Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel

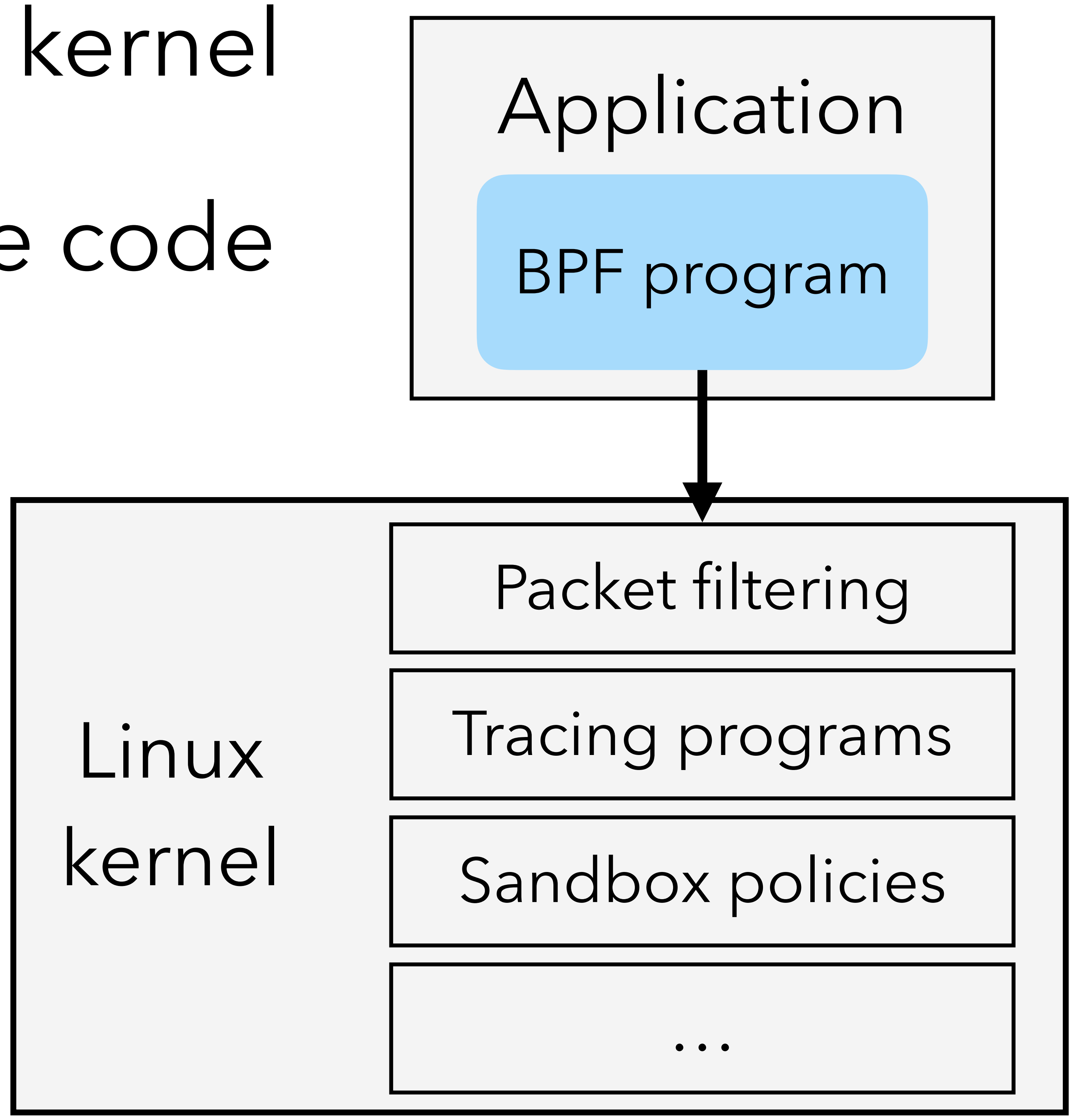
Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang
University of Washington

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

UNSAT

Goal: formally verified (e)BPF JITs in the Linux kernel

- BPF is widely deployed for extending the Linux kernel
- In-kernel JIT compilers translate BPF to machine code for performance
- Correctness is critical
  - Code runs directly in kernel
  - Makes decisions throughout kernel



