Analysis/Bug-finding/Verification for Security

VIJAY GANESH
University of Waterloo
Winter 2013
Analysis/Test/Verify for Security

• Instrument code for testing
  – Heap memory: Purify
  – Perl tainting (information flow)
  – Java race condition checking

• Black-box testing
  – Fuzzing and penetration testing
  – Black-box web application security analysis

• Static code analysis
  – FindBugs, Fortify, Coverity, MS tools, ...

• Model Checking tools
  – NuSMV to verify program properties

Slide courtesy John Mitchell
Manual Testing Doesn’t Scale

Software

Slides courtesy John Mitchel
Manual Testing Doesn’t Scale

Software

Behaviors

Entry

1

2

3

4

Exit

Slides courtesy John Mitchel
Manual Testing Doesn’t Scale

Software

Behaviors

Entry

1

2

3

4

Exit

1 → 2 → 4

Slides courtesy John Mitchel
Manual Testing Doesn’t Scale

Entry

1

2  3

4

Exit

Software

Behaviors

Slides courtesy John Mitchel
Manual Testing Doesn’t Scale

Software

Exit

Behaviors

1 → 2 → 4

Slides courtesy John Mitchel
Manual Testing Doesn’t Scale

Software

Entry

1

2

3

4

Exit

Behaviors

1 → 2 → 4

1 → 3 → 4

1 → 2 → 4

1 → 3 → 4

1 → 2 → 3 → 1 → 2 → 4 → 1 → 3 → 4

1 → 2 → 3 → 1 → 2 → 4 → 1 → 3 → 4

1 → 2 → 3 → 1 → 2 → 3 → 1 → 3 → 4

1 → 2 → 4 → 1 → 2 → 3 → 1 → 3 → 4

1 → 2 → 4 → 1 → 2 → 3 → 1 → 3 → 4

1 → 2 → 4 → 1 → 2 → 4 → 1 → 3 → 4

Slides courtesy John Mitchel
Manual Testing Doesn’t Scale

Manual testing only examines small subset of behaviors

Software

Behaviors

Manual testing only examines small subset of behaviors

Manual Testing Doesn’t Scale

Manual testing only examines small subset of behaviors

Manual testing only examines small subset of behaviors

Manual testing only examines small subset of behaviors

Manual testing only examines small subset of behaviors

Manual testing only examines small subset of behaviors

Manual testing only examines small subset of behaviors

Slides courtesy John Mitchel
clipconv8x8_u8_s16_c(ptr...){

  for (i = 0; i < 8; i++) {
    for (j = 0; j < 8; j++) {

      x = BLOCK8x8_S16(src,sstr,i,j);
      if (x < 0) x = 0;
      if (x > 255) x = 255;

      (*((uint8_t *)((void *) ptr + stride*row) + column)) = x;
    }
  }
}

Code From LibOIL Library version 0.3.x (GNOME Windowing System)
Bug in Swfdec Adobe Flash Movie Player

```c
jpeg_decoder(JpegDecoder* dec) {
    dec->width_blocks = (dec->width + 8*max_h_sample - 1)/(8*max_h_sample);
    dec->height_blocks = (dec->height + 8*max_v_sample - 1)/(8*max_v_sample);
    int rowstride, image_size;
    ...
    rowstride = dec->width_blocks * 8*max_h_sample / dec->comps[i].h_subsample;
    imagesize = rowstride * (dec->height_blocks * 8*max_v_sample/dec->comps[i].v_subsample);

    dec->c[i].image=malloc(imagesize);
    ...

    //LibOIL API function call
    clipconv8x8_u8_s16_c(dec->c[i].image...);
    ...
}
```

Code from Swfdec Shockwave Flash Movie Player

But
imagesize 0

malloc OK

Bug in Swfdec Adobe Flash Movie Player

Thursday, 17 January, 13
Bug in Swfdec Adobe Flash Movie Player

```c
jpeg_decoder(JpegDecoder* dec) {
    dec->width_blocks = (dec->width + 8*max_h_sample - 1)/(8*max_h_sample);
    dec->height_blocks = (dec->height + 8*max_v_sample - 1)/(8*max_v_sample);
    int rowstride, image_size;
    ... 
    rowstride =   dec->width_blocks * 8*max_h_sample / dec->comps[i].h_subsample;
    imagesize =  rowstride * (dec->height_blocks * 8*max_v_sample/dec->comps[i].v_subsample);
    dec->c[i].image = malloc(imagesize);
    ...

    //LibOIL API function call
    clipconv8x8_u8_s16_c(dec->c[i].image);
    ...
}
```

from input movie

malloc OK
But imagesize 0

Code from Swfdec Shockwave Flash Movie Player
Essence of this Bug

Overflow in imagesize computation

```c
jpegdecode(image) {
    ...
    imagesize = f(image->height)*g(image->width);
    ptr = malloc(imagesize);
    LibraryCall(ptr,…);
}
```
Difficulty of finding this Bug

