, Cristian Cadar, Daniel Dunbar, Dawson Engler, OSDI’08
KLEE

• Shares main idea with EXE, but completely redesigned
• Deals with the external environment
• More optimizations, better implemented
• Targeted at checking system-intensive programs “out of the box”
• Thoroughly evaluated on real, more complicated, environment-intensive programs
KLEE

- A hybrid between an **operating system for symbolic processes** and an **interpreter**
  - Programs are compiled to virtual instruction sets in LLVM assembly language
  - Each symbolic process (“state”) has a symbolic environment
    - register file   stack   heap
    - program counter path condition
  - Symbolic environment of a state (unlike a normal process)
    - Refers to symbolic expressions and not concrete data values

[Leibowitz’13]
KLEE

- Able to execute a large number of states simultaneously
- At its core – an interpreter loop
  - Selects a state to run (search heuristics)
  - Symbolically executes a single instruction in the context of the state
  - Continues until no remaining states
  - (or reaches user-defined timeout)
int badAbs(int x) {
    if (x < 0)
        return -x;
    if (x == 1234)
        return -x;
    return x;
}

x ≥ 0
x ≠ 1234
x = 3

STP Solver
Execution

• Conditional Branches
  • Queries STP to determine if the branch condition is true or false
  • The state’s instruction pointer is altered suitably
  • Both branches are possible?
    • State is cloned, and each clone’s instruction pointer and path condition are updated appropriately
Execution

• Targeted Errors
  • As in EXE
    • Division by 0
    • Overflow
    • Out-of-bounds memory reference

[Leibowitz’13]
Modeling the Environment

- Code reads/writes values from/to its environment
  - Command line arguments
  - Environment variables
  - File data
  - Network packets
- Want to return all possible values for these reads
- How?
  - Redirecting calls that access the environment to custom models
Modeling the Environment

- Example: Modeling the File System
  - File system operations
    - Performed on an actual concrete file on disk?
      - Invoke the corresponding system call in the OS
    - Performed on a symbolic file?
      - Emulate the operation’s effect on a simple symbolic file system (private for each state)
  - Defined simple models for 40 system calls
Modeling the Environment

- Example: Modeling the File System
  - Symbolic file system
    - Crude
      - Contains a single directory with N symbolic files
      - User can specify N and size of files
    - Coexists with real file system
      - Applications can use files in both

[Leibowitz’13]
Modeling the Environment

• Failing system calls
  • Environment can fail in unexpected ways
    • `write()` when disk is full
    • Unexpected, hard-to-diagnose bugs
  • Optionally simulates environmental failures
    • Failing system calls in a controlled manner
Optimizations

1. Compact State Representation
   • Number of concurrent states grows quickly (even >100,000)
   • Implements copy-on-write at object level
     • Dramatically reduces memory requirements per state
     • Heap structure can be shared amongst multiple states
     • Can be cloned in constant time (very frequent operation)

[Leibowitz’13]
2. Simplifying queries
   • Cost of constraint solving dominates everything else
     • Make solving faster
     • Reduce memory consumption
     • Increase cache hit rate (to follow)
Optimizations

2. Simplifying queries
   a. Expression Rewriting
      • Simple arithmetic simplifications
        \[ x + 0 \rightarrow x \]
        \[ x * 2^n \rightarrow x << n \]
      • Strength reduction
      • Linear simplification
        \[ 2x - x \rightarrow x \]
Optimizations

2. Simplifying queries
   b. Constraint Set Simplification
      • Constraints on same variables tend to become more specific
      • Rewrites previous constraints when new, equality constraints, are added to the set

\[ x < 10 \]
Optimizations

2. Simplifying queries
   b. Constraint Set Simplification
      • Constraints on same variables tend to become more specific
      • Rewrites previous constraints when new, equality constraints, are added to the set

\[
\begin{align*}
x &< 10 \\
x & = 5
\end{align*}
\]

[Leibowitz’13]
2. Simplifying queries
   b. Constraint Set Simplification
   • Constraints on same variables tend to become more specific
   • Rewrites previous constraints when new, equality constraints, are added to the set

\[
\begin{align*}
\text{x} &< 10 \\
\text{x} & = 5
\end{align*}
\rightarrow
\begin{align*}
\text{true}
\end{align*}
\]
Optimizations