Recent work on formal verification of systems

seL4

Ironclad Apps

CERTIKOS

fscq

Serval

- **This talk: how to apply formal verification to the BPF JITs in the Linux kernel**

Challenges: verifying BPF JITs in the Linux kernel

- Not designed for verification
  - Practical specification of JIT correctness
  - Prevents real-world bugs, enables optimizations
- Rapidly evolving JITs
  - Scale automated verification to JIT compilers
  - Catch up with new features being added
- Integration with kernel development
  - Write JITs in domain-specific language; extract to C code
  - Auditable without requiring formal methods background

Contributions

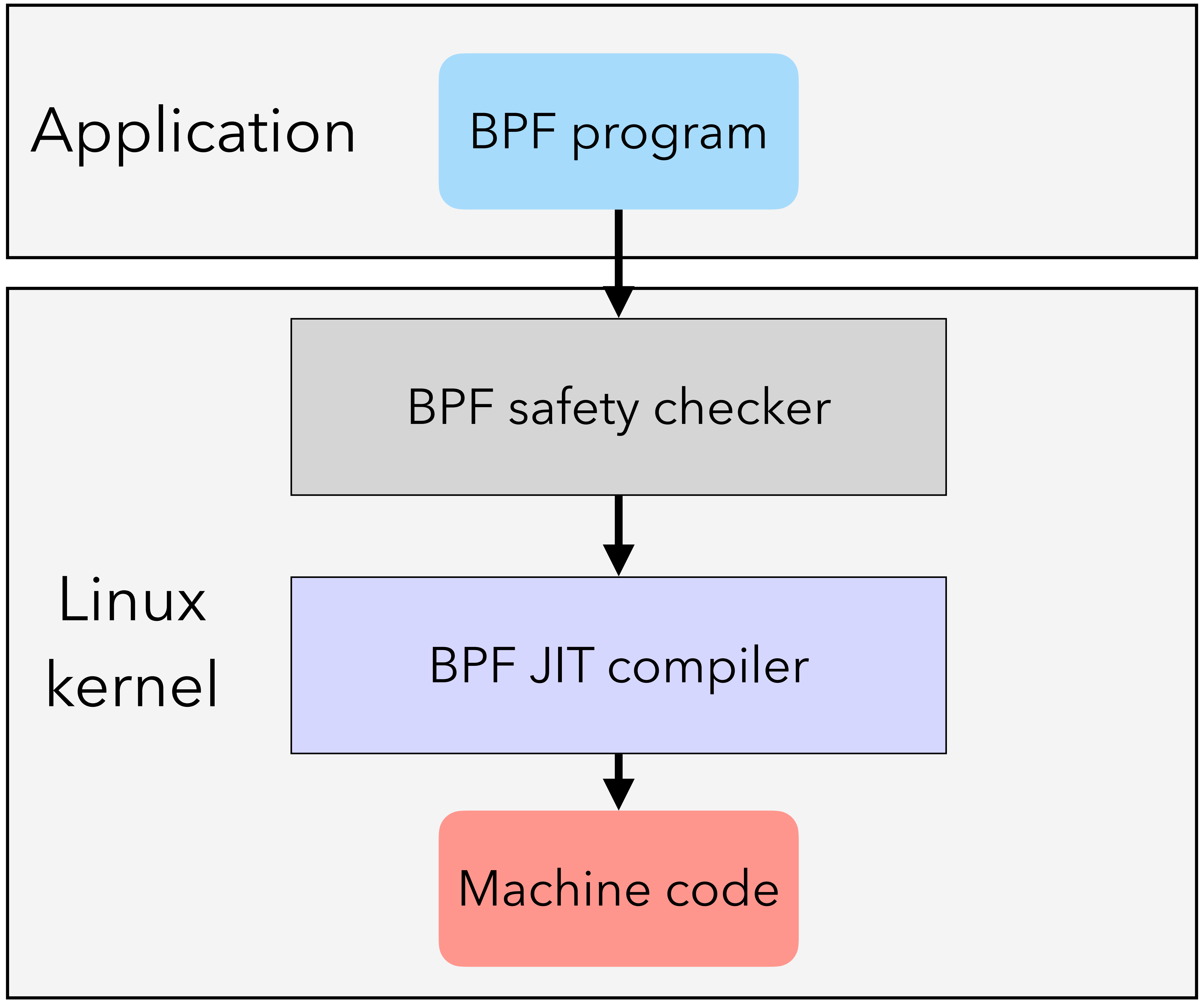
- Jitterbug: automated formal verification of BPF JITs
  - Specification for reasoning about JITs
  - Automated proof strategy
- Upstreamed changes in the Linux kernel
  - New BPF JIT for RISC-V (32-bit) since v5.7
  - Found and fixed new bugs and wrote new optimizations for existing JITs for x86 (32 & 64-bit), Arm (32 & 64-bit), RISC-V (64-bit)
- Clarification changes in RISC-V instruction-set manual

