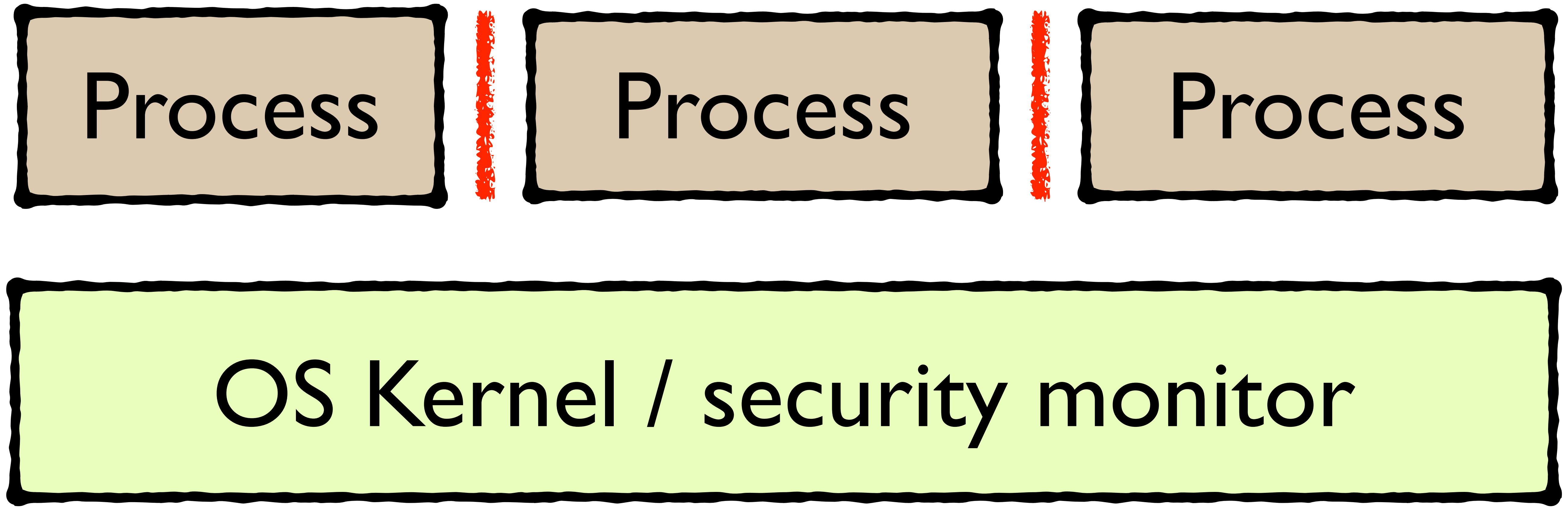
Scaling symbolic evaluation for automated verification of systems code with Serval

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

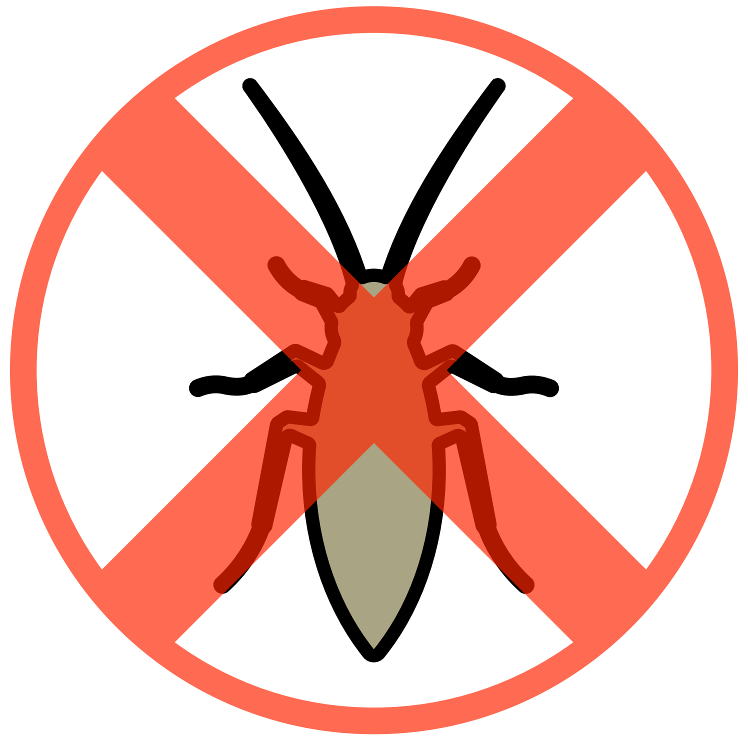
PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

UNSAT

Eliminating bugs with formal verification



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



Eliminating bugs with formal verification

Process

OS K

DI'14)

6)