• Deep in the program
  – Stack depth 50
  – Number of instructions in path: ~ 7 million
  – Program source (excluding libraries) ~70 KLOC

• Complex input format
  – Movie file, arbitrarily large and complex

• Few regions of input (height, width)

• Construct a test input to find bug automatically
Random Fuzzing

- Multiple Fuzzed Movie Files
- The Fuzzer randomly mutates various fields in the file
- It can work sometimes
- However, often produces mal-formed inputs rejected by parser
Problem with Random Fuzzing

Fuzzed Input Movie File

Program under test

Parsing And Syntactic Check

Semantic Core of the Program
Problem with Random Fuzzing

Fuzzed Input Movie File

Random Fuzzed inputs often Rejected by Parser

Program under test

Parsing And Syntactic Check

Semantic Core of the Program
Problem with Random Fuzzing

Random Fuzzed inputs often Rejected by Parser

Fuzzed Input Movie File

Program under test Rejected In Parsing

Parsing And Syntactic Check

Semantic Core of the Program
Information-flow based Fuzzer

Program Source → BuzzFuzz → Bug-revealing Input

Valid Seed Input
Information-flow based Execution

- Instrument source for information-flow analysis (taint-tracking)

- Track input regions to sinks
  - Dynamic data dependency analysis

- Map from values to set of input bytes
### Information-flow based Execution

```c
jpeg_decoder(Jpeg* dec) {
    dec->width_blocks = (dec->width + 8*max_h_sample - 1)/(8*max_h_sample);
    dec->height_blocks = (dec->height + 8*max_h_sample - 1)/(8*max_h_sample);
    ...
    rowstride = dec->width_blocks * 8*max_h_sample / dec->comps[i].h_subsample;
    image_size = rowstride * (dec->height_blocks * 8*max_v_sample) / dec->comps[i].v_subsample;
    dec->c[i].image=malloc(image_size);
    ...
    //LibOIL API function call
    clipconv8x8_u8_s16_c(dec->c[i].image...);
    ...
} //End of Movie Player Code
```
Information-flow based Fuzzer
Selecting Attack Points

- Library API as Sinks or Attack Points
Information-flow based Fuzzer
Fuzzing with Extremal Values

Extremal values for integers: 0, -1, int_max

Valid Seed Input

Fuzzer

Fuzzed Input

Generated Taint Report

Height: 1862, 1863
Width: 1864, 1865

Valid Seed Input: 0xffff, 0xffff
Information-flow based Fuzzer
Smarter way to Fuzz

Fuzzed Input

Program under test

0xffff
0xffff

CRASH!!
Program Analyzers

Code

Spec
Program Analyzers

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer oflow</td>
<td>4,353,245</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>23,212</td>
</tr>
<tr>
<td>4</td>
<td>stack oflow</td>
<td>86,923</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>8,491</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,502</td>
<td>info leak</td>
<td>10,921</td>
</tr>
</tbody>
</table>
Program Analyzers

Program Analyzer

Code

Spec

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer oflow</td>
<td>4,353,245</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>23,212</td>
</tr>
<tr>
<td>4</td>
<td>stack oflow</td>
<td>86,923</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>8,491</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,502</td>
<td>info leak</td>
<td>10,921</td>
</tr>
</tbody>
</table>

analyze large code bases

analyze large
code bases

analyze large code bases
Program Analyzers

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer overflow</td>
<td>4,353,245</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>23,212</td>
</tr>
<tr>
<td>4</td>
<td>stack overflow</td>
<td>86,923</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>8,491</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,502</td>
<td>info leak</td>
<td>10,921</td>
</tr>
</tbody>
</table>

analyze large code bases

potentially reports many warnings
### Program Analyzers

**Table:**

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer oflow</td>
<td>4,353,245</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>23,212</td>
</tr>
<tr>
<td>4</td>
<td>stack oflow</td>
<td>86,923</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>8,491</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,502</td>
<td>info leak</td>
<td>10,921</td>
</tr>
</tbody>
</table>

