Vulnerability Analysis (II): Symbolic Execution

Slide credit: Vijay D’Silva
<table>
<thead>
<tr>
<th></th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Efficiency of Fuzzing</td>
</tr>
<tr>
<td>2</td>
<td>Symbolic Reasoning</td>
</tr>
<tr>
<td>3</td>
<td>Path Predicates</td>
</tr>
<tr>
<td>4</td>
<td>Bug Finding</td>
</tr>
<tr>
<td></td>
<td>Topic</td>
</tr>
<tr>
<td>---</td>
<td>--------------------------</td>
</tr>
<tr>
<td>1</td>
<td>Efficiency of Fuzzing</td>
</tr>
<tr>
<td>2</td>
<td>Symbolic Reasoning</td>
</tr>
<tr>
<td>3</td>
<td>Path Predicates</td>
</tr>
<tr>
<td>4</td>
<td>Bug Finding</td>
</tr>
</tbody>
</table>
Quiz: Coverage

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
}
```
# Quiz: Coverage

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
}
```

```c
if (input < UINT_MAX - 2)
  false ->
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else
      s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
  true

   true
    s = len;
    if (len % 2 == 0)
      s = len;
    false ->
      s = len + 2;
      buf = malloc(s);
      read(fd, buf, len);
```

<table>
<thead>
<tr>
<th># of</th>
<th># of inputs for full</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dawn Song
Quiz: Coverage

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
}
```

<table>
<thead>
<tr>
<th>Lines</th>
<th>Branche</th>
<th>Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

# of inputs for full coverage

Dawn Song
Quiz: Coverage

### Code Snippet

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
}
```

### Coverage Table

<table>
<thead>
<tr>
<th></th>
<th>Lines</th>
<th>Branches</th>
<th>Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td># of</td>
<td>10</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td># of inputs for full coverage</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Quiz: Coverage

What is the expected number of inputs required to cover the highlighted line, using random test-case generation? Assuming unsigned is 32 bits.
Efficiency of Test-Case Generation

We can evaluate the efficiency of a test-case generation technique with respect to a coverage metric by comparing

\[ \text{minimum } \# \text{ of inputs vs. expected } \# \text{ of inputs} \]

required for full coverage using that metric

A technique is

• \textit{efficient} if the minimum value is close to expected value
• \textit{not efficient} if minimum \(<\) expected value
Inputs and Paths

<table>
<thead>
<tr>
<th>input</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
Inputs and Paths

There are many examples where

*minimum # << expected #*

of inputs for random fuzzing.

Can we do better if we take program structure into account?

Dawn Song
<table>
<thead>
<tr>
<th></th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Efficiency of Fuzzing</td>
</tr>
<tr>
<td>2</td>
<td>Symbolic Reasoning</td>
</tr>
<tr>
<td>3</td>
<td>Path Predicates</td>
</tr>
<tr>
<td>4</td>
<td>Bug Finding</td>
</tr>
</tbody>
</table>
Focus on Sets of Values

```
unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)
    s = len;
else
    s = len + 2;
```

Set of all inputs
Focus on Sets of Values

Goal: find one element of each set
Symbolic analysis provides a way to directly manipulate sets
Symbolic vs. Explicit Representation

Explicit representation

<table>
<thead>
<tr>
<th>x</th>
<th>-3</th>
<th>-1</th>
<th>1</th>
<th>3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Symbolic representation

- $x > -4 \land x < 4$
- $\land x \mod 2 == 1 \land y == x + 3$

Explicit representation

<table>
<thead>
<tr>
<th>x</th>
<th>-7</th>
<th>-5</th>
<th>-3</th>
<th>-1</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>-4</td>
<td>-2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Symbolic representation

- $x > -8 \land x < 8$
- $\land x \mod 2 == 1 \land y == x + 3$

Explicit representation

<table>
<thead>
<tr>
<th>x</th>
<th>...</th>
<th>-5</th>
<th>-3</th>
<th>-1</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>...</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>...</td>
<td>-2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Symbolic representation

- $x \mod 2 == 1 \land y == x + 3$
## Symbolic Representation

A symbolic representation encodes a set of values in terms of properties of those values.

