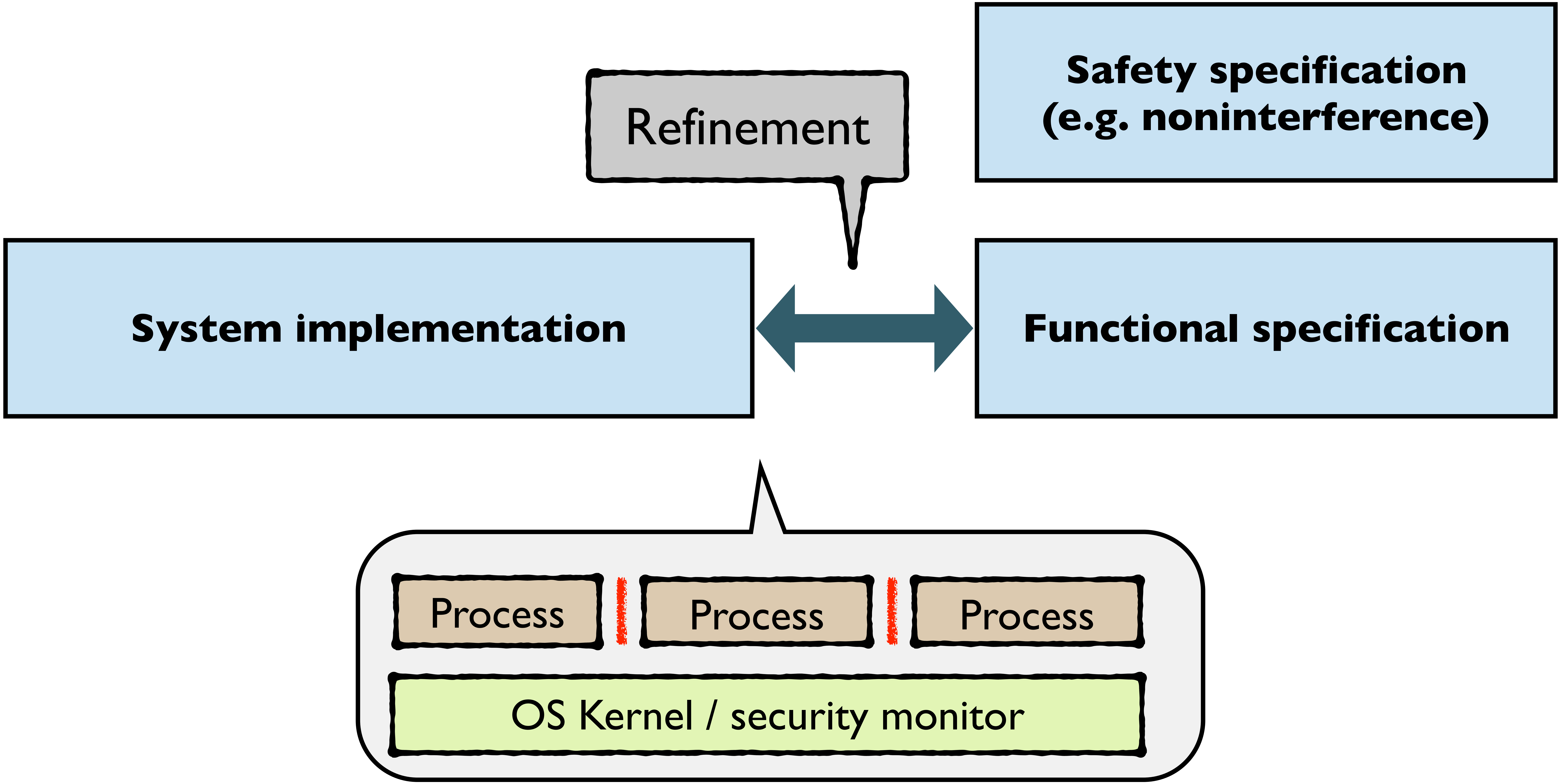
Scaling symbolic evaluation for automated verification of systems code with Serval

Luke Nelson[1], James Bornholt[1], Ronghui Gu[2], Andrew Baumann[3], Emina Torlak[1], Xi Wang[1]
[1]University of Washington, [2]Columbia University, [3]Microsoft Research