- **Code**
- **Spec**
- **Program Analyzer**

- Analyze large code bases
- Potentially reports many warnings
- False alarms

---

*Thursday, 17 January, 13*
Program Analyzers

- Potentially reports many warnings
- May emit false alarms
- Analyze large code bases
- False alarm

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer oflow</td>
<td>4,353,245</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>23,212</td>
</tr>
<tr>
<td>4</td>
<td>stack oflow</td>
<td>86,923</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>8,491</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,502</td>
<td>info leak</td>
<td>10,921</td>
</tr>
</tbody>
</table>
Static Analysis Goals

• Bug finding
  – Identify code that the programmer wishes to modify or improve

• Correctness
  – Verify the absence of certain classes of errors

Note: some fundamental limitations...
# Soundness and Completeness

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Soundness</strong></td>
<td>If the program contains an error, the analysis will report a warning. “Sound for reporting correctness”</td>
</tr>
<tr>
<td><strong>Completeness</strong></td>
<td>If the analysis reports an error, the program will contain an error. “Complete for reporting correctness”</td>
</tr>
</tbody>
</table>

Slides courtesy John Mitchell
<table>
<thead>
<tr>
<th>Complete</th>
<th>Incomplete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sound</td>
<td></td>
</tr>
<tr>
<td>Unsound</td>
<td></td>
</tr>
</tbody>
</table>

Slides courtesy John Mitchell
Complete

Reports all errors
Reports no false alarms

Incomplete

Sound

Unsound

Slides courtesy John Mitchell
Complete

Incomplete

Sound

Reports all errors
Reports no false alarms

Unsound

Undecidable

Slides courtesy John Mitchell

Thursday, 17 January, 13
Complete

Sound
- Reports all errors
- Reports no false alarms

Incomplete

Sound
- Reports all errors
- May report false alarms

Unsound

Undecidable

Slides courtesy John Mitchell
<table>
<thead>
<tr>
<th>Complete</th>
<th>Incomplete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sound</td>
<td>Undecidable</td>
</tr>
<tr>
<td><strong>Reports all errors</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Reports no false alarms</strong></td>
<td></td>
</tr>
<tr>
<td>Unsound</td>
<td>Decidable</td>
</tr>
<tr>
<td><strong>Reports all errors</strong></td>
<td></td>
</tr>
<tr>
<td><strong>May report false alarms</strong></td>
<td></td>
</tr>
</tbody>
</table>

Slides courtesy John Mitchell
Complete

Sound
- Reports all errors
- Reports no false alarms

Incomplete

Sound
- Reports all errors
- May report false alarms

Unsound
- May not report all errors
- Reports no false alarms

Slides courtesy John Mitchell
Complete

Sound

Reports all errors
Reports no false alarms

Undecidable

Unsound

May not report all errors
Reports no false alarms

Decidable

Incomplete

Reports all errors
May report false alarms

Decidable
Complete

Sound
- Reports all errors
- Reports no false alarms

Undecidable

Unsound
- May not report all errors
- Reports no false alarms

Decidable

Incomplete

Sound
- Reports all errors
- May report false alarms

Decidable

Unsound
- May not report all errors
- May report false alarms

Slides courtesy John Mitchell
<table>
<thead>
<tr>
<th>Complete</th>
<th>Incomplete</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td><strong>Sound</strong></td>
</tr>
<tr>
<td>Reports all errors</td>
<td>Reports all errors</td>
</tr>
<tr>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td><strong>Undecidable</strong></td>
<td><strong>Decidable</strong></td>
</tr>
<tr>
<td>May not report all errors</td>
<td>May not report all errors</td>
</tr>
<tr>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
</tbody>
</table>

**Decidable**

Slides courtesy John Mitchell
Static Analysis
char *p;
if(x == 0)
  p = foo();
else
  p = 0;

if(x != 0)
  s=*p;
else
  ...
return;
char *p;
if (x == 0)
    p = foo();
else
    p = 0;

if (x != 0)
    s = *p;
else
    ...
return;
char *p;
if (x == 0)
  p = foo();
else
  p = 0;

if (x != 0)
  s = *p;
else ...
return;