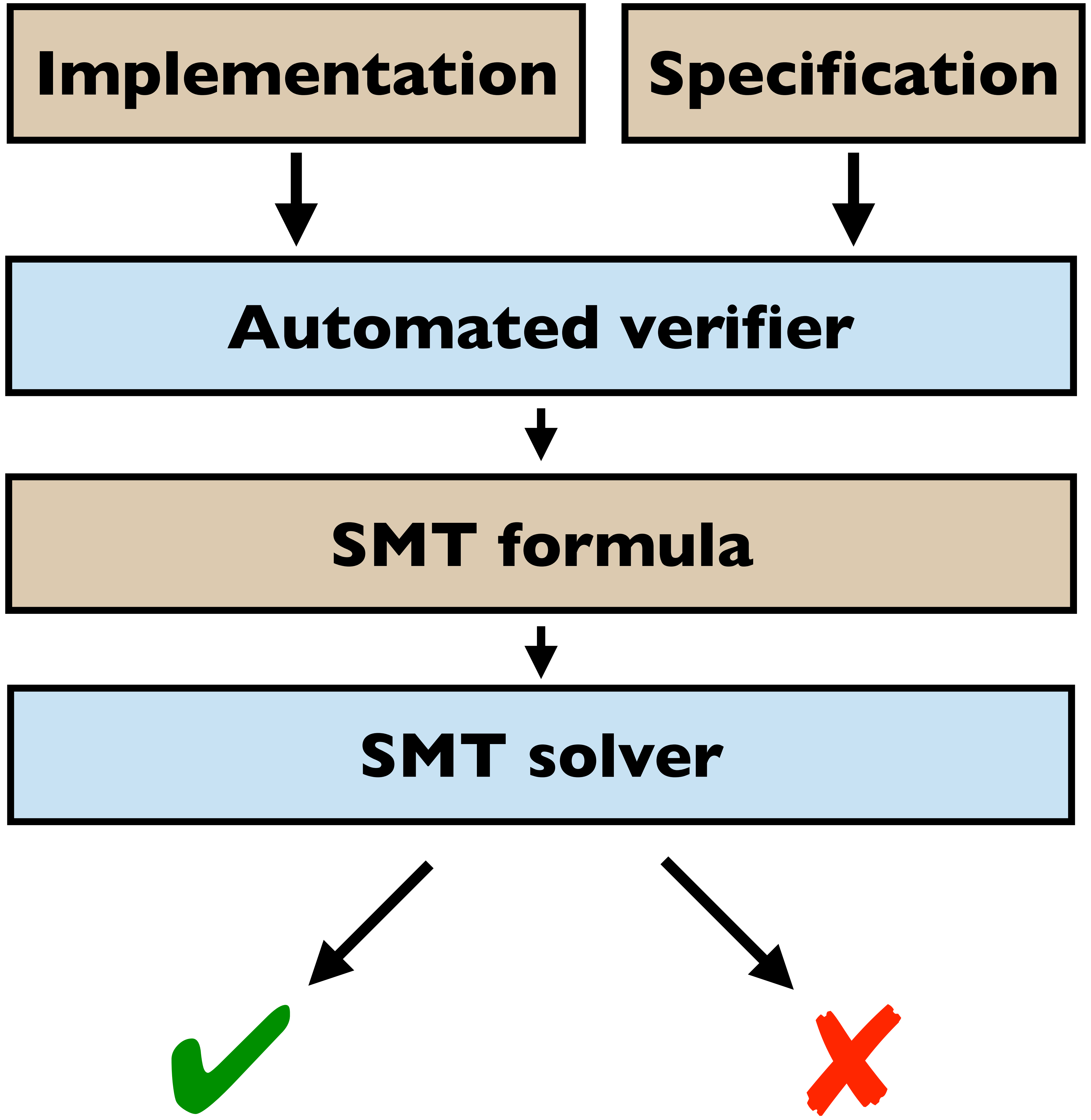
)

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

Prior work: automated (push-button) verification

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

Example: Hyperkernel (SOSP'17)

