Advanced Software Testing and Debugging (CS598)
Symbolic Execution

Spring 2022
Lingming Zhang
Brief 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
Program execution paths

• **Program** can be viewed as binary tree with possibly infinite depth

• 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
Example

Code under test

```csharp
void CoverMe(int[] a) {
    if (a == null)
        return;
    if (a.Length > 0)
        if (a[0] == 1234567890)
            throw new Exception("bug");
}
```
Random testing?

**Code under test**
```java
void CoverMe(int[] a) {
    if (a == null)
        return;
    if (a.Length > 0)
        if (a[0] == 1234567890)
            throw new Exception("bug");
}
```

**Random Testing**
- Generate random inputs
- Execute the program on those (concrete) inputs

**Problem:**
- Probability of reaching error could be astronomically small

**Probability of ERROR for the gray branch:**
\[ \frac{1}{2^{32}} \approx 0.000000023\% \]
The spectrum of program testing/verification

- Random/fuzz testing
- Concolic testing & whitebox fuzzing
- Bounded verification & symbolic execution
- Verification

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

• KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs (OSDI'08)
• Hybrid Concolic Testing (ICSE'07)
Symbolic execution

• Symbolic Execution
  • Use symbolic values for inputs
  • Execute program symbolically on symbolic input values
  • Collect symbolic path constraints (PCs)
  • Use SMT/SAT solvers to check if a branch can be taken

```
int foo(int i) {
  int j=2*i;
  i=i++;
  i=i*j;
  if(i<1)
    i=-i;
  return i;
}
```

Code under test

Symbolic execution engine

Constraint solver

High-quality tests

Test generation

Path constraints

Solutions
Symbolic execution: example

**Code under test**

```c
int foo(int i) {
    int j=2*i;
    i=i++;
    i=i*j;
    if(i<1)
        i=-i;
    return i;
}
```

**Concrete execution**

- \( i = 1 \)
- \( i = 1, j = 2 \cdot 1 \)
- \( i = 1 + 1 \)
- \( i = 2 \cdot 2 \)
- \( j = 2 \cdot i \)
- \( PC = true \)

**Symbolic execution**

- \( i = I_0 \)
- \( PC = true, i = I_0, j = 2 \cdot I_0 \)
- \( PC = true, i = I_0 + 1, j = 2 \cdot I_0 \)
- \( PC = true, i = (I_0 + 1) \cdot 2I_0, j = 2 \cdot I_0 \)
- \( PC = true, i = (I_0 + 1) \cdot 2I_0, j = 2 \cdot I_0 \)
- \( PC = (I_0 + 1) \cdot 2I_0 < 1 \), \( i = -(I_0 + 1) \cdot 2I_0 \)
- \( PC = (I_0 + 1) \cdot 2I_0 < 1 \), \( return -(I_0 + 1) \cdot 2I_0 \)
- \( PC = (I_0 + 1) \cdot 2I_0 \geq 1 \), \( return (I_0 + 1) \cdot 2I_0 \)

**Generated test 1**

- \( i = I_0 = 0 \)

**Generated test 2**

- \( i = I_0 = 1 \)
Symbolic execution: bug finding

• How to extend symbolic execution to catch non-crash bugs?
• Add dedicated checkers at dangerous code locations!
  • Divide by zero example: \( y = \frac{x}{z} \) where \( x \) and \( z \) are symbolic variables and assume current PC is \( p \)
    • Check if \( z==0 && p \) is possible!

```
int foo(int i) {
    int j=2*i;
    i=i++;
    i=i*j;
    if(i<1)
        i=-i;
    i=j/i;
    return i;
}
```

We can easily generate a dedicated checker for each kind of bug (e.g., buffer overflow, integer overflow, ...)

