Building automated verification tools for systems software

Luke Nelson

Based on work with Jacob Van Geffen, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang
Goal: Eliminate bugs in systems software

Low-level bugs: division-by-zero, buffer overflow, etc.

Logic bugs: implementation does something unintended

Design bugs: intended design is not secure
Approach: formal verification

Formally prove the absence of bugs:

Write a specification of the intended behavior

Prove the implementation meets the specification
Outline

**Jitterbug**: framework for building and verifying JIT compilers for BPF (OSDI’20)

**Serval**: framework for verifying low-level systems code (SOSP’19)

**Rosette**: solver-aided programming language for creating verification and synthesis tools (PLDI’14)
Outline

- **Jitterbug**: framework for building and verifying JIT compilers for BPF (OSDI’20)
- **Serval**: framework for verifying low-level systems code (SOSP’19)
- **Rosette**: solver-aided programming language for creating verification and synthesis tools (PLDI’14)
Eliminating bugs in low-level systems using verification

Process | Process | Process

OS Kernel / security monitor

seL4 (SOSP’09)
Ironclad, Jitk (OSDI’14)
CertiKOS (PLDI’16)
Komodo (SOSP’17)
Eliminating bugs in low-level systems using verification

Strong correctness guarantees

Require manual proofs

CertiKOS: 200k lines of proof

Multiple person-years to write

Komodo (SOSP'17)
Prior work on automated verification

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

![Diagram showing automated verifier and SMT solver]

- ✔ Implementation
- ✘ Specification

Automated verifier

SMT solver
Challenges

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

Diagram:

- Implementation
- Specification
- Automated verifier
- SMT solver

✔️

✘
Serval Main Results

Serval: Framework for writing automated verifiers

- RISC-V, x86, Arm, LLVM, BPF
- Scale verification with *symbolic optimizations*

Experience retrofitting existing verified systems for Serval

- CertiKOS OS kernel
- Komodo enclave security monitor
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
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: verifying a sign function in RISC-V

