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
**Goal:** eliminating bugs with formal verification

- **System implementation**
- **Functional specification**
- **Safety specification (e.g. noninterference)**

Refinement

- **OS Kernel / security monitor**
Using interactive / auto-active verification

- seL4 (SOSP’09)
- Ironclad Apps (OSDI’14)
- FSCQ (SOSP’15)
- CertiKOS (PLDI’16)
- Komodo (SOSP’17)

• Require manual proof annotations/tactics
• Expensive: CertiKOS 200k LOC proof
• Multiple person-years
This talk: automated (push-button) verification

- Trade-off: automation vs expressivity
- No proofs on implementation
- Requires bounded implementation
- Restricts spec to first-order logic
- Examples: Hyperkernel (SOSP’17), Nickel (OSDI’18)
This talk: automated (push-button) verification

How to write and maintain automated verifiers?

How to systematically fix verification bottlenecks?

How to retrofit to existing systems?
Contributions

- Serval: a framework for writing automated verifiers
- Lift interpreters into verifiers: RISC-V, BPF, x86-32, LLVM
- Symbolic optimization for repairing verification bottlenecks
- Experience with Serval
  - Retrofitted CertiKOS and Komodo for automated verification
  - Found 18 new bugs in Linux BPF JIT and Keystone
- Assumption: no guarantees on concurrency or side channels
Verification stack

**Serval:**
Specification library, symbolic optimizations, machine code support

**Rosette:**
Symbolic evaluation, symbolic profiling, symbolic reflection

**SMT solver:**
Constraint solving, counterexample generation
Verification stack

**Z3**
Constraint solving, counterexample generation

**Serval**
Specification library, symbolic optimizations, machine code support

**Rosette**
Symbolic evaluation, symbolic profiling, symbolic reflection

**SMT solver**
Constraint solving, counterexample generation
Verification stack

System specification
RISC-V instructions
\[\text{RISC-V verifier}\]
\[\text{Serval:}\]
Specification library, symbolic optimizations, machine code support
\[\text{Rosette:}\]
Symbolic evaluation, symbolic profiling, symbolic reflection
\[\text{SMT solver:}\]
Constraint solving, counterexample generation
\[\text{x86-32 instructions}\]
\[\text{x86-32 verifier}\]
\[\text{LLVM instructions}\]
\[\text{LLVM verifier}\]
\[\text{BPF instructions}\]
\[\text{BPF verifier}\]
Verification stack

- **System specification**
- **RISC-V instructions**
- **x86-32 instructions**
- **LLVM instructions**
- **BPF instructions**

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

- **Serval:** Specification library, symbolic optimizations, machine code support
- **Rosette:** Symbolic evaluation, symbolic profiling, symbolic reflection
- **SMT solver:** Constraint solving, counterexample generation
Verification stack

System specification
RISC-V instructions
x86-32 instructions
LLVM instructions
BPF instructions

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

Serval:
Specification library, symbolic optimizations, machine code support

Rosette:
Symbolic evaluation, symbolic profiling, symbolic reflection

SMT solver:
Constraint solving, counterexample generation
Verification stack

- **System specification**
- **RISC-V instructions**
- **x86-32 instructions**
- **LLVM instructions**
- **BPF instructions**

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

- **Serval**: Specification library, symbolic optimizations, machine code support
- **Rosette**: Symbolic evaluation, symbolic profiling, symbolic reflection
- **SMT solver**: Constraint solving, counterexample generation
Verifier = interpreter + symbolic optimization

Write a verifier as an interpreter

Symbolic profiling to find bottleneck

Develop / apply symbolic optimizations

✔
Example: proving refinement for sign

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

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

Serval

Specification library

RISC-V verifier

0: sltz a0 a1
1: bnez a1 4
2: sgtz a0 a0
3: ret
4: li   a0 -1
5: ret
Verifier [1/3]: writing an interpreter

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

(define (interpret c program)
  (define pc (cpu-pc c))
  (define insn (fetch c 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 c 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 c 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 c 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 c 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)))]
    ...
))

• Natural to write
• Easy to audit
• Can 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])))

Specification library

RISC-V verifier

Serval

0: sltz a0 a1
1: bnez a1 4
2: sgtz a0 a0
3: ret
4: li  a0 -1
5: ret
Verifier [2/3]: identifying bottlenecks in symbolic evaluation

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

Serval

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

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

Profile for run.rkt generated 2019-10-18 13:19:20

Function | Score | Time (ms) | Term Count | Unused Terms | Union Size | Merge Cases
---|---|---|---|---|---|---
execute | 3.6 | 8 | 13 | 13 | 0 | 22
@vector-ref | 1.8 | 3 | 0 | 0 | 6 | 14
Bottleneck: state explosion due to symbolic PC

(struct cpu (pc regs) #:mutable)

(define (interpret c program)
  (define pc (cpu-pc c))
  (define insn (fetch c 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 a0 a1
1: bnez a1 4
2: sgtz a0 a0
3: ret
4: li a0 -1
5: ret
Bottleneck: state explosion due to symbolic PC

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

(define (int cpu)
  (define pccpu)
  (define insn fetch)
  (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)))]
    ...)

