Qsym : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing

INSU YUN, SANGHO LEE, AND MENG XU, TAESOO KIM, *GEORGIA INSTITUTE OF TECHNOLOGY;*

YEONGJIN JANG, *OREGON STATE UNIVERSITY;*

Presented by: Oskars Dauksts
October 30, 2018

Overview

- Background
- Introduction
- Motivation
- Design
- Implementation
- Evaluation
- Discussion
- Conclusion

WAYNE STATE

Background

Background – Key Terms

- **Concolic Execution –** Combines symbolic execution with concrete execution
  - **Symbolic execution –** Allows for execution of all possible paths
  - **Concrete execution –** Concrete values that guide the execution through constraints
- **Fuzzing –** QA technique that involves inputting large amount of inputs to test coding errors, input filtering and other loopholes.
- **Hybrid Fuzzing –** Concolic Execution + Fuzzing

Background

- Limitations to hybrid fuzzing
  - Scaling to real-world software
- Introduction of Qsym
  - Native execution with symbolic emulation.
- Results

Introduction

Introduction

- Fuzzing
  - Quickly discover inputs to execution path with lose conditions
  - x > 0;
- Concolic Execution
  - Good at finding inputs that use complex conditions
    - x == 0xdeadbeef;
  - Expensive and Slow
- Past Solution: Hybrid Fuzzing

Introduction - Limits

- Suffer from scaling in non-trivial inputs
- Symbolic emulation is too slow

- Introduced Solution Qsym
  - Integrate symbolic emulation to native execution using dynamic binary translation
  - Optimistically solve constraints
  - Prune basic blocks

Introduction - Qsym

- Fast concolic execution through efficient emulation
  - Optimized emulation speed
- Efficient repetitive testing and concrete environment
  - Allows fast re-execution, eliminates snapshots
- New heuristics for hybrid fuzzing
  - Prune compute intensive blocks
- Real-world bugs
  - New bugs discovered

Motivation

Motivation – Slow Symbolic Emulation

- Why IR
  - Machine language → IR instructions for easy modeling
  - Easy to develop
- Why not IR
  - Big overhead
  - Sometimes 1 machine instruction = 2 IR instructions
  - Caching of IR instructions forces execution on the basic block level

Motivation – Ineffective Snapshot

- Why Snapshots
  - Eliminates execution overhead
- Why not Snapshots
  - Sometimes hybrid fuzzing does not share a common branch
    - Leads fuzzer to wrong code path
  - Interaction with external environments

Motivation - Slow and Inflexible Sound Analysis

- Why Sound Analysis
  - Guaranteed soundness by collecting complete constraints
  - No false expectations
- Why not
  - Could lead to never ending analysis
    - Ex: file_zmagic()
    - Decompression of zlib contains complex constraints
    - Other interesting code is missed
  - Over-constraining the path

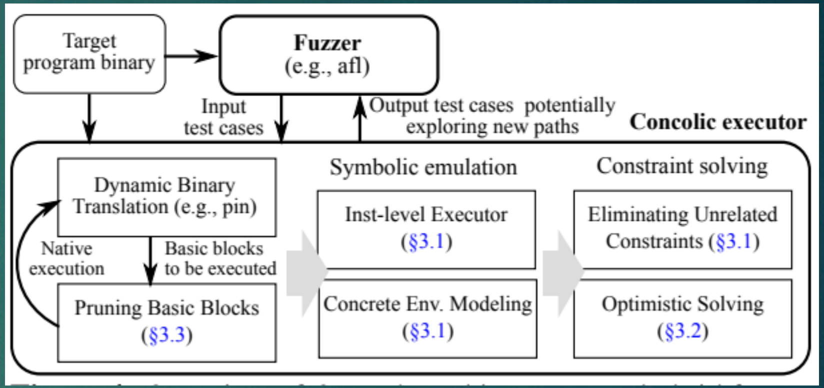
Motivation - Approach

- Slow Symbolic Emulation → Remove IR translations
  - Pay for implementation complexity
- Ineffective Snapshot → Remove snapshot mechanism
  - Concrete Execution to model external environment
- Slow and Inflexible Sound Analysis → Solve only portion of overly-constrained paths

Design

Design - Qsym Architecture

- 1) Input: Test case and Binary file
- 2) Attempts to generate new test cases
- 3) Uses DBT to natively execute the input
- 4) Prunes Basic blocks
- 5) Symbolic emulation integrated with native execution
- 6) Solving all of the constraints while generating new test cases