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

UNSAT

Goal: eliminating bugs with formal verification



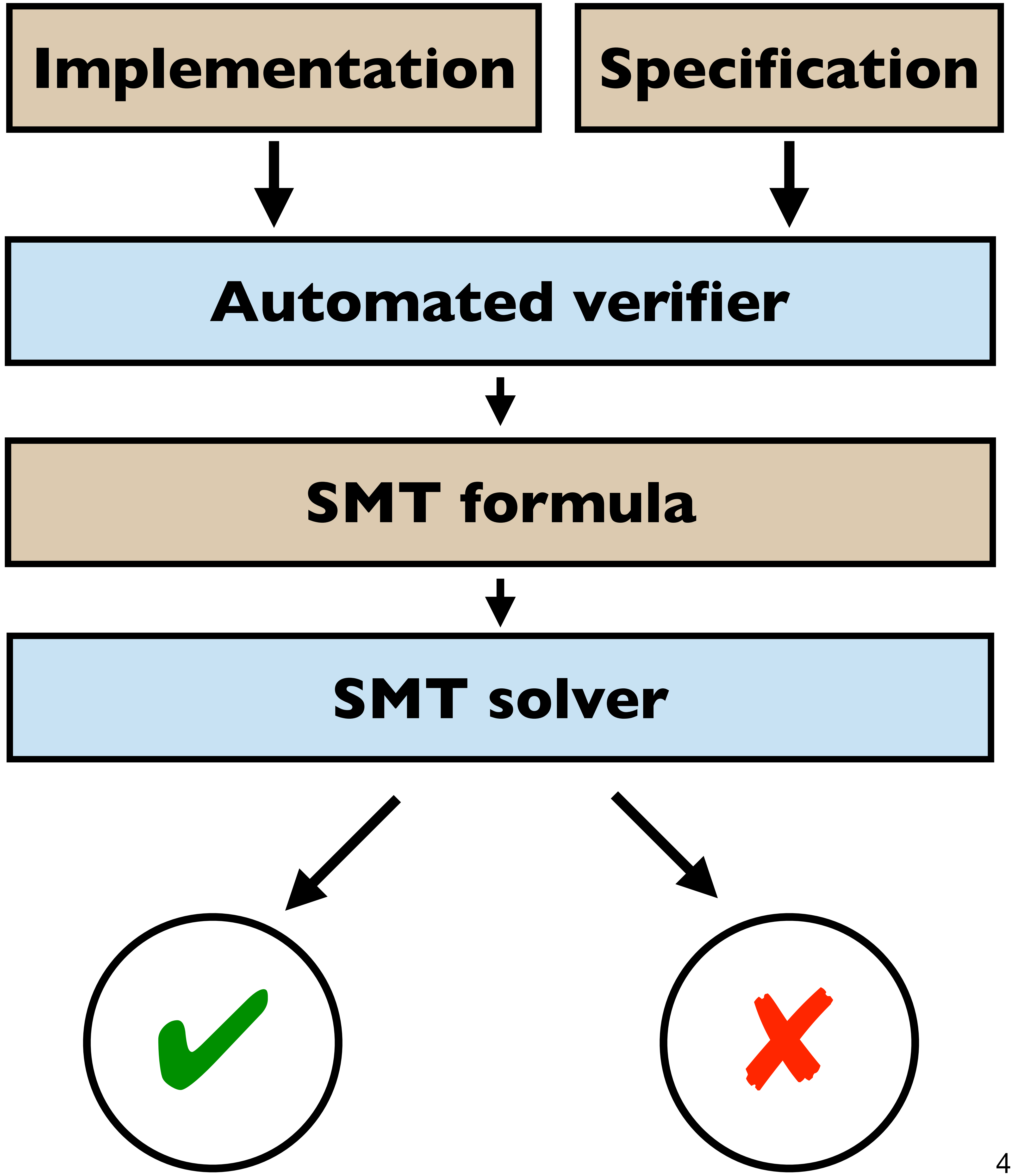
Using interactive / auto-active verification

- 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)

