Announcements

• No class December 14
• Makeup class January 13 10-13
Concrete, Symbolic, and Concolic Testing

Mooly Sagiv

Slides taken from Cristian Cadar
Recap

• Bounded model checking is powerful for small and intermediate size code
• But how can we deal with large code with complicated features?
  – Limitations of the SMT power
  – Complexity of SMT
Program Path

- **Program Path**
  - A path in the control flow of the program
    - Can start and end at any point
    - Appropriate for imperative programs

- **Feasible program path**
  - There exists an input that leads to the execution of this path

- **Infeasible program path**
  - No input that leads to the execution
void grade(int score) {
    A: if (score < 45) {
        B: printf("fail");
    } else 
    C: printf("pass");
    } 
    D: if (score > 85) {
    E: printf("with honors");
    } 
    F: 
}
Concrete vs. Symbolic Executions

• Real programs have many infeasible paths
  – Ineffective concrete testing
• Symbolic execution aims to find rare errors
Symbolic Testing Tools

- EFFIGY [King, IBM 76]
- PEX [MSR]
- SAGE [MSR]
- SATURN [Stanford]
- KLEE [Stanford]
- Java pathfinder [NASA]
- Bitscope [Berkeley]
- Cute [UIUC, Berkeley]
- Calysto [UBC]
Finding Infeasible Paths Via SMT

void grade(int score) {
A: if (score < 45) {
B: printf("fail");
} else
C: printf("pass");
} D: if (score > 85) {
E: printf("with honors");
} F:
}
score < 45 \land score > 85 \quad \text{UNSAT}
Plan

• Random Testing
• Symbolic Testing
• Concolic Testing
Fuzzing [Miller 1990]

• Test programs on random unexpected data
• Can be realized using black/white testing
• Can be quite effective
  – Operating Systems
  – Networks
• ...
• Usually implemented via instrumentation
• Tricky to scale for programs with many paths

If (x == 10001) {
  int f(int *p) {
    ....
    if (f(*y) == *z) {
      ....
    }
    ....
    if (p != NULL) {
      return q;
    }
  }
}
Symbolic Exploration

• Execute a program on symbolic inputs
• Track set of values symbolically
• Update symbolic states when instructions are executed
• Whenever a branch is encountered check if the path is feasible using a theorem prover call
Symbolic Execution Tree

- The constructed symbolic execution paths
- Nodes
  - Symbolic Program States
- Edges
  - Potential Transitions
- Constructed during symbolic evaluation
- Each edge requires a theorem prover call
```
1) int x, y;
2) if (x > y) {
3) x = x + y;
4) y = x - y;
5) x = x - y;
6) if (x > y)
7) assert false;
8})
```
Another Example

int f(int x) { return 2 * x ;}
int h(int x, int y) {
  1) if (x != y) {
      2) if (f(x) == x + 10) {
          3) abort() // * error */
      }
  }
  4) return 0;

Pointers

struct foo {int i; char c;}
bar(struct foo *a) {
  1) if (a->c == 0) {
      2) *((char *)a + sizeof(int)) = 1 ;
      3) if (a->c !=0) {
          4) abort();
          5)
      6)
  7) }
Non-Deterministic Behavior

```c
int x; y;
1) if (nondet()) {
   2) x = 7;
}
else {
   3) x = 19;
}
4)
```
Loops

1) int i;
2) while i < n {
   i = i + 1;
}
3) if (n == 10^6) {
4)    abort();
5) }

Scaling Issues for Symbolic Exploration
Challenge 1: Limitations of Theorem Provers

```c
foobar(int x, int y) {
1)   if (x * x * x > 0) {
2)      if (x>0 && y ==10) {
3)          abort();
4)      } 
5)   else {
6)      if (x > 0 && y == 20) {
7)          abort ;
8)      } 
9)   }
```
Challenge 2: External Calls

1) FILE *fp;
2) fp = fopen("test.txt", "w");
3) if (fp) {
4)    struct stat buffer;
5)    if (stat ("text.txt", &buffer) != 0) {
6)        abort();
7)    }
8) }

1) int i = 0;
2) while i < n {
   i = i + 1;
}
3) if (n==10^6) {
   4) abort();
5) }

Concolic Testing

**Concrete + Symbolic = Concolic**

- Combine *concrete testing* (concrete execution) and *symbolic testing* (symbolic execution)
- Trade coverage (miss bugs) for scalability
- Reduce the number of theorem prover calls
- Reduce the complexity of path formulas
- Can cope with external calls
Motivating Example