### Specification

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

### Implementation

```
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 optimizations

1. Write a verifier as interpreter in Rosette
2. Symbolic profiling to find bottleneck
3. Apply symbolic optimizations
Building a verifier (1/3): Write an interpreter in Rosette

(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))))
    ...)))
Building a verifier (1/3): Write an interpreter in Rosette

(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)))]
    ...
  ))
Building a verifier (1/3): Write an interpreter in Rosette

(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)))]
   ...
))
Building a verifier (1/3): Write an interpreter in Rosette

(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)))]
    ...)))
Building a verifier (2/3): Identify symbolic evaluation bottlenecks

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

\texttt{0: sltz a1 a0}
\texttt{1: bnez a1 4}
\texttt{2: sgtz a0 a0}
\texttt{3: ret}
\texttt{4: li a0 -1}
\texttt{5: ret}

RISC-V verifier

Serval
Building a verifier (2/3): Identify symbolic evaluation bottlenecks

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

slow or times out

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

RISC-V verifier

Serval
Building a verifier (2/3): Identify symbolic evaluation bottlenecks

Sympro (OOPSLA’18): Tool for profiling symbolic evaluation

<table>
<thead>
<tr>
<th>Function</th>
<th>Score</th>
<th>Time (ms)</th>
<th>Term Count</th>
<th>Unused Terms</th>
<th>Union Size</th>
<th>Merge Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute</td>
<td>3.7</td>
<td>6</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>calls</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>interpret</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@vector-ref</td>
<td>2.0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Building a verifier (2/3): Identify symbolic evaluation bottlenecks

Sympro (OOPSLA’18): Tool for profiling symbolic evaluation

Hints that fetch might be the issue

The diagram shows a call stack with time intervals (0.000s to 0.030s) and various functions with execution times. The highlighted function 'fetch' indicates potential issues.

Function: execute (run.rkt:42, 3 calls)
Function: interpret (run.rkt:10)
Function: @vector-ref (1 call)

Execution times:
- execute: 3.7
- interpret: 6
- @vector-ref: 2.0

Unused Terms: 6
Union Size: 13
Merge Cases: 22
Building a verifier (2/3): Identify symbolic evaluation bottlenecks

(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)))]
    ...)
Rosette merges states to avoid path explosion

\( X < 0 \)  
\( \neg (X < 0) \)

PC → 0  
a0 → X  
a1 → Y

PC → 1  
a0 → X  
a1 → 1

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

PC → 1  
a0 → X  
a1 → 0

a1 = (a0 < 0) ? 1 : 0;
Bottleneck: state explosion due to symbolic PC

PC → 1
a0 → X
a1 → if(X < 0, 1, 0)

... conditional jump ...

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

PC $\rightarrow$ if(...)  
a0 $\rightarrow$ X  
a1 $\rightarrow$ if(...)  

PC $\rightarrow$ 0  
PC $\rightarrow$ 1  
PC $\rightarrow$ 2  

PC $\rightarrow$ 3  
PC $\rightarrow$ 4  
PC $\rightarrow$ 5  

Conditional jump

0: sltz a1 a0  
1: bnez a1 4  
2: sgtz a0 a0  
3: ret  
4: li a0 -1  
5: ret
Building a verifier (3/3): Fix bottlenecks with symbolic optimizations

- "peephole" optimization / rewrite on the symbolic state
- Fine-tuned control over symbolic evaluation
- Use domain knowledge
Building a verifier (3/3): Fix bottlenecks with symbolic optimizations

((define (interpret c program)

(define pc (cpu-pc c))

(define (match : ...))

(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
Building a verifier (3/3): Fix bottlenecks with symbolic optimizations

\[ \text{PC} \rightarrow \text{if}(X < 0, 4, 2) \]
\[ a0 \rightarrow X \]
\[ a1 \rightarrow \text{if}(...) \]

\[ \text{PC} \rightarrow 0 \]
\[ \text{PC} \rightarrow 1 \]
\[ \text{PC} \rightarrow 2 \]

\[ \text{PC} \rightarrow 3 \]
\[ \text{PC} \rightarrow 4 \]
\[ \text{PC} \rightarrow 5 \]

\[ \text{PC} \rightarrow \text{if}(X < 0, 4, 2) \]
\[ a0 \rightarrow X \]
\[ a1 \rightarrow \text{if}(...) \]

\[ \text{split-pc} \]

\[ \text{PC} \rightarrow 4 \]
\[ \text{PC} \rightarrow 2 \]
Building a verifier (3/3): Fix bottlenecks with symbolic optimizations

**Domain knowledge:**
- Split PC to avoid state explosion
- Merge other registers to avoid path explosion
Symbolic optimizations are necessary to scale 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

SymFix (VMCAI’2020): automating fixes for symbolic evaluation bottlenecks

Caveat:

Symbolic profiling cannot identify expensive SMT operations
Retrofitting previous systems for verification with Serval

**Port two systems to RISC-V**

- CertiKOS (PLDI'16): OS kernel with strict isolation verified in Coq
- Komodo (SOSP’17): Enclave security monitor verified in Dafny

**Retrofit to automated verification with Serval**

- Apply RISC-V verifier to binary image
- Prove functional correctness and noninterference
- ~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

Process | Process | Process

CertiKOS
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”
Serval summary

Writing verifiers as interpreters

Systematic approach to scaling verification

Retrofitting Serval to existing systems

  Security monitors good fit for automated verification

  No unbounded loops; no inductive data structures
**Outline**

**Jitterbug**: framework for building and verifying JIT compilers for BPF (OSDI'20)

**Serval**: framework for verifying low-level systems code (SOSP’19)

**Rosette**: solver-aided programming language for creating verification and synthesis tools (PLDI’14)
(e) BPF: extended Berkeley Packet Filter

**Widely deployed: tracing, networking, security, etc.**
- Applications submit BPF programs to extend the kernel
- Kernel translates BPF to machine code via JIT compilers
- Kernel attaches machine code to subsystems for execution

**Correctness is critical**
- Compiled code runs in kernel.
- Makes decisions throughout the system.

---

**Goal: Formally verified BPF JITs in the Linux kernel**
Challenges and approach

**Not designed for verification**
Need to rule out real bugs, enable optimizations, and amenable to verification
Develop a practical specification for JIT correctness

**Rapidly evolving**
Can catch up with new features and respond to code reviews
Scale automated verification to JIT compilers

**Integration with kernel development**
Auditable C code requiring no knowledge of formal methods
Write JITs in domain-specific language & extract to C code
Jitterbug main results

Jitterbug: automated formal verification of BPF JITs
   Specification for reasoning about JIT correctness
   Automated proof strategy that scales to new & existing JITs

Upstreamed changes in the Linux kernel
   New BPF JIT for RISC-V (32-bit) since v5.7
   30+ new bug fixes and optimizations for x86 (32/64), Arm (32/64), RISC-V (64)
   2 bugs in Arm64 instruction encoding library (shared by KVM, etc.)

Clarification changes in the RISC-V instruction-set manual

Finding bugs in well-tested code shows the effectiveness of verification
BPF history

Classic BPF in 1992

Extended BPF in 2014

11 64-bit registers, function calls, 100+ opcodes, etc.

In-kernel JIT to machine code for performance
Overview of BPF JIT compilation

- Application
- BPF program
- BPF verifier
- Linux kernel
- BPF JIT compiler
- Machine code
- Hook point

**BPF JIT**
- Considers only safe programs
- Combines compiler + assembler + linker

**Kernel accepts safe BPF programs**
- No out-of-bounds access, division by zero
- Bounded execution

64-bit registers R0—R10 & 100+ instruction opcodes
Flexible control flow, shared memory as BPF maps
Overview of BPF program execution

Interacts with the kernel
- Return value, memory accesses, function calls
- Behaves like a regular function call

machine code

input data

prologue  body  epilogue

memory accesses & function calls

return value
BPF JITs are hard to get right

82 correctness bugs in x86 (32 & 64), Arm (32 & 64), RISC-V (64) JITs in 2014 – 2020

Bugs in every category of instructions

Difficult to audit and test exhaustively
Example: load 32-bit value from memory (x86)

case BPF_LDX | BPF_MEM | BPF_W:
...
/* Emit code to clear high bits */
if (!bpf_prog->aux->verifier_zext)
  break;
if (dstk) {
  /* MOV [ebp+off], 0 */
  EMIT3(0xC7, add_1reg(0x40, IA32_EBP), STACK_VAR(dst_hi));
  EMIT(0x0, 4);
} else {
  /* MOV dst_hi, 0 */
  EMIT3(0xC7, add_1reg(0xC0, dst_hi), 0);
}

Optimization (analyzed by kernel)

JIT control flow: dst_hi spilled to stack

JIT control flow: dst_hi mapped to reg

Can you spot any bugs?
Writing correct JITs is difficult

Must reason about complex artifacts simultaneously

- JIT configuration (e.g., optimizations)
- Control flow in both JIT and emitted code
- Semantics of source and target instructions

Must reason about low-level machine state

- BPF JITs emit machine instructions as raw bytes
- Requires precise model of target architecture

Approach: formal specification of JIT correctness

- Restricted form of compiler correctness
- Intuition: Machine code must behave equivalently to source BPF program
For any safe source program (i.e., terminates with no undefined behavior), JIT configuration, target program produced by the JIT, and for any input data:
The executions of source & target programs produce the same trace and return value.
JIT correctness pros & cons

+ Intuitive and effective at preventing bugs
  Target program preserves behavior of source program
  Behavior of target program permitted by source program

+ Tailored for in-kernel execution
  Target program preserves architectural invariants (e.g., callee-saved registers)
  Parameterized by target semantics

- Not amenable to automated verification
  Reasons about entire trace of program execution
  Hard to encode to SMT
Exploit JIT structure: per-instruction translation

Existing JITs in Linux: emit_prologue + $N \times$ emit_insn + emit_epilogue

Each BPF instruction is separately compiled to some number of target instructions
Jitterbug proof strategy: metatheory

Exploit JIT structure: per-instruction translation and proof.
Jitterbug encodes and proves, for each opcode, a stepwise correctness specification. The stepwise spec composes so that the correctness of each translation step implies JIT correctness (proved in Lean).