- Simplifying queries
  - c. Implied Value Concretization
    - The value of a variable effectively becomes concrete
    - Concrete value is written back to memory

\[ x + 1 = 10 \quad \rightarrow \quad x = 9 \]
Optimizations

• Simplifying queries
  d. Constraint Independence

• As in EXE
Optimizations

3. Counter-Example Cache

- More sophisticated than in EXE
- Allows efficient searching for cache entries for both subsets and supersets of a given set

$$\{ i < 10, i = 10 \}$$ unsatisfiable

$$\{ i < 10, j = 8 \}$$ (i = 5, j = 8)
3. Counter-Example Cache

- More sophisticated than in EXE
- Allows efficient searching for cache entries for both subsets and superset of a given set

\[
\begin{align*}
\{ i < 10, i = 10, j = 12 \} & \quad \text{Superset of (1)} \\
\{ i < 10, i = 10 \} & \quad \text{unsatisfiable} \\
\{ i < 10, j = 8 \} & \quad (i = 5, j = 8) \\
\{ i < 10, j = 8 \} & \quad \text{unsatisfiable}
\end{align*}
\]
Optimizations

3. Counter-Example Cache
   • More sophisticated than in EXE
   • Allows efficient searching for cache entries for both subsets and supersets of a given set

\[
\begin{align*}
\{ i < 10 \} & \quad \text{Subset of (2)} \\
\{ i < 10, i = 10 \} & \quad \text{unsatisfiable} \\
\{ i < 10, j = 8 \} & \quad (i = 5, j = 8) \\
( i = 5, j = 8 ) & 
\end{align*}
\]
3. Counter-Example Cache

• More sophisticated than in EXE

• Allows efficient searching for cache entries for both subsets and supersets of a given set

\[
\begin{align*}
\{ i < 10, j = 8, \ i \neq 3 \} & \quad \text{Superset of (2)} \\
\{ i < 10, i = 10 \} & \quad \text{unsatisfiable} \\
\{ i < 10, j = 8 \} & \quad ( i = 5, j = 8 ) \\
\end{align*}
\]
Optimizations

4. Search Heuristics – State Scheduling
   • The state to run at each instruction is selected by interleaving 2 strategies
   • Each is used in a Round-Robin fashion
   • Each state is run for a “time slice”
   • Ensures a state which frequently executes expensive instructions will not dominate execution time
Optimizations

4. Search Heuristics – State Scheduling
   a. Random Path Selection
      • Traverses tree of paths from root to leaves
        (internal nodes – forks, leaves – states)
      • At branch points – randomly selects path to follow
      • States in each subtree have equal probability of being selected
      • Favors states higher in the tree – less constraints, freedom to reach uncovered code
      • Avoids starvation (loop + symbolic condition = “forks bomb”)
Optimizations

4. Search Heuristics – State Scheduling
   b. Coverage-Optimized Search
      • Tries to select states more likely to cover new code
      • Computes min. distance to uncovered instruction, call stack size & whether state recently covered new code
      • Randomly selects a state according to these weights

[Leibowitz’13]
Optimizations

• Experimental Performance
  • Used to generate tests in
    • GNU COREUTILS Suite (89 programs)
    • BUSYBOX (72 programs)
    • Both have variety of functions, intensive interaction with the environment
    • Heavily tested, mature code
  • Used to find bugs in
    • Total of 450 applications
Optimizations

• Experimental Performance
  • Query simplification + caching
    • Number of STP queries reduced to 5% (!) of original
    • Time spent solving queries to STP reduced from 92% of overall time to 41% of overall time
  • Speedup

[Leibowitz’13]
Results – Line Coverage

- GNU COREUTILS
- Overall: 84%, Average 91%, Median 95%

Coverage (ELOC %)

Apps sorted by KLEE coverage

16 at 100%

[Leibowitz’13]
Results – Line Coverage

- GNU COREUTILS

<table>
<thead>
<tr>
<th>Avg/utility</th>
<th>KLEE</th>
<th>91%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>68%</td>
<td></td>
</tr>
</tbody>
</table>

15 Years of manual testing beaten in less than 89 hours

Apps sorted by KLEE coverage - Manual coverage

[Leibowitz’13]
Results – Line Coverage

• High coverage with few test cases
  • Average of 37 tests per tool in GNU COREUTILS
• “Out of the box” – utilities unaltered
• Entire tool suite (no focus on particular apps)
• However
  • Checks only low-level errors and violations
  • Developer tests also validate output to be as expected
Results – Bugs found

- 10 memory error crashes in GNU COREUTILS
  - More than found in previous 3 years combined
  - Generates actual command lines exposing crashes

