SymQEMU

Compilation-based symbolic execution for binaries

Sebastian Poeplau
EURECOM and Code Intelligence

Aurélien Francillon
EURECOM

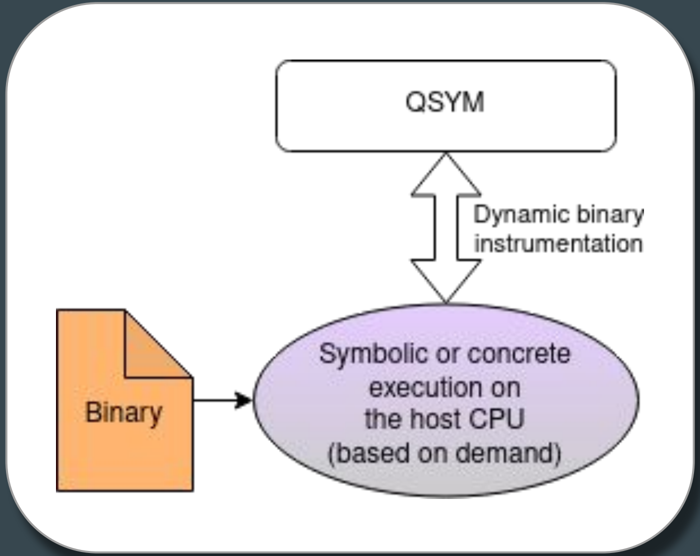
Motivation

- Want fast and flexible binary-only symbolic execution
    - Idea: apply compilation-based symbolic execution [1] to binaries
- Why would you want to work without sources?
    - Proprietary dependencies
    - Security audits (e.g., firmware analysis)
    - Large projects with complex build systems, multiple source languages, etc.
- Why not use one of the existing solutions?
    - Often need to choose between speed and flexibility
    - High complexity

[1] Poeplau and Francillon: Symbolic execution with SymCC: Don't interpret, compile! USENIX Security 2020

QSYM

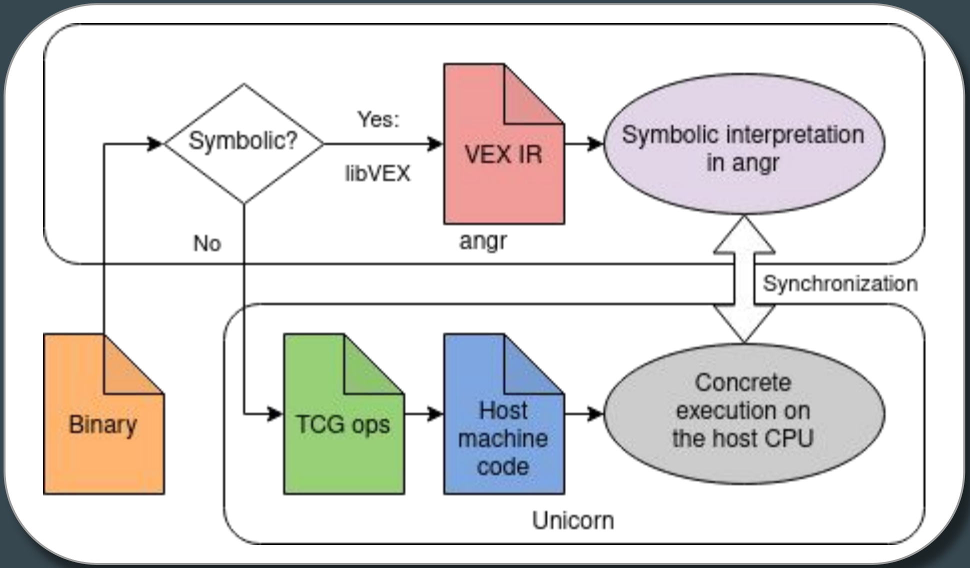
- Based on dynamic binary instrumentation
  - Intel Pin to insert symbolic handling at run time
  - Symbolic semantics at the x86 machine-code level
- High performance, conceptually simple
- Architecturally inflexible
  - Tied to the x86 instruction set
- Tedious implementation
  - Need to implement symbolic handling for *each x86 instruction*



