

Eliminating bugs in BPF JITs using automated formal verification

Luke Nelson

with Jacob van Geffen, Emina Torlak, and Xi Wang

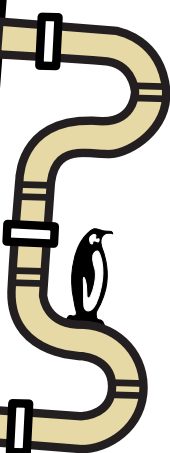
W PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

UNIVERSITY *of* WASHINGTON

UNSAT 

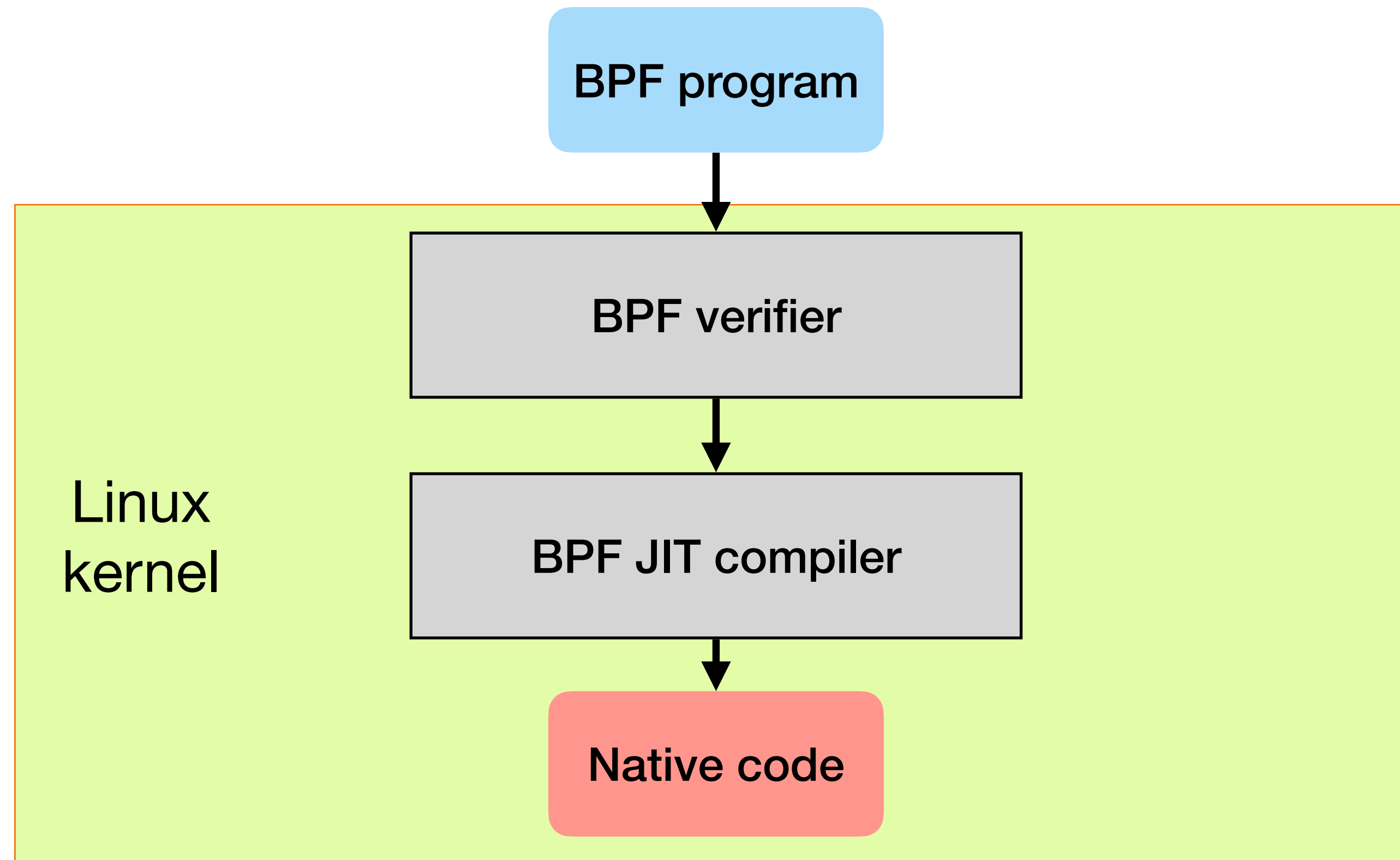
LINUX
PLUMBERS
CONFERENCE

AUGUST 24-28, 2020



BPF is used throughout the kernel

- Many uses for BPF: tracing, networking, security, etc.
- In-kernel JIT compilers for better performance



BPF JITs are hard to get right

- Developers have to think about code at multiple levels
 - The JIT itself and the machine code produced by the JIT
- Kernel selftests + fuzzing are effective at preventing many bugs
 - But the search space is too large to exhaust all possibilities
 - Many corner cases in the input to JIT and input to the BPF program
- Compiled code runs in kernel; JIT bugs can become kernel vulnerabilities

BPF JITs are hard to get right

```
case BPF_LDX | BPF_MEM | BPF_W:
...
switch (BPF_SIZE(code)) {
case BPF_W:
    if (!bpf_prog->aux->verifier_zext)
        break;
    if (dstk) {
        EMIT3(0xC7, add_1reg(0x40, IA32_EBP),
              STACK_VAR(dst_hi));
        EMIT(0x0, 4);
    } else {
        EMIT3(0xC7, add_1reg(0xC0, dst_hi), 0);
    }
    break;
}
```

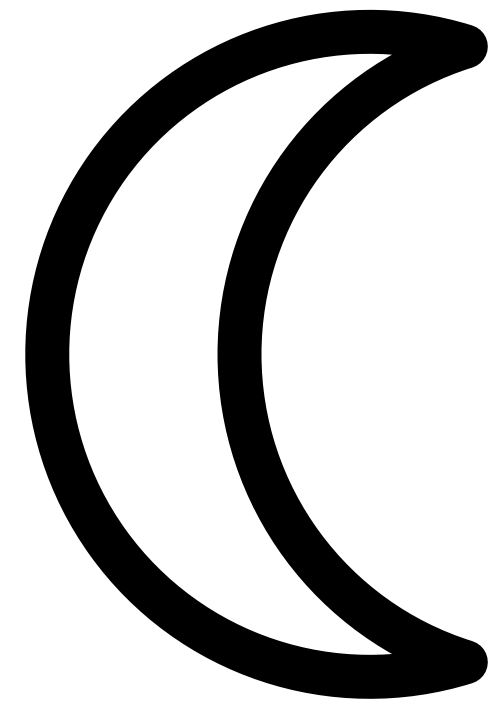
Control flow in the JIT
or even compiled code

Emitting instructions as raw bytes

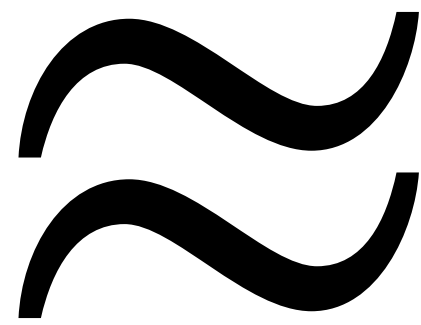
There's a bug in this code: can you
spot it? 🐜

Eliminating bugs with formal verification

- Formally prove the absence of bugs
- Specification: abstract description of intended behavior
- Prove that implementation satisfies the specification



specification



proof



implementation

Developer burden using formal verification

- Formal verification requires more manual effort compared to testing
- Requires writing down a specification
 - Specification must prevent bugs and cover existing implementations
- Requires proving the implementation meets that specification
 - Manual proofs are time-consuming, can be $>10\times$ LOC proof to implementation
 - Existing automated techniques will not scale well

Main results

- Jitterbug is a tool for automated formal verification of the BPF JITs in Linux.
 - JIT specification + automated proof strategy
 - Implementation in a domain-specific language (DSL)
- Found and fixed 30+ new bugs in existing BPF JITs across 11 patches.
 - Manual translation of JITs to DSL for verification, several weeks per JIT
- Developed a new BPF JIT for 32-bit RISC-V (riscv32 / RV32).
 - Written in DSL; automated extraction to C code
- Developed 12 new optimization patches for existing JITs.