Design - Taming Concolic Executor

- Instruction-level Symbolic Execution
  - Only executes small set of instructions that are required to generate symbolic constraints (Figure 1)
- Solving constraints that are relevant to the target branch
  - Other concolic executors do it incrementally (Figure 2)
- Re-execution over snapshotting
  - Qsym runs natively
- Concrete external environment
  - Executes them by concrete values

```
// If rdx (size) is symbolic
__memset_sse2:
  movd     xmm0,esi
  mov      rax,rdi
  punpcklbw xmm0,xmm0
  punpcklwd xmm0,xmm0   ──→
  pshufd xmm0,xmm0,0x0
  cmp      rdx,0x40
  ja __memset_sse2+80
```

```
def _op_generic_InterleaveLO(self, args):
    s = self._vector_size
    c = self._vector_count
    left_vector = [args[0][(i+1)*s-1:i*s]
                         for i in xrange(c/2)]
    right_vector = [args[1][(i+1)*s-1:i*s]
                         for i in xrange(c/2)]
    return claripy.Concat(*itertools.chain.from_iterable(
             reversed(zip(left_vector, right_vector))))
```

Figure 1

Initial PoV	Qsym	Driller
1 # create user	1 # create user	1 # create user
userone	userone	\xfb\xfb\xfb\xfb\xf4\xf1\xf1
1 # create user	1 # create user	1 # create user
usertwo	usertwo	\xfb\xfb\xfb\xfb\x0b\xfb\xf1
2 # login	2 # login	2 # login
userone	userone	\xfb\xfb\xfb\xfb\xf4\xf1\xf1
1 # send message	4 # delete message	4 # delete message

Figure 2

Design – Optimistic Solving

- Qsym generates new test cases from over-constraint problems
  - Utilization of Hybrid fuzzer
    - Formulates new test inputs
  - Optimistically selects some portion of constraints

Design – Basic Block Pruning

- Elimination of repetitive code execution
  - Ex: Compute intensive operations
- Uses Exponential Back off
  - Executes block with power of 2
- Grouping multiple execution and Context Sensetivity
  - Prevents excessive pruning

Implementation

Implementation

- Total 16k LoC
- Intel Pin for Dynamic Binary Translation (DBT)
- Utilizes libdft for handling system calls
- Supports part of Intel-64 instructions
  - Adding support to different type of instructions in the future

Component	Lines of code
Concolic execution core	12,528 LoC of C++
Expression generation	1,913 LoC of C++
System call abstraction	1,577 LoC of C++
Hybrid fuzzing	565 LoC of Python

Discussion

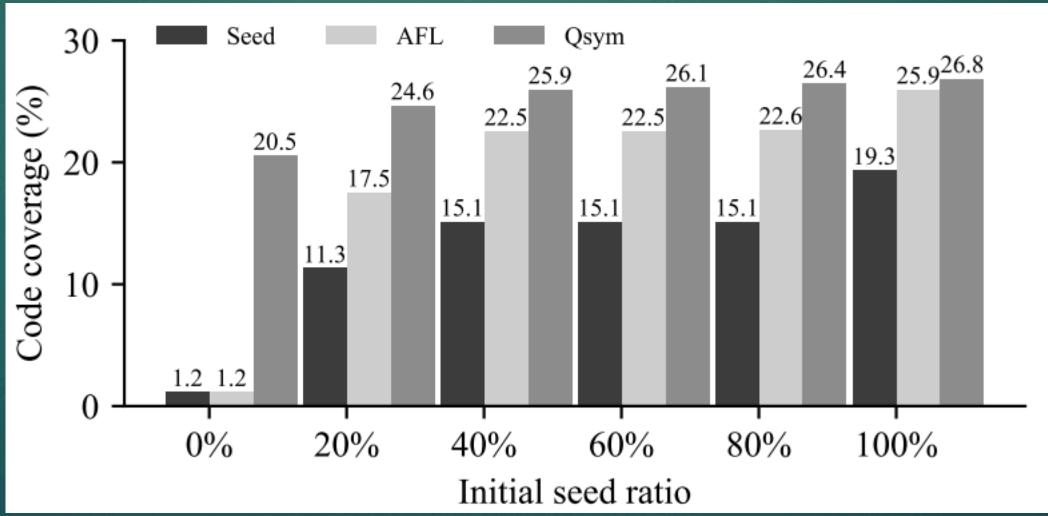
Evaluation - Scaling real-world problems

- Found 13 new bugs
  - Stack and Heap overflows
  - NULL references
- Reason for better scaling than state of art fuzzers
  - Ability to detect errors in Incomplete or Incorrect systems calls