Yun et al.: QSYM: A practical concolic execution engine tailored for hybrid fuzzing, USENIX 2018

angr

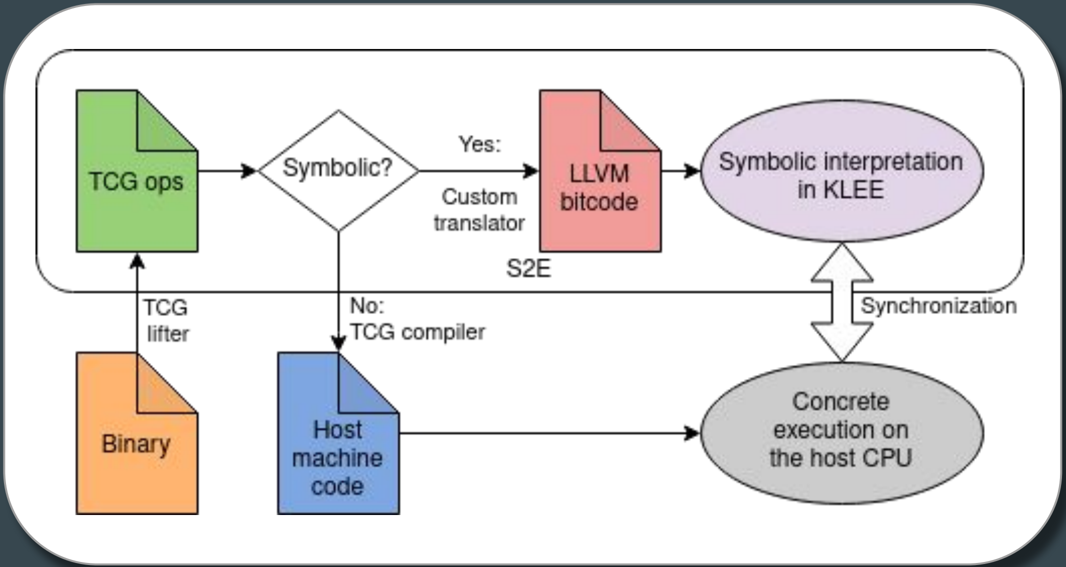
- Dynamically translate binary to VEX, then interpret symbolically
- Fast path for concrete execution: Unicorn CPU emulator
- Very flexible
- Low execution speed
  - Python implementation
  - Interpretation is slower than compiled code



Shoshitaishvili et al.: SoK: (State of) The art of war: Offensive techniques in binary analysis, S&P 2016

S2E

- Basic idea: QEMU + KLEE
  - QEMU's TCG ops are lifted to LLVM bitcode
  - Bitcode is fed to KLEE
- Entire operating system inside
- Conceptually very flexible
  - Implemented for x86 only
- Highly complex



Chipounov et al.: Selective symbolic execution, HotDep 2009 *and*
The S2E platform: Design, implementation, and applications, ACM TOCS 2012

Goals

- Speed!
- Architectural flexibility
  - Firmware analysis requires support for many CPU types
  - Analysis host may be different from target architecture
- Robustness
  - Don’t want to write disassemblers ourselves
- Simplicity
  - Make a system that others can extend