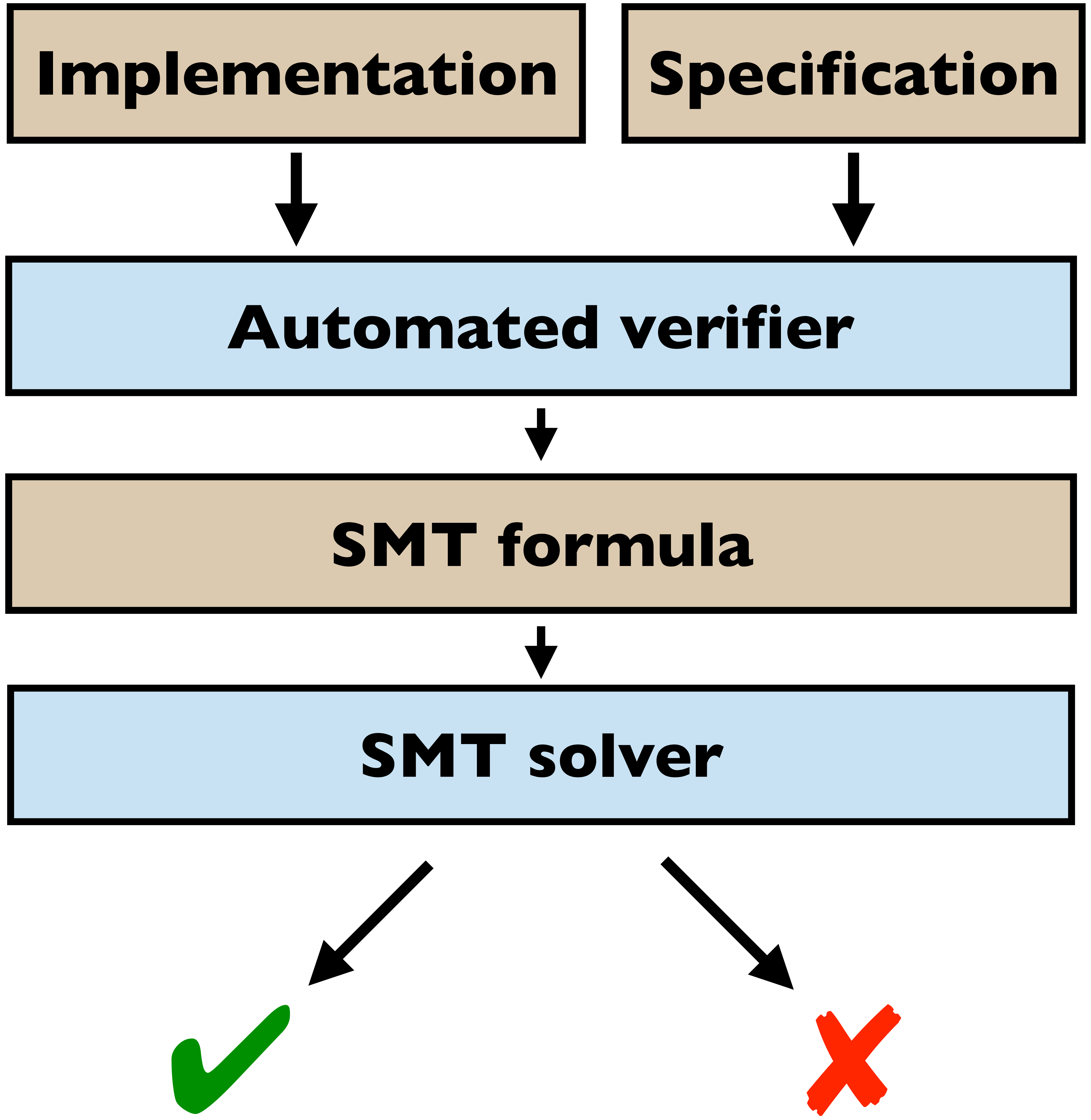


Challenges

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

System specification

RISC-V instructions

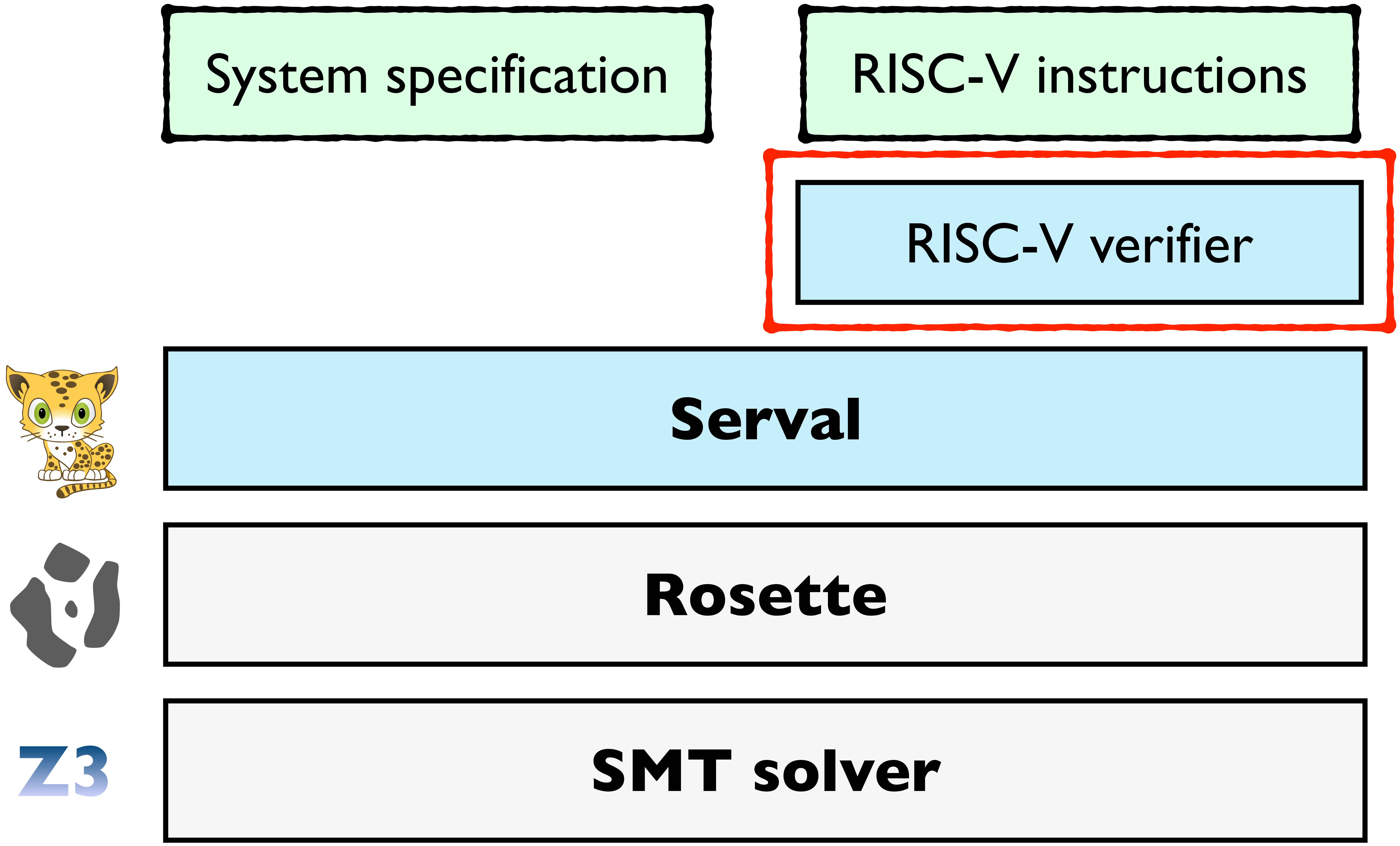
RISC-V verifier

Serval

Rosette