Contributions

- Jitterbug: automated formal verification of BPF JITs
  - **Specification for reasoning about JITs (this talk)**
  - Automated proof strategy (see paper for details)
- Upstreamed changes in the Linux kernel
  - New BPF JIT for RISC-V (32-bit) since v5.7
  - Found and fixed new bugs and wrote new optimizations for existing JITs for x86 (32 & 64-bit), Arm (32 & 64-bit), RISC-V (64-bit)
- Clarification changes in RISC-V instruction-set manual

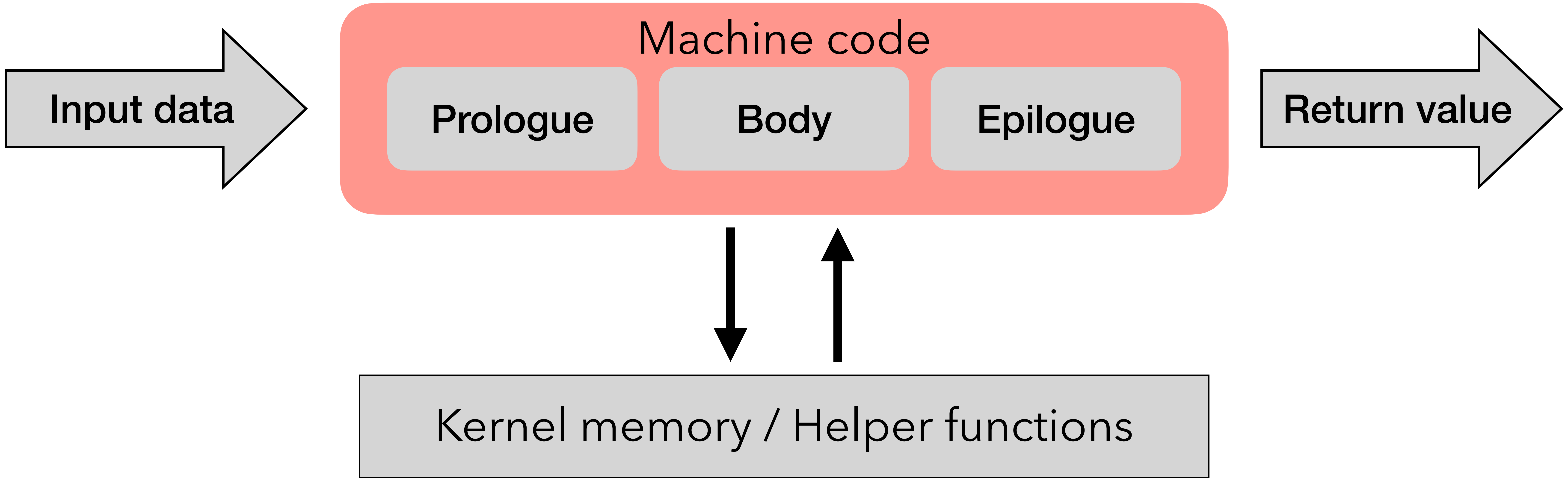
BPF JIT overview: compilation

- Application submits BPF program to kernel
- In-kernel checker ensures safety of BPF program
- JIT compiler translates to machine code



