eBPF Misbehavior Detection:
Fuzzing with a **Specification-Based Oracle**

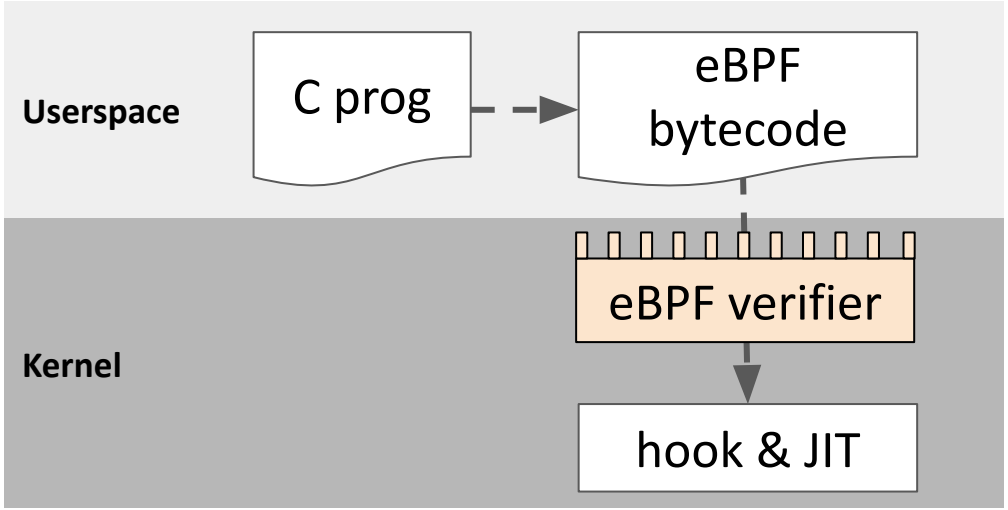
Tao Lyu Kumar Kartikeya Dwivedi
Thomas Bourgeat Mathias Payer Meng Xu* Sanidhya Kashyap

EPFL

* UNIVERSITY OF WATERLOO

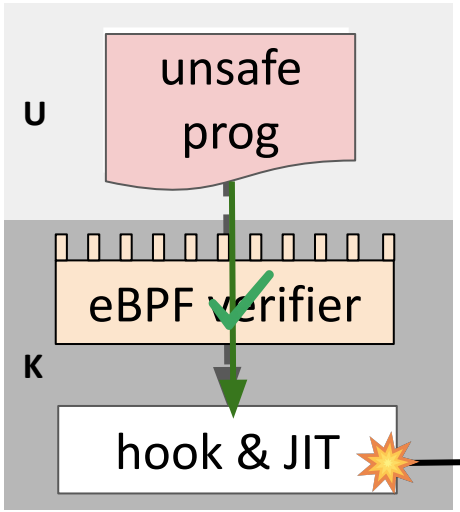
Kernel Extensions Are Essential

- Extending kernels at runtime with safety guarantees
- Widely deployed for observability, security, networking, etc.



Two Tiny eBPF Programs, Two Big Verifier Issues

- **Security issues**

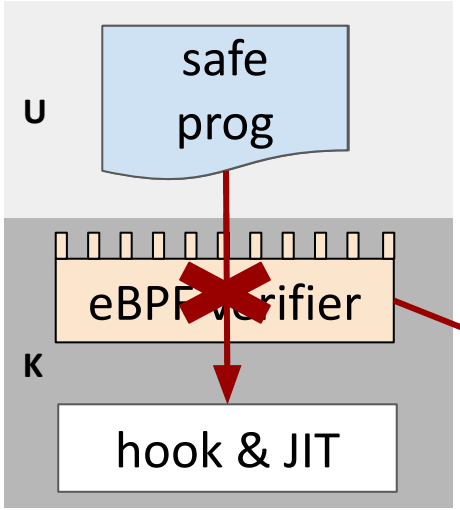



```
...
r1 = r10
if cond goto pc+2
*(u64*)(r1-120) = 0
goto pc+1
...
```

```
tlyu@syzkaller:~$ ./exp-get-root
[+] Created BPF maps and confirmed BPF is enabled on this kernel.
[+] OOB read succeeded! The kernel is vulnerable
[*] Now we get every rights. See you on the other side!
# id
uid=0(root) gid=0(root) groups=0(root),1000(tlyu) context=system_u:system_r:kernel_t:s0
# whoami
root
#
```

Two Tiny eBPF Programs, Two Big Verifier Issues

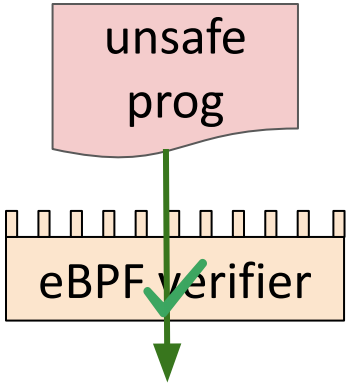
- **Usability issues**



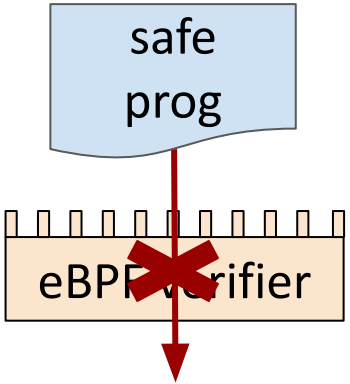

```
...
if (x >= 0 && x < 11) {
    res = compute_value();
    array[x] = res;
}
...
```

ERROR: unbounded memory access

So… What's Wrong with the eBPF Verifier?



Security issues



Usability issues

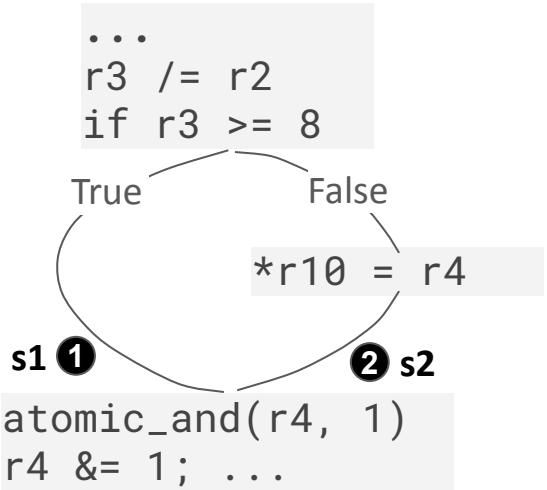
over-approximated state | incorrect checks | mis-implementations

Verifier Internals & Root Causes

verifier = path enumeration + abstract interpretation + safety checks