SMT solver

Verifying a system with Serval



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

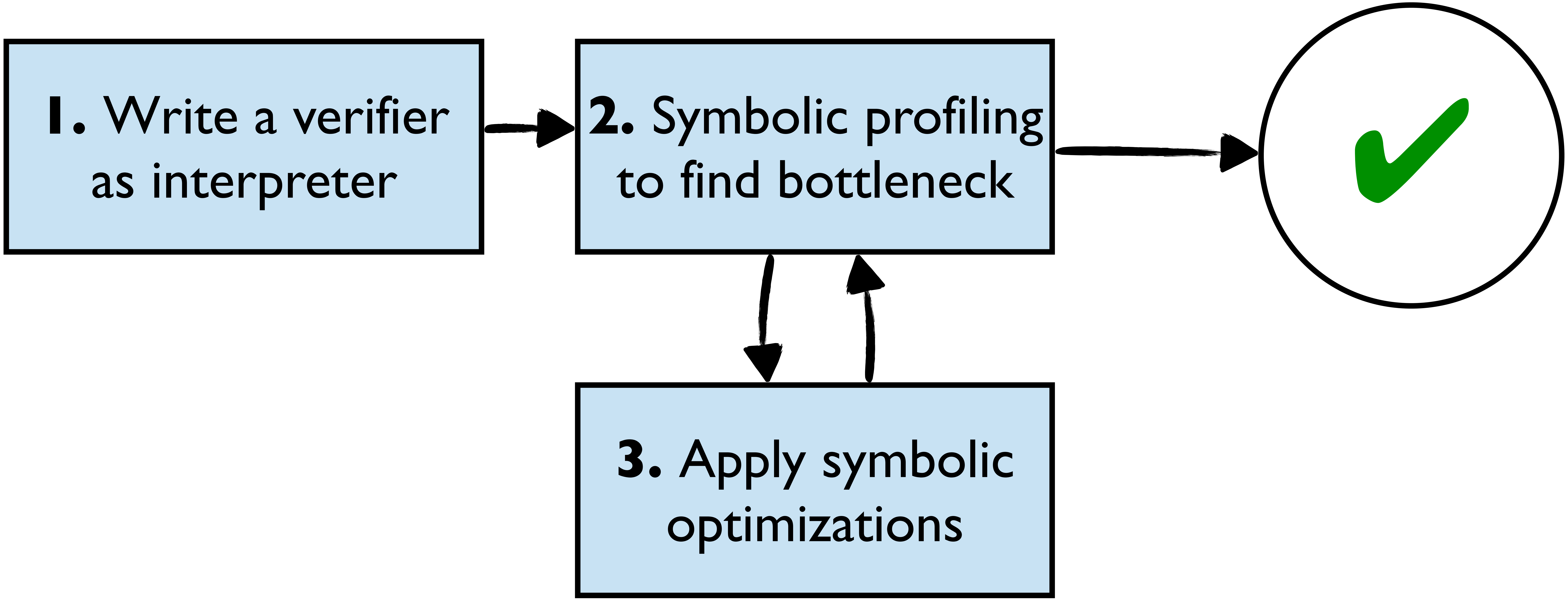
```
0: sltz a1 a0
1: bnez a1 4
2: sgtz a0 a0
3: ret
4: li   a0 -1
5: ret
```