Code under test
Challenges: path explosion

• Interleaving two search heuristics:
  • **Random Path Selection**: when a branch point is reached, the set of states in each subtree has equal probability of being selected
  • **Coverage-Optimized Search**: selects states likely to cover new code in the immediate future, based on
    • The minimum distance to an uncovered instruction
    • The call stack of the state
    • Whether the state recently covered new code
Challenges: 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**
  - \(x < 10 \land x = 5 \rightarrow x = 5\)

- **Implied value concretization**
  - \(x + 1 = 10 \rightarrow x = 9\)

- **Constraint independence**
  - \(i < j \land j < 20 \land k > 0 \land i = 20 \rightarrow i < j \land i < 20 \land i = 20\)
Challenges: optimizing SMT queries (cont.)

- Counter-example cache
  - $i < 10 \land i = 10$ (no solution)
  - $i < 10 \land j = 8$ (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
  - $\{i < 10\}$ (satisfiable with $i \rightarrow 5$, $j \rightarrow 8$)
- Superset of satisfiable constraints
  - Same variable assignments might work

![Graph showing the effect of KLEE’s solver optimizations over time](image)

**Figure 2:** The effect of KLEE’s solver optimizations over time, showing they become more effective over time, as the caches fill and queries become more complicated. The number of executed instructions is normalized so that data can be aggregated across all applications.
Challenges: environment modeling

- If all arguments are concrete, forward to OS directly

- Otherwise, provide models that can handle symbolic files
  - Goal is to explore all possible interactions with the environment
  - About 2,500 LoC to define simple models for roughly 40 system calls
    - e.g., open, read, write, stat, lseek, ftruncate, ioctl

```c
int fd = open("t.txt", O_RDONLY);

int fd = open(sym_str, O_RDONLY);
```

Sketch of KLEE’s model for read()
KLEE implementation
Benchmarks

• 89 programs in GNU **Coreutils** (version 6.10), roughly 80,000 lines of library code and 61,000 lines in the actual utilities, including ones
  • Managing the file system (e.g., ls, dd, chmod)
  • Displaying and configuring system properties (e.g., logname, printenv)
  • Controlling command invocation (e.g., nohup, nice, env)
  • Processing text files (e.g., sort, od, patch)

• Two other UNIX utility suites: **Busybox**, a widely-used distribution for embedded systems, and the latest release for **Minix**

• The **HiStar** operating system kernel
We now give 5.2 GNU C that wants high library coverage can change this setting.

new statement or branch in the main utility code. A user generates by only emitting tests cases for paths that hit a median coverage per application.

gate coverage achieved by each method and the average and (library code not included). The last rows shows the aggregate coverage achieved on any tool is 62.6%. We get 100% line coverage on 16 tools, over 90% on 56 tools, and over 80% on 77 tools (86.5% of all tools). The minimum was 512 and (to pick a random number) 160 were no matter how many times the branch ran dynamically).

The average path length was 76 (median: 53), the maximum was 512 and (to pick a random number) 160 were.

The maximum number needed was 129 (for the "system calls (see §).

The one exception is significantly improving the overall results (which it improves last few lines of high-coverage tools, rather than signif-

Hitting system call failure paths is useful for getting the tool, with and without failing system call invocations.

The first two columns in Table 2 give aggregate line coverage results. On average our tests cover 90.9% of the lines in each tool (median: 94.7%), with an overall coverage of 84.5%.

The second best improvement for a single tool requires system call failures to go from a dismal 21.2% to 72.6%.

As specified by the --max-time option, we ran each tool for about 60 minutes (some finished before this limit, while relatively small, the tools are not toys — including library code called, the smallest five have between 3K and 10-20%.

While relatively small, the tools are not toys — including library code the tool calls.

Almost all tools were tested using the same command arguments explained in cheating through the use of fragile optimizations.

Figure 4 breaks down the tools by executable lines of code (ELOC), including library code the tool calls.

Table 2: Number of COREUTILS tools which achieve line coverage in the given ranges for KLEE and developers’ tests (library code not included). The last rows shows the aggregate coverage achieved by each method and the average and median coverage per application.