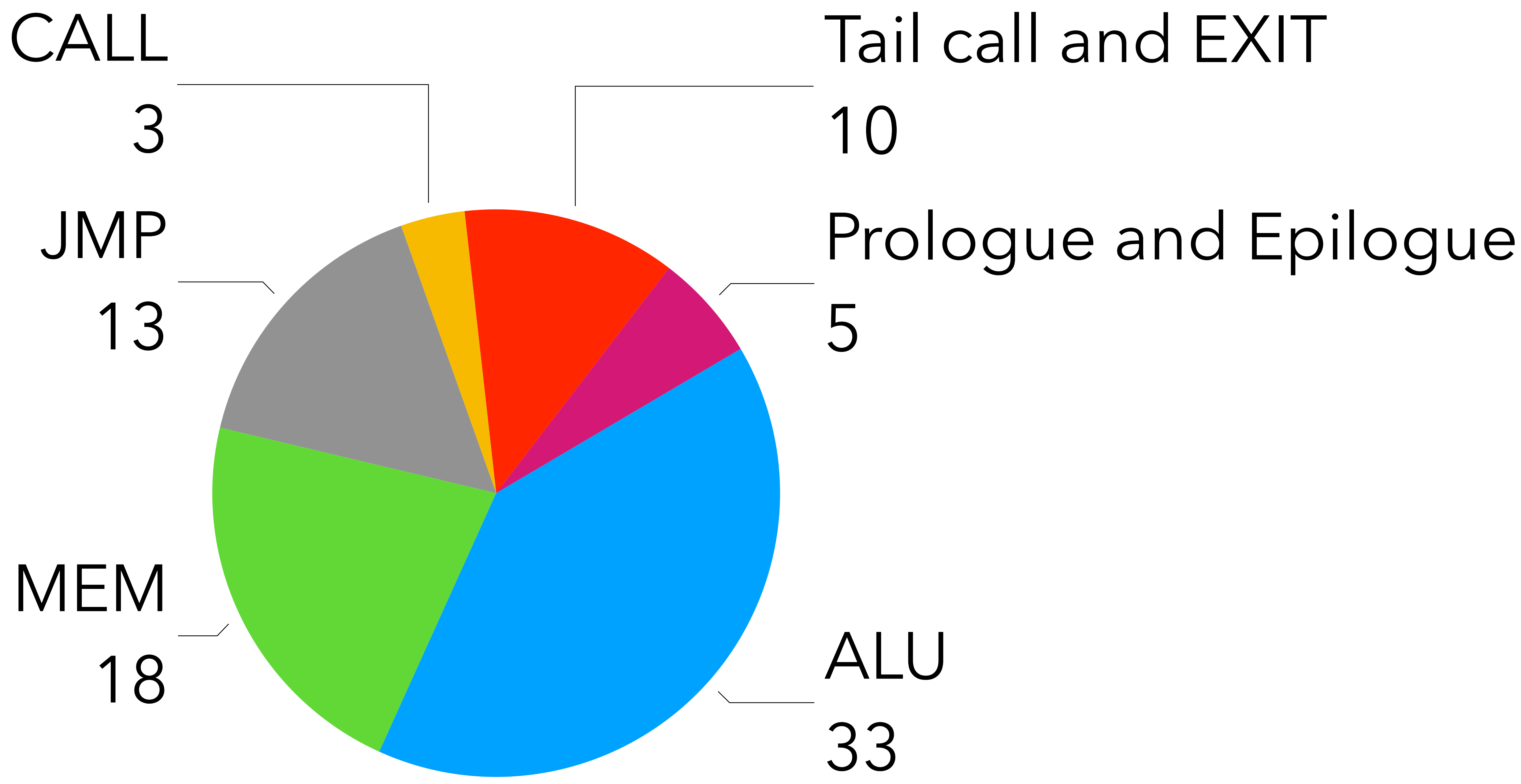
BPF JIT overview: run time

- Behaves like a regular kernel function
- Interacts with kernel through return value, memory accesses, function calls



Bugs in the BPF JITs in Linux: May 2014– Apr. 2020

- 82 JIT correctness bugs in x86 (32- & 64-bit), Arm (32- & 64-bit), RISC-V (64-bit)
- Bugs in every category of instructions
- Difficult to exhaustively test



