Scaling symbolic evaluation for automated verification of systems code with Serval

Luke Nelson¹, James Bornholt¹, Ronghui Gu², Andrew Baumann³, Emina Torlak¹, Xi Wang¹
¹University of Washington, ²Columbia University, ³Microsoft Research
Eliminating bugs with formal verification

Process

Process

Process

OS Kernel / security monitor

seL4 (SOSP’09)
Iron clad Apps (OSDI’14)
FSCQ (SOSP’15)
CertiKOS (PLDI’16)
Komodo (SOSP’17)
Eliminating bugs with formal verification

- **Strong correctness guarantees**
- **Require manual proofs**
- **CertiKOS 200k lines of proof**
- **Multiple person-years**

- seL4 (SOSP’09)
- Ironclad Apps (OSDI’14)
- FSCQ (SOSP’15)
- CertiKOS (PLDI’16)
- Komodo (SOSP’17)
Prior work: automated (push-button) verification

- No proofs on implementation
- Requires bounded implementation
- Restricts specification

Example: Hyperkernel (SOSP’17)
Challenges

How to lower effort of writing automated verifiers?

How to find and fix performance bottlenecks?

How to retrofit to existing systems?
Contributions

• Serval: a framework for writing automated verifiers
  • RISC-V, x86-32, LLVM, BPF
  • Scaling via symbolic optimizations
• Experience
  • Retrofitted CertiKOS and Komodo for Serval
  • Found 15 new bugs in Linux BPF JIT
Contributions

- Serval: a framework for writing automated verifiers
  - RISC-V, x86-32, LLVM, BPF
  - Scaling via symbolic optimizations

Experience

- Retrofitted CertiKOS and Komodo for Serval
- Found 15 new bugs in Linux BPF JIT

no guarantees on concurrency or side channels
Verifying a system with Serval

System specification
RISC-V instructions
RISC-V verifier
Serval
Rosette
SMT solver
Verifying a system with Serval

Serval

Rosette

Z3

System specification

RISC-V instructions

RISC-V verifier

SMT solver
Verifying a system with Serval

System specification

RISC-V instructions

RISC-V verifier

Serval

Rosette

SMT solver

Z3
Verifying a system with Serval

System specification

RISC-V instructions

RISC-V verifier

Serval

Rosette

SMT solver
Verifying a system with Serval

System specification

RISC-V instructions

RISC-V verifier

Serval

Rosette

SMT solver
Verifying a system with Serval

- System specification
- RISC-V instructions
- RISC-V verifier
- Serval
- Rosette
- SMT solver
Example: proving refinement for sign

\[
\text{(define (sign x)}
\text{  (cond)}
\text{    [(negative? x) -1]}
\text{    [(positive? x) 1]}
\text{    [(zero? x) 0]]})
\]

RISC-V verifier

Serval
Verifier = interpreter + symbolic optimization

1. Write a verifier as interpreter
2. Symbolic profiling to find bottleneck
3. Apply symbolic optimizations

✔
Verifier [1/3]: writing an interpreter

(struct cpu (pc regs ...) #:mutable)

(define (interpret c program)
  (define pc (cpu-pc c))
  (define insn (fetch pc program))
  (match insn
    [('li rd imm)
      (set-cpu-pc! c (+ 1 pc))
      (set-cpu-reg! c rd imm)]
    [('bnez rs imm)
      (if (! (= (cpu-reg c rs) 0))
        (set-cpu-pc! c imm)
        (set-cpu-pc! c (+ 1 pc))))
...
))
Verifier [1/3]: writing an interpreter

(struct cpu (pc regs ...) #:mutable)

(define (interpret c program)
  (define pc (cpu–pc c))
  (define insn (fetch pc program))
  (match insn
    [(‘li rd imm)
      (set-cpu–pc! c (+ 1 pc))
      (set-cpu–reg! c rd imm)]
    [(‘bnez rs imm)
      (if (! (= (cpu–reg c rs) 0))
        (set-cpu–pc! c imm)
        (set-cpu–pc! c (+ 1 pc)))]
    ...))
Verifier [1/3]: writing an interpreter