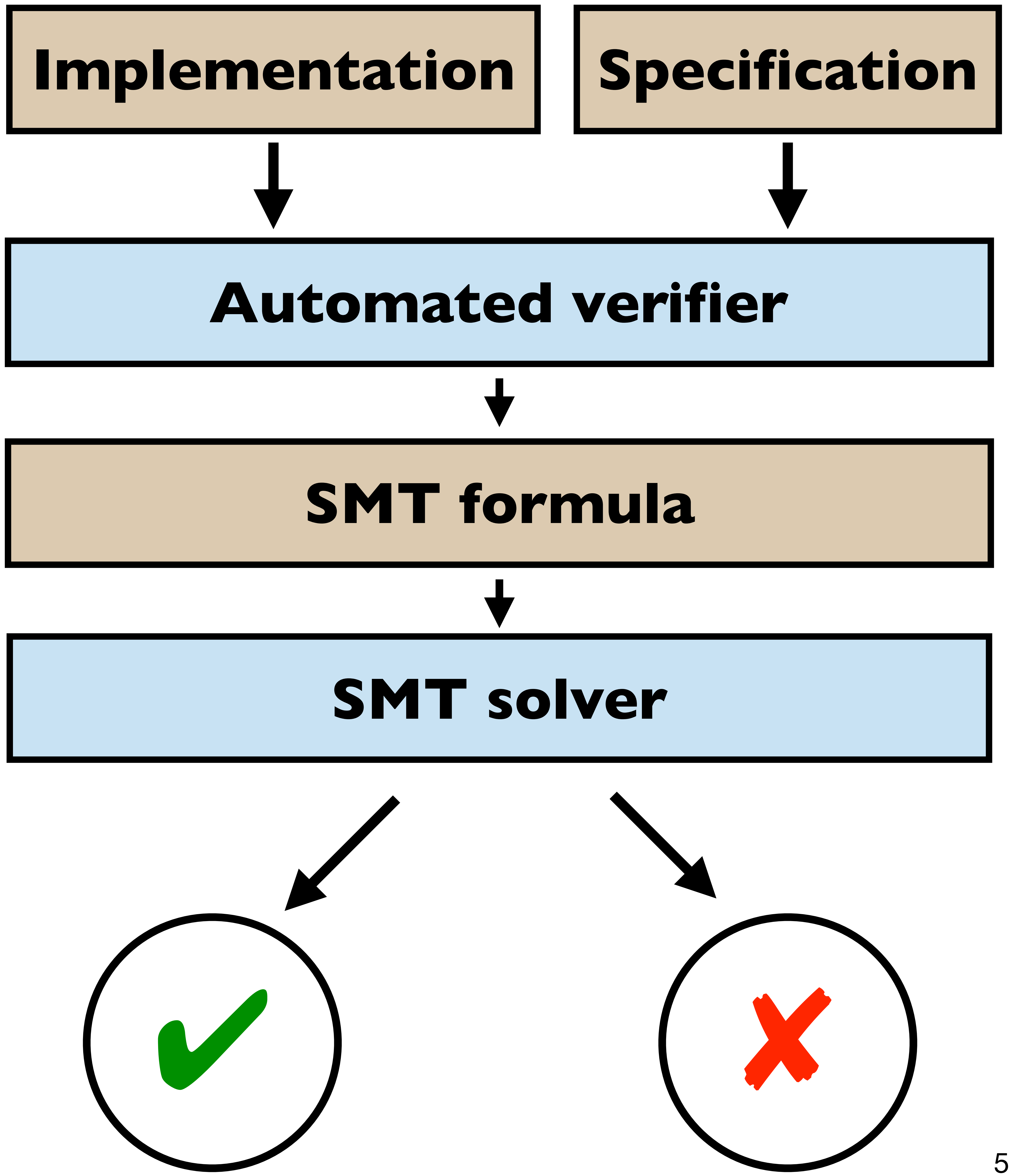


This talk: automated (push-button) verification

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
- <u>Assumption</u>: no guarantees on concurrency or side channels

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

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

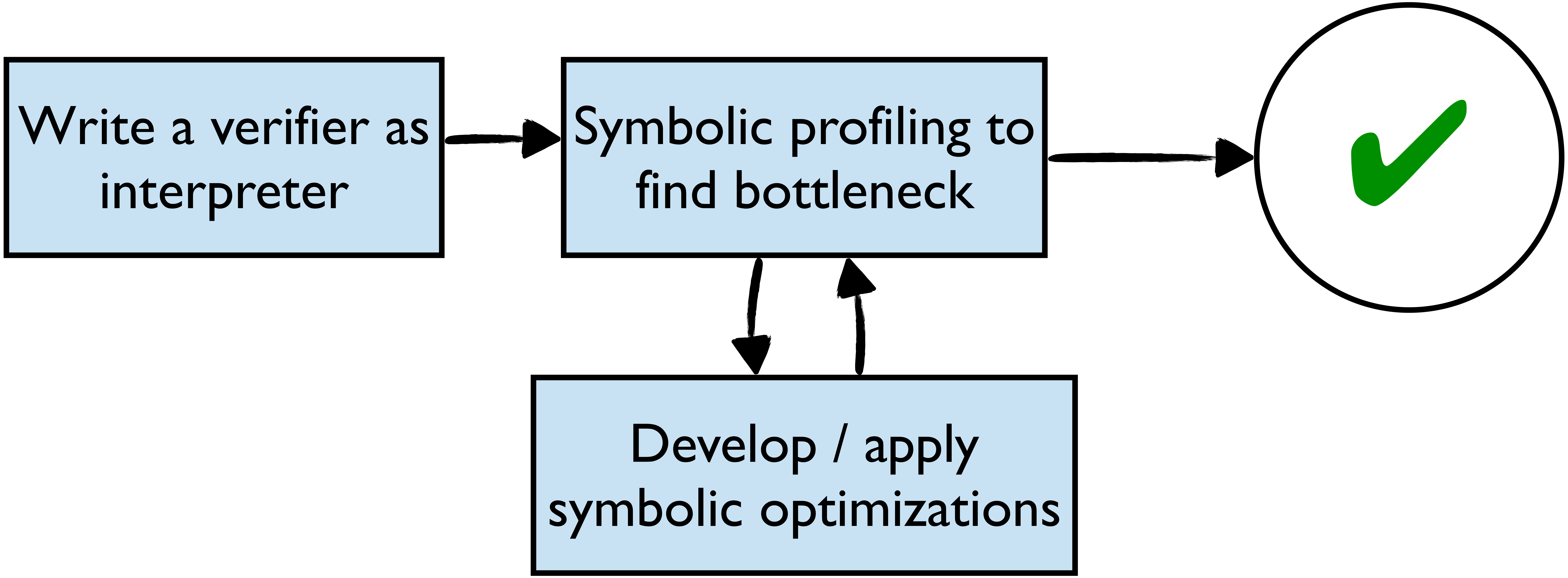
Serval:
Specification library, symbolic optimizations, machine code support