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

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

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
Concolic Testing Approach

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

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

Concrete Execution

Symbolic Execution

Concrete state

z = 14

Symbolic state

x = x₀, y = y₀, z = 2*y₀

Path condition

x = 22, y = 7, z = 14
Concolic Testing Approach

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

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
Concolic Testing Approach

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

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

Concrete Execution

Symbolic Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>2*y₀ == x₀</td>
<td>Solution: x₀ = 2, y₀ = 1</td>
<td>2*y₀ != x₀</td>
</tr>
</tbody>
</table>

Solve: 2*y₀ == x₀

Solution: x₀ = 2, y₀ = 1

x = 22, y = 7, z = 14

x = x₀, y = y₀, z = 2*y₀
Concolic Testing Approach

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

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

Concrete Execution |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>concrete state</td>
</tr>
<tr>
<td>x = 2, y = 1</td>
</tr>
</tbody>
</table>

Symbolic Execution |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>symbolic state</td>
</tr>
<tr>
<td>x = x₀, y = y₀</td>
</tr>
</tbody>
</table>

Path Condition
Concolic Testing Approach

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

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

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

Symbolic Execution
symbolic state
x = x₀, y = y₀,
z = 2*y₀

path condition
Concolic Testing Approach

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

void testme (int x, int y) {
    z = double (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>$x = 2, y = 1, z = 2$</td>
<td>$x = x_0, y = y_0, z = 2*y_0$</td>
<td>$2*y_0 == x_0$</td>
</tr>
</tbody>
</table>
Concolic Testing Approach

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

void testme (int x, int y) {
    z = double (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, y = 1, z = 2</td>
<td>x = x₀, y = y₀, z = 2*₂y₀</td>
<td>2*₂y₀ == x₀ x₀ · y₀ + 10</td>
</tr>
</tbody>
</table>
Concolic Testing Approach

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

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

Concrete Execution
- concrete state: $x = 2, y = 1, z = 2$
- symbolic state: $x = x_0, y = y_0, z = 2y_0$
- path condition: $2y_0 = x_0$

Symbolic Execution
- solve: $(2y_0 = x_0) \land (x_0 > y_0 + 10)$
- solution: $x_0 = 30, y_0 = 15$
Concolic Testing Approach

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

Concrete Execution | Symbolic Execution
--- | ---
concrete state | symbolic state | path condition

- $x = 30, y = 15$
- $x = x_0, y = y_0$
Concolic Testing Approach

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

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

Concrete Execution

Symbolic Execution

Concrete state

Symbolic state

Path condition

Program Error

```
x = 30, y = 15
```

```
x = x_0, y = y_0
```

```
2*y_0 == x_0
```

```
x_0 > y_0+10
```
The Concolic Testing Algorithm

1. Classify input variables into symbolic / concrete
2. Instrument to record symbolic vars and path conditions
3. Choose an arbitrary input
4. Execute the program
5. Symbolically re-execute the program
6. Negate the unexplored last path condition
7. Is there an input satisfying constraint
Interesting Example Concolic Testing

```cpp
foobar(int x, int y) {
    1) if (x * x * x > 0) {
        2) if (x > 0 && y == 10) {
            3) abort();
        }
    4) }
    5) else {
        6) if (x > 0 && y == 20) {
            7) abort();
        }
        8) }
    9) }
```
Pointers

struct foo {int i; char c;}
bar(struct foo *a) {
  1) if (a->c == 0) {
      2) *((char *)a + sizeof(int)) = 1;
      3) if (a->c != 0) {
          4) abort();
      }
  }
  5)
}
Loops

1) int i = 0;
2) while i < n {
   i = i + 1;
}
3) if (n == 10^6) {
4)   abort();
5) }

Handling External Calls

1) FILE *fp;
2) fp = fopen("test.txt", "w");
3) if (fp) {
4)    struct stat buffer;
5)    if (stat("text.txt", &buffer) != 0) {
6)        abort();
7)    }
8) }

Concolic Testing Techniques

- Linear underapproximation
- Modeling System call models
- Simple heuristics
  - Two possible values for pointers
  - ...
- Efficient instrumentation
- Interface extraction
- Generating random inputs of arbitrary types
- ...
Original DART Approach [PLDI’05]

• Tailored for C

• Three types of functions
  – Program functions
  – External functions handled non-deterministically
  – Library functions handled as black-box (concrete only)
SAGE: Whitebox Fuzzing for Security Testing

• Check correctness of Win’7, Win’8
• 200+ machine years
• 1 Billion+ SMT constraints
• 100s of apps, 100s of bugs
• 1/3 of all Win7 WEX security bugs found
• Millions of dollars saved
KLEE

- Symbolic testing for system’s code
- Optimize search space
- Models for common functions
- Built on top of LLVM
- SMT agnostic
Writing Systems Code Is Hard

• Code complexity
  – Tricky control flow
  – Complex dependencies
  – Abusive use of pointer operations