Outline

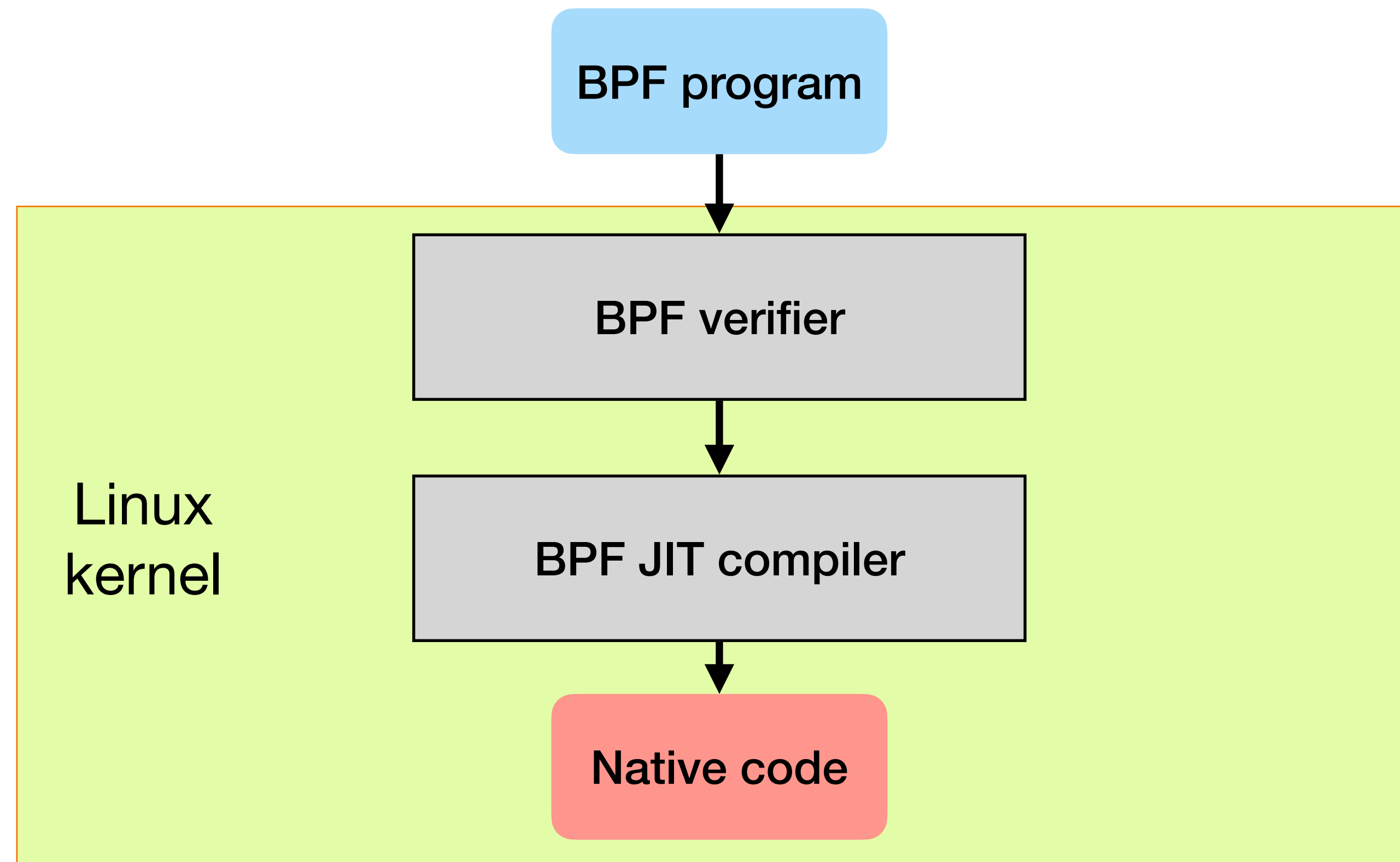
- Overview of how the BPF JITs in Linux work
- Case study of bugs in BPF JITs
- Overview of Jitterbug's JIT specification
- How to use Jitterbug
- Verification effort
- Demonstration
- Future directions for JIT verification

Outline

- **Overview of how the BPF JITs in Linux work**
- Case study of bugs in BPF JITs
- Overview of Jitterbug's JIT specification
- How to use Jitterbug
- Verification effort
- Demonstration
- Future directions for JIT verification

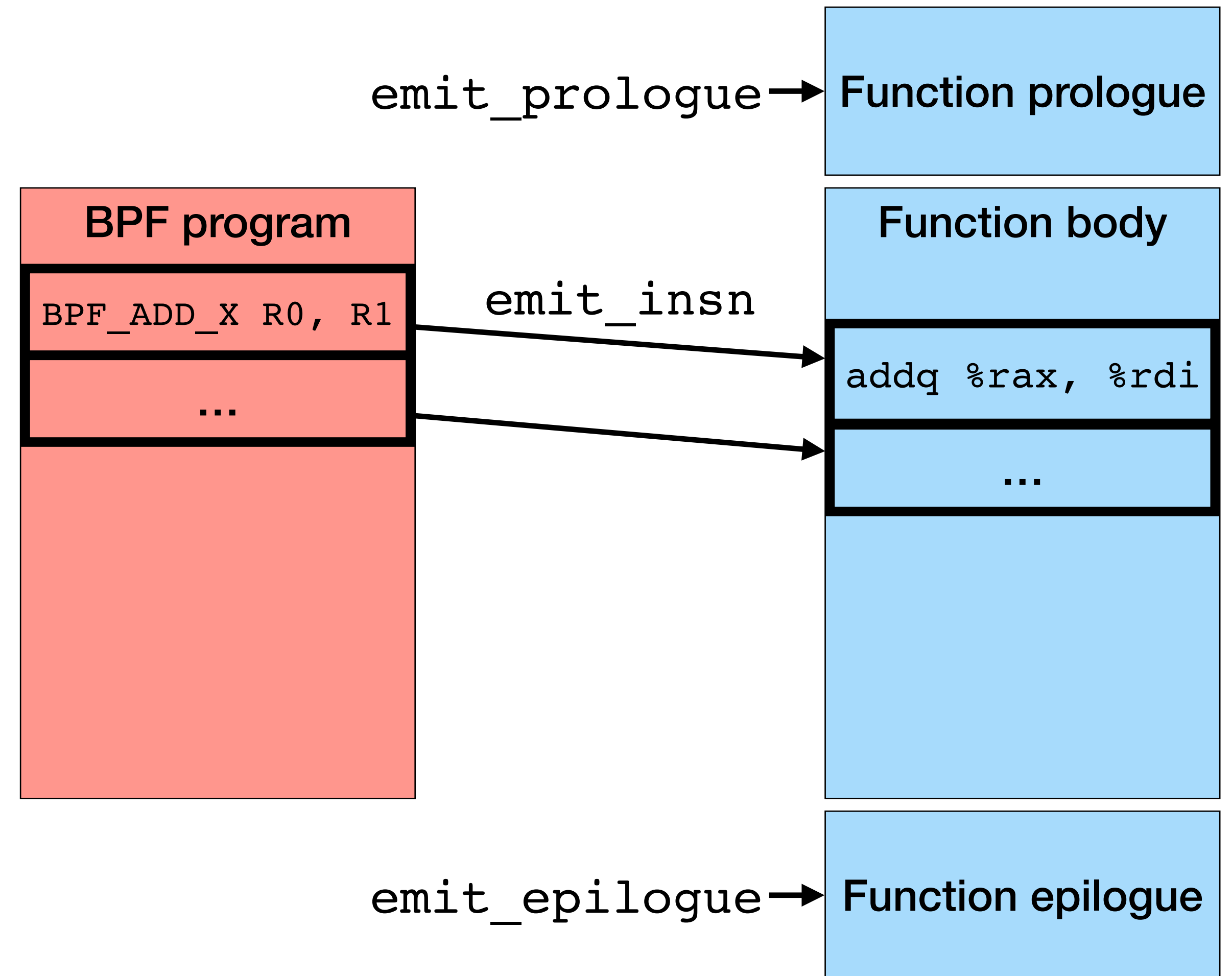
BPF JIT overview (1/2)

- Verifier checks if BPF program is safe to execute.
- JIT compiles program if verifier deems it safe
- Jitterbug focuses on BPF JIT — Assumes BPF verifier to be correct



BPF JIT overview (2/2)

