QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing

Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang †, and Taesoo Kim

Georgia Institute of Technology & Oregon State University †

27th USENIX Security Symposium
August 16, 2018
Two popular ways to find security bugs: Fuzzing & Concolic execution
Fuzzing and Concolic execution have their own pros and cons

• Fuzzing
  • Good: Finding general inputs
  • Bad: Finding specific inputs

• Concolic execution
  • Good: Finding specific inputs
  • Bad: State explosion
Hybrid fuzzing can address their problems

• Use both techniques: Fuzzing + Concolic execution
• Find specific inputs: Using concolic execution
• Limit state explosion: Only fork at branches that are hard to fuzzing
Hybrid fuzzing has achieved great success in small-scale study

- e.g.) Driller: a state-of-the-art hybrid fuzzer
  - Won 3rd place in CGC competition
  - Found 6 new crashes: cannot be found by fuzzing nor concolic execution
However, current hybrid fuzzing suffers from problems to scale to real-world applications

• Very slow to generate constraint

• Cannot support complete system calls

• Not effective in generating test cases
Our system, QSYM, addresses these issues by introducing several key ideas

• Discard intermediate layer for performance

• Use concrete environment to support system calls

• Introduce heuristics to effectively generate test cases
QSYM is scalable to real-world software

• 13 previously unknown bugs in open-source software

• All applications are already fuzzed (OSS-Fuzz, AFL, ...)
  • Including ffmpeg that is fuzzed by OSS-Fuzz for 2 years

• Bugs are hard to pure fuzzing – require complex constraints
Overview: Hybrid fuzzing in general

Program

```
push ebp
mov ebp, esp
...
```

Basic block

```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
...
```

Intermediate Representations

```
A[0] == 'A'
...
```

Constraints

Coverage

Fuzzing

State forking

Test cases
Overview: Hybrid fuzzing in general

Program

push ebp
mov ebp, esp
...

Basic block

Intermediate Representations

t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
...

Intermediate Representations

A[0] == 'A'
...

Constraints

Coverage

State forking

Fuzzing

Performance overhead

Test cases

Performance overhead
Overview: QSYM

1. Instruction-level execution

Program

push ebp
mov ebp, esp
...

Basic block

A[0] == 'A'
...

Constraints

Coverage

Fuzzing

Test cases
Overview: Hybrid fuzzing in general

Program

push ebp
mov ebp, esp
...

Basic block

Intermediate Representations

t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
...

Intermediate Representations

A[0] == 'A'
...

Constraints

State forking

Coverage

Fuzzing

Incomplete
Environment modeling

Test cases
Overview: QSYM

1. *Instruction-level execution*
2. *Concrete environment modeling*

Program

```
push ebp
mov ebp, esp
...
```

Basic block

```
A[0] == 'A'
...
```

Constraints

Coverage

Test cases
Overview: Hybrid fuzzing in general

Program

push ebp
mov ebp, esp
...

Basic block

t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
...

Intermediate Representations

A[0] == 'A'
...

Constraints

Test cases

Coverage

State forking

Ineffective test case generation due to unsatisfiable paths.
Overview: QSYM

1. Instruction-level execution
2. Concrete environment modeling
3. Optimistic Solving

Program

Basic block

Constraints

Coverage

Test cases

Fuzzing

Instruction-level execution
Concrete environment modeling
Optimistic Solving
Overview: Hybrid fuzzing in general

- Program
  - push ebp
  - mov ebp, esp
  - ...

- Basic block

- Intermediate Representations
  - $t0 = \text{GET:I32}(ebp)$
  - $t1 = \text{GET:I32}(esp)$
  - $t2 = \text{Sub32}(t1, 0x00000004)$
  - ...

- Constraints
  - $A[0] == 'A'$
  - ...

- State forking

- Coverage

- Fuzzing

- Blocked by complex logics

- Test cases
Overview: QSYM

1. Instruction-level execution
2. Concrete environment modeling

3. Optimistic Solving

4. Basic block pruning
Refer our paper

Program

Basic block

push ebp
mov ebp, esp ...

Constraints

A[0] == 'A'
...

Coverage

Fuzzing

Test cases
Overview: Hybrid fuzzing in general

Program:
```
push ebp
mov ebp, esp
...  
```

Basic block:
```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
...  
```

Intermediate Representations:
```
A[0] == 'A'
...  
```

Constraints:
```
A[0] == 'A'
...  
```

Performance overhead:

Coverage:

Fuzzing:

State forking:

Test cases:

18
Intermediate representations (IR) are good to make implementations easier

• Provide architecture-independent interpretations

• Can re-use code for all architectures

• e.g. angr works on many architectures: x86, arm, and mips
Problem 1: IR incurs significant performance overhead

- Increase the number of instructions
  - 4.7 times in VEX (IR used by angr)

- Need to execute a whole basic block symbolically
  - Due to caching and optimization
  - Only 30% of instructions need to be symbolically executed
Solution 1: Execute instructions directly without using intermediate layer

• Remove the IR translation layer
• Pay for the implementation complexity
QSYM reduces the number of instructions to execute symbolically

- 126 CGC binaries
Overview: Hybrid fuzzing in general

Program

push ebp
mov ebp, esp
...

