Automated verification of systems software with Serval

Xi Wang
Joint work with Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, and Emina Torlak
University of Washington  Columbia University  Microsoft Research
Today: eliminating bugs in low-level systems software

• Low-level bugs: buffer overflow, div by zero
• Logic bugs: implementation does something unintended
• Design bugs: unintended design is insecure
Example: undefined behavior

```c
uint64_t mul(uint16_t a, uint16_t b) {
    uint32_t c = a * b;
    return c;
}
```

Question: what’s the result of `mul(60000, 60000)`?

- (a) 3,600,000,000
- (b) 18,446,744,073,014,584,320
- (c) something else
Eliminating bugs with formal verification

- seL4 (SOSP’09)
- Ironclad, Jitk (OSDI’14)
- 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
Prior work: automated (push-button) verification

- Yggdrasil
  - OSDI'16
- Hyperkernel
  - SOSP'17
- Nickel
  - OSDI'18
Prior work: automated (push-button) verification

- No proofs on implementation
- Requires finite implementation
- Restricts specification
Challenges

- How to lower effort of writing automated verifiers?
- How to find and fix performance bottlenecks?
- How to retrofit to existing systems?

Diagram:

- Specification
  - Automated verifier
    - Implementation
    - SMT solver

- ✔️
- ✘
Contributions

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

• Experience
  • Retrofitted CertiKOS and Komodo for Serval
  • Found 30+ new bugs in Linux BPF JIT and 3 in Keystone

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

- System specification
- RISC-V instructions
- RISC-V verifier

Serval

Rosette

Z3

SMT solver
Verifying a system with Serval

System specification

RISC-V instructions

RISC-V verifier

Serval

Rosette

Z3

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

Z3

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
Example: proving refinement for sign

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

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

RISC-V verifier

```
0: sltz a1 a0
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]))
```

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

Serval

RISC-V verifier

slow/timeout
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) | 3.7 | 3.0 | 6 | 13 | 13 | 0 | 22
@vector-ref 1 call | 2.0 | 3.0 | 3 | 0 | 0 | 6 | 14
Verifier [2/3]: identifying bottlenecks in symbolic evaluation

Call Stack:
- the-profiled-thunk
- parameter-pr...

Function:
- fetch
- execute
- run.rkt:42
- calls
- interpret
- run.rkt:25

Caller Context:
- 1

Term Count: 6
Unused Terms: 13
Union Size: 13
Merge Cases: 0

@vector-ref
- 1 call
- fetch
- run.rkt:25

Aggregate [More]
Collapse solver time
Signatures
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))))
    ...
))
Merge states to avoid path explosion

\[
\begin{array}{c}
\text{PC} \rightarrow 0 \\
a0 \rightarrow X \\
a1 \rightarrow Y \\
\hline
\text{PC} \rightarrow 1 \\
a0 \rightarrow X \\
a1 \rightarrow 1 \\
\hline
\text{PC} \rightarrow 1 \\
a0 \rightarrow X \\
a1 \rightarrow 0 \\
\hline
\text{PC} \rightarrow 1 \\
a0 \rightarrow X \\
a1 \rightarrow \text{if}(X < 0, 1, 0) \\
\end{array}
\]

\[X < 0\]  
\[
\neg(X < 0)
\]

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

Conditional jump

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

PC → 1
a0 → X
a1 → if(X < 0, 1, 0)

PC → if(X < 0, 4, 2)
a0 → X
a1 → if(X < 0, 1, 0)
Bottleneck: state explosion due to symbolic PC

0: sltz a1 a0
a0 → X
a1 → if(...)

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" on symbolic state
- Fine-tune symbolic evaluation
- Use domain knowledge
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 $\rightarrow$ if($X < 0$, 4, 2)

a0 $\rightarrow$ X

a1 $\rightarrow$ if(...)
Verifier [3/3]: Repairing with symbolic optimizations

Domain knowledge:
- Split PC to avoid state explosion
- Merge other registers to avoid path explosion
Symbolic optimizations are essential to scaling verification

- Symbolic program counter
- Symbolic memory address
- Symbolic system register
- ... and more
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 - recent work SymFix (VMCAI'20)
Experience

- Can existing systems be retrofitted for Serval?

- Are Serval’s verifiers reusable?
Retrofitting previously verified systems

- Port CertiKOS (PLDI’16) and Komodo (SOSP’17) to RISC-V
- Retrofit to automated verification
- Apply the RISC-V verifier to binary image
- Prove functional correctness and noninterference
- \( \approx 4 \) weeks each
Retrofitting overview

Is the implementation free of unbounded loops?

System implementation

Is the specification expressible in Serval?

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

- Linux kernel's BPF JIT compilers
  - Found 30+ new bugs
  - Bug fixes and new tests upstreamed

- Keystone
  - Open-source enclave platform
  - Found 3 new bugs in implementation and design
Example: BPF in the Linux kernel

BPF bytecode

Application

User

Kernel

1

2

3

Bugs in JIT can compromise the entire systems!

Jitk (OSDI’14); JitSynth (CAV’20)
Example bug found using Serval

- Difficult to audit and test
- Can have serious security impact
- Key: scale symbolic evaluation for both JIT and emitted machine instructions

```c
/* Do LSH operation */
if (val < 32) {
  /* shl dreg_hi,imm8 */
  EMIT3(0xC1, add_1reg(0x0E0, dreg_hi), val);
  /* mov ebx,dreg_lo */
  EMIT2(0x8B, add_2reg(0x0C0, dreg_lo, IA32_EBX));
  /* shld dreg_hi,dreg_lo,imm8 */
  EMIT4(0x0F, 0xA4, add_2reg(0xC0, dreg_hi, dreg_lo), val);
  EMIT3(0xC1, add_1reg(0x0E0, dreg_lo), val);
  /* IA32_ECX = 32 - val */
  EMIT2(0xB1, val);
  /* movzx ecx,ecx */
  EMIT3(0x0F, 0xB6, add_2reg(0xC0, IA32_ECX, IA32_ECX));
  /* neg ecx */
  EMIT2(0xF7, add_1reg(0xD8, IA32_ECX));
  /* add ecx,32 */
  EMIT3(0x83, add_1reg(0xC0, IA32_ECX), 32);
  /* shr ebx,cl */
  EMIT2(0xD3, add_1reg(0xE8, IA32_EBX));
  /* or dreg_hi,ebx */
  EMIT2(0x09, add_2reg(0xC0, dreg_hi, IA32_EBX));
}
```
Conclusion

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

http://serval.UNSAT.org