(struct cpu (pc regs ...) #:mutable)

(define (interpret c program)
  (define pc (cpu-pc c))
  (define insn (fetch pc program))
  (match insn
   [('li rd imm)
    (set-cpu-pc! c (+ 1 pc))
    (set-cpu-reg! c rd imm)]
   [('bnez rs imm)
    (if (! (= (cpu-reg c rs) 0))
     (set-cpu-pc! c imm)
     (set-cpu-pc! c (+ 1 pc)))]
   ...))

(define (fetch pc program)
  ...)

...
Verifier [1/3]: writing an interpreter

(struct cpu (pc regs ...) #:mutable)

(define (interpret c program)
  (define pc (cpu-pc c))
  (define insn (fetch pc program))
  (match insn
    [(l): define (li rd imm)
      (set-cpu-pc! c (+ 1 pc))
      (set-cpu-reg! c rd imm)]
    [(bnez): define (bnez rs imm)
      (if (! (= (cpu-reg c rs) 0))
        (set-cpu-pc! c imm)
        (set-cpu-pc! c (+ 1 pc)))]
    ...
))
Verifier [1/3]: writing an interpreter

```
(struct cpu (pc regs ...) #:mutable)

(define (interpret c program)
  (define pc (cpu-pc c))
  (define insn (fetch pc program))
  (match insn
    [("li" rd imm)
      (set-cpu-pc! c (+ 1 pc))
      (set-cpu-reg! c rd imm)]
    [("bnez" rs imm)
      (if (! (= (cpu-reg c rs) 0))
        (set-cpu-pc! c imm)
        (set-cpu-pc! c (+ 1 pc)))]
    ...))
```

- Easy to write
- Reuse CPU test suite
Verifier [2/3]: identifying bottlenecks in symbolic evaluation

(define (sign x)
  (cond
    [(negative? x) -1]
    [(positive? x) 1]
    [(zero? x) 0])))

RISC-V verifier

Serval
Verifier [2/3]: identifying bottlenecks in symbolic evaluation

(define (sign x)
  (cond
    [(negative? x) -1]
    [(positive? x) 1]
    [(zero? x) 0])))
Verifier [2/3]: identifying bottlenecks in symbolic evaluation

![Call Stack Diagram]

**Function** | Score | Time (ms) | Term Count | Unused Terms | Union Size | Merge Cases
---|---|---|---|---|---|---
execute run.rkt:42 3 calls | 3.7 | 6 | 13 | 13 | 0 | 22
@vector-ref 1 call | 2.0 | 3 | 0 | 0 | 6 | 14
Verifier [2/3]: identifying bottlenecks in symbolic evaluation

- **Function**: fetch
- **Term Count**: 2
- **Unused Terms**: 13
- **Union Size**: 13
- **Merge Cases**: 0

- **Bug For**: bug-for...
- **Bug On**: bug-on8
- **Bug On**: bug-on

- **Fetcher**: fetch
- **Execute**: execute

- **Signatures**: 22
- **Collapse Solver Time**: 1
- **Aggregate**: 1

- **Caller Context**: 1

- **Execute**: run.rkt:42
- **Calls**: 3

- **Interpret**: run.rkt:18

- **@vector-ref**: run.rkt:25
- **1 Call**: 2.0
Verifier [2/3]: identifying bottlenecks in symbolic evaluation

(struct cpu (pc regs) #:mutable)

(define (interpret c program)
  (define pc (cpu-pc c))
  (define insn (fetch pc program))
  (match insn
    [(['li rd imm)
      (set-cpu-pc! c (+ 1 pc))
      (set-cpu-reg! c rd imm)]
    [(['bnez rs imm)
      (if (! (= (cpu-reg c rs) 0))
        (set-cpu-pc! c imm)
        (set-cpu-pc! c (+ 1 pc)))]
    ...
  ))