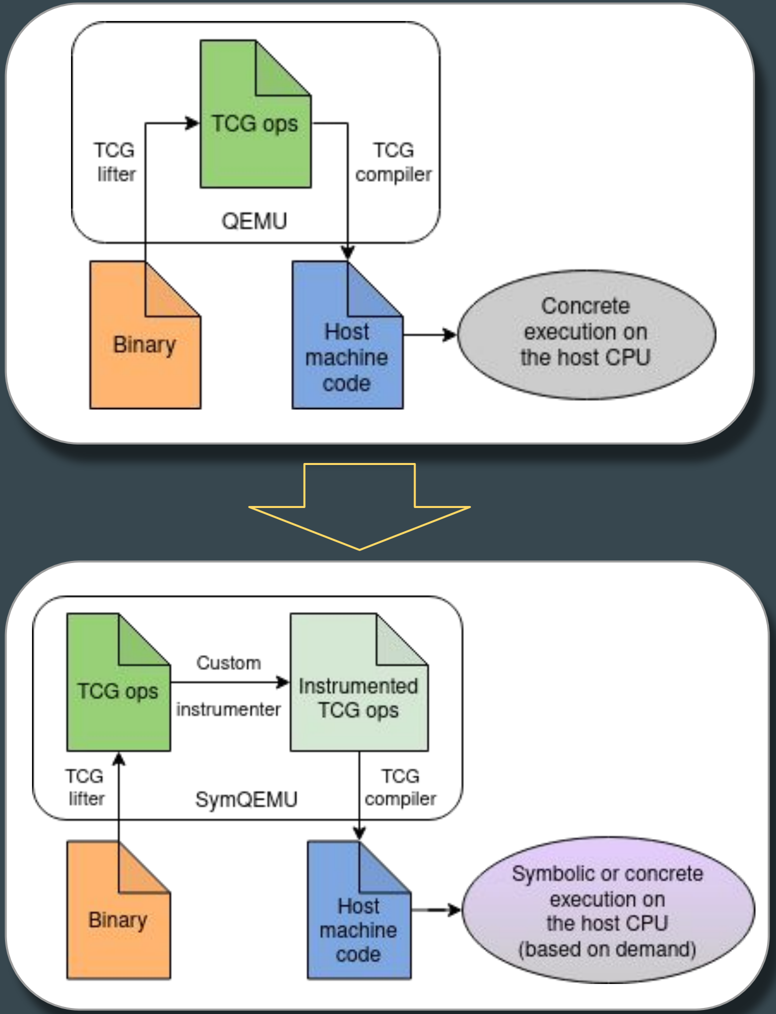
SymQEMU

Design and implementation

- QEMU is reliable and flexible
- Compilation-based symbolic execution is fast
- Approach: insert symbolic handling during binary translation

SymQEMU: Implementation

- Modified QEMU
  - Insert symbolic handling during binary translation (~2,000 lines of C code)
  - Symbolic semantics at the level of TCG ops
- Simple implementation
  - Small instruction set
  - Backend reused from SymCC (i.e., QSYM)
- Flexibility (inherited from QEMU)
  - Support AArch64 with 17 lines of code
- High performance (see next slides)



Evaluation

Three sets of experiments:

1. Google FuzzBench
2. Whole-program analysis
3. Benchmark comparison

FuzzBench: Summary

- Google FuzzBench: evaluation service for fuzzers
  - Tests fuzzers on open-source targets
  - 12 fuzzers, 21 targets, 24 hours, 15 iterations (~10 CPU core years)
  - Experiments performed by Google, resulting in a detailed report
    (special thanks to Google's Abhishek Arya, Jonathan Metzman and Laurent Simon)
- SymQEMU
  - Hybrid fuzzing with AFL: one AFL process in distributed mode, one SymQEMU process, exchanging new inputs between the two (like SymCC and QSYM evaluations)
  - Second-highest score overall (without using source code)
  - Outperformed all others on 3 out of 21 targets
  - Better than pure AFL on 14 out of 21

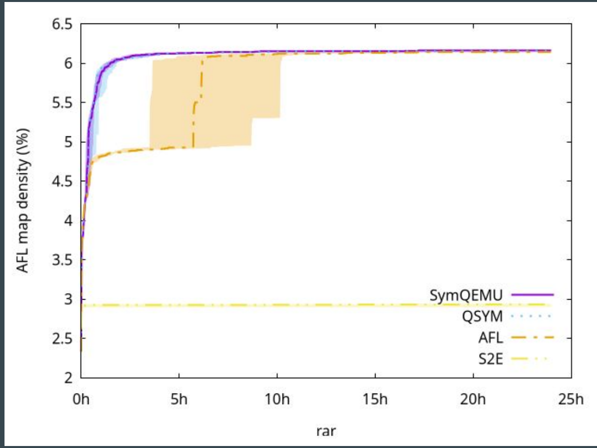
Full report available at http://s3.eurecom.fr/tools/symbolic_execution/symqemu.html