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

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);
}
```

Bug: inverted check for optimization

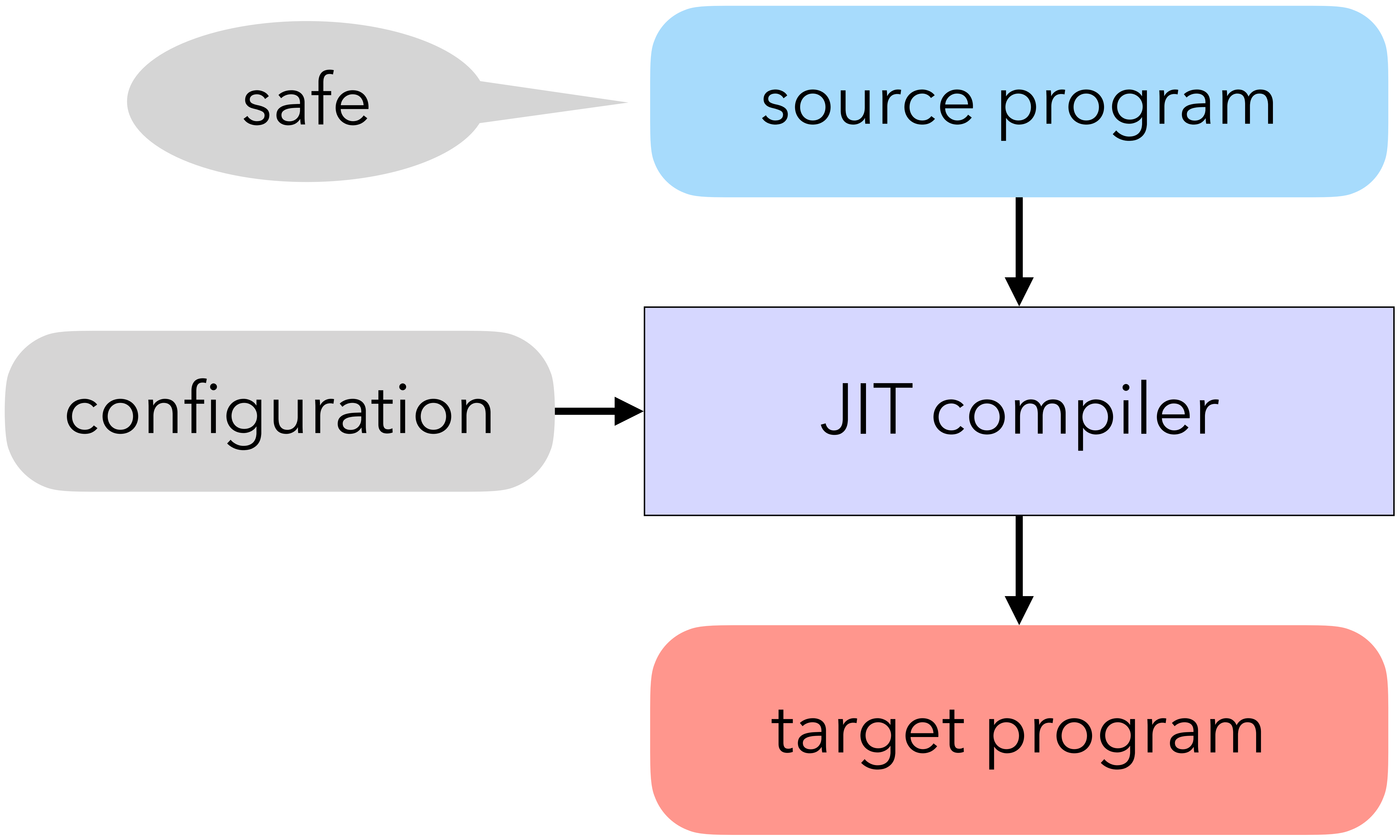
Bug: mov encoding missing 3 bytes of immediate

Writing correct JITs is difficult

- Must consider multiple levels
  - JIT configuration (e.g., optimizations)
  - Control flow in both JIT and emitted code
  - Semantics of source and target instructions
- Need a specification to rule out bugs
  - Restricted form of compiler correctness
  - Intuition: Machine code must behave equivalently to source BPF program

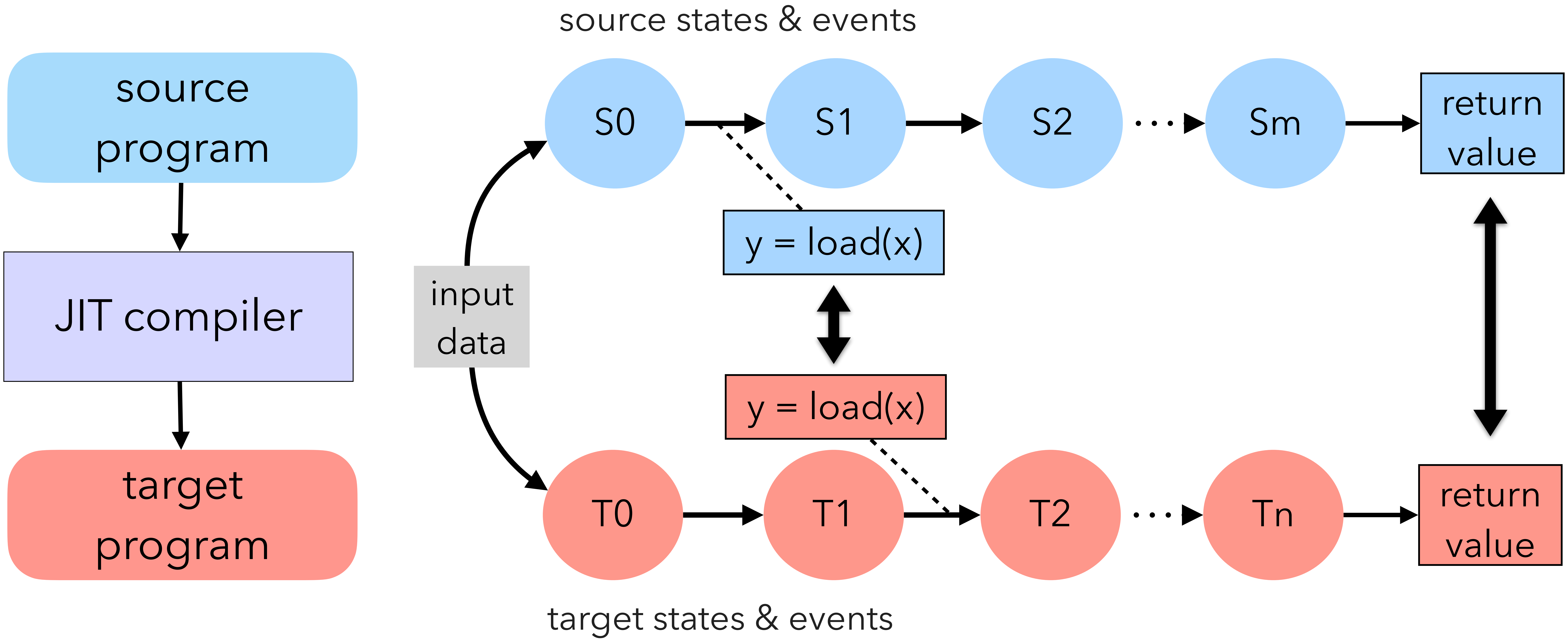
JIT correctness specification (1/3)