0: sltz a1 a0
1: bnez a1 4
2: sgtz a0 a0
3: ret
4: li a0 -1
5: ret
Merge states to avoid path explosion

0: sltz a1 a0
1: bnez a1 4
2: sgtz a0 a0
3: ret
4: li a0 -1
5: ret

PC → 0
a0 → X
a1 → Y

PC → 1
a0 → X
a1 → 1

¬(X < 0)

PC → 1
a0 → X
a1 → 0

PC → 1
a0 → X
a1 → if(X < 0, 1, 0)
Conditional jump

PC → 1
a0 → X
a1 → if(X < 0, 1, 0)

PC → if(X < 0, 4, 2)
a0 → X
a1 → if(X < 0, 1, 0)

0: sltz a1 a0
1: bne a1 4
2: sgtz a0 a0
3: ret
4: li a0 -1
5: ret

Bottleneck: state explosion due to symbolic PC
Bottleneck: state explosion due to symbolic PC

PC \rightarrow \text{if}(\ldots)
\begin{align*}
a0 &\rightarrow X \\
a1 &\rightarrow \text{if}(\ldots)
\end{align*}

Conditional jump

0: sltz a1 a0
1: bnez a1 4
2: sgtz a0 a0
3: ret
4: li a0 -1
5: ret
Verifier [3/3]: Repairing with symbolic optimizations

- Symbolic optimization:
  - “Peephole” optimization on symbolic state
  - Fine-tune symbolic evaluation
  - Use domain knowledge
- Serval provides set of symbolic optimizations for verifiers
Verifier [3/3]: Repairing with symbolic optimizations

- Match on symbolic structure of PC
- Evaluate separately using each concrete PC value
- Merge states afterwards
Verifier [3/3]: Repairing with symbolic optimizations

PC → if(X < 0, 4, 2)
a0 → X
a1 → if(...)

PC → 0
PC → 1
PC → 2
PC → 3
PC → 4
PC → 5

PC → if(X < 0, 4, 2)
a0 → X
a1 → if(...)

split-pc

PC → 4
PC → 2
Verifier [3/3]: Repairing with symbolic optimizations

Domain knowledge:
- Split PC to avoid state explosion
- Merge other registers to avoid path explosion
Verifier summary

- Verifier = interpreter + symbolic optimizations
- Easy to test verifiers
- Systematic way to scale symbolic evaluation

Caveats:
- Symbolic profiling cannot identify expensive SMT operations
- Repair requires expertise
Implementation

- RISC-V verifier
- x86-32 verifier
- LLVM verifier
- BPF verifier

Serval

Rosette

SMT solver
Experience

• Can existing systems be retrofitted for Serval?

• Are Serval’s verifiers reusable?
Retrofitting previously verified security monitors

- Port CertiKOS (PLDI’16) and Komodo (SOSP’17) to RISC-V

- Retrofit the systems to automated verification

- Apply the RISC-V verifier to binary image

- Prove functional correctness and noninterference

- ≈4 weeks each
Retrofitting overview

Is the implementation free of unbounded loops?

Is the specification expressible in Serval?

System implementation

System specification
Example: retrofitting CertiKOS

- OS kernel providing strict isolation
- Physical memory quota, partitioned PIDs
- Security specification: noninterference
Example: retrofitting CertiKOS

• Implementation

  • Already free of unbounded loops

  • Tweak spawn to close two potential information leaks

• Specification

  • Noninterference using traces of unbounded length

  • Broken down into 3 properties of individual “actions”
Retrofitting summary

• Security monitors good fit for automated verification

• No unbounded loops

• No inductive data structures
Reusing verifiers to find bugs

- Combine RISC-V, x86-32, and BPF verifiers
- Found 15 bugs in the Linux kernel’s BPF JIT compiler
- Bug fixes and new tests upstreamed
Conclusion

• Writing automated verifiers using lifting
• A systematic method for scaling symbolic evaluation
• Retrofit Serval to verify existing systems

• For paper and more info:
  • https://serval.unsat.systems