Rosette:
Symbolic evaluation, symbolic profiling, symbolic reflection

Z3

SMT solver:
Constraint solving, counterexample generation

Verifier = interpreter + symbolic optimization



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
```

Specification library

RISC-V verifier

Serval

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)))]
    ...))
```

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]))
```

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

Specification library

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]))
```

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

Slow / Timeout

Specification library

RISC-V verifier

Serval

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

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

☑ Aggregate ? Caller Context: 1 ? ☐ Collapse solver time ? ☐ Signatures ? [More]

Function	Score	Time (ms)	Term Count	Unused Terms	Union Size	Merge Cases
execute run.rkt:41 3 calls └interpret run.rkt:10	3.6	8	13	13	0	22
@vector-ref 1 call └fetch run.rkt:24	1.8	3	0	0	6	14

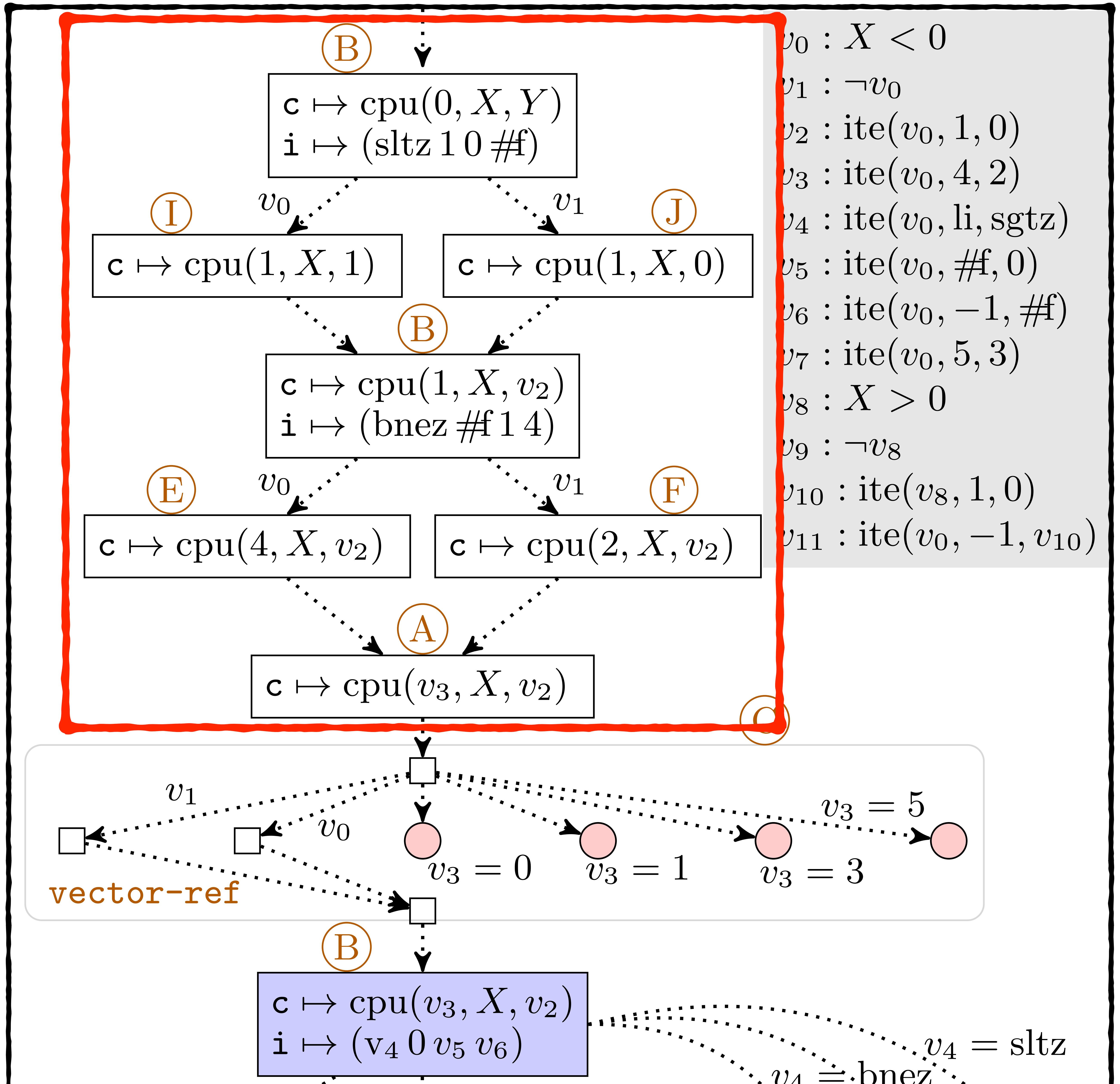
Bottleneck: state explosion due to symbolic PC

```
(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

(define (int
  (define pc
  (define in
  (match ins
    [('li rd
       (set-
       (set-
    [('bnez
       (if (
           (
           (
  ...))
```




```
sltz a0 a1
bnez a1 4
sgtz a0 a0
ret
li   a0 -1
ret
```