• Environmental dependencies
  – Code has to anticipate all possible interactions
  – Including malicious ones
KLEE
[OSDI 2008, Best Paper Award]

Based on symbolic execution and constraint solving techniques

Automatically generates high coverage test suites
  – Over 90% on average on ~160 user-level apps

Finds deep bugs in complex systems programs
  – Including higher-level correctness ones
```c
int bad_abs(int x) {
    if (x < 0)
        return -x;
    if (x == 1234)
        return -x;
    return x;
}
```
KLEE Architecture

C code → LLVM → LLVM bytecode → SYMBOLIC ENVIRONMENT → KLEE → Constraint Solver (STP)

x ≥ 0
x ≠ 1234
x = 3

x = -2
x = 1234
x = 3
Outline

Motivation
Example and Basic Architecture

→ Scalability Challenges
Experimental Evaluation
Three Big Challenges

Motivation

Example and Basic Architecture

Scalability Challenges

– Exponential number of paths
– Expensive constraint solving
– Interaction with environment

Experimental Evaluation
Exponential Search Space

Naïve exploration can easily get “stuck”

Use search heuristics:

Coverage-optimized search
  – Select path closest to an uncovered instruction
  – Favor paths that recently hit new code

Random path search
  – See [KLEE – OSDI’08]
Three Big Challenges

Motivation
Example and Basic Architecture

Scalability Challenges
– Exponential number of paths
– Expensive constraint solving
– Interaction with environment

Experimental Evaluation
Constraint Solving

Dominates runtime
  – Inherently expensive (NP-complete)
  – Invoked at every branch

Two simple and effective optimizations
  – Eliminating irrelevant constraints
  – Caching solutions
    Dramatic speedup on our benchmarks
Eliminating Irrelevant Constraints

In practice, each branch usually depends on a small number of variables

```java
...  ...
if (x < 10) {
  ...
}
```

- $x + y > 10$
- $z \& -z = z$
- $x < 10$
Caching Solutions

Static set of branches: lots of similar constraint sets

- $2 \cdot y < 100$
- $x > 3$
- $x + y > 10$

$x = 5$
$y = 15$

Eliminating constraints cannot invalidate solution

- $2 \cdot y < 100$
- $x + y > 10$

Adding constraints often does not invalidate solution

- $2 \cdot y < 100$
- $x > 3$
- $x + y > 10$
- $x < 10$

$x = 5$
$y = 15$
Dramatic Speedup

Aggregated data over 73 applications

- Blue: Base
- Dashed: Irrelevant Constraint Elimination
- Green: Caching
- Red: Irrelevant Constraint Elimination + Caching

Time (s) vs. Executed instructions (normalized)
Three Big Challenges

Motivation
Example and Basic Architecture

Scalability Challenges
  – Exponential number of paths
  – Expensive constraint solving
  – Interaction with environment

Experimental Evaluation
Environment: Calling Out Into OS

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

If all arguments are concrete, forward to OS

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

Otherwise, provide *models* that can handle symbolic files

- Goal is to explore all possible *legal* interactions with the environment
Plain C code run by KLEE

- Users can extend/replace environment w/o any knowledge of KLEE internals

Currently: effective support for symbolic command line arguments, files, links, pipes, ttys, environment vars
Does KLEE work?

Motivation
Example and Basic Architecture
Scalability Challenges

Evaluation

– Coverage results
– Bug finding
– Crosschecking
GNU Coreutils Suite

Core user-level apps installed on many UNIX systems
89 stand-alone (i.e. excluding wrappers) apps (v6.10)

- File system management: `ls`, `mkdir`, `chmod`, etc.
- Management of system properties: `hostname`, `printenv`, etc.
- Text file processing: `sort`, `wc`, `od`, etc.
- ...

Variety of functions, different authors, intensive interaction with environment

Heavily tested, mature code
Coreutils ELOC (incl. called lib)
Methodology

Fully automatic runs
Run KLEE one hour per utility, generate test cases
Run test cases on *uninstrumented* version of utility
Measure line coverage using *gcov*
  - Coverage measurements not inflated by potential bugs in our tool
High Line Coverage
(Coreutils, non-lib, 1h/utility = 89 h)

Overall: 84%, Average 91%, Median 95%
Beats 15 Years of Manual Testing

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

Manual tests also check correctness
Busybox Suite for Embedded Devices

Overall: 91%, Average 94%, Median 98%

Apps sorted by KLEE coverage

Coverage (ELOC %)

31 at 100%
Busybox – KLEE vs. Manual

Avg/utility

<table>
<thead>
<tr>
<th></th>
<th>KLEE</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>94%</td>
<td>44%</td>
</tr>
</tbody>
</table>

Apps sorted by KLEE coverage - Manual coverage
Does KLEE work?

