SimuVEX

Using VEX in Symbolic Analysis

Yan Shoshitaishvili
yans@cs.ucsb.edu
2014
Who am I?

My name is Yan Shoshitaishvili, and I am a PhD student in the Seclab at UC Santa Barbara.

Email: yans@cs.ucsb.edu
Twitter: @Zardus
Github: http://github.com/zardus
Blog: http://blog.yancomm.net

This work is a collaboration between the UCSB Seclab and the Northeastern Seclab!
Don't Panic!

This presentation **does** have a design!

1. Who (are we)?
2. What (is Symbolic Analysis)?
3. Why (did we choose VEX)?
4. How (do we do it)?
5. Where (does all of this get us)?
6. When (will it be released)?
Why Symbolic Analysis?

"How do I trigger path X or condition Y?"

- Dynamic analysis
  - Based on concrete inputs to application.
- (Concrete) static analysis
  - "You can't"/"You might be able to"
  - Based on various static techniques.

We need something slightly different.
What is Symbolic Analysis?

"How do I trigger path X or condition Y?"

1. Interpret the application.
2. Track "constraints" on variables.
3. When the required condition is triggered, "concretize" to obtain a possible input.
"Concretize"?

<table>
<thead>
<tr>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \geq 10$</td>
</tr>
<tr>
<td>$x &lt; 100$</td>
</tr>
</tbody>
</table>

Constraint solving:

- Conversion from set of constraints to set of concrete values that satisfy them.
- NP-complete, in general.
Symbolic Execution Example

```python
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```
Symbolic Execution Example

```python
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```

<table>
<thead>
<tr>
<th>State A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
</tr>
<tr>
<td>x = ???</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
</tr>
</tbody>
</table>
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
Symbolic Execution Example

```python
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```

<table>
<thead>
<tr>
<th>State AA</th>
<th>State AB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Variables</td>
</tr>
<tr>
<td>x = ???</td>
<td>x = ???</td>
</tr>
<tr>
<td>Constraints</td>
<td>Constraints</td>
</tr>
<tr>
<td>x &lt; 10</td>
<td>x &gt;= 10</td>
</tr>
</tbody>
</table>
Symbolic Execution Example

```python
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"

State ABA

<table>
<thead>
<tr>
<th>Variables</th>
<th>x = ???</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>x &gt;= 10</td>
</tr>
<tr>
<td></td>
<td>x &lt; 100</td>
</tr>
</tbody>
</table>

Concretized ABA

<table>
<thead>
<tr>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 99</td>
</tr>
</tbody>
</table>
Symbolic Analysis Is Useful

Lots of uses:

- Reasoning about reachability
- Bughunting
- Test-case generation
Symbolic Analysis Is Hard

Two main challenges unique to symbolic analysis:

1. Constraint Solving
   a. NP-complete, in general
   b. "not our field"

2. State Explosion
   a. All outcomes of a piece of code must be considered.
   b. Loops!
Reinventing the Wheel

Existing systems:

1. Source level: EXE, CUTE, KLEE, AEG
2. Binary level: Mayhem, Fuzzball, Avalanche
3. System level: S2E

Hard to find a balance of flexibility, usability, and support.
Stand on the Shoulders of Giants

Balance between fine-grained control and existing tool/idea reuse:

Concepts: related work

Binary translation: VEX

Constraint solving: Z3
Why Z3?

"Shared-source" constraint solver from Microsoft Research.

- Actively developed
- Powerful and flexible
- Python bindings!
- Not too hard to switch away from!
VEX Crash Course

VEX is Valgrind's intermediate language, allowing Valgrind's tools to be implemented once for cross-platform analyses.

VEX IR

- t0 = GET:i64(48)
- t1 = LDle:i64(t0)
- t2 = Add64(t0,0x8:i64)
- PUT(48) = t2
- PUT(184) = t1
- t4 = GET:i64(184)
- PUT(184) = t4
Code VEXonomy

VEX translates instructions to IRExprs, IRStmts, IRSBs.