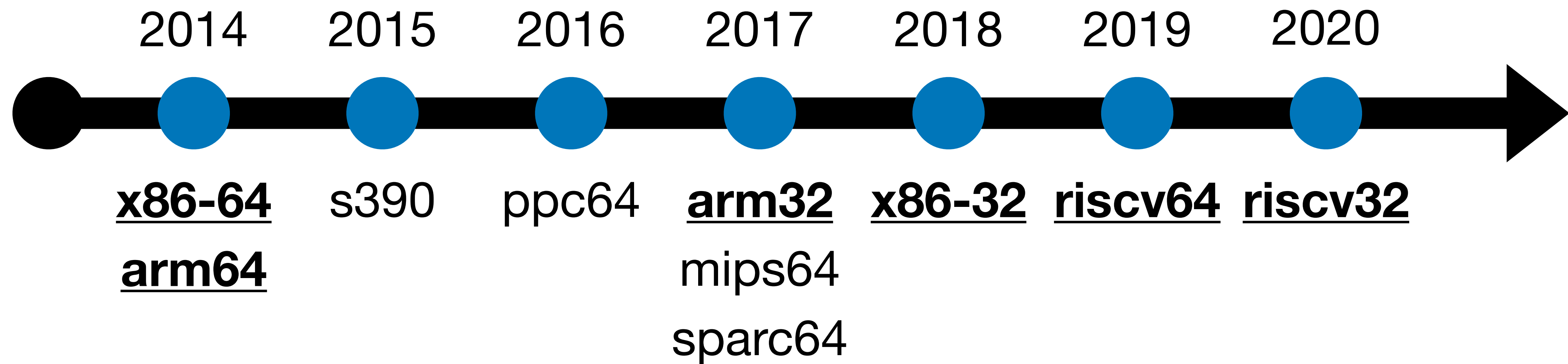
- Static register allocation
- Emits prologue / epilogue to set up stack, etc.
- Compiles one BPF instruction at a time
- Repeats JIT until code converges



Outline

- Overview of how the BPF JITs in Linux work
- **Case study of bugs in BPF JITs**
- Overview of Jitterbug's JIT specification
- How to use Jitterbug
- Verification effort
- Demonstration
- Future directions for JIT verification

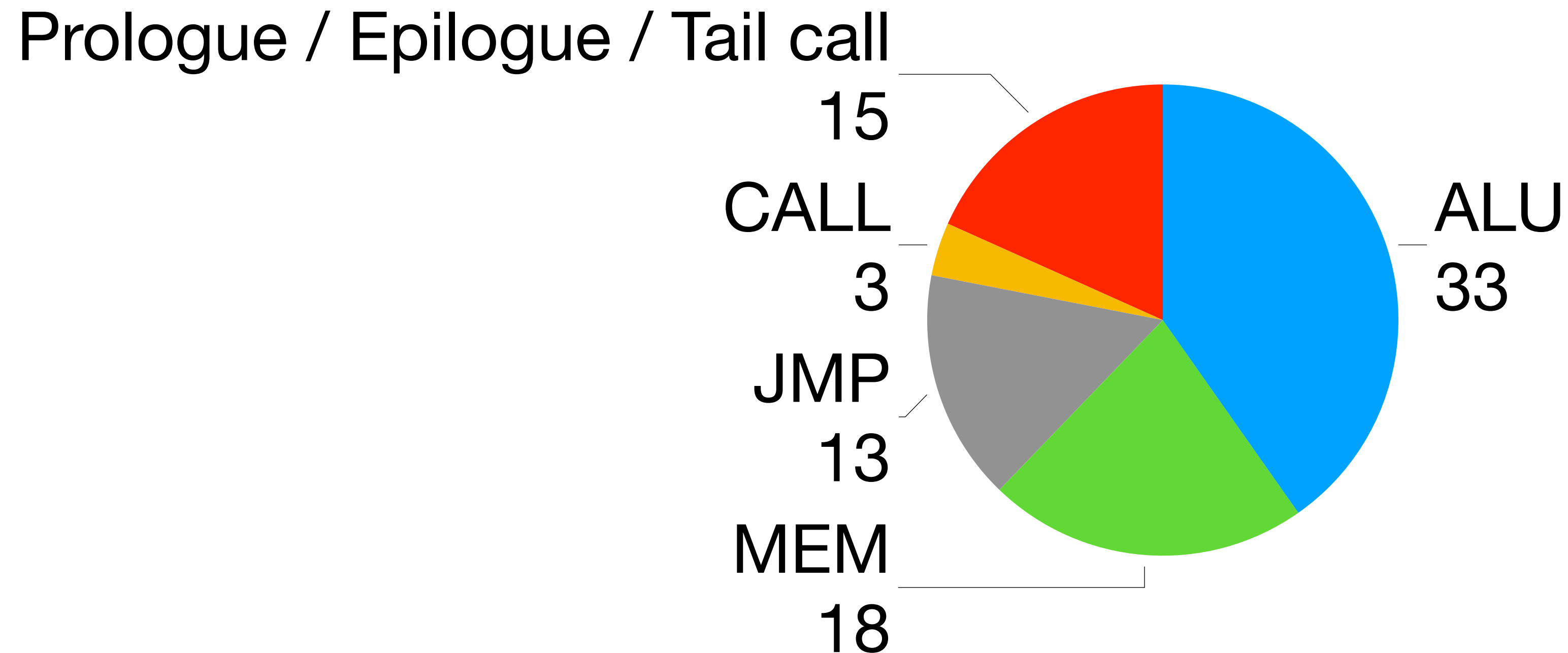
eBPF JIT History



- JIT support for eBPF added over past ~7 years
- We looked at x86, Arm, & RISC-V (32- and 64-bit)

Bugs in BPF JITs

- We manually reviewed bug-fixing commits in existing BPF JITs
- 82 JIT correctness bugs across 41 commits from from May 2014— Apr. 2020
- Correctness bug: JIT produces wrong native code for a BPF instruction



Bugs found using Jitterbug

- bpf, riscv: clear high 32 bits for ALU32 add/sub/neg/lsh/rsh/arsh
- bpf, x86_32: Fix incorrect encoding in BPF_LDX zero-extension
- arm, bpf: Fix bugs with ALU64 {RSH, ARSH} BPF_K shift by 0
- arm, bpf: Fix offset overflow for BPF_MEM BPF_DW
- arm64: insn: Fix two bugs in encoding 32-bit logical immediates
- riscv, bpf: Fix offset range checking for auipc+jalr on RV64
- bpf, x32: Fix bug with ALU64 {LSH, RSH, ARSH} BPF_K shift by 0
- bpf, x32: Fix bug with ALU64 {LSH, RSH, ARSH} BPF_X shift by 0
- bpf, x32: Fix bug with JMP32 JSET BPF_X checking upper bits
- bpf, x86_32: Fix clobbering of dst for BPF_JSET
- bpf, x86: Fix encoding for lower 8-bit registers in BPF_STX BPF_B

Bugs found using Jitterbug

- bpf, riscv: clear high 32 bits for ALU32 add/sub/neg/lsh/rsh/arsh
- bpf, x86_32: Fix incorrect encoding in BPF_LDX zero-extension
- arm, bpf: Fix bugs with ALU64 {RSH, ARSH} BPF_K shift by 0
- arm, bpf: Fix offset overflow for BPF_MEM BPF_DW
- arm64: insn: Fix two bugs in encoding 32-bit logical immediates
- riscv, bpf: Fix offset range checking for auipc+jalr on RV64
- bpf, x32: Fix bug with ALU64 {LSH, RSH, ARSH} BPF_K shift by 0
- bpf, x32: Fix bug with ALU64 {LSH, RSH, ARSH} BPF_X shift by 0
- bpf, x32: Fix bug with JMP32 JSET BPF_X checking upper bits
- bpf, x86_32: Fix clobbering of dst for BPF_JSET
- bpf, x86: Fix encoding for lower 8-bit registers in BPF_STX BPF_B

Example bug (1/2)

Zero-extension of 32-bit ALU instructions on riscv64

- BPF 32-bit ALU instructions (BPF_ALU) zero-extend to 64 bits
- riscv64 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

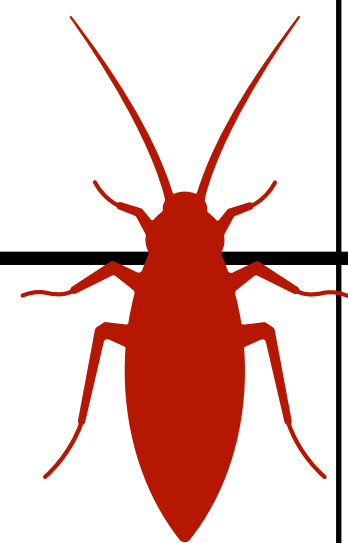
```
case BPF_ALU | BPF_SUB | BPF_X:
case BPF_ALU64 | BPF_SUB | BPF_X:
    emit(is64 ? rv_sub(rd, rd, rs) : rv_subw(rd, rd, rs), ctx);
+   if (!is64)
+       emit_zext_32(rd, ctx);
    break;
```