Motivation
Example and Basic Architecture
Scalability Challenges

Evaluation
  – Coverage results
  – Bug finding
  – Crosschecking
GNU Coreutils Bugs

Ten crash bugs
- More crash bugs than approx last three years combined
- KLEE generates actual command lines exposing crashes
## Ten command lines of death

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>md5sum -c t1.txt</code></td>
<td><code>pr -e t2.txt</code></td>
<td><code>tac -r t3.txt t3.txt</code></td>
<td><code>paste -d\abcdefghijklmnopqrstuvwxyz</code></td>
</tr>
<tr>
<td><code>mkdir -Z a b</code></td>
<td><code>mkfifo -Z a b</code></td>
<td><code>ptx -F\abcdefghijklmnopqrstuvwxyz</code></td>
<td><code>ptx x t4.txt</code></td>
</tr>
<tr>
<td><code>mknod -Z a b p</code></td>
<td><code>seq -f %0 1</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**t1.txt:** \t \tMD5 (  
**t2.txt:** \b\b\b\b\b\b\b\t  
**t3.txt:** \n  
**t4.txt:** A
Does KLEE work?

Motivation
Example and Basic Architecture
Scalability Challenges

Evaluation
  – Coverage results
  – Bug finding
  – Crosschecking
Finding Correctness Bugs

KLEE can prove asserts on a per path basis

– Constraints have no approximations
– An assert is just a branch, and KLEE proves feasibility/infeasibility of each branch it reaches
– If KLEE determines infeasibility of false side of assert, the assert was proven on the current path
Crosschecking

Assume $f(x)$ and $f'(x)$ implement the same interface

1. Make input $x$ symbolic
2. Run KLEE on $\texttt{assert}(f(x) == f'(x))$
3. For each explored path:
   a) KLEE terminates w/o error: paths are equivalent
   b) KLEE terminates w/ error: mismatch found

Coreutils vs. Busybox:

1. UNIX utilities should conform to IEEE Std.1003.1
2. Crosschecked pairs of Coreutils and Busybox apps
3. Verified paths, found mismatches
## Mismatches Found

<table>
<thead>
<tr>
<th>Input</th>
<th>Busybox</th>
<th>Coreutils</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tee &quot;&quot; &lt;t1.txt</code></td>
<td>[infinite loop]</td>
<td>[terminates]</td>
</tr>
<tr>
<td><code>tee -</code></td>
<td>[copies once to stdout]</td>
<td>[copies twice]</td>
</tr>
<tr>
<td><code>comm t1.txt t2.txt</code></td>
<td>[doesn’t show diff]</td>
<td>[shows diff]</td>
</tr>
<tr>
<td><code>cksum /</code></td>
<td>&quot;4294967295 0 /&quot;</td>
<td>&quot;/: Is a directory&quot;</td>
</tr>
<tr>
<td><code>split /</code></td>
<td>&quot;/: Is a directory&quot;</td>
<td></td>
</tr>
<tr>
<td><code>tr</code></td>
<td>[duplicates input]</td>
<td>&quot;missing operand&quot;</td>
</tr>
<tr>
<td><code>[ 0 &quot;&lt;&quot; 1 ]</code></td>
<td></td>
<td>&quot;binary op. expected&quot;</td>
</tr>
<tr>
<td><code>tail -2l</code></td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
<tr>
<td><code>unexpand -f</code></td>
<td>[accepts]</td>
<td>[rejects]</td>
</tr>
<tr>
<td><code>split -</code></td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
<tr>
<td>t1.txt: a t2.txt: b</td>
<td>(no newlines!)</td>
<td></td>
</tr>
</tbody>
</table>
Related Work

Very active area of research. E.g.:

• EGT / EXE / KLEE [Stanford]
• DART [Bell Labs]
• CUTE [UIUC]
• SAGE, Pex [MSR Redmond]
• Vigilante [MSR Cambridge]
• BitScope [Berkeley/CMU]
• CatchConv [Berkeley]
• JPF [NASA Ames]

KLEE
  – Hundred distinct benchmarks
  – Extensive coverage numbers
  – Symbolic crosschecking
  – Environment support
KLEE can effectively:

- Generate high coverage test suites
  Over 90% on average on ~160 user-level applications

- Find deep bugs in complex software
  Including higher-level correctness bugs, via crosschecking
Summary

• Concolic testing is powerful
• Scaling is an issue
• Future progress in symbolic reasoning can help
  – Example: handling dynamically allocated memory
Suggested Projects Concolic

• Study the effectives of KLEE and CBMC for regression testing
  – Run KLEE and CBMC in order to find differences
  – The trick is to find an interesting application
    • Small and meaningful
    • Not too many external calls
    • Can compare code to spec

• Apply CBMC and Concolic to test temporal properties of code