<table>
<thead>
<tr>
<th>Coverage (w/o lib)</th>
<th>COREUTILS</th>
<th>BUSYBOX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KLEE tests</td>
<td>Devel. tests</td>
</tr>
<tr>
<td>100%</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>90-100%</td>
<td>40</td>
<td>6</td>
</tr>
<tr>
<td>80-90%</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>70-80%</td>
<td>7</td>
<td>23</td>
</tr>
<tr>
<td>60-70%</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>50-60%</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>40-50%</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>30-40%</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>20-30%</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>10-20%</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>0-10%</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Overall cov.</td>
<td>84.5%</td>
<td>67.7%</td>
</tr>
<tr>
<td>Med cov/App</td>
<td>94.7%</td>
<td>72.5%</td>
</tr>
<tr>
<td>Ave cov/App</td>
<td>90.9%</td>
<td>68.4%</td>
</tr>
</tbody>
</table>
Coreutils bugs detected

```
paste -d\\
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\\
ptx x t4.txt
seq -f %0 1
```

<table>
<thead>
<tr>
<th>t1.txt:</th>
<th>&quot;\t\tMD5(&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2.txt:</td>
<td>&quot;\b\b\b\b\b\b\b\t&quot;</td>
</tr>
<tr>
<td>t3.txt:</td>
<td>&quot;n&quot;</td>
</tr>
<tr>
<td>t4.txt:</td>
<td>&quot;a&quot;</td>
</tr>
</tbody>
</table>

**Figure 7:** KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.
Busybox bugs detected

| t1.txt: a | t2.txt: A | t3.txt: \t
| --- | --- | --- |

| date -I | ls --co | printf "%s" B |
| chown a.a - | kill -l a | printf "%s x" B |
| od t1.txt | od t2.txt | printf % |
| printf % | printf %Lo | tr [ |
| tr [= | tr [a-z |

| cut -f t3.txt | install --m | nmeter - |
| envdir - | envdir - | envuidgid |
| envdir - | envdir - | envdir - |
| arp -Ainet | top d | cut -f t3.txt |
| setarch " " | " " | " " |
| " " | " " | " " |
| hexdump-e | ping6 - | " " |

Figure 10: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in BUSYBOX. When multiple applications crash because of the same shared (buggy) piece of code, we group them by shading.
Inconsistencies between Coreutils and Busybox

<table>
<thead>
<tr>
<th>Input</th>
<th>BUSYBOX</th>
<th>COREUTILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm t1.txt t2.txt</td>
<td>[does not show difference]</td>
<td>[shows difference]</td>
</tr>
<tr>
<td>tee -</td>
<td>[does not copy twice to stdout]</td>
<td>[does]</td>
</tr>
<tr>
<td>tee &quot; &quot; &lt;t1.txt</td>
<td>[infinite loop]</td>
<td>[terminates]</td>
</tr>
<tr>
<td>cksum /</td>
<td>&quot;4294967295 0 /&quot;</td>
<td>&quot;/: Is a directory&quot;</td>
</tr>
<tr>
<td>split /</td>
<td>&quot;/: Is a directory&quot;</td>
<td>&quot;missing operand&quot;</td>
</tr>
<tr>
<td>tr [ 0 '='&lt; 1 ]</td>
<td>[duplicates input on stdout]</td>
<td>&quot;binary operator expected&quot;</td>
</tr>
<tr>
<td>sum -s &lt;t1.txt</td>
<td>&quot;97 1 -&quot;</td>
<td>&quot;97 1&quot;</td>
</tr>
<tr>
<td>tail -2l</td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
<tr>
<td>unexpand -f</td>
<td>[accepts]</td>
<td>[rejects]</td>
</tr>
<tr>
<td>split -</td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
<tr>
<td>ls --color-blah</td>
<td>[accepts]</td>
<td>[rejects]</td>
</tr>
<tr>
<td>t1.txt: a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2.txt: b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Very small subset of the mismatches KLEE found between the BUSYBOX and COREUTILS versions of equivalent utilities. The first three are serious correctness errors; most of the others are revealing missing functionality.
Inconsistencies between Coreutils and Busybox: how?

```
unsigned mod_opt(unsigned x, unsigned y) {
    if ((y & -y) == y) // power of two?
        return x & (y - 1);
    else
        return x % y;
}
```

```
unsigned mod_opt(unsigned x, unsigned y) {
    return x % y;
}
```

```
int main() {
    unsigned x, y;
    make_symbolic(&x, sizeof(x));
    make_symbolic(&y, sizeof(y));
    assert(mod(x, y) == mod_opt(x, y));
    return 0;
}
```