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

RISC-V verifier

Serval

Verifier = interpreter + symbolic optimization



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

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

- 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]))
```

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

RISC-V verifier

Serval

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

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

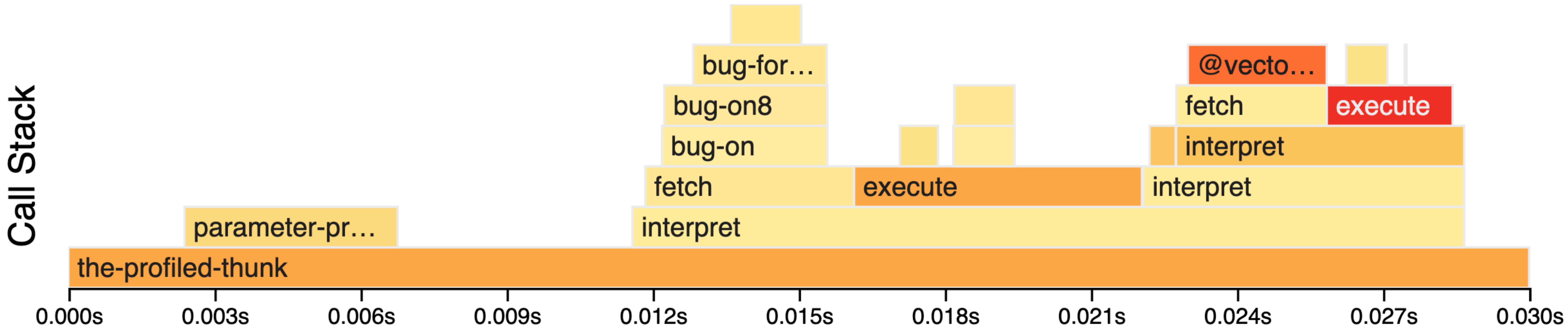
```
0: sltz a1 a0
1: bnez a1 4
2: sgtz a0 a0
3: ret
4: li   a0 -1
```

Slow/Timeout

RISC-V verifier

Serval

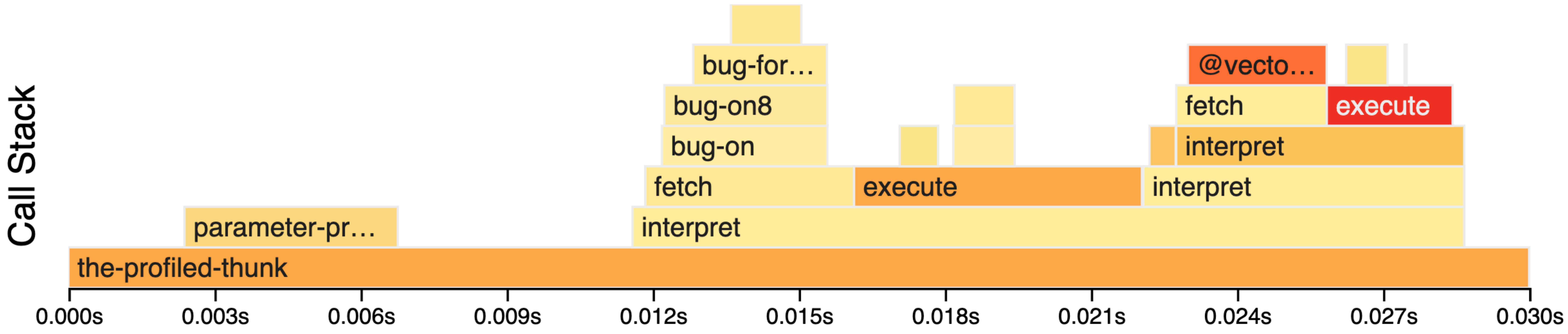
Verifier [2/3]: identifying bottlenecks in symbolic evaluation



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