<table>
<thead>
<tr>
<th>Representation</th>
<th>Example</th>
<th>Set Represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formula</td>
<td>$x &gt; 8 &amp;&amp; x % 4 = 0 &amp;&amp; x &lt; 24$</td>
<td>8, 12, 16, 20</td>
</tr>
<tr>
<td>Regular expression</td>
<td>report_*[012].pdf</td>
<td>report_0.pdf, report0.pdf, report_1.pdf,...</td>
</tr>
</tbody>
</table>
Tradeoff of Symbolic Representation

Advantages
• Can be exponentially smaller than explicit representation of finite sets
• Can represent infinite sets (e.g. regular expressions)
  • Generic algorithms (e.g. same algorithms for a certain type of formulas)

Tradeoff
• Performing basic operations may be expensive
• Specialized algorithms are required
• Difficult to predict size of representation
Satisfiability

A formula is *satisfiable* if there is a way to assign values to variables and make the formula.

\[(x > 0 \land x < 20 \land x = y + y)\] is *satisfied* by \((x:10,y:5)\)

\[(x > 0 \land x < 20 \land x = y + y)\] is *not satisfied* by \((x:13,y:6)\)

A formula is satisfied by a *satisfying assignment*.

A formula is *unsatisfiable* if every assignment of values to variables makes the formula false

\[(x > 0 \land x < 20 \land x = y + y \land x \% 2 = 1)\] is *unsatisfiable*
A solver determines if a formula is satisfiable.

• A SAT solver is a solver for propositional logic
• An SMT solver is a solver for formulas in a first-order logic
## Theories

A *theory* specifies the meaning of special symbols.

<table>
<thead>
<tr>
<th>Theory</th>
<th>Symbols</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural numbers</td>
<td>0,1,2, +, -, ...</td>
<td>Standard</td>
</tr>
<tr>
<td>Bit-Vectors</td>
<td>0,1,2,+,-, ^, &amp;,</td>
<td>Bitwise operations, machine arithmetic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1, ...</td>
</tr>
<tr>
<td>Strings</td>
<td>a,b,c, a.b, e*, ...</td>
<td>Concatenation, Kleene-star, etc.</td>
</tr>
<tr>
<td>Arrays</td>
<td>a, a[x], &lt;=, a[x] +4, ...</td>
<td>Indexing, reading, comparison</td>
</tr>
</tbody>
</table>
# Examples of Solvers for Specific Theories

<table>
<thead>
<tr>
<th>Solver</th>
<th>Theory</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>STP</td>
<td>Bit-vectors and arrays</td>
<td><a href="https://sites.google.com/site/stpfastprover/">https://sites.google.com/site/stpfastprover/</a></td>
</tr>
<tr>
<td>Beaver</td>
<td>Bit-vectors</td>
<td><a href="http://uclid.eecs.berkeley.edu/jha/beaver-dist/beaver.html">http://uclid.eecs.berkeley.edu/jha/beaver-dist/beaver.html</a></td>
</tr>
</tbody>
</table>
# Examples of Solvers for Multiple Theories

<table>
<thead>
<tr>
<th>Solver</th>
<th>Theories Supported</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z3</td>
<td>Equality, linear, non-linear arithmetic, arrays, bit-vectors, etc.</td>
<td><a href="http://z3.codeplex.com/">http://z3.codeplex.com/</a></td>
</tr>
<tr>
<td>CVC4</td>
<td>Equality, linear arithmetic, arrays, bit-vectors, strings, etc.</td>
<td><a href="http://cvc4.cs.nyu.edu/web/">http://cvc4.cs.nyu.edu/web/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>----------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Efficiency of Fuzzing</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Symbolic Reasoning</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Path Predicates</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Bug Finding</td>
<td></td>
</tr>
</tbody>
</table>
Write a formula for the values of len and input that execute the colored path.

```
if (input < UINT_MAX - 2)
false
true
```

```
unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)
false
true
```

```
s = len;
if (len % 2 == 0)
false
true
```

```
s = len + 2;
s = len;
```

```
buf = malloc(s);
read(fd, buf, len);
```
Paths as Formulas