B0 : X < 0
B1 : ~v0
B2 : ite(v0, 1, 0)
B3 : ite(v0, 4, 2)
B4 : ite(v0, li, sgtz)
B5 : ite(v0, #f, 0)
B6 : ite(v0, ~1, #f)
B7 : ite(v0, 5, 3)
B8 : X > 0
B9 : ~v8
B10 : ite(v8, 1, 0)
B11 : ite(v0, ~1, v10)
B12 : sltz a0 a1
B13 : bnez a1 4
B14 : sgtz a0 a0
B15 : ret
B16 : li a0 -1
B17 : ret
```
Bottleneck: state explosion due to symbolic PC

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

(define (interpret c program)
  (define pc (cpu-pc c))
  (define insn (fetch c 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)))]
    ...
    ))
```

- Conditional jump

```
0: sltz a0 a1
1: bnez a1 4
2: sgtz a0 a0
3: ret
4: li a0 -1
5: ret
```
(struct cpu
  (pc regs))

#:mutable

(define (interpret c program)
  (define pc (cpu-pc c))
  (define insn (fetch c 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)))]
    ...
    )
)

Bottleneck: state explosion due to symbolic PC

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

v0: X<0
v1: ¬v0
v2: ite (v0, 1, 0)
v3: ite (v0, 4, 2)
v4: ite (v0, li, sg tz)
v5: ite (v0, #f, 0)
v6: ite (v0, 1, #f)
v7: ite (v0, 5, 3)
v8: X>0
v9: ¬v8
v10: ite (v8, 1, 0)
v11: ite (v0, v10, v2)

sltz a0 a1
bnez a1 4
sg tz a0 a0
ret
li a0 -1
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

(struct cpu (pc regs) #:mutable)

(define (interpret c program)
  (define pc (cpu-pc c))
  (define insn (fetch c program))
  (match insn ...

Developer uses symbolic optimization

(struct cpu (pc regs) #:mutable)

(define (interpret c program)
  + (serval:split-pc [cpu pc] c
  (define insn (fetch c program))
  (match insn ...

...))

...))
Verifier [3/3]: Repairing with symbolic optimizations

3x fewer merges

\( \text{Verifier} \) [3/3]: Repairing with symbolic optimizations

\[ \begin{align*}
&v_0 : X < 0 \\
v_1 : \neg v_0 \\
v_2 : \text{ite}(v_0, 1, 0) \\
v_3 : \text{ite}(v_0, 4, 2) \\
v_4 : \text{ite}(v_0, \# 0) \\
v_5 : \text{ite}(v_0, -1, \# 0) \\
v_6 : \text{ite}(v_0, 5, 3) \\
v_7 : X > 0 \\
v_8 : \neg v_7 \\
v_9 : \text{ite}(v_0, 1, 0) \\
v_{10} : \text{ite}(v_0, -1, v_{10})
\end{align*} \]
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
Retrofitting previously verified security monitors

- Port CertiKOS (PLDI’16) and Komodo (SOSP’17) to RISC-V
- Prove functional correctness and noninterference
Retrofitting overview

Is the implementation free of unbounded loops?

Is the specification expressible in Serval?

System implementation

System specification

RISC-V verifier

Serval
CertiKOS (PLDI’16)

• OS kernel providing strict isolation

• Physical memory quota, partitioned PIDs
Retrofitting: implementation

- CertiKOS interface already expressible without unbounded loops

- Tweak spawn system call to close potential information leaks

- Did not account for memory consumed by ELF loading

- Leaked number of children
Retrofitting: specification

• CertiKOS specifies noninterference using traces of unbounded length
  • Broken down into 3 properties of individual “actions”
  • Local action, yield to another process, yield back
• We reuse the properties as our noninterference specification
• We also prove noninterference spec as in Nickel (OSDI’18)
Retrofitting summary

- Security monitors good fit for automated verification
  - No unbounded loops
  - No inductive data structures
- Verify binary images directly
  - Develop in standard languages (C / Asm)
  - No need to trust compiler / linker / etc.
15 new bugs found in Linux BPF JIT

**Serval:**
Specification library, symbolic optimizations, machine code support
Conclusion

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

• Come to SOSP to learn more
• For paper and more info:
  • https://serval.unsat.systems