Function	Score	Time (ms)	Term Count	Unused Terms	Union Size	Merge Cases
execute run.rkt:42 3 calls └interpret run.rkt:10	3.7	6	13	13	0	22
@vector-ref 1 call └fetch run.rkt:25	2.0	3	0	0	6	14

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



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

fetch

Function			Term Count	Unused Terms	Union Size	Merge Cases
execute run.rkt:42 3 calls └interpret run	3.7	6	13	13	0	22
@vector-ref 1 call └fetch run.rkt:25	2.0	3	0	0	6	14

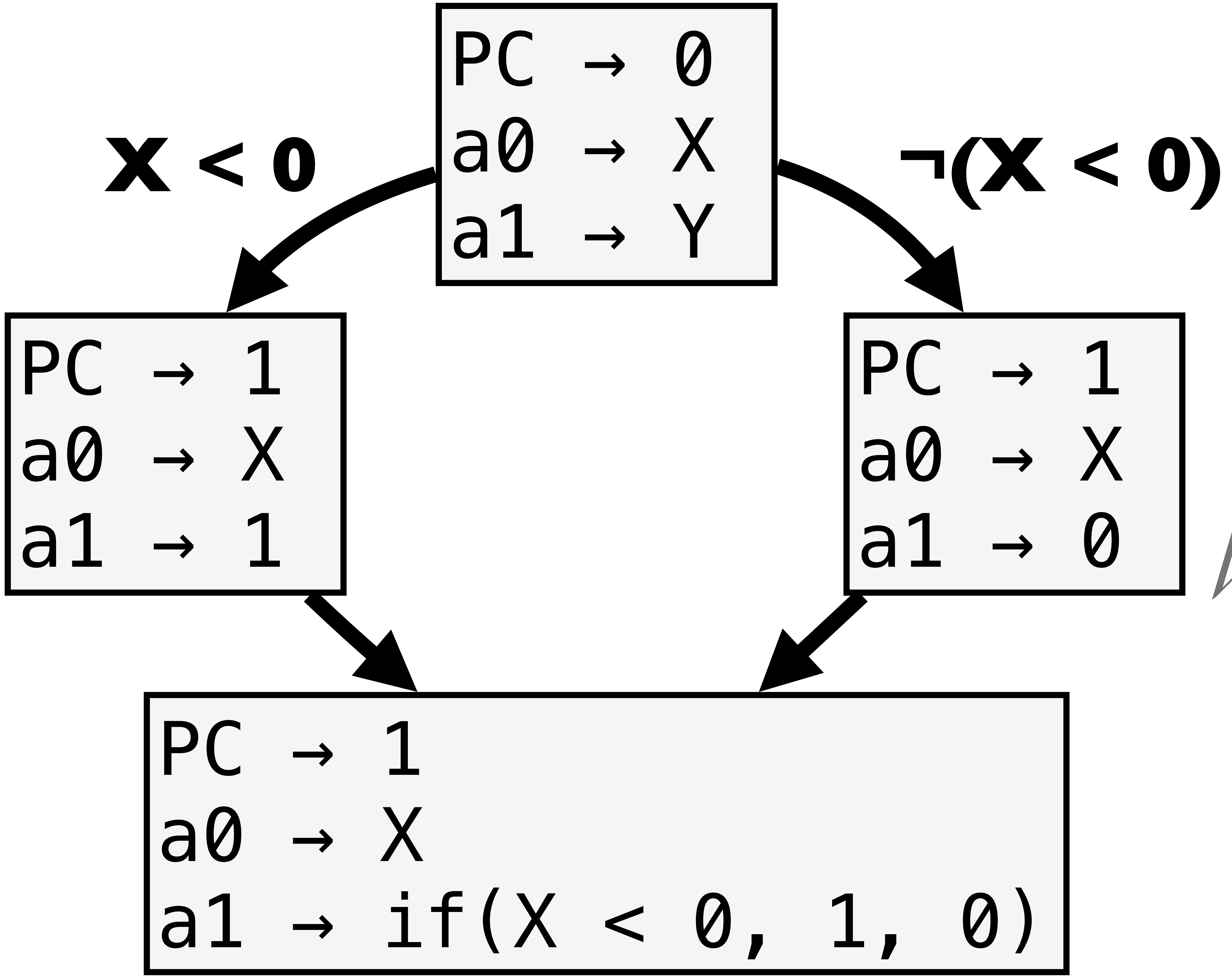
Verifier [2/3]: identifying bottlenecks in symbolic evaluation

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

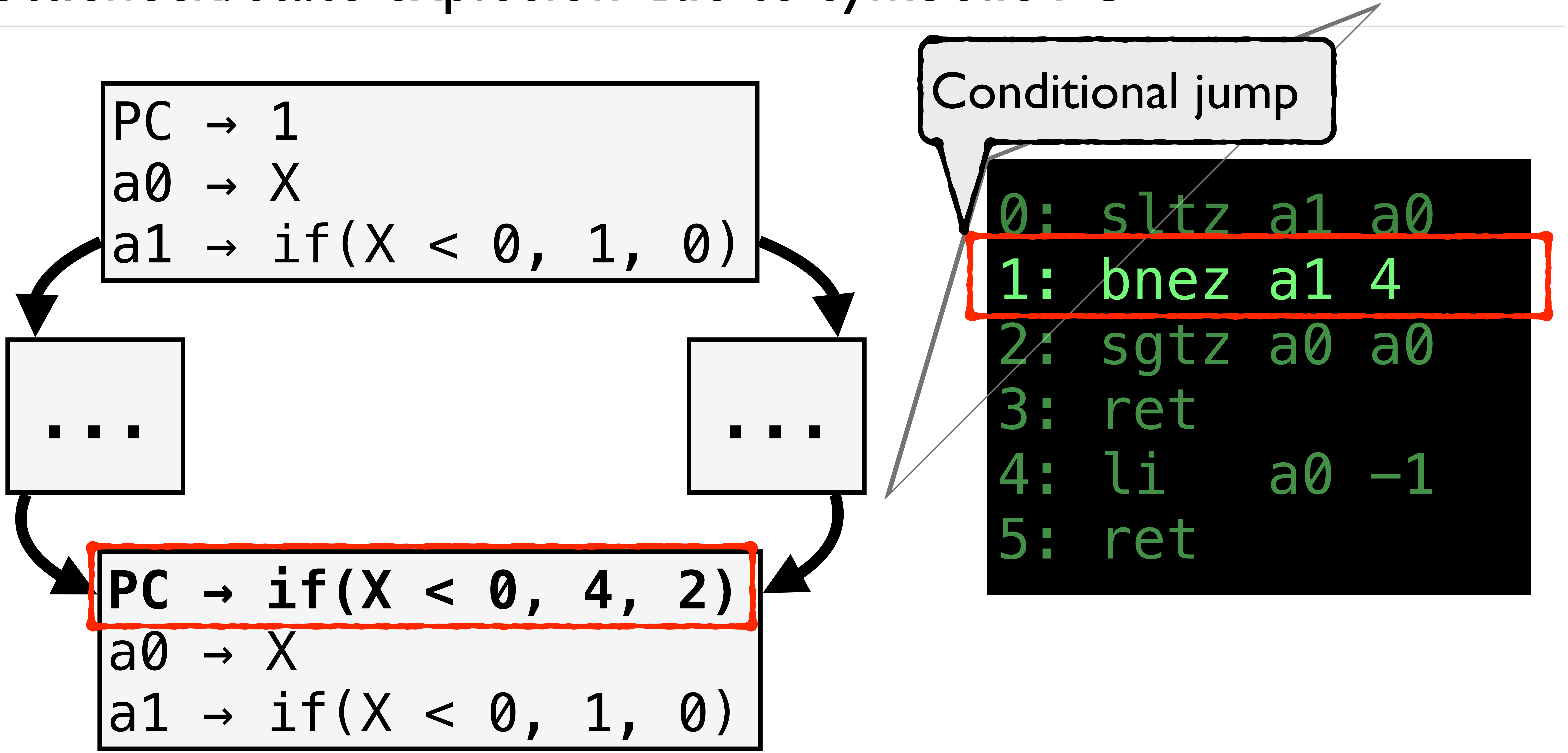
```
0: sltz a1 a0
1: bnez a1 4
2: sgtz a0 a0
3: ret
4: li   a0 -1
5: ret
```

Merge states to avoid path explosion

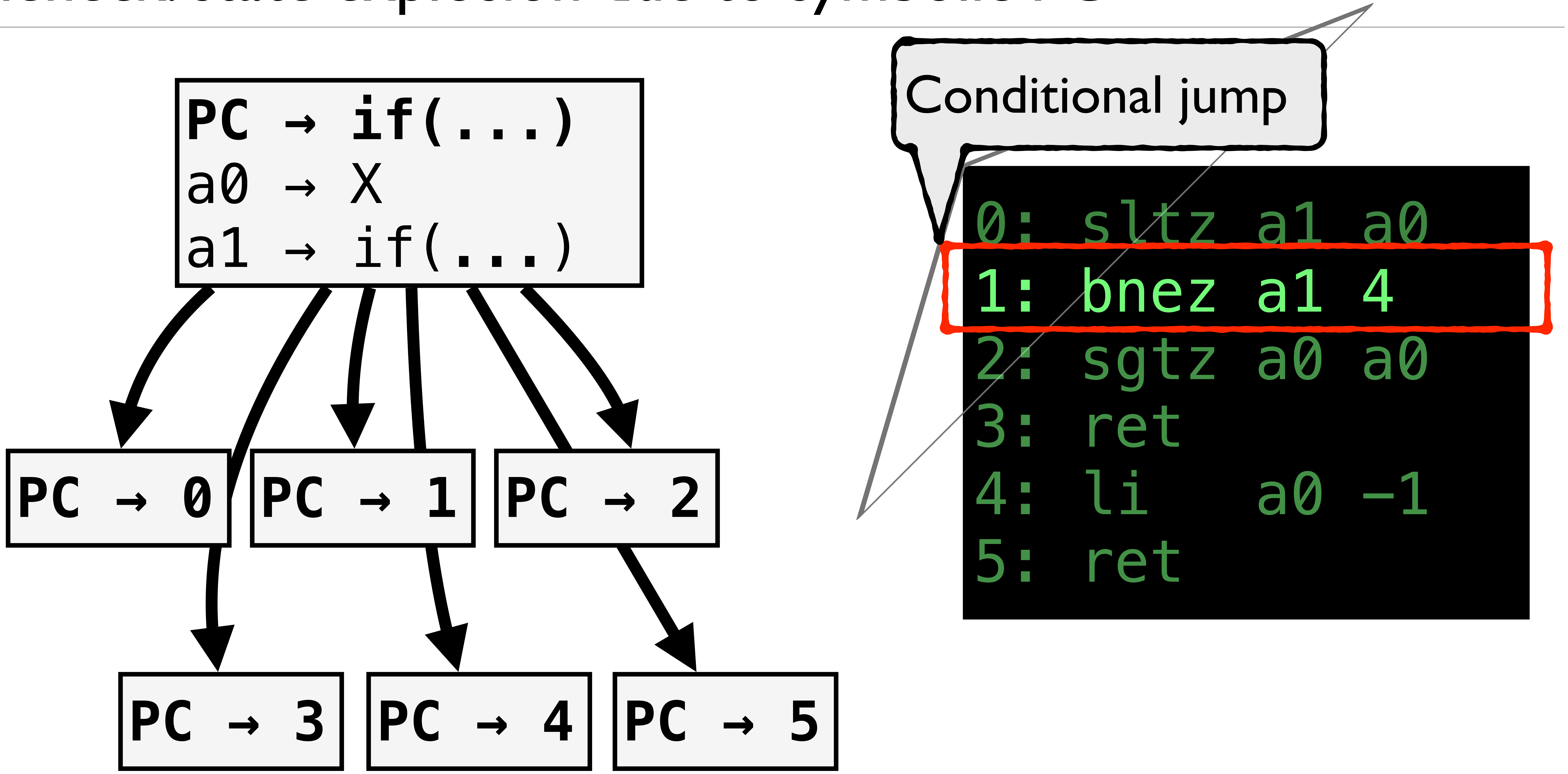