Write a formula for the values of len and input that execute the colored path.

```c
if (input < UINT_MAX - 2)
false  true

unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)

s = len;

if (len % 2 == 0)
false  true

s = len + 2;
s = len;

buf = malloc(s);
read(fd, buf, len);
```
Paths as Formulas

Write a formula for the values of `len` and `input` that execute the colored path.

```
if (input < UINT_MAX - 2)
false
  true

unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)

s = len;
if (len % 2 == 0)
false
  true

s = len + 2;
s = len;

buf = malloc(s);
read(fd, buf, len);
```
Write a formula for the values of len and input that execute the colored path.

\[
\text{input} < \text{UINT\_MAX} - 2 \quad \text{&&} \quad \text{len} == \text{input} + 3
\]
Write a formula for the values of `len` and `input` that execute the colored path.

```
if (input < UINT_MAX - 2) {
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
        s = len;
    else
        s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);
    return 0;
}
```
Write a formula for the values of \( \text{len} \) and \( \text{input} \) that execute the colored path.

\[
\text{if (input < UINT\_MAX - 2)} \quad \text{true}
\]

\[
\text{unsigned \ len, s; char* \ buf; len = input + 3; if (len < 10)} \quad \text{false}
\]

\[
\text{s = len; if (len % 2 == 0)} \quad \text{true}
\]

\[
\text{s = len + 2; s = len; buf = malloc(s); read(fd, buf, len);}
\]

\[
\text{input < UINT\_MAX - 2} \quad \text{false}
\]

\[
\text{& & len == input + 3} \\
\text{& & ! (len < 10)} \\
\text{& & ! (len % 2 == 0)}
\]
Paths as Formulas

Write a formula for the values of `len` and `input` that execute the colored path.

```c
if (input < UINT_MAX - 2)
    false
else
    true

unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)
    input < UINT_MAX - 2
else
    false

s = len;
if (len % 2 == 0)
    if (len < 10)
        true
    else
        false

s = len + 2;
s = len;
buf = malloc(s);
read(fd, buf, len);

Satisfying assignments to the `path` predicate:

| input | 8  | 10 | 12 | 14 | 16 | 18 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
<td>19</td>
<td>21</td>
<td>...</td>
</tr>
</tbody>
</table>
```
Path Predicates

A *path predicate* encodes the constraints that must be satisfied for a program path to be executed.

It symbolically represents all inputs for executing the path.

To construct a path predicate

• Rename variables to have unique occurrences
• Assignments become equalities
• Branches are themselves, or negated
• Sequence is conjunction

Theory used should support a proper model of program statements and memory model
Quiz: Path Predicates

Write a formula for the values of len and input that execute the colored path.

```c
if (input < UINT_MAX - 2)
false

unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)

true

s = len;

false

if (len % 2 == 0)
false

s = len + 2;

true

s = len;

buf = malloc(s);
read(fd, buf, len);
```

Dawn Song
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Efficiency of Fuzzing</td>
</tr>
<tr>
<td>2</td>
<td>Symbolic Reasoning</td>
</tr>
<tr>
<td>3</td>
<td>Path Predicates</td>
</tr>
<tr>
<td>4</td>
<td>Bug Finding</td>
</tr>
</tbody>
</table>
Quiz: Spot the Bug

Can you spot the bug involving the integer variables?

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
}
```
Quiz: Spot the Bug

Can you spot the bug involving the integer variables?

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)  
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
}
```
Quiz: Spot the Bug

Can you add an assertion to catch the bug?

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
    }
}
```
Quiz: Spot the Bug

Can you add an assertion to catch the bug?