Basic block

t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
...

Intermediate Representations

Intermediate Representations

A[0] == 'A'
...

Constraints

Coverage

State forking

Fuzzing

Incomplete Environment modeling

Test cases
State forking can reduce re-execution overhead for constraint generation

• No need to re-execute to reach the state
  • Recover from the snapshot
State forking for kernel is non-trivial

- State in concolic execution = Program state + Kernel state

- Forking program state is trivial
  - Save application memory + register
  - Save constraints

- Forking kernel state is non-trivial
  - Need to maintain all kernel data structures
  - e.g., file system, network state, memory system ...
Problem 2: State forking introduces problems in either completeness or performance

- **Kernel modeling**
  - e.g.) angr
  - Pros: Small performance overhead
  - Cons: Incompleteness – angr supports only 22 system calls in Linux

- **Full kernel emulation**
  - e.g.) S2E
  - Pros: Completeness
  - Cons: Large performance overhead
Solution 2: Re-execute to use concrete environment instead of kernel state forking

• Instead of state forking, re-execute from start

• High re-execution overhead
  • Instruction-level execution
  • Basic block pruning

• Limit constraint solving: Based on coverage from fuzzing
Models minimal system calls and uses concrete values

• Only model system calls that are relevant to user interactions
  • e.g.) standard input, file read, …

• Other system calls: Call system call using concrete values
  • e.g.) mprotect(addr, sym_size, PROT_R)
    → mprotect(addr, conc_size, PROT_R)
Problem: Concrete environment results in incomplete constraints

- Add implicit constraints
  - e.g.) `mprotect(addr, sym_size, PROT_R)`
    \[\rightarrow mprotect(addr, conc_size, PROT_R)\]

- Without knowing semantics of system calls
  - Concretize: Over-constrained
  - Ignore: Under-constrained
Unrelated constraint elimination can tolerate incomplete constraints

\[
x = \text{int} \left( \text{input()} \right)
\]
\[
y = \text{int} \left( \text{input()} \right)
\]

# Incomplete constraints
mprotect(addr, x, PROT_R)

if \( y \times y == 1337 \times 1337 \):
   bug()

Constraints for x (Incomplete)
&& \( y \times y == 1337 \times 1337 \)

Path constraints
\( y \times y == 1337 \times 1337 \)

Branch dependent constraints
x = Use concrete value
y = 1337
Overview: Hybrid fuzzing in general

Program

Basic block

Intermediate Representations

Ineffective test case generation due to unsatisfiable paths

Fuzzing

Coverage

State forking

Constraints

Test cases

...
Problem3: Over-constrained paths results in no test cases

```python
type = int(input())

if type == TYPE1:
    parse_TYPE1()
...

if type == TYPE2:
    parse_TYPE2()
```

Unsatisfiable: No test case
Problem 3: Over-constrained paths results in no test cases

```python
type = int(input())
if type == TYPE1:
    parse_TYPE1()
...
if type == TYPE2:
    parse_TYPE2()
```

If these branches are independent:

- `type == TYPE1`
- `type != TYPE1` + long time
- `type == TYPE2`
Solution3: Solve constraints optimistically

type = int(input())

if type == TYPE1:
    parse_TYPE1()

...

if type == TYPE2:
    parse_TYPE2()
Our decision: Solve only the last constraint in the path

- Simple: Only one constraint
- High chance to pass the branch
- Only waste a small solving time

```python
type = int(input())

if type == TYPE1:
    parse_TYPE1()

...  

if type == TYPE2:
    parse_TYPE2()
```
In hybrid fuzzing, generating incorrect inputs are fine due to fuzzing

Program

push ebp
mov ebp, esp
...

Basic block

A[0] == 'A'
...

Constraints

Fuzzing will filter out incorrect inputs based on coverage

Coverage

Test cases

Fuzzing will filter out incorrect inputs based on coverage
Optimistic solving helps to find more bugs

• LAVA-M dataset
Implementation

• 16K LoC of C++

• Intel Pin: emulation

• Z3: constraint solving

• Will be available at https://github.com/sslab-gatech/qsym
Evaluation questions

• Scaling to real-world software?