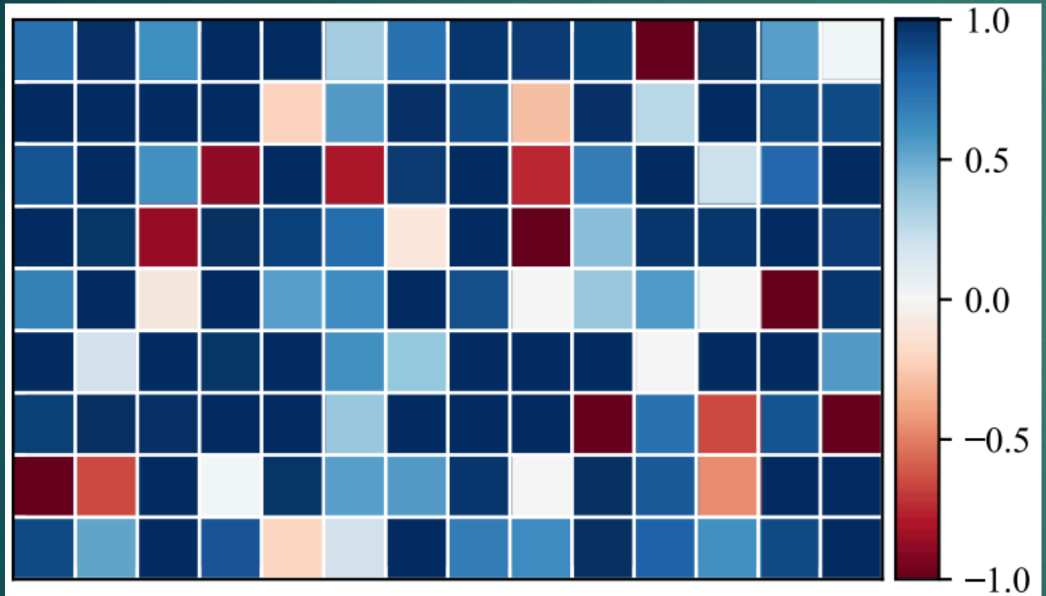
Program	Bug Type	Syscall
libtiff	Erroneous system calls	`mmap`
openjpeg	Unsupported system calls	`set_robust_list`
tcpdump	Erroneous system calls	`mmap`
libarchive	Unsupported system calls	`fcntl`
ffmpeg	Unsupported system calls	`rt_sigaction`

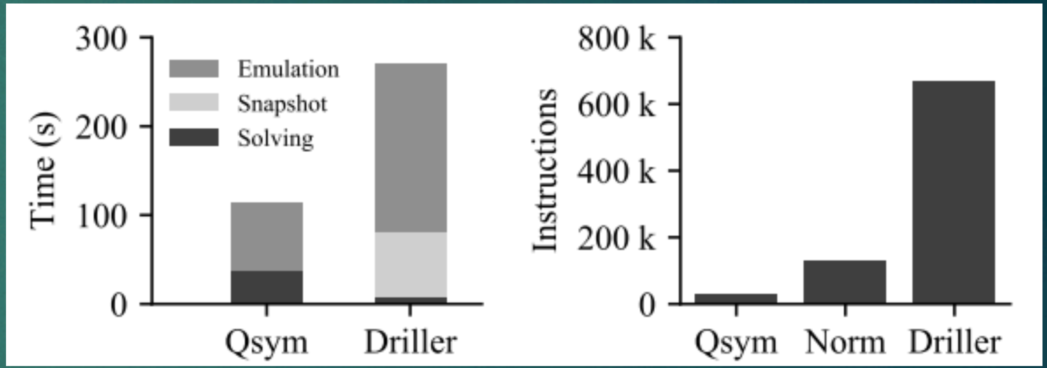
Evaluation – Code Coverage Effectiveness

- Qsym vs AFL fuzzer on libPNG project
- 6 hour run
- Dummy input at 0%
- 141 samples