```
0: sltz a1 a0
1: bnez a1 4
2: sgtz a0 a0
3: ret
4: li   a0 -1
5: ret
```

Bottleneck: state explosion due to symbolic PC



Bottleneck: state explosion due to symbolic PC



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

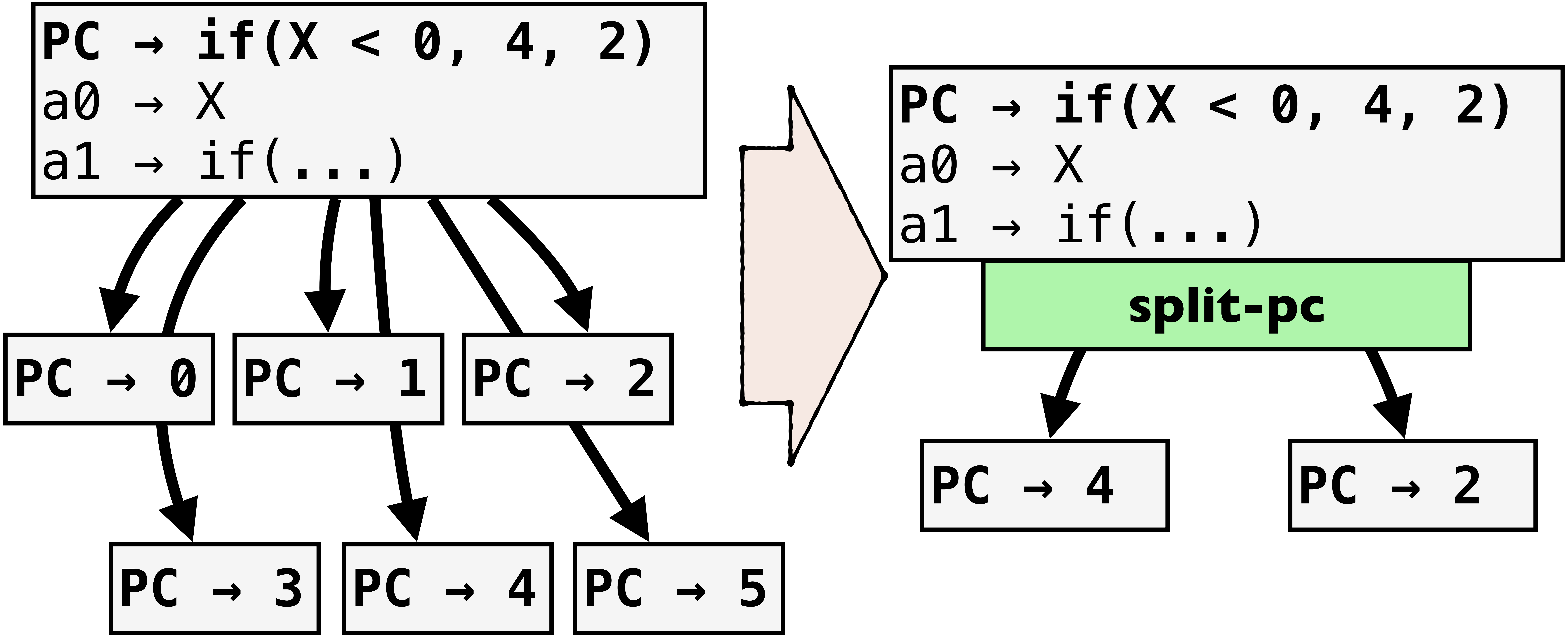
Verifier [3/3]: Repairing with symbolic optimizations

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

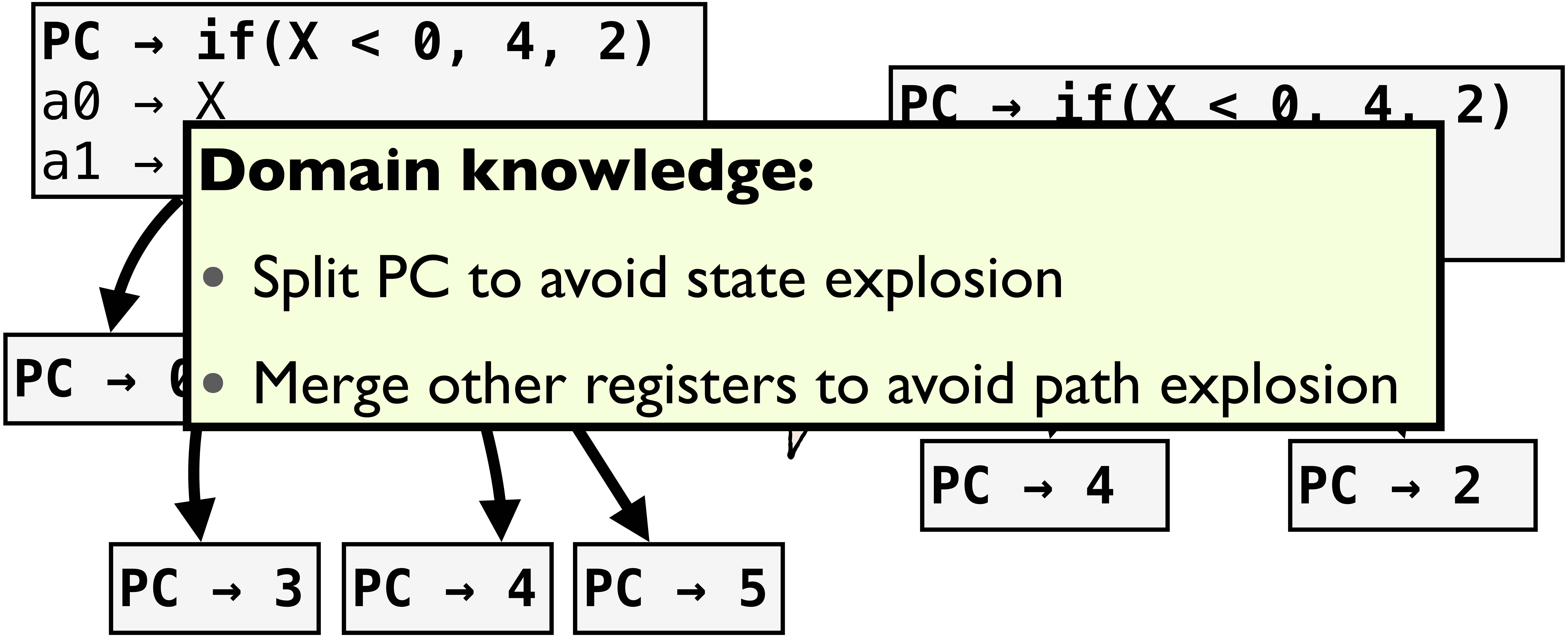
```
(define (interpret c program)
+ (serval:split-pc [cpu pc] c
  (define insn (fetch pc program))
  (match insn
    ...)))
```

- Match on symbolic structure of PC