Every assertion can be treated as a branch statement with two outgoing branches (i.e., hold or not); symbolic execution will try to cover both!
Discussion

• Strengths
• Limitations
• Future work
This class

• KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs (OSDI'08)
• Hybrid Concolic Testing (ICSE'07)
Symbolic execution: coverage problem

Symbolic execution may not reach deep into the execution tree. Specially when encountering loops
Solution: concolic execution

**Concolic=Concrete+Symbolic**

- Generate a random seed input to dive into the program execution tree
- Concretely execute the program with the random seed input and collect the path constraint, e.g., `a && b && c`
- In the next iteration, negate the last conjunct to obtain the constraint: `a && b && !c`
- Solve it to get input to the path which matches all the branch decisions except the last one
Concolic execution

Code under test

```c
void CoverMe(int[] a) {
    if (a == null)
        return;
    if (a.Length > 0)
        if (a[0] == 1234567890)
            throw new Exception("bug");
}
```

Choose next path

<table>
<thead>
<tr>
<th>Constraints to solve</th>
<th>Data</th>
<th>Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>a==null</td>
<td>null</td>
<td>a==null</td>
</tr>
<tr>
<td>a!=null &amp; &amp; a.Length&gt;0</td>
<td>{}</td>
<td>a!=null &amp; &amp; !(a.Length&gt;0)</td>
</tr>
<tr>
<td>a!=null &amp; &amp; a.Length&gt;0</td>
<td>{1}</td>
<td>a!=null &amp; &amp; a.Length&gt;0 &amp; &amp; a[0]!=1234567890</td>
</tr>
<tr>
<td>a!=null &amp; &amp; a.Length&gt;0 &amp; &amp; a[0]==1234567890</td>
<td>{123...}</td>
<td>a!=null &amp; &amp; a.Length&gt;0 &amp; &amp; a[0]==1234567890</td>
</tr>
</tbody>
</table>

DONE, no path left!
Concolic execution: another example

Code under test

```java
void CoverMe(int[] a, int b) {
    if (a == null)
        return;
    if (a.Length > 0)
        if (a[0] == hash(b))
            throw new Exception("bug");
}
```

Choose next path

<table>
<thead>
<tr>
<th>Constraints to solve</th>
<th>Data</th>
<th>Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>null, 0</td>
<td>a==null</td>
<td></td>
</tr>
</tbody>
</table>
| a!=null              | {}, 0  | a!=null &
| a.Length>0           | {1}, 0 | !(a.Length>0) &
|                      |        | a.Length>0 &
|                      |        | a[0]!==hash(b) &
|                      |        | a[0]==hash(0) &
|                      |        | a[0]==434...

Concretized!

DONE, no path left!
Limitations

Execution path space

Explored by concolic execution
Limitations: a comparative view

Concolic testing: wide and shallow

Random testing: narrow and deep
Limitations: example

Example ( ) {
  state = 0;
  while(1) {
    s = input();
    c = input();
    if(c==':' && state==0)
      state=1;
    else if(c=='\n' && state==1)
      state=2;
    else if (s[0]=='U' &&
      s[1]=='I' &&
      s[2]=='U' &&
      s[3]=='C' &&
      state==2) {
      COVER_ME;;
    }
  }
}

• **COVER_ME** can be hit on an input sequence
  • s = ‘UIUC’
  • c : ‘:’ ‘\n’

• **Random testing** can get to state = 2, but difficult to get ‘UIUC’ as a sequence
  • Probability: \(1/(2^8)^4 \approx 2.3 \times 10^{-10}\)