Source Code

```
char *p;
if (x == 0)
  p = foo();
else
  p = 0;

if (x != 0)
  s = *p;
else ...
return;
```

Symbolic CFG Analysis

```
char *p

if (x == 0)
  p = 0
else
  p = foo()

if (x != 0)
  s = *p
else ...
return
```

Defects detected

- Assigning: p=0
- x!=0 taking true branch
- Dereferencing null pointer p

Slides courtesy Andy Chou
Example

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

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

Thursday, 17 January, 13
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;
        }
    }
}
```
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>$x = 22, y = 7$</td>
<td>$x = x_0, y = y_0$</td>
<td></td>
</tr>
</tbody>
</table>

Concrete State: $x = 22, y = 7$

Symbolic State: $x = x_0, y = y_0$

Path Condition: $x > y + 10$
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>x = 22, y = 7, z = 14</td>
<td>x = x₀, y = y₀, z = 2*y₀</td>
<td></td>
</tr>
</tbody>
</table>

Concrete state:
- `x = 22, y = 7, z = 14`

Symbolic state:
- `x = x₀, y = y₀, z = 2*y₀`

Path condition:
- The path condition is not defined in the given code snippet.
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
- \( x = 22, y = 7,\)
- \( z = 14 \)

### Symbolic Execution
- \( x = x_0, y = y_0,\)
- \( z = 2*y_0 \)

### Path Condition
- \( 2*y_0 \neq x_0 \)
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

Concrete Execution

Symbolic Execution

Solve: 2*y₀ == x₀
Solution: x₀ = 2, y₀ = 1

x = 22, y = 7, z = 14
x = x₀, y = y₀, z = 2*y₀

2*y₀ != x₀
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;
        }
    }
}
```

<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 = 2, y = 1, z = 2</td>
<td>x = x₀, y = y₀, z = 2*y₀</td>
</tr>
<tr>
<td></td>
<td>path condition</td>
</tr>
<tr>
<td>2*y₀ == x₀</td>
<td></td>
</tr>
</tbody>
</table>

**Concrete Execution**

- Define `double(int v)` function to return `2*v`.

**Symbolic Execution**

- Define `testme(int x, int y)` function to assign `z = double(y)` and check if `z == x`.
  - If `x > y+10`, raise an error.

**Concrete State**

- `x = 2`, `y = 1`, `z = 2`.

**Symbolic State**

- `x = x₀`, `y = y₀`, `z = 2*y₀`.

**Path Condition**

- `2*y₀ == x₀`
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**: $2y_0 == x_0$, $x_0 > y_0 + 10$

---

Thursday, 17 January, 13
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

Solve: (2*y₀ == x₀) AND (x₀ > y₀ + 10)

Solution: x₀ = 30, y₀ = 15

2*y₀ == x₀

x₀ · y₀+10

x = 2, y = 1,
z = 2

x = x₀, y = y₀,
z = 2*y₀
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;
        }
    }
}

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>concrete state</td>
<td>symbolic state</td>
</tr>
<tr>
<td>x = 30, y = 15</td>
<td>x = x₀, y = y₀</td>
</tr>
</tbody>
</table>

Thursday, 17 January, 13
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

2\ast y_0 =\ast x_0

x_0 > y_0 + 10

x = 30, y = 15

x = x_0, y = y_0
Explicit Path (not State) Model

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with valgrind to discover memory errors
Reliability through Logical Reasoning
Engineering, Usability, Novelty

Program Specification

Program Reasoning Tool

Logic Formulas

Solver

SAT/UNSAT

Program is Correct?

or Generate Counterexamples (Test cases)
What is at the Core?
The SAT/SMT Problem

- Rich logics (Modular arithmetic, Arrays, Strings,...)
- NP-complete, PSPACE-complete,...
- Practical, scalable, usable, automatic
- Enable novel software reliability approaches

Logic

Formula

Solver

(q ∨ p ∨ ¬r)
(q ∨ ¬p ∨ r)
...

SAT

UNSAT
Why Bit-vectors and Arrays

- STP logic tailored for software reliability applications
- Support **symbolic execution**/program analysis

<table>
<thead>
<tr>
<th>C/C++/Java/...</th>
<th>Bit-vectors and Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int Var</td>
<td>32 bit variable</td>
</tr>
<tr>
<td>Char Var</td>
<td>8 bit variable</td>
</tr>
<tr>
<td>Arithmetic operation (x+y, x-y, x*y, x/y,...)</td>
<td>Arithmetic function (x+y,x-y,x*y,x/y,...)</td>
</tr>
<tr>
<td>assignments</td>
<td>equality</td>
</tr>
<tr>
<td>x = expr;</td>
<td>x = expr;</td>
</tr>
<tr>
<td>if conditional</td>
<td>if-then-else construct</td>
</tr>
<tr>
<td>if(cond) x = expr¹ else x = expr²</td>
<td>x = if(cond) expr¹ else expr²</td>
</tr>
<tr>
<td>inequality</td>
<td>inequality predicate</td>
</tr>
<tr>
<td>Memory read/write</td>
<td>Array read/write</td>
</tr>
<tr>
<td>x = *ptr + i;</td>
<td>ptr[]; x = Read(ptr,i);</td>
</tr>
<tr>
<td>Structure/Class</td>
<td>Serialized bit-vector expressions</td>
</tr>
<tr>
<td>Function</td>
<td>Symbolic execution</td>
</tr>
<tr>
<td>Loops</td>
<td>Bounding</td>
</tr>
</tbody>
</table>

Thursday, 17 January, 13
How to Automatically Crash Programs?
Concolic Execution & STP

Problem: Automatically generate crashing tests given only the code

Program

Symbolic Execution Engine with Implicit Spec

Automatic Tester

Formulas

SAT/UNSAT

Crashing Tests

STP
How to Automate Testing?
Concolic Execution & STP

Structured input processing code:
PDF Reader, Movie Player,...

Buggy_C_Program(int* data_field, int len_field) {
    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
        // 1. Integer overflow causing NULL deref
        // 2. Buffer overflow
        *(ptr+i) = process_data(*(data_field+i));
    }
}

• Formula captures computation
• Tester attaches formula to capture spec
How to Automate Testing?
Concolic Execution & STP

Structured input processing code:
PDF Reader, Movie Player,...

```c
Buggy_C_Program(int* data_field, int len_field) {
    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized

    while (i++ < process(len_field)) {
        //1. Integer overflow causing NULL deref
        //2. Buffer overflow
        *(ptr+i) = process_data(*(data_field+i));
    }
}
```

Equivalent Logic Formula derived using symbolic execution

```c
data_field, mem_ptr : ARRAY;
len_field : BITVECTOR(32); //symbolic
i, j, ptr : BITVECTOR(32); //symbolic

mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);
```

• Formula captures computation
• Tester attaches formula to capture spec
How to Automate Testing?
Concolic Execution & STP

Structured input processing code:
PDF Reader, Movie Player, ...

Buggy_C_Program(int* data_field, int len_field) {
    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized
    while (i++ < process(len_field)) {
        //1. Integer overflow causing NULL deref
        //2. Buffer overflow
        *(ptr+i) = process_data(*(data_field+i));
    }
}

Equivalent Logic Formula derived using symbolic execution

data_field, mem_ptr : ARRAY;
len_field : BITVECTOR(32); //symbolic
i, j, ptr : BITVECTOR(32); //symbolic
.
mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);
.
.

• Formula captures computation
• Tester attaches formula to capture spec
How to Automate Testing?
Concolic Execution & STP

Structured input processing code:
PDF Reader, Movie Player,

Buggy_C_Program(int* data_field, int len_field) {
    int * ptr = malloc(len_field*sizeof(int));
    int i; //uninitialized
    while (i++ < process(len_field)) {
        //1. Integer overflow causing NULL deref
        //2. Buffer overflow
        *(ptr+i) = process_data(*(data_field+i));
    }
}

Equivalent Logic Formula derived using symbolic execution

data_field, mem_ptr : ARRAY;
len_field : BITVECTOR(32); //symbolic
i, j, ptr : BITVECTOR(32); //symbolic

mem_ptr[ptr+i] = process_data(data_field[i]);
mem_ptr[ptr+i+1] = process_data(data_field[i+1]);

//INTEGER OVERFLOW QUERY
0 <= j <= process(len_field);
ptr + i + j = 0?

• Formula captures computation
• Tester attaches formula to capture spec