```
source program

configuration

JIT compiler

target program

JIT assumptions

∧

prologue correctness

∧

per-instruction correctness

∧

epilogue correctness

⇒

JIT correctness
```
Jitterbug proof strategy

Symbolic evaluation of JIT implementation to produce symbolic target instructions. Symbolic evaluation of source & target instructions to encode behavior. Feed into stepwise specification & discharge using SMT solver.
Scaling automated verification to JITs

Symbolic evaluation of symbolic code
- Serval: symbolic input & concrete code
- Jitterbug: symbolic input & symbolic code

Scalability challenge due to path explosion
- Example: BPF’s LD64_IMM is compiled to 307 types of RISC-V instruction sequences
- Pure symbolic execution doesn’t scale

Approach: fine-tune symbolic evaluation
- Aggressively merge state at every control-flow join
- But split on every possible concrete opcode
- Compact encoding + easier-to-solve constraints

See paper: instantiating existential & axiomatizing SMT operations
Developing a BPF JIT for RISC-V (32-bit)

Development process

Started in June 2019
Co-designed with specification and proof strategy
Implemented in DSL, then automated extraction to C code
Accepted in March 2020 after five iterations of code review

Automated verification enables rapid development

Feature-complete: 100+ BPF opcodes
Caught up with new features and optimizations
Addressed code reviews in a timely manner
Example: implementing BPF JIT for RISC-V (32-bit)

### DSL / Rosette implementation