• How good is QSYM compared to
  • Driller (a state-of-the-art hybrid fuzzing)
  • Vuzzer (a state-of-the-art fuzzing)
  • Fuzzing and symbolic execution
QSYM scales to real-world software

- 13 bugs in real-world software

<table>
<thead>
<tr>
<th>Program</th>
<th>CVE</th>
<th>Bug Type</th>
<th>Fuzzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>lepton</td>
<td>CVE-2017-8891</td>
<td>Out-of-bounds read</td>
<td>AFL</td>
</tr>
<tr>
<td>openjpeg</td>
<td>CVE-2017-12878</td>
<td>Heap overflow</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td></td>
<td>Fixed by other patch</td>
<td>NULL dereference</td>
<td></td>
</tr>
<tr>
<td>tcpdump</td>
<td>CVE-2017-11543*</td>
<td>Heap overflow</td>
<td>AFL</td>
</tr>
<tr>
<td>file</td>
<td>CVE-2017-1000249*</td>
<td>Stack overflow</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>libarchive</td>
<td>Wait for patch</td>
<td>NULL dereference</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>audiofile</td>
<td>CVE-2017-6836</td>
<td>Heap overflow</td>
<td>AFL</td>
</tr>
<tr>
<td></td>
<td>Wait for patch</td>
<td>Heap overflow × 3</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td></td>
<td>Wait for patch</td>
<td>Memory leak</td>
<td></td>
</tr>
<tr>
<td>ffmpeg</td>
<td>CVE-2017-17081</td>
<td>Out-of-bounds read</td>
<td>OSS-Fuzz</td>
</tr>
<tr>
<td>objdump</td>
<td>CVE-2017-17080</td>
<td>Out-of-bounds read</td>
<td>AFL</td>
</tr>
</tbody>
</table>
QSYM can generate test cases that fuzzing is hard to find

- e.g.) ffmpeg: Not reachable by fuzzing

```c
if( ((ox^ox+dxw))
    | (ox^ox+dxh))
    | (ox^ox+dxw+ dxh))
    | (oy^oy+dyw))
    | (oy^oy+dyh))
    | (oy^oy+dyw+ dyh))) >> (16 + shift)
   || (dxx | dxy | dyx | dyy) & 15
   || (need_emu && (h > MAX_H || stride > MAX_STRIDE)))
{ ... return; }
// the bug is here
```
Compare QSYM with Driller, a state-of-the-art hybrid fuzzing

- Dataset: 126 binaries from CGC

- Run only *one* instance of concolic execution for *5 min*
  - i.e., remove fuzzing
- Compare code coverage
QSYM achieved more code coverage than Driller in most cases of CGC

• Among 126 challenges
  • QSYM achieved more: 104 challenges
  • Driller achieved more: 18 challenges
QSYM achieved more code coverage due to its better performance

- e.g., CROMU_00001

- To achieve new code coverage, seven stages are required
  - Add one user → Add another user → login → send to message → ...

- QSYM can reach the stage, but Driller cannot in time
Driller achieved more code coverage if nested branches exist

• Driller can find inputs for nested branches by a single execution due to forking

• QSYM requires re-execution
  • NOTE: Our experiment allows only one instance of concolic execution
QSYM outperforms other techniques in LAVA-M dataset

- LAVA-M dataset: inject hard-to-find bugs in real-world software
- 5 hour run

<table>
<thead>
<tr>
<th></th>
<th>uniq</th>
<th>base64</th>
<th>md5sum</th>
<th>who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>28</td>
<td>44</td>
<td>57</td>
<td>2,136</td>
</tr>
<tr>
<td>FUZZER</td>
<td>7 (25%)</td>
<td>7 (16%)</td>
<td>2 (4%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>SES</td>
<td>0 (0%)</td>
<td>9 (21%)</td>
<td>0 (0%)</td>
<td>18 (1%)</td>
</tr>
<tr>
<td>VUzzer</td>
<td>27 (96%)</td>
<td>1 (2%)</td>
<td>0 (0%)</td>
<td>23 (1%)</td>
</tr>
<tr>
<td>QSYM</td>
<td>28 (100%)</td>
<td>44 (100%)</td>
<td>57 (100%)</td>
<td>1,238 (58%)</td>
</tr>
</tbody>
</table>
Discussions & Limitation

• Use of less accurate test cases
  • Requires efficient validators
  • e.g., exploit generation

• Implementation status
  • Only support x86, x86_64
  • No floating point support
Conclusion

• Hybrid fuzzing scalable to real-world software
  • 13 bugs in real-world software
• Outperform a state-of-the-art hybrid fuzzing and other bug finding

• https://github.com/sslab-gatech/qsym
Thank you
Using only the last constraint is good for time and bug finding
Number of instructions that are not emulated by QSYM due to its limitation

- 13 / 126 challenges: At least one
- 3 / 126 challenges: More than 1% of total instructions