Example bug (2/2)

mov encoding in LDX on x86-32

- 4-byte BPF memory load zero-extends upper 32 bits
- 2 x86 registers per 1 BPF register
 - zero-extending is setting reg holding the high bits to 0
- JIT uses the following instruction:

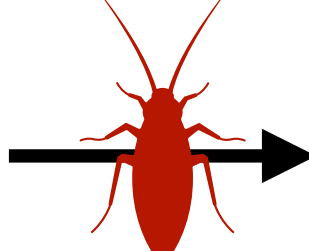
```
movl $0, %dst_hi
```




```
case BPF_LDX | BPF_MEM | BPF_W:
...
switch (BPF_SIZE(code)) {
case BPF_W:
...
if (dstk) {
    EMIT3(0xC7, add_1reg(0x40, IA32_EBP),
          STACK_VAR(dst_hi));
    EMIT(0x0, 4);
} else {
    EMIT3(0xC7, add_1reg(0xC0, dst_hi), 0);
}
break;
```

Example bug (2/2)

mov encoding in LDX on x86-32

`movl $0, %dst_hi`  `EMIT3(0xC7, add_1reg(0xC0, dst_hi), 0);`

- `EMIT3`: Emit 3 bytes of instruction
 - `0xC7`: Opcode for “`mov r/m32, imm32`”
 - `add_1reg(0xC0, dst_hi)`: Encodes destination register
 - `0`: one byte of immediate
- Bug: “`mov`” expects `imm32`, missing 3 bytes of the immediate!
- Fix: Use “`xor`” instead: correct encoding, fewer bytes

`xorl %dst_hi, %dst_hi`  `EMIT2(0x33, add_2reg(0xC0, dst_hi, dst_hi));`

Outline

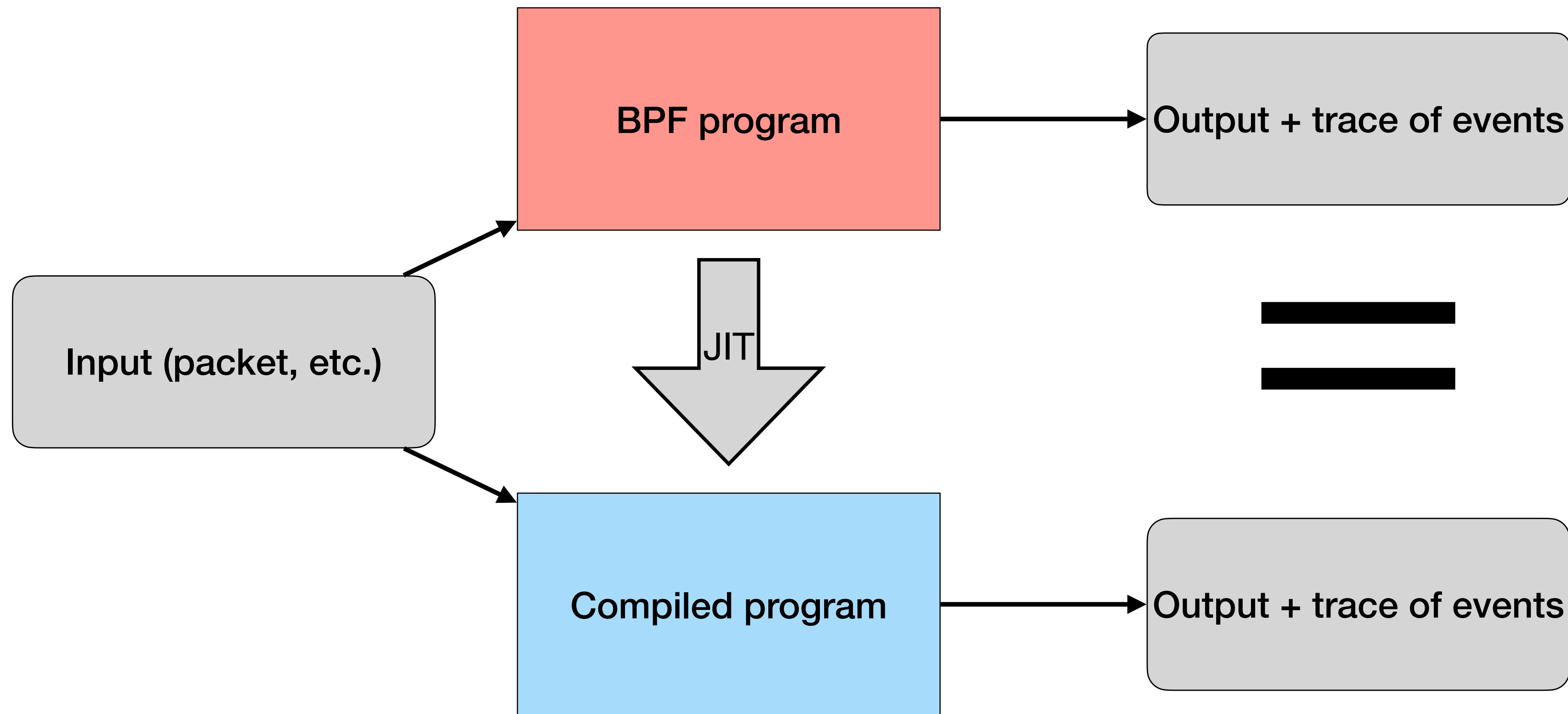
- Overview of how the BPF JITs in Linux work
- Case study of bugs in BPF JITs
- **Overview of Jitterbug's JIT specification**
- How to use Jitterbug
- Verification effort
- Demonstration
- Future directions for JIT verification

How to systematically rule out bugs?

- Need a specification that rules out classes of bugs in BPF JITs
 - Encoding bugs, semantics bugs, etc.
 - ALU, JMP, MEM, CALL, etc.
- What does JIT correctness even mean?
- How to prove implementation meets specification?

Specification: End-to-end correctness

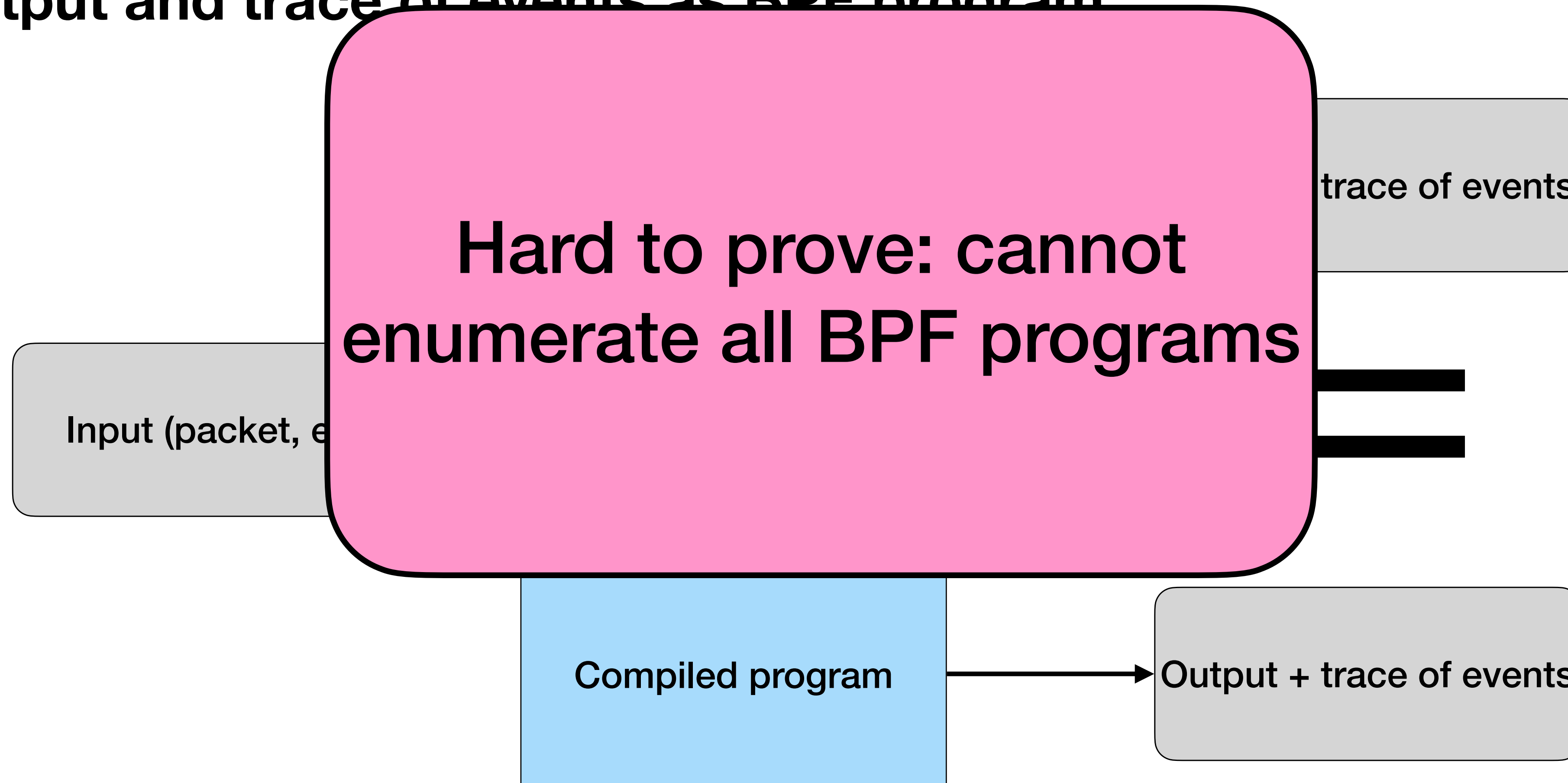
For all BPF programs, for all inputs, compiled code should produce same output and trace of events as BPF program