For any safe source program, JIT configuration (e.g., optimizations), and target program produced by JIT:



JIT correctness specification (2/3)

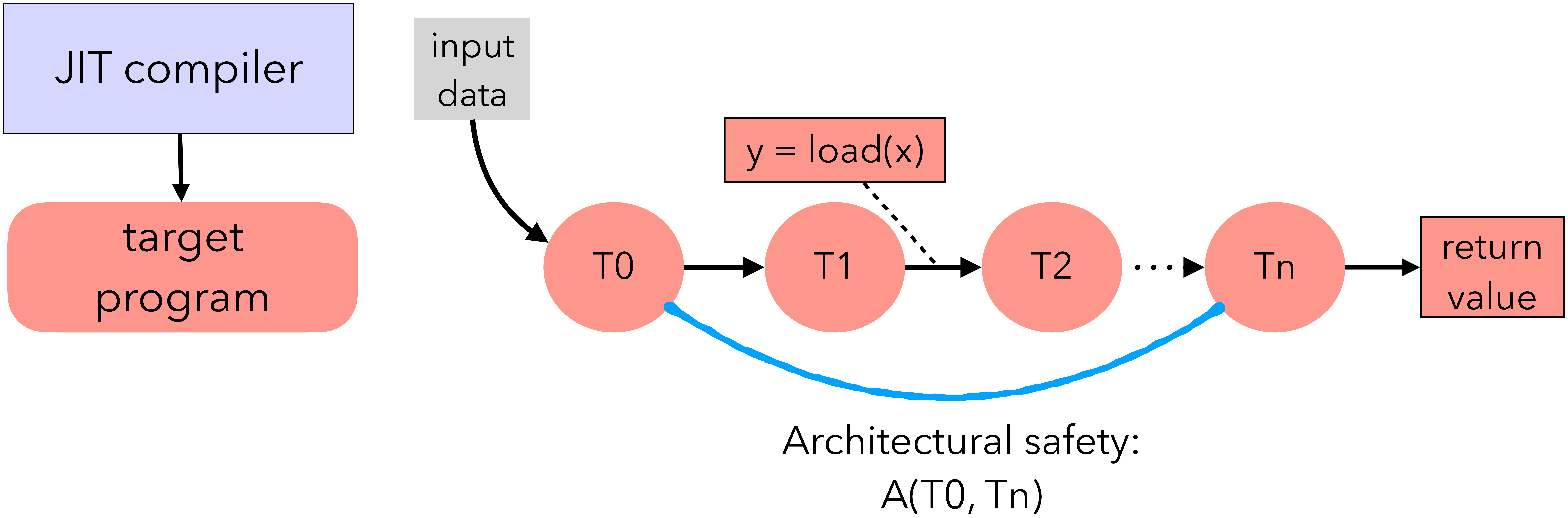
For any input data, execution of source and target programs produce same trace and return value



JIT correctness specification (3/3)

Execution of target program preserves *architectural safety*

Example: callee-saved registers preserved



JIT correctness pros & cons

Advantages:

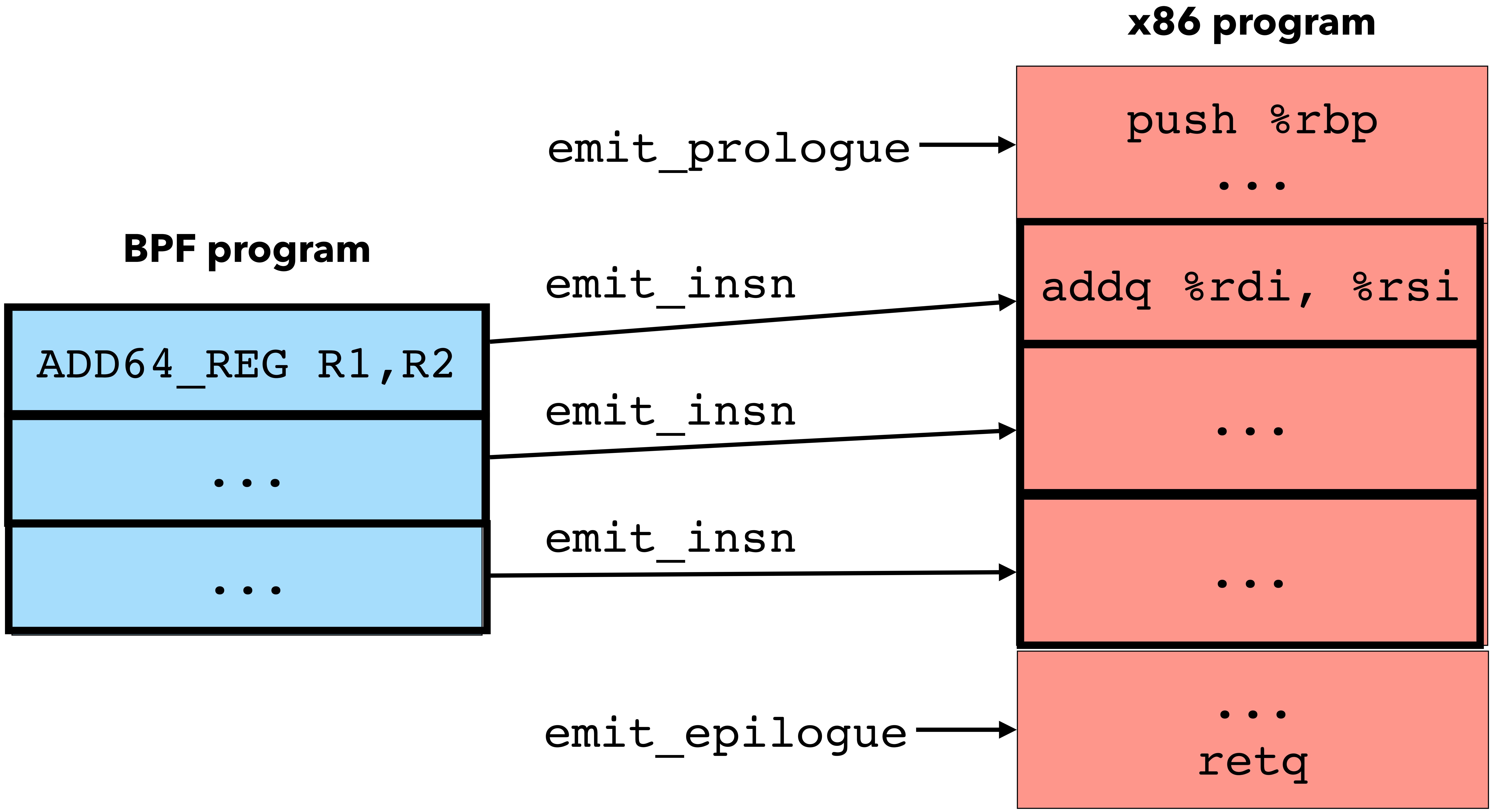
- Intuitive & effective at preventing bugs
- Tailored for in-kernel execution

Disadvantages:

- Not amenable to automated verification (hard to encode to SMT)

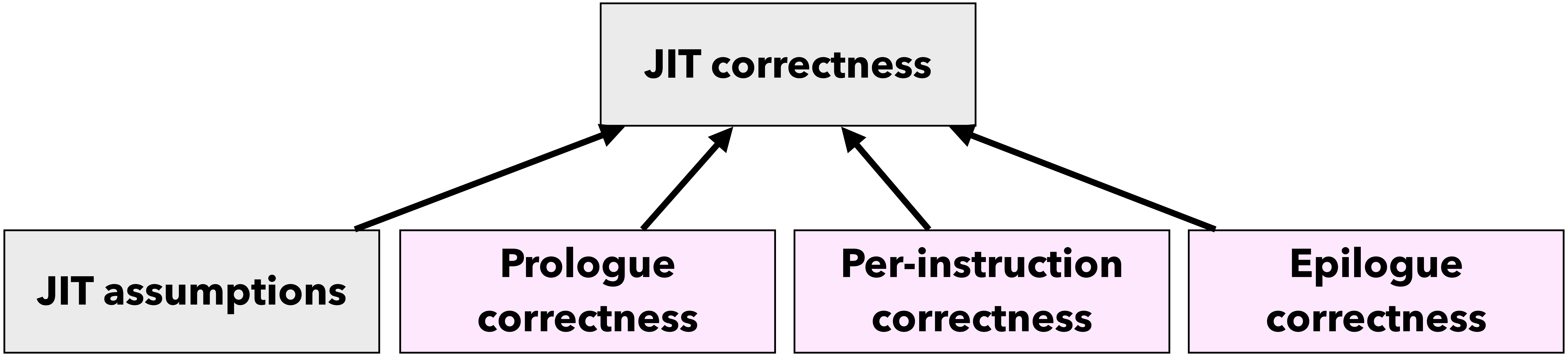
Exploit JIT structure: per-instruction translation