- IRExprs provide the values
- IRStmts "describe" state changes
- IRSBs maintain structure/order

Creates a reproducible, side-effects-free representation.
Step-by-step VEXample

0x8000: dec eax

\[ t0 = \text{GET:} \text{l32}(8) \]
\[ t1 = \text{Sub}(t0, 1) \]
\[ \text{PUT}(8) = t1 \]
\[ \text{PUT}(68) = 0x8001 \]

IRStmt: set t0 to... IRExpr: value of eax

IRStmt: set t1 to... IRExpr: t0 - 1

IRStmt: put into eax... IRExpr: t1

IRStmt: put into eip... IRExpr: addr of next instruction
Step-by-step VEXample (2)

0x8001: jz 0x9000

VEX

t2 = Z_FLAG()

Exit 0x9000 if t2

PUT(68) = 0x8003

IRStmt: set t0 to... IRExpr: value of eax

IRStmt: exit to 0x9000 if... IRExpr: t0

IRStmt: put into eip... IRExpr: addr of next instruction
VEXamorphosis

SimuVEX creates a symbolic interpretation layer over VEX:
VEXterpretation

- SimIRExprs represent symbolic values.
- SimIRStmts modify a symbolic state.

What's a symbolic state?

**SimState**

- symbolic memory
- symbolic registers
- constraints
- plugins
  - (symbolic) 'kernel' state for userspace binaries
Symbolic Interpretation (IRStmt)

Every SimIRStmt takes a state, makes changes to memory, registers, and constraints, and outputs a set of states.

Initial SimState
- symbolic memory
- symbolic registers
- constraints
- plugins
  - (symbolic) 'kernel' state for userspace binaries

SimIRStmt

New SimState
- symbolic memory
- symbolic registers
- constraints
  - … etc

New SimState
- symbolic memory
- symbolic registers
- constraints
  - … etc
Symbolic Interpretation (IRSB)

These statements are aggregated in SimIRSBs.

- Initial SimState
  - symbolic memory
  - symbolic registers
  - constraints
  - plugins
    - (symbolic) 'kernel' state for userspace binaries

- SimIRSB
- SimIRStmt
  - symbolic memory
  - symbolic registers
  - constraints
  - ... etc

- New SimState
  - symbolic memory
  - symbolic registers
  - constraints
  - ... etc
Complications...

The naive approach has some issues.

```c
void *memcpy(void *dst, void *src, int n)
{
    for (int i = 0; i < n; i++)
        dst[i] = src[i];

    return dst;
}
```

What happens with a symbolic "n"?
for (int i = 0; i < n; i++) {...}

<table>
<thead>
<tr>
<th>State Initial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
</tr>
<tr>
<td>Constraints</td>
</tr>
</tbody>
</table>

| State A+ |
| Variables |
| Constraints |

| State B+ |
| Variables |
| Constraints |

| State C+ |
| Variables |
| Constraints |

| State A- |
| Variables |
| Constraints |

| State B- |
| Variables |
| Constraints |

| State C- |
| Variables |
| Constraints |
Symbolic Summaries

Solution: replace it with a manually written "symbolic summary".

Pro: intelligently reason about conditions
Pro: increased analysis speed
Con: manual implementation

Also used to abstract away system calls.
To support symbolic summaries, we abstract anything that takes an input state and produces output states as a "SimRun".

**Useful Abstractions**

- Initial SimState
  - symbolic memory
  - symbolic registers
  - constraints
  - plugins
    - (symbolic) 'kernel' state for userspace binaries

- SimRun

- New SimState
  - symbolic memory
  - symbolic registers
  - constraints
  - ... etc
A SimRun can be one of several things:

- A SimIRSB, to support direct binary analysis
- A path of SimIRSBs, to aid in program slicing
- A summary of state modifications.
Why?

The SimRun abstraction provides several powerful capabilities:

- Simplifies the analysis
  - most analyses just use SimRun
  - transparency enable/disable symbolic summaries
- SimRuns can execute in symbolic or concrete mode
  - enables concolic execution on a SimRun-granularity
What do we use this for?

We can leverage all this complex stuff to search for bugs or vulnerabilities! For example, authentication bypass vulnerabilities.

```
get_credentials
authenticate
failure
success
evil_strcmp
```
Demo time!
Wow!

We've been gradually releasing stuff!

- So far, the non-symbolic underpinnings.
  - PyVEX ([http://github.com/zardus/pyvex](http://github.com/zardus/pyvex))
  - IDALink ([http://github.com/zardus/idalink](http://github.com/zardus/idalink))
- Other minor, uninteresting things

- More to come!
Questions?
Comments?
Collaboration Ideas?