Trace: sequence of memory loads / stores + function calls

Specification: End-to-end correctness

For all BPF programs, for all inputs, compiled code should produce same output and trace of events as BPF program



Trace: sequence of memory loads / stores + function calls

Specification: Breaking down to three parts

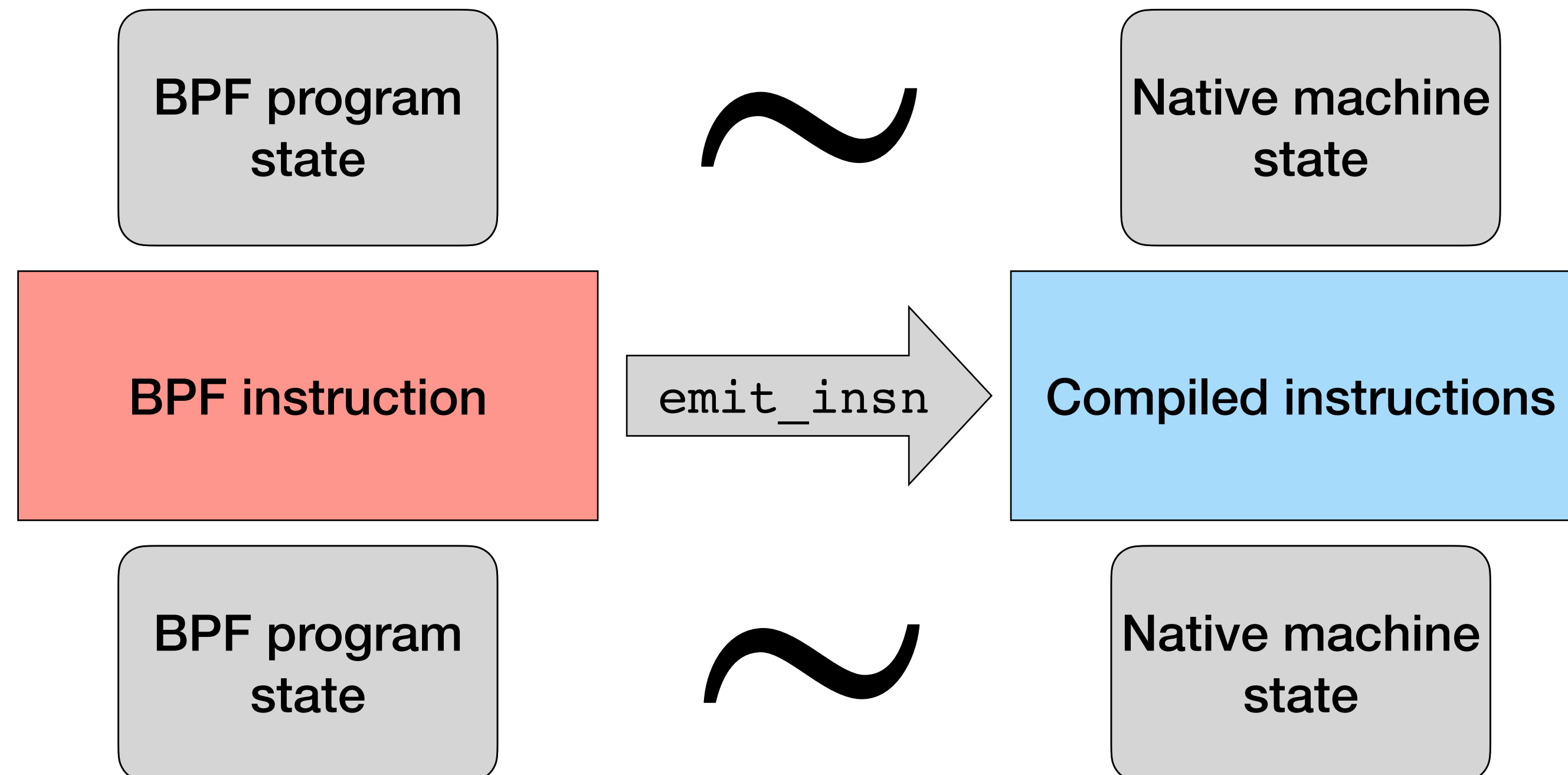
- Prologue correctness
 - The JIT prologue sets up BPF state (e.g., the stack) correctly
- Per-instruction correctness
 - The JIT produces correct machine code for each individual BPF instruction
- Epilogue correctness
 - The JIT epilogue tears down BPF state correctly and returns correct value

Specification: Breaking down to three parts

- Prologue correctness
 - The JIT prologue sets up BPF state (e.g., the stack) correctly
- Per-instruction correctness
 - The JIT produces correct machine code for each individual BPF instruction
- Epilogue correctness
 - The JIT epilogue tears down BPF state correctly and returns correct value

Specification: Per-instruction correctness

- Exploit per-instruction nature of JIT compilers
- Reason about correctness one BPF instruction at a time
- If BPF program state and machine state are related, interpreting the BPF instruction and executing the compiled code should produce related states



Why JIT correctness is hard (1/3)