Bottleneck: state explosion due to symbolic PC

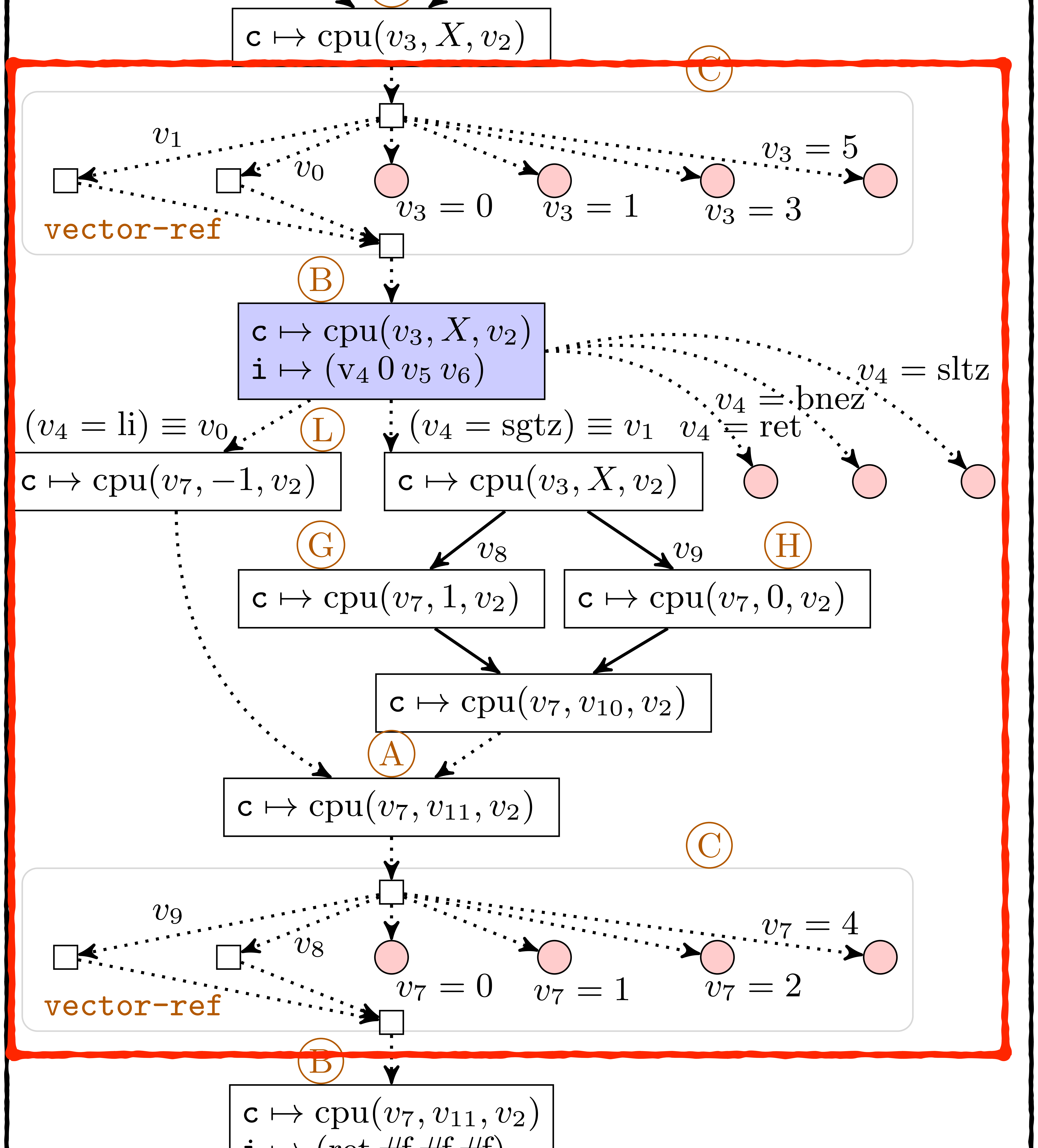
```
(struct cpu (pc regs) #:mutable)

(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
```