• **Concolic testing** can generate ‘UIUC’, but explores many paths to state = 2

Similar code structure in
• Text editors (vi)
• Parsers (lexer)
• Event-driven programs (GUI)
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 for deep & broad search to increase coverage
Hybrid concolic testing: example

Example (

\begin{verbatim}
state = 0;
while(1) {
 s = input();
c = input();
if(c=='.' & state==0)
 state=1;
else if(c=='\n' & state==1)
 state=2;
else if (s[0]=='U' &
 s[1]=='I' &
 s[2]=='U' &
 s[3]=='C' &
 state==2) {
 COVER_ME:;
 }
}
\end{verbatim}

\end{verbatim}

}\end{verbatim}

- **Random phase:** \{'\$', '\&', '-', '6', ':', '%', '^', '\n', 'x', '~' ...\}
  - Saturates after many (~10000) iterations
  - In less than 1 second
  - **COVER_ME** is not reached

  - Reaches **COVER_ME**!
Implementation

• An extension on the CUTE:
  • A concolic execution engine for C
  • Code instrumentation via CIL, a framework for parsing and transforming C programs\(^1\)
  • Constraint solving via lp_solve, a library for integer linear programming\(^2\)

---

\(^1\) [https://github.com/cil-project/cil](https://github.com/cil-project/cil)
\(^2\) [http://lpsolve.sourceforge.net/5.5/](http://lpsolve.sourceforge.net/5.5/)
Testing red-black tree

<table>
<thead>
<tr>
<th>Seed</th>
<th>Random Testing</th>
<th>Concolic Testing</th>
<th>Hybrid Concolic Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>523</td>
<td>32.27</td>
<td>52.48</td>
<td>66.67</td>
</tr>
<tr>
<td>7487</td>
<td>32.27</td>
<td>52.48</td>
<td>67.02</td>
</tr>
<tr>
<td>6726</td>
<td>32.27</td>
<td>52.48</td>
<td>66.67</td>
</tr>
<tr>
<td>5439</td>
<td>32.27</td>
<td>52.48</td>
<td>67.73</td>
</tr>
<tr>
<td>4494</td>
<td>32.27</td>
<td>52.48</td>
<td>69.86</td>
</tr>
<tr>
<td>Average</td>
<td>32.27</td>
<td>52.48</td>
<td>67.59</td>
</tr>
</tbody>
</table>

Table 1. Results of Testing Red-Black Tree
Testing Vim editor

<table>
<thead>
<tr>
<th>Seed</th>
<th>Random Testing</th>
<th>Concolic Testing</th>
<th>Hybrid Concolic Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>877443</td>
<td>8.01</td>
<td>21.43</td>
<td>41.93</td>
</tr>
<tr>
<td>67532</td>
<td>8.16</td>
<td>21.43</td>
<td>40.39</td>
</tr>
<tr>
<td>98732</td>
<td>8.72</td>
<td>21.43</td>
<td>33.67</td>
</tr>
<tr>
<td>32761</td>
<td>7.80</td>
<td>21.43</td>
<td>35.45</td>
</tr>
<tr>
<td>28683</td>
<td>9.75</td>
<td>21.43</td>
<td>40.53</td>
</tr>
<tr>
<td>Average</td>
<td>8.17</td>
<td>21.43</td>
<td>37.86</td>
</tr>
</tbody>
</table>

Table 2. Results of Testing the VIM Test Editor
Discussion

• Strengths
• Limitations
• Future work
Symbolic execution engines you may want to try

• C family: KLEE (http://llvm.org/)
• C#: Pex/IntelliTest (http://research.microsoft.com/en-us/projects/pex/)
• Java: Java PathFinder (https://github.com/SymbolicPathFinder/jpf-symbc)
• JavaScript: Jalangi2 (https://github.com/Samsung/jalangi2)
• Binaries (x86, ARM, ...): S2E (https://s2e.systems/)
Further readings

• Koushik Sen, Darko Marinov, Gul Agha. CUTE: A Concolic Unit Testing Engine for C. 2005, FSE.


Thanks and stay safe!