Branches in JIT

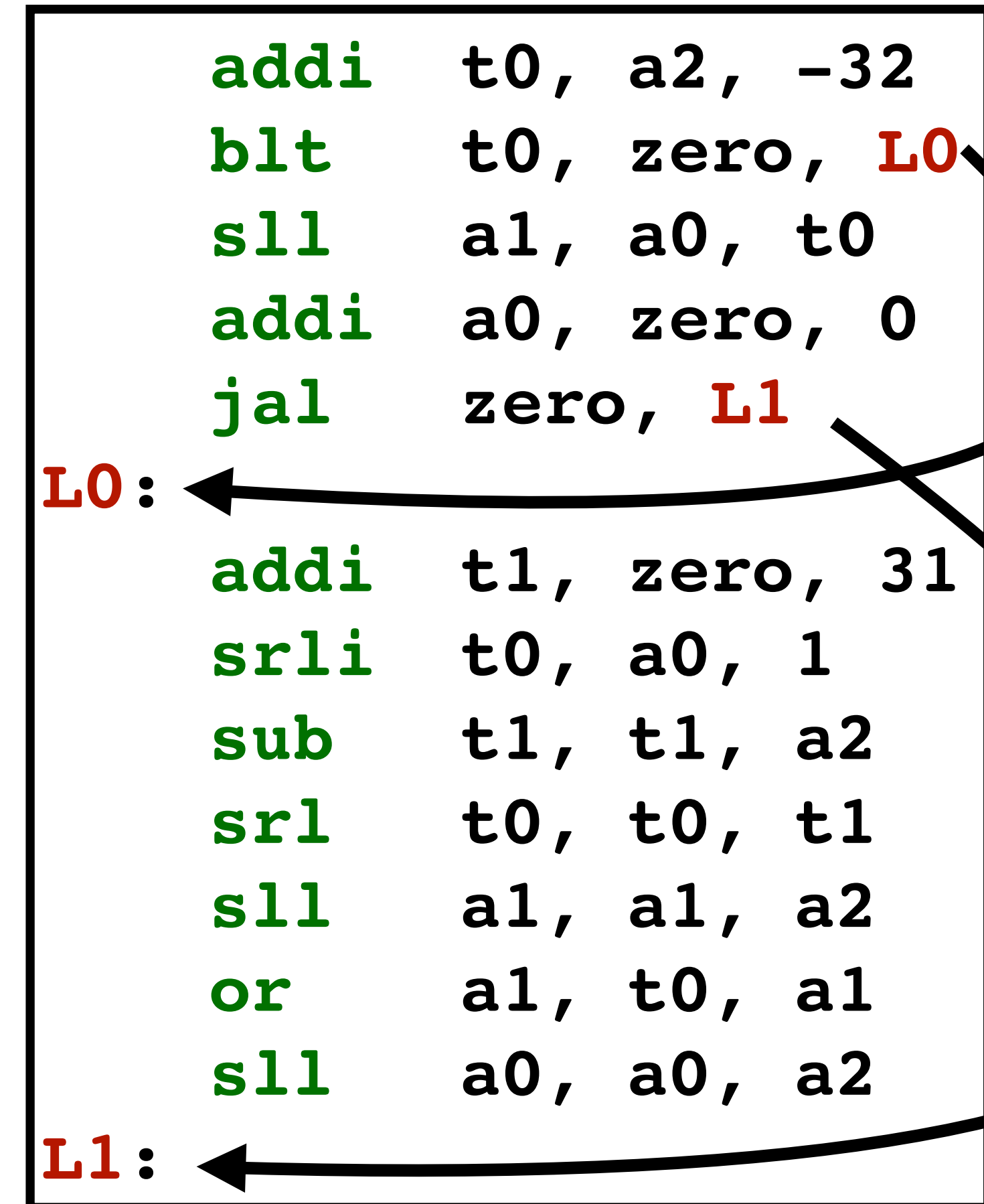
- JIT can contain branches based on immediate size, registers used, etc.
- Example: at least 7 branches for STX on x86-32
- Many possible paths to consider: JIT must be correct for all possible executions

```
case BPF_STX | BPF_MEM | BPF_B:
case BPF_STX | BPF_MEM | BPF_H:
case BPF_STX | BPF_MEM | BPF_W:
case BPF_STX | BPF_MEM | BPF_DW:
→ if (dstk)
    ...
    else
        ...
→ if (sstk)
    ...
    else
        ...
→ switch (BPF_SIZE(code)) {
    ...
}
→ if (is_imm8(insn->off))
    ...
    else
        ...
→ if (BPF_SIZE(code) == BPF_DW) {
→     if (sstk)
        ...
        else
            ...
→     if (is_imm8(insn->off + 4))
        ...
        else
            ...
}
break;
```

Why JIT correctness is hard (2/3)

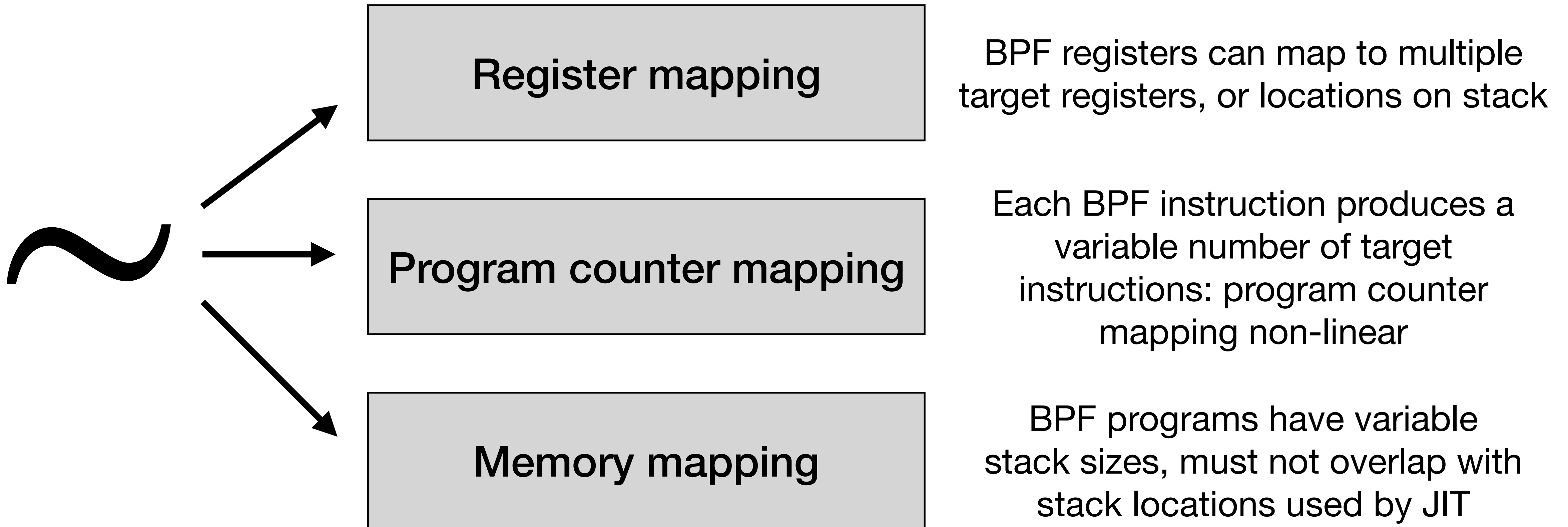
Branches in compiled code

- JIT will generate native code that contains branches
- Example: riscv32 JIT for “BPF_ALU64_REG(BPF_LSH, R1, R2)” needs branches to emulate 64-bit shift
- Must be correct for all executions in compiled code



Why JIT correctness is hard (3/3)

