HASE
Hardware-Assisted Symbolic Execution

Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, Baris Kasikci

KLEE Symposium, 2018
Motivation

- Reproducing bugs that occur in production is hard
  - To fix bugs, developers have to reproduce them to understand the root cause

- Developers currently rely on
  - Stack traces
  - OS environment information
  - Coredumps
  - User reports
State-of-the-art: Replay debugging

- Limitation: High overheads for production
  - RR [Mozilla]: 1.2x - 1.4x
  - ODR [SOSP'09]: 1.6x - 3.5x
  - DoublePlay [ASPLOS'11]: 1.15x - 1.28x

How can we reduce the overhead to allow continuous logging?
System design

Phase #1: In-production monitoring
- Monitoring
- Program (E.g. MySQL)
- Processor trace

Phase #2: Replay debugging
- Symbolic execution
- Control flow + data constraints
- Replay debugging interface

Control flow + memory snapshot
- Coredump

Program interface
Background: Intel Processor Trace (PT)

- Since Broadwell generation (2014)
- Records full instruction history with low overhead (3\%)*
- Major limitation: **High log bandwidth** (200MB/s - 2GB/s)
- *Ringbuffer mode* - only keep last X instructions
- E.g. Firefox playing a YouTube video:
  - 5MB trace buffer ~ 192,000 branches ~ 3.3ms

*FlowGuard [HPCA '17]
Background: Intel Processor Trace (PT) - bonus

- 1 bit per conditional jump
- (optional) time stamps
- Address filtering
- Full system trace possible
- Easy to use:
  - `$ perf record -e intel_pt// program`
- Use cases:
  - Reconstruct every instruction (not just sampling)
  - Very accurate profiling
  - Code coverage (for fuzzing)
Background: Coredump

Snapshot of mutable virtual memory

... Stack Heap BSS Data User program text

Coredump

CPU Registers

Process context (uid,pid ...)

User program text
Phase #1: In-production monitoring

Program e.g. MySQL

Crash or failure

Monitoring module

Error report

Core dump & processor trace snapshot

Operating system

Intel PT
Phase #2: Replay debugging

Error report → Symbolic Engine → Control flow + data constraints → Replay debugging interface

- Without user interaction: Pre-compute/pre-constrain symbolic states
- Interactive: Pull serialized states & replay debugging
HASE frontend

- Uses debug symbols
- Evaluate expressions
- Print backtrace
- Timeslider

- Angr
- x86 machine code

Angr

x86 machine code
HASE’s symbolic execution

- **Uses core dump**
  - Simplifies constraints by using concrete values from the core dump for the final state

- **Follows single path**
  - Avoids path explosion by following the processor trace snapshot

- **Lazy**
  - Does not compute all memory values, only those requested by the developer

- **Consistent**
  - Concrete values computed show to the developer are added as constraints to the session
Open challenges

● **Comprehensibility:**
  ○ Generate data values that help programmer to understand the problem

● **System model:**
  ○ Idea #1: Extend Angr’s system model
  ○ Idea #2: Full system tracing, symbolic or concrete execution of kernel code

● **Multi-threading:**
  ○ Processor trace has optional timestamps for partial ordering
Summary

● **Motivation:** Reproducing production bugs is difficult
  ○ Existing record/replay systems have high overheads

● **HASE:**
  ○ Replay debugging tool with low overhead
  ○ Combines symbolic execution and Intel PT
  ○ Operates on unmodified binary application code and kernel

● **Project page:** [https://github.com/hase-project/hase](https://github.com/hase-project/hase)