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

```
t1.txt: "\t \tMD5 ("
t2.txt: "\b\b\b\b\b\b\b\b\b\b\b\b\t"
t3.txt: "\n"
t4.txt: "a"
```
Contributions

- Technique + tool for finding deep bugs in real, open source C/C++ code
  - No manual testcases
  - No functional specification
- Bugs reported may have security implications; exploitability must be determined manually
  - Memory access, heap management, assertion failures, division-by-zero
- Found 77 new bugs in BIND, OpenSSL, Linux kernel
  - 14 Linux kernel vulnerabilities (mostly minor DoS issues)
Problem: Scalability

- Path explosion
  - $|\text{paths}| \sim 2^{\text{if-statements}}$
- Path length and complexity
  - Undecidable: infinite-length paths (halting problem)
- SMT query complexity (NP-complete)
Solution: Under-Constrained

- Directly execute individual functions within a program
  - Less code = Fewer paths
  - Function calls executed (inter-procedural)
  - Able to test previously-unreachable code
- Challenges
  - Complex inputs (e.g., pointer-rich data structures)
  - Under-constrained: inputs have unknown preconditions
    - False positives
UC-KLEE tool

- Extends KLEE tool (OSDI 2008)
- Runs LLVM bitcode compiled from C/C++ source
- Automatically synthesizes complex inputs
  - Based on lazy initialization (Java PathFinder)
  - Supports pointer manipulation and casting in C/C++ (no type safety)
  - User-specified input depth (k-bound) [Deng 2006]

[Ramos’15]
Lazy Initialization

• Symbolic (input) pointers initially unbound
• On first dereference:
  • New object allocated
  • Symbolic pointer bound to new object’s address
• On subsequent dereferences:
  • Pointer resolves to object allocated above
Example

Unbound Symbolic Input

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
Example

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
Example

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```
Example

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```

\[ n \neq 0 \quad n \rightarrow \text{NULL} \]
\[ n \neq 0 \]

[Ramos’15]
Example

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```

[Ramos’15]
Example

int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}

[Ramos’15]
Example

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```

- \( n == 0 \)
- \( n \) NULL
- \( n != 0 \)
- \( n == \&uc\_node1 \)
- \( uc\_node1 \)
  - va1
  - next

[Ramos’15]
```
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```

[Ramos’15]
```
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```

[Ramos’15]
Example

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```

[Ramos’15]
Example

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```

[Ramos’15]
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
Example

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```

[Ramos’15]
Example

```c
int listSum(node *n) {
    int sum = 0;
    while (n) {
        sum += n->val;
        n = n->next;
    }
    return sum;
}
```

[Ramos’15]
Use Cases

- Equivalence checking: patches
  - Yesterday’s code vs. today’s code (i.e., fewer bugs today)
  - Goal: detect (and prevent!) new crashes introduced by patches
  - Other uses discussed in CAV 2011 paper
- General bug-finding: rule-based checkers
  - Single version of a function; under-constrained + additional checker rules
  - Memory leaks, uninitialized data, unsafe user input
  - Simple interface for adding new checkers
Equivalence Checking

- Value equivalence
  - Return value
  - Arguments passed by reference
  - Global/static variables
  - System call effects (modeled)
- Error (crash) equivalence
  - Both versions typically have the same same (unknown) preconditions!
  - Neither version crashes on an input
  - Both versions crash on an input

USE CASE: whether patches introduce crashes

[Ramos’15]
Equivalence Checking

• Check per path equivalence of two functions
• If all paths exhausted, equivalence verified (up to input bound)
Evaluation

- BIND, OpenSSL
  - Mature, security-critical codebases (~400 KLOC each)
- Patches
  - BIND: 487 patches to 9.9 stable (14 months)
  - OpenSSL: 324 patches to 1.0.1 stable (27 months)
- Ran UC-KLEE for 1 hour on each patched function
Evaluation: Patches

- Discovered 10 new bugs (4 in BIND, 6 in OpenSSL)
  - 2 OpenSSL DoS vulnerabilities:
    - CVE-2014-0198: NULL pointer dereference
    - CVE-2015-0292: Out-of-bounds memcpy read
- Verified (w/ caveats) that patches do not introduce crashes
  - 67 (13.8%) for BIND, 48 (14.8%) for OpenSSL

- More results available in the publication
Acknowledgments/References

• [Chowdhury’15] Information Security, CS 526, Omar Chowdhury, University of Iowa, 2015