Bottleneck: s

```
(struct cpu

(define (int
  (define pc
  (define in
  (match ins
    [('li rd
       (set-
       (set-
    [('bnez
       (if (
           (
           (
  ...))
```




```
sltz a0 a1
bnez a1 4
sgtz 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

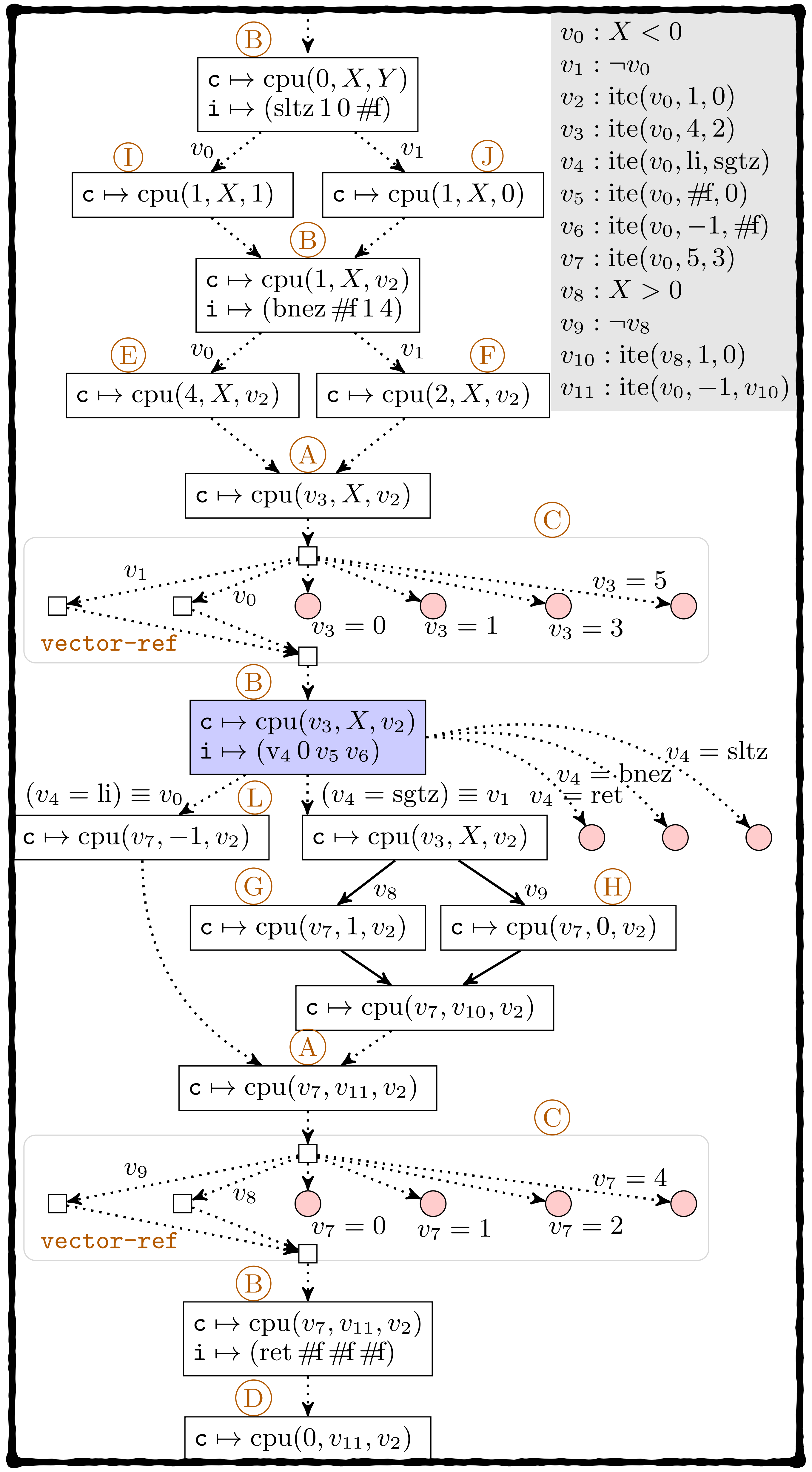
Verifier [3/3]: Repairing with symbolic optimizations

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

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

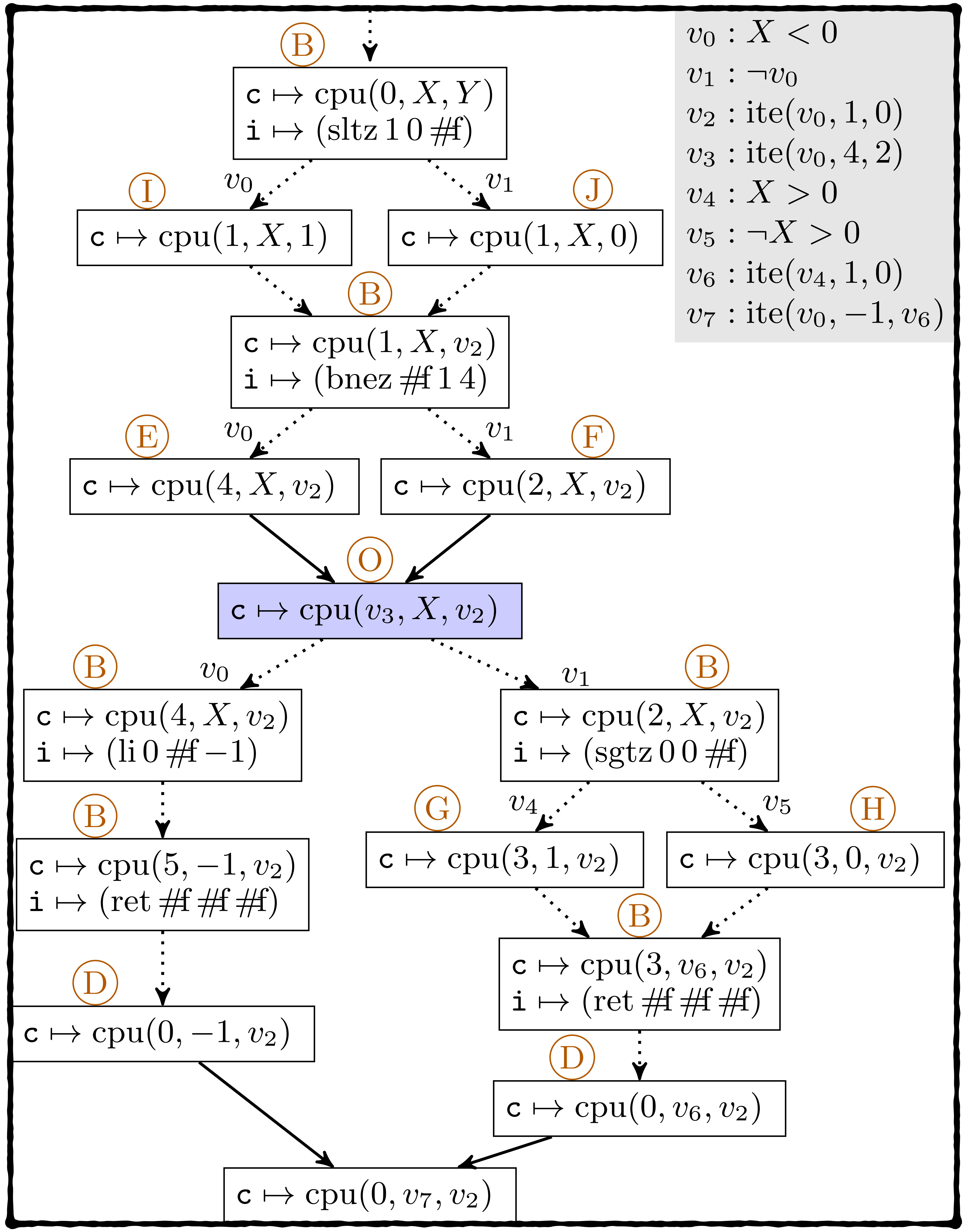
Developer uses symbolic optimization