Need to map BPF state to machine state

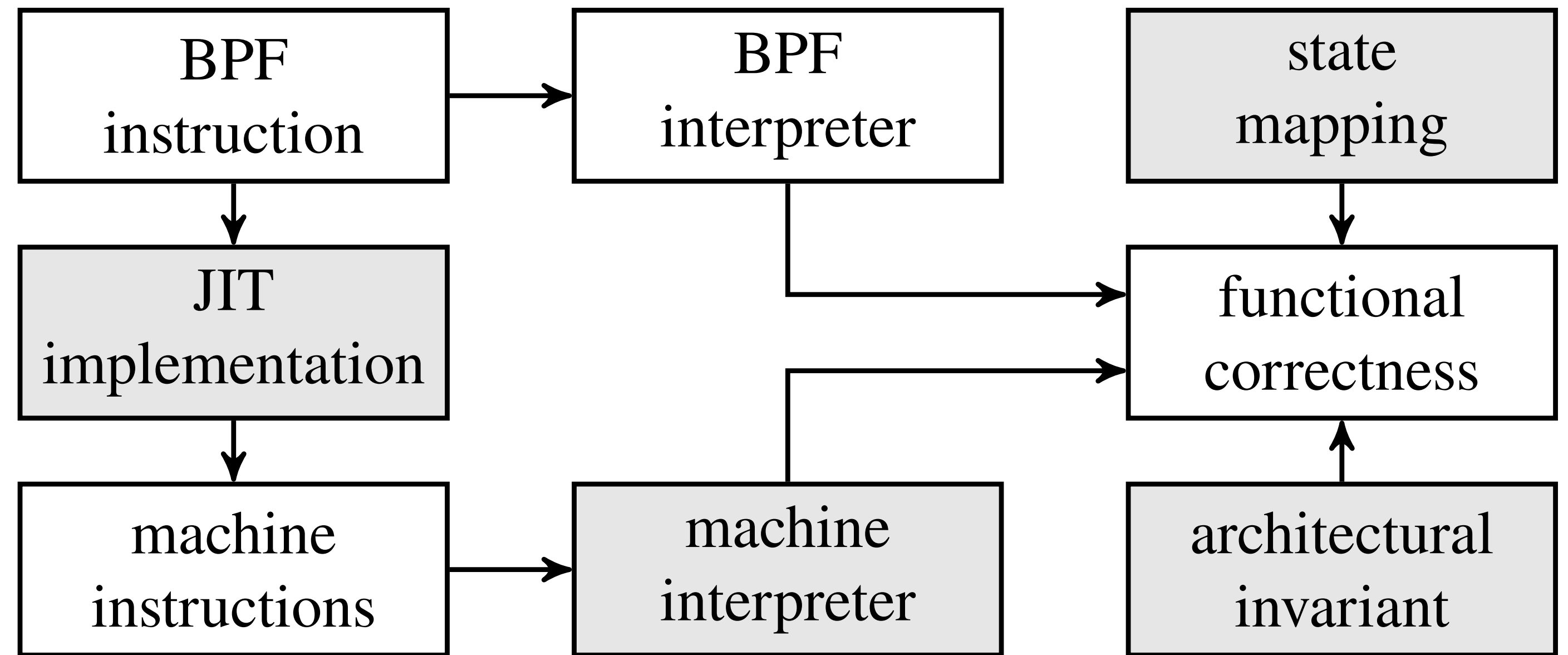


Outline

- Overview of how the BPF JITs in Linux work
- Case study of bugs in BPF JITs
- Overview of Jitterbug's JIT specification
- **How to use Jitterbug**
- Verification effort
- Demonstration
- Future directions for JIT verification

Proving JIT correctness

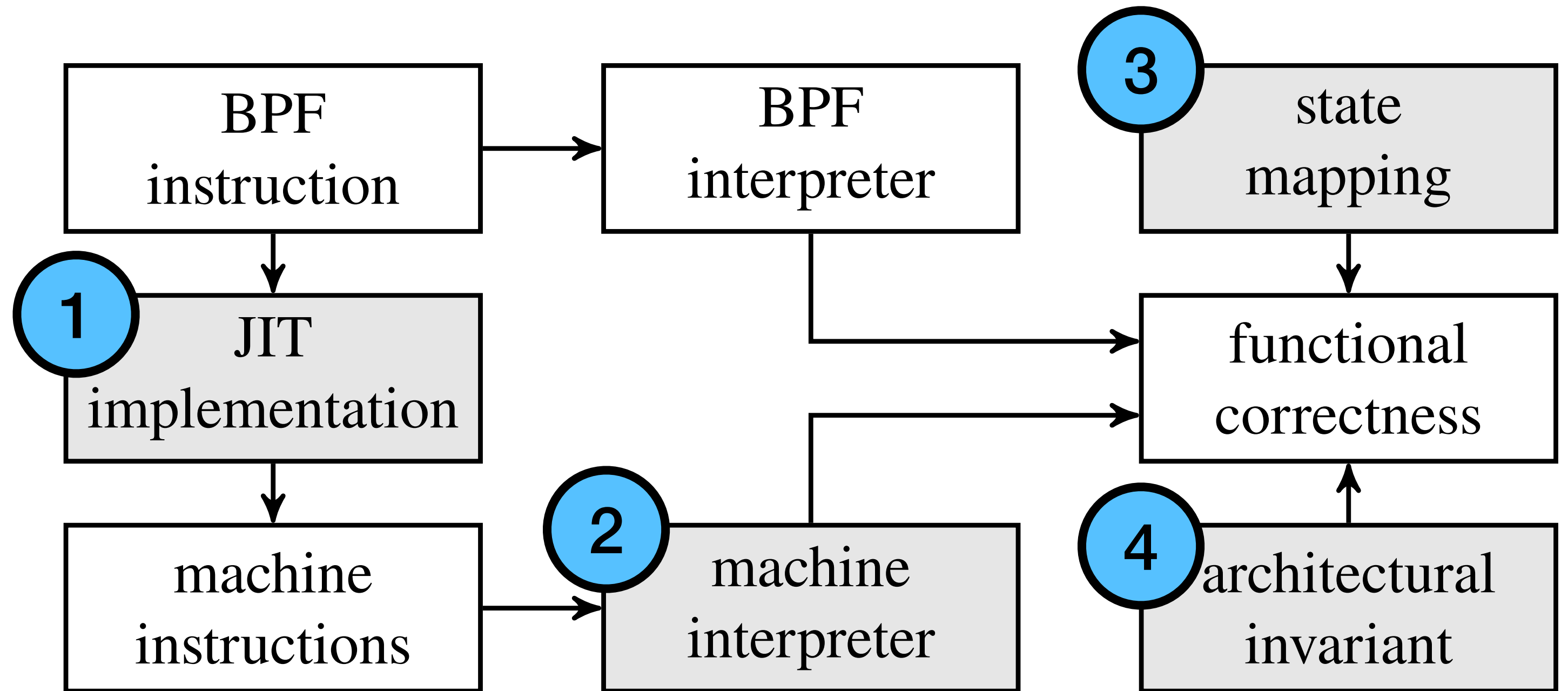
- Symbolic evaluation of JIT in DSL to produce (symbolic) machine code.
- Symbolic evaluation of machine interpreter on machine instructions
- Symbolic evaluation of BPF interpreter on BPF instruction
- Prove per-instruction functional correctness using SMT solver



Proving JIT correctness

What do JIT developers have to provide?

1. JIT implementation in Jitterbug's DSL
2. Symbolic interpreter for machine instructions
3. State mapping from BPF to machine state
4. Any additional architectural invariants (e.g., stack alignment)

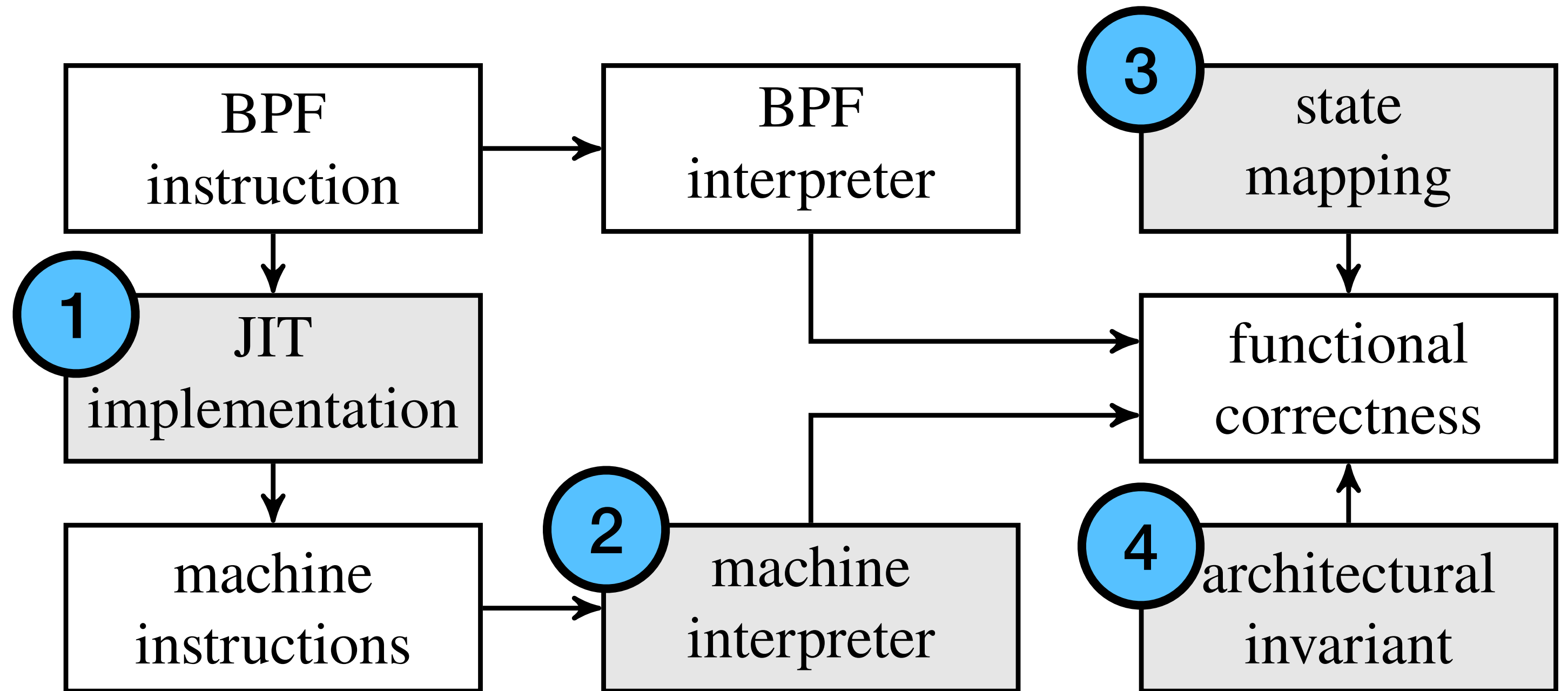


Shaded boxes: provided by JIT developer

Proving JIT correctness

What do JIT developers have to provide?