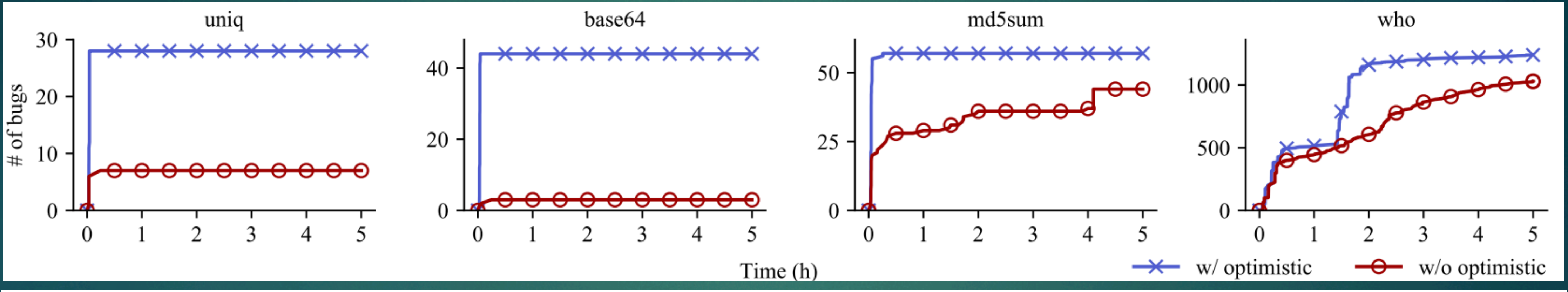
Evaluation – Fast Symbolic Evaluation

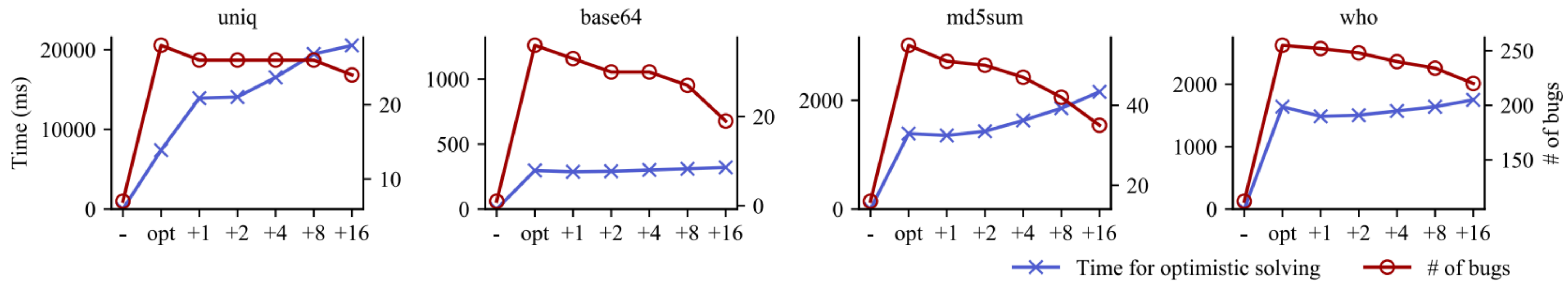




Evaluation – Optimistic Solving

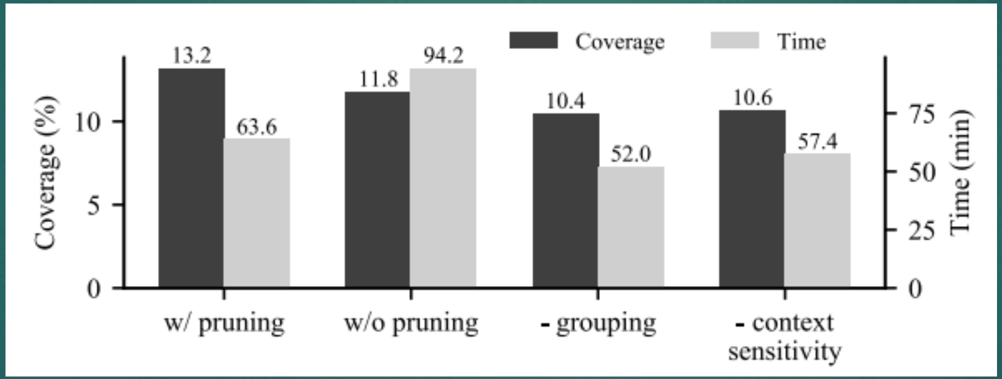
- Relax on over constraint variables





Evaluation – Pruning Basic Blocks

- Effect of pruning basic blocks
  - Reduced execution time
  - Bigger code coverage



Discussion

- Adoption beyond fuzzing
  - Basic block pruning can be applied to parsers
  - Applied to other concolic executors
- Complementing each other with other fuzzers
  - Can be used with fuzzers that enhance currently used AFL fuzzer
- Limitations
  - Bound to theoretical limits to constrain solving
  - X86_64
  - Not all instructions are supported

Conclusion

Conclusion

- Fast concolic execution engine tailored to use with hybrid fuzzers
- Scalable for real world applications
- Outperformed current fuzzing tools
- Found new undiscovered bugs