```
(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



3× fewer merges



Verifier summary

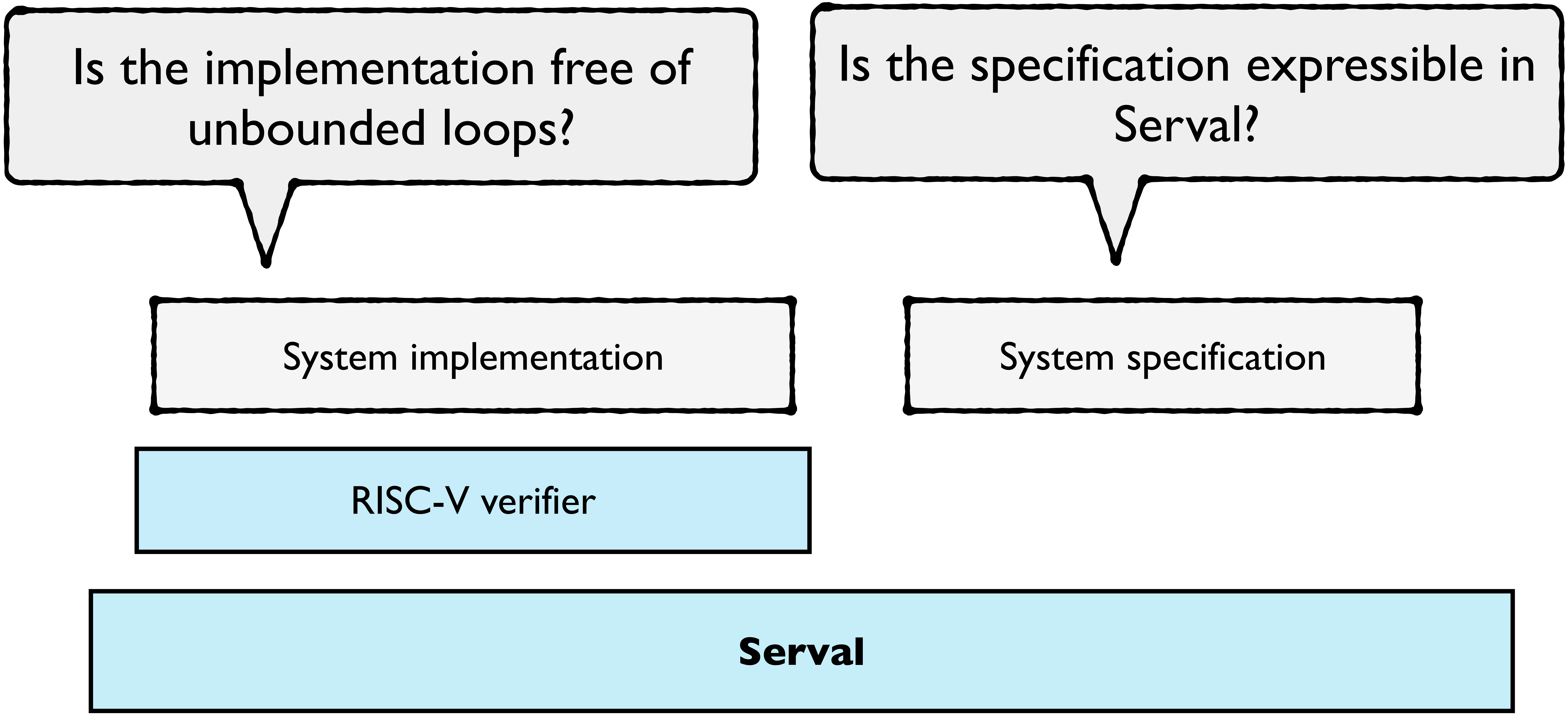
- Verifier = interpreter + symbolic optimizations
- Easy to test verifiers
- Systematic way to scale symbolic evaluation