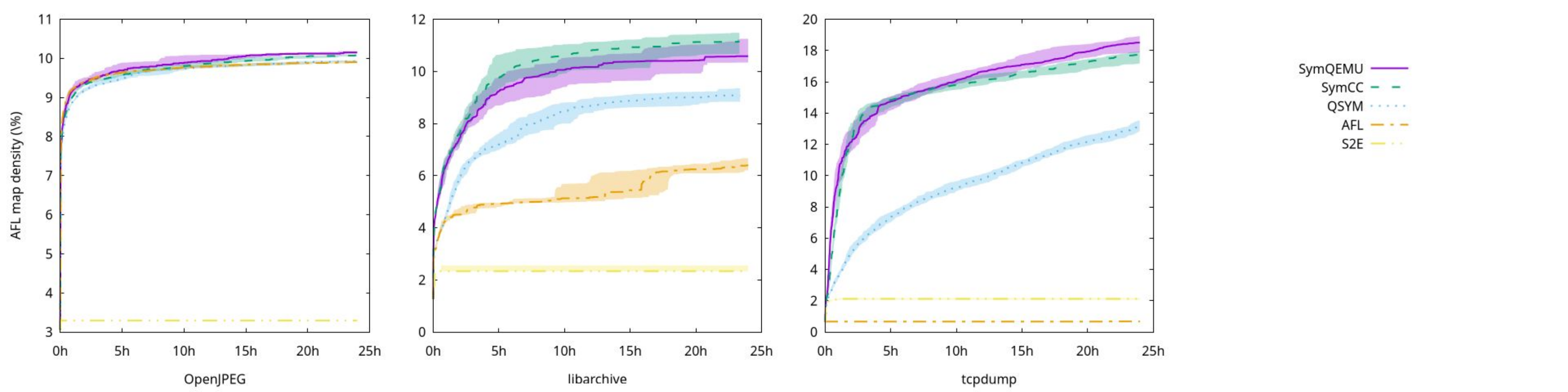
Whole-program analysis: Setup

- Targets
  - Open source: OpenJPEG, libarchive, tcpdump (like SymCC evaluation)
  - Closed source: rar (freely available, friendly license)
- Systems under test
  - SymQEMU, QSYM, SymCC (open-source targets only): hybrid fuzzing with AFL
  - S2E: symbolic exploration with default search strategy
  - Pure AFL
  - 3 CPU cores for each configuration
- Intel Xeon Platinum 8260 CPU with 2GB of RAM *per core*
  - See the paper for fineprint regarding S2E
- 24 hours, 30 iterations (~5 CPU core years)

Whole-program analysis: Results

- SymQEMU significantly outperforms QSYM, S2E and pure AFL
- Performance comparable with SymCC (but without using source code)



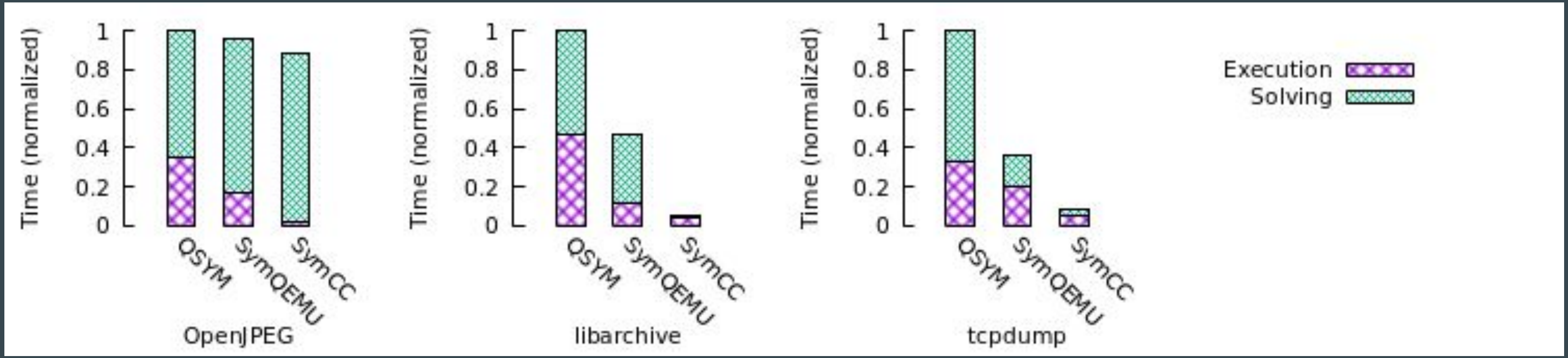


Benchmark experiments: Setup

- Goal
  - Investigate performance differences in a more controlled environment
- Methodology
  - Concolic execution of fixed paths
  - OpenJPEG, tcpdump, libarchive
  - 1,000 randomly selected test cases each (generated during whole-program analysis)
  - Execute in SymQEMU, QSYM and SymCC
  - Measure time spent in execution and SMT solving, respectively

Benchmark experiments: Results

- SymQEMU executes faster than QSYM, closer to SymCC
- Side note: SymCC's queries are the easiest to solve
  - See discussion in the paper



Compilation-based symbolic execution works on binaries and yields a highly flexible system.

SymQEMU inserts symbolic handling into binaries during dynamic binary translation
Significantly faster than state of the art, performance comparable with source-based SymCC
Works on closed-source software

Thank you!

sebastian.poeplau@eurecom.fr
aurelien.francillon@eurecom.fr

https://github.com/eurecom-s3/symqemu
(code, docs, evaluation details)