- Evaluate separately using each concrete PC value
- Merge states afterwards

Verifier [3/3]: Repairing with symbolic optimizations



Verifier [3/3]: Repairing with symbolic optimizations



Domain knowledge:

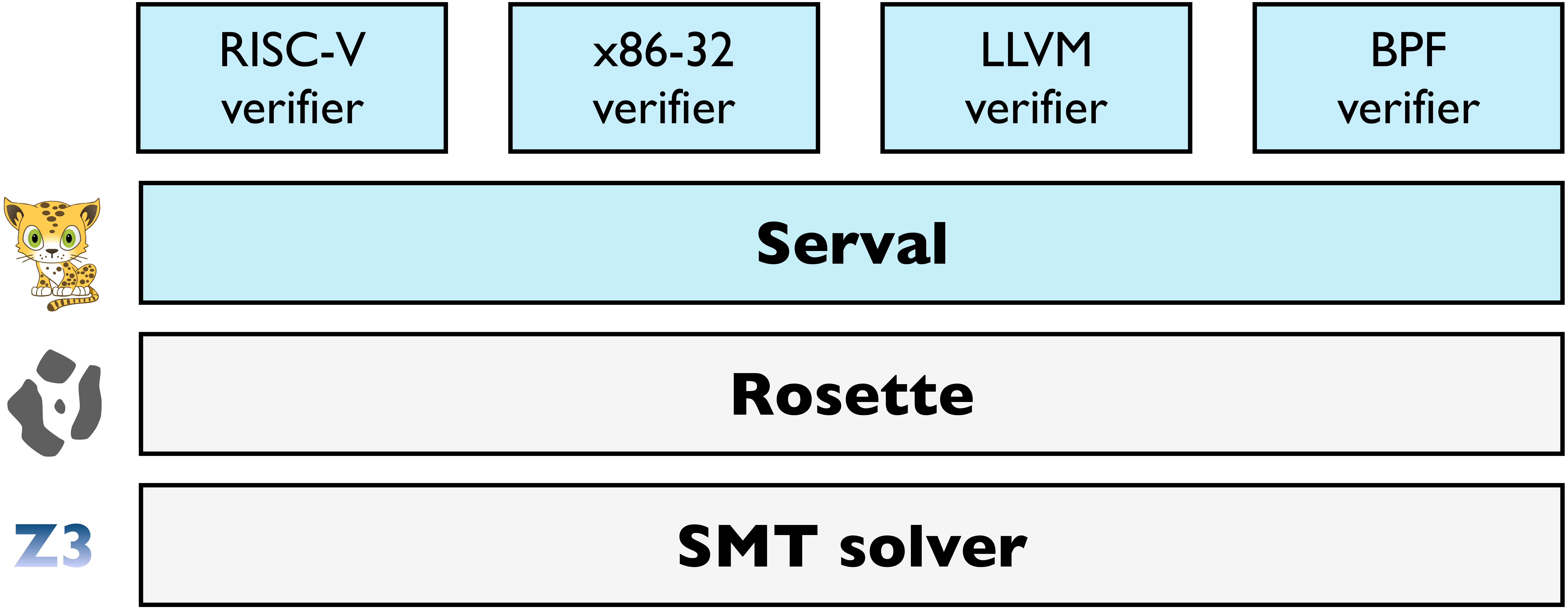
- Split PC to avoid state explosion
- Merge other registers to avoid path explosion

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



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?

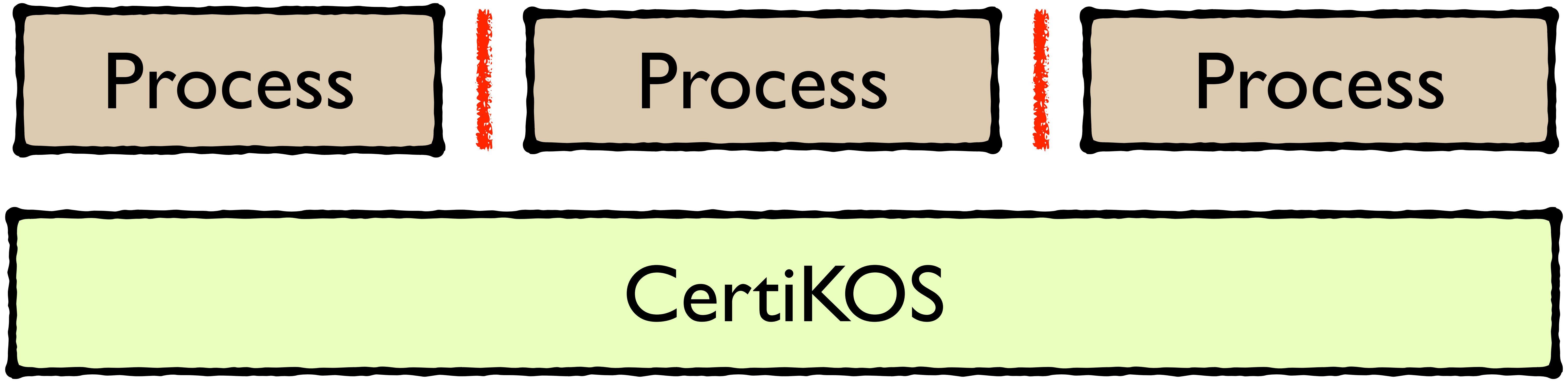
Is the specification expressible in Serval?

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

UNSAT