1. JIT implementation in Jitterbug's DSL
2. Symbolic interpreter for machine instructions
3. State mapping from BPF to machine state
4. Any additional architectural invariants (e.g., stack alignment)



Shaded boxes: provided by JIT developer

JIT implementation (1/3)

DSL for verification

- Jitterbug introduces a DSL for implementing+verifying JITs
 - Subset of Rosette with a one-to-one mapping to C code
 - Rosette: programming language for symbolic reasoning tools
 - Allows for finer-grained control over symbolic evaluation
- New riscv32 JIT: automated extraction from DSL to C code
- Existing JITs: manual translation from C code to DSL for verification
 - Future work: reasoning about C code directly

JIT implementation (1/3)

DSL for verification: example from riscv32 JIT

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

DSL implementation

Automated
extraction

```
void emit_alu_r64(const s8 *dst, const s8 *src,
                 struct rv_jit_context *ctx, 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;
  ...
}
```

C implementation

Machine interpreter (2/3)

Example: RISC-V add

- Jitterbug needs to have semantics of machine instructions
- Write interpreter for machine instructions in Rosette
- Automatically lifted to work on symbolic instructions

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

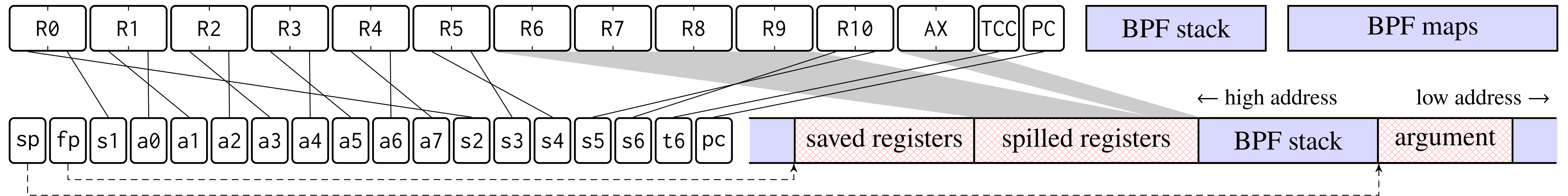
(define (interpret-add c rd rs1 rs2)
  (gpr-set! c rd
            (bvadd (gpr-ref c rs1)
                   (gpr-ref c rs2))))

(cpu-next! c))

(define (interpret c program)
  (define pc (cpu-pc c))
  (define insn (fetch pc program))
  (cond
    [(add? insn)
     (interpret-add c (add-rd insn)
                    (add-rs1 insn)
                    (add-rs2 insn))]
    ...))
```

State mapping (3/3)

- Developer needs to define mapping from BPF state to machine state
- Registers, program counter, memory
- Example: riscv32 registers
 - BPF registers map to two RISC-V registers, or to spilled register region on stack



Beyond bug finding

- Developed+verified new JIT for riscv32 architecture
- Developed+verified optimizations for existing JITs
 - bpf, arm: Optimize ALU64 ARSH X using orrpl conditional instruction
 - bpf, arm: Optimize ALU ARSH K using asr immediate instruction
 - bpf, arm64: Optimize AND,OR,XOR,JSET BPF_K using arm64 logical immediates
 - bpf, arm64: Optimize ADD,SUB,JMP BPF_K using arm64 add/sub immediates
 - bpf, riscv: Enable zext optimization for more RV64G ALU ops
 - bpf, riscv: Enable missing verifier_zext optimizations on RV64
 - bpf, riscv: Optimize FROM_LE using verifier_zext on RV64
 - bpf, riscv: Optimize BPF_JMP BPF_K when imm == 0 on RV64
 - bpf, riscv: Optimize BPF_JSET BPF_K using andi on RV64
 - bpf, riscv: Modify JIT ctx to support compressed instructions
 - bpf, riscv: Add encodings for compressed instructions
 - bpf, riscv: Use compressed instructions in the rv64 JIT
- See links / discuss after for details

Outline

- Overview of how the BPF JITs in Linux work
- Case study of bugs in BPF JITs
- Overview of Jitterbug's JIT specification
- How to use Jitterbug
- **Verification effort**
- Demonstration
- Future directions for JIT verification

Development effort

- Jitterbug is ~2,200 lines of Rosette
 - Excluding machine interpreters and architecture-specific specifications
- riscv32 JIT co-developed with Jitterbug
 - 10 months of development and shipped with Linux v5.7
 - Developed JIT specification + proof technique with riscv32 JIT
- Porting other architectures took several weeks each
- JIT optimizations several weeks each as well

JIT verification effort

	JIT implementation (loc)		Specification (loc)	
	C	DSL	JIT-specific	Interpreter
riscv32	1,580	1,119	316	1,195
riscv64	1,473	863	217	“
arm32	1,620	777	118	1,233
arm64	956	610	89	1,058
x86-32	1,683	991	107	2,274
x86-64	1,382	599	115	“

- Main effort is in writing the machine interpreter
- Effort to port to DSL and write specification is small (roughly 3 weeks each)
 - Added effort to keep DSL code in sync with C code

Verification performance

	Per-insn verification time (s)	
	mean	median
riscv32	31.7	13.6
riscv64	60.8	3.2
arm32	19.7	17.3
arm64	10.7	3.8
x86-32	14.8	12.5
x86-64	9.2	5.0

- Verification time usually <1 min. for one instruction: rapid feedback

Outline

- Overview of how the BPF JITs in Linux work
- Case study of bugs in BPF JITs
- Overview of Jitterbug's JIT specification
- How to use Jitterbug
- Verification effort
- **Demonstration**
- Future directions for JIT verification

Outline

- Overview of how the BPF JITs in Linux work
- Case study of bugs in BPF JITs
- Overview of Jitterbug's JIT specification
- How to use Jitterbug
- Verification effort
- Demonstration
- **Future directions for JIT verification**

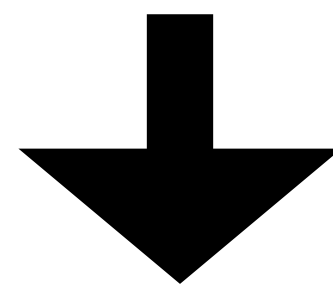
Future directions for JIT verification

- Currently have to manually translate code to DSL for verification
 - Hard to maintain with changing C code
- We want JIT developers to be able verify C code directly without knowing how to write Rosette
- Two main barriers:
 - Need additional research on scaling automated verification / symbolic evaluation of C code
 - Need a common JIT interface in the kernel amenable to the specification

Proposal: General JIT interface for verification

- Factor common code among architectures
- Example: We factored riscv32 and riscv64 to share `build_body`, etc.
- JIT interface now mirrors specification: prologue, single instruction, epilogue

```
struct bpf_prog *bpf_int_jit_compile(struct bpf_prog *prog);
```



```
void bpf_jit_build_prologue(struct rv_jit_context *ctx);  
int bpf_jit_emit_insn(const struct bpf_insn *insn,  
                      struct rv_jit_context *ctx, bool extra_pass);  
void bpf_jit_build_epilogue(struct rv_jit_context *ctx);
```

The BPF verifier

- The Linux BPF verifier is also hard to get right
- Verifier accepts code that it should reject
 - Safety bugs have been found in verifier in the past
- Verifier rejects code that it should accept
 - Can be hard to get verifier to accept BPF program
 - Complex interaction with LLVM when compiling C code to BPF
- Need more research on how to develop BPF verifiers that accept more safe programs while still rejecting the unsafe ones

Conclusion

- Jitterbug is a tool for automated verification of BPF JITs
 - Effective at finding bugs in existing JITs, developing new JITs, and optimizing existing JITs
- Ideally will be integrated into BPF JIT development process
- Code available at <https://github.com/uw-unsat/jitterbug>
- Acknowledgements: H. Peter Anvin, Daniel Borkmann, Will Deacon, Brian Gerst, Song Liu, Andy Shevchenko, Alexei Starovoitov, Björn Töpel, Jiong Wang, Yanqing Wang, Marc Zyngier