Existing JITs in Linux: `emit_prologue` + N × `emit_insn` + `emit_epilogue`



Breaking down JIT correctness

- Assume per-instruction JIT
- Correctness of each translation step implies JIT correctness
- Amenable to automated verification



Breaking down JIT correctness

- As
- Co
- An

JIT as

ue
ness

Scaling automated verification

- Requires reasoning about symbolic machine code produced by JIT
- Prior work works on concrete code
- See paper for details on how to scale

Developing and verifying the BPF JIT for RISC-V (32-bit)

- Written in DSL; extracted to C
- Started in 2019, co-developed with specification and proof technique over ~10 months
- Five iterations of code review; accepted in March 2020
- Automated verification enables catching up with features (e.g. zero-extension optimization, 100+ opcodes)

Improving existing JITs

- x86 (32- & 64-bit), Arm (32- & 64-bit), RISC-V (64-bit)
- Manually translate C code to DSL; less than 3 weeks each
- Found and fixed 16 new correctness bugs across 10 patches
- Developed and verified 12 optimization patches
- Demonstrates effectiveness of specification

Conclusion

- Case study of applying formal verification to BPF JITs in the Linux kernel
  - Jitterbug: specification + automated proof strategy
  - Developed new BPF JIT for RISC-V (32-bit)
  - Improved existing JITs with bug fixes and optimizations
- Extending automated verification to a restricted class of JIT compilers