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

Retrofitting previously verified security monitors

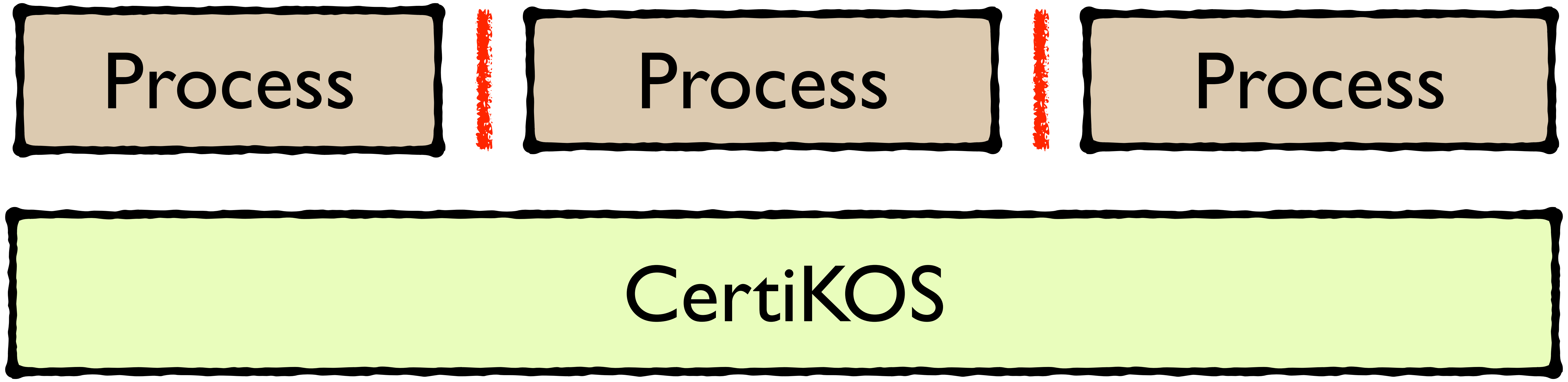
- Port CertiKOS (PLDI'16) and Komodo (SOSP'17) to RISC-V
- Prove functional correctness and noninterference

Retrofitting overview



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

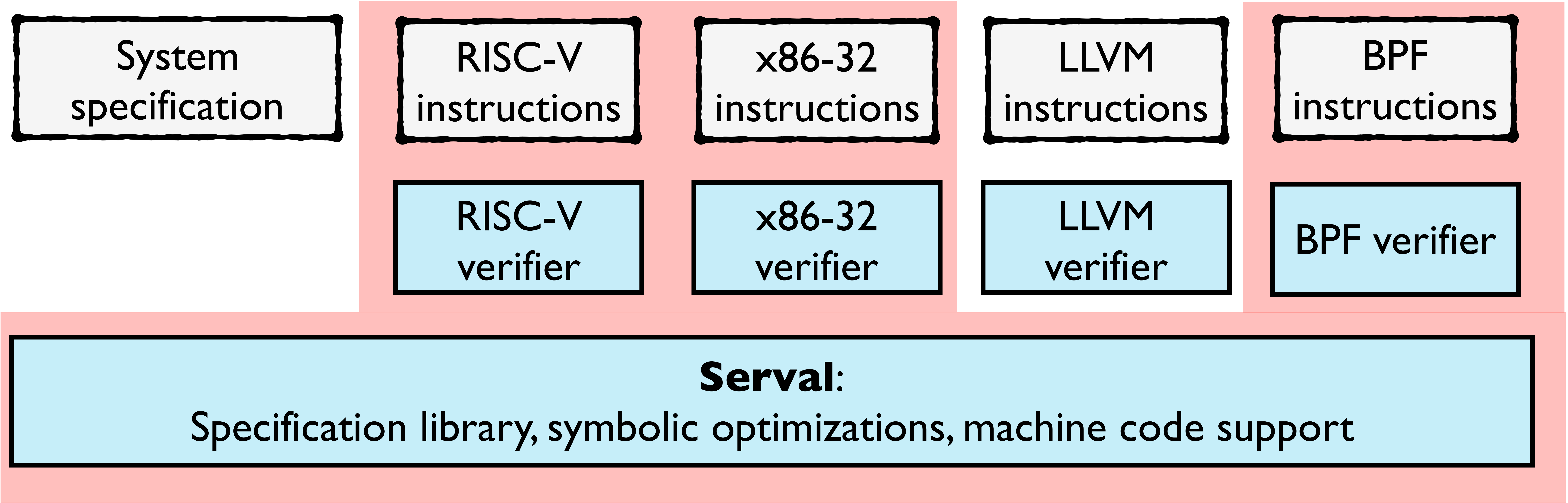
Retrofitting: specification

- 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



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