```plaintext
(func (emit_alu_r64 dst src ctx op)
  (var [tmp1 (@ bpf2rv32 TMP_REG_1)]
    [tmp2 (@ bpf2rv32 TMP_REG_2)]
    [rd (bpf_get_reg64 dst tmp1 ctx)]
    [rs (bpf_get_reg64 src tmp2 ctx)])
  (switch op)
    [(BPF_ADD)
      (cond
        [(equal? rd rs)
          (emit (rv_srli RV_REG_T0 (lo rd) 31) ctx)
          (emit (rv_slli (hi rd) (hi rd) 1) ctx)
          (emit (rv_or (hi rd) RV_REG_T0 (hi rd)) ctx)
          (emit (rv_slli (lo rd) (lo rd) 1) ctx)]
        [else
          (emit (rv_add (lo rd) (lo rd) (lo rs)) ctx)
          (emit (rv_sltu RV_REG_T0 (lo rd) (lo rs)) ctx)
          (emit (rv_add (hi rd) (hi rd) (hi rs)) ctx)
          (emit (rv_add (hi rd) (hi rd) RV_REG_T0) ctx)]]
    ])
...
```

### C implementation

```c
void emit_alu_r64(const s8 *dst, const s8 *src, const u8 op)
{
    const s8 *tmp1 = bpf2rv32[TMP_REG_1];
    const s8 *tmp2 = bpf2rv32[TMP_REG_2];
    const s8 *rd = bpf_get_reg64(dst, tmp1, ctx);
    const s8 *rs = bpf_get_reg64(src, tmp2, ctx);

    switch (op) {
    case BPF_ADD:
        if (rd == rs) {
            emit(rv_srli(RV_REG_T0, lo(rd), 31), ctx);
            emit(rv_slli(hi(rd), hi(rd), 1), ctx);
            emit(rv_or(hi(rd), RV_REG_T0, hi(rd)), ctx);
            emit(rv_slli(lo(rd), lo(rd), 1), ctx);
        } else {
            emit(rv_add(lo(rd), lo(rd), lo(rs)), ctx);
            emit(rv_sltu(RV_REG_T0, lo(rd), lo(rs)), ctx);
            emit(rv_add(hi(rd), hi(rd), hi(rs)), ctx);
            emit(rv_add(hi(rd), hi(rd), RV_REG_T0), ctx);
        }
    break;
    ...
    }
}
```

Automated extraction
Improving existing BPF JITs

x86 (32- & 64-bit), Arm (32- & 64-bit), RISC-V (64-bit)

Manually translate C code to Rosette/DSL

Adopted & tweaked invariants from JIT for RISC-V (32-bit)

Less than 3 weeks per JIT

Found and fixed new bugs

30+ new bugs in BPF JITs

Demonstrates effectiveness of specification

Developed & upstreamed new optimizations

12 optimization patches

Example: RISC-V compressed instruction support for 64-bit JIT

Verification gives confidence for complex optimizations
Demo: Zero-extension bug for 32-bit arithmetic on RISC-V (64)

BPF 32-bit ALU instructions (BPF_ALU|BPF_SUB) zero-extend to 64 bits

RV64 32-bit ALU instructions (e.g., subw) sign-extend to 64 bits

Bug: Mismatch between BPF and RISC-V semantics

Fix: Emit additional instructions to zero-extend result

```c
    case BPF_ALU | BPF_SUB | BPF_X:
    case BPF_ALU64 | BPF_SUB | BPF_X:
        emit(is64 ? rv_sub(rd, rd, rs) : rv_subw(rd, rd, rs), ctx);
        + if (!is64)
        + emit_zext_32(rd, ctx);
    break;
```
Ongoing research: Improving the BPF verifier

BPF verifier is hard to get right

Accepts unsafe programs (correctness bug)

Rejects safe programs (usability issue)

Idea: revisit proof-carrying code

Require applications to submit proofs that their programs are correct

Leverage automated techniques to generate proofs
Conclusion

**Automated verification is effective at finding and preventing bugs**

- Less developer effort required than for manual proofs
- Systematic approaches to scaling verification

**Real-world impact**

- More than 30 new bugs found and fixed
- New verified BPF JIT and optimizations deployed to the Linux kernel

**Papers+code available at**

- https://unsat.org
- https://github.com/uw-unsat/jitterbug
- https://github.com/uw-unsat/serval