```c
foo(unsigned input){
    if (input < UINT_MAX - 2){
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len % 2 == 0)
            s = len;
        else
            s = len + 2;
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
    .....}
}
```
Adding Assertion to the CFG

```
if (input < UINT_MAX - 2)
false | true

unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)

s = len;
if (len % 2 == 0)
false | true

s = len + 2;
s = len;

buf = malloc(s);
read(fd, buf, len);
```

```
if (input < UINT_MAX - 2)
false | true

unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)

s = len;
if (len % 2 == 0)
false | true

s = len;
assert(len < UINT_MAX - 1);
true | false

s = len + 2;

buf = malloc(s);
read(fd, buf, len);
err
```
Adding Assertion to the CFG

A path to the potential overflow becomes a path to a potential assertion violation.
Path Predicate for Assertion Violation

```c
if (input < UINT_MAX - 2) {
  unsigned len, s;
  char* buf;
  len = input + 3;
  if (len < 10) {
    s = len;
    if (len % 2 == 0) {
      assert(len < UINT_MAX - 1);
      s = len;
      buf = malloc(s);
      read(fd, buf, len);
    }
    buf = malloc(s);
    read(fd, buf, len);
  }
}
```
Path Predicate for Assertion Violation

input < UINT_MAX - 2
&& len == input + 3
&& ! (len < 10)
&& ! (len % 2 == 0)
Path Predicate for Assertion Violation

1. input < UINT_MAX - 2
2. && len == input + 3
3. && !(len < 10)
4. && !(len % 2 == 0)
5. && !(len < UINT_MAX - 1)

```
if (input < UINT_MAX - 2)
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
        s = len;
        if (len % 2 == 0)
            if (len < UINT_MAX - 1)
                buf = malloc(s);
                read(fd, buf, len);
```
Path Predicate for Assertion Violation

In a theory that correctly encodes the program’s semantics, this formula is satisfiable if and only if the assertion can be violated.

input < UINT_MAX - 2
&& len == input + 3
&& !(len < 10)
&& !(len % 2 == 0)
&& !(len < UINT_MAX - 1)
Assertion Violation as Satisfiability

In the appropriate theory, the formula

\[
\text{input} < \text{UINT_MAX} - 2 \\
\&\& \text{len} == \text{input} + 3 \\
\&\& ! (\text{len} < 10) \\
\&\& ! (\text{len} \% 2 == 0) \\
\&\& !(\text{len} < \text{UINT_MAX}) \\
\]

is satisfied by the assignment

\[
\text{input} \quad \text{UINT_MAX} - 3 \\
\text{len} \quad \text{UINT_MAX}
\]
Constraint-Based Automatic Test Case Generation

• Look inside the box
  • Use the code itself to guide the fuzzing
• Encode security/safety properties as assertions
• Explore program paths on which assertions occur
• Steps involved
  1. Find inputs going down different execution paths
  2. For a given path, check if there are inputs that cause a violation of the security property
| **DART**       | *DART: Directed Automated Random Testing*, Godefroid, Klarlund, Sen, PLDI 2005  
|               | [http://dl.acm.org/citation.cfm?id=1065036](http://dl.acm.org/citation.cfm?id=1065036) |
| **CUTE**       | *CUTE: A Concolic Unit Testing Engine for C*, Sen, Marinov, Agha, ESEC/FSE 2005  
|               | [http://dl.acm.org/citation.cfm?id=1081750](http://dl.acm.org/citation.cfm?id=1081750) |
|               | [https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar_html/](https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar_html/) |
# Articles about Symbolic Execution for Security

http://bitblaze.cs.berkeley.edu/papers/bitblaze_iciss08.pdf |
|---|---|
http://www.ece.cmu.edu/~ejschwar/papers/cav11.pdf |
| S2E | *S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems*, Chipounov, Kuznetsov, Candea, ASPLOS 2011  
http://dslab.epfl.ch/pubs/s2e.pdf?attredirects=0 |
http://dl.acm.org/citation.cfm?id=2093564 |
Summary of Symbolic Execution for Bug Finding

• Augment a program with appropriate assertions
• Symbolically execute a path
  • Create formula representing path constraint and assertion failure
  • Solve constraints with a solver
  • A satisfying assignment, if found, is an input triggering a bug
• Reverse a branch condition to explore a different path
  • Give solver the new constraint
  • If the constraint is satisfiable
    • The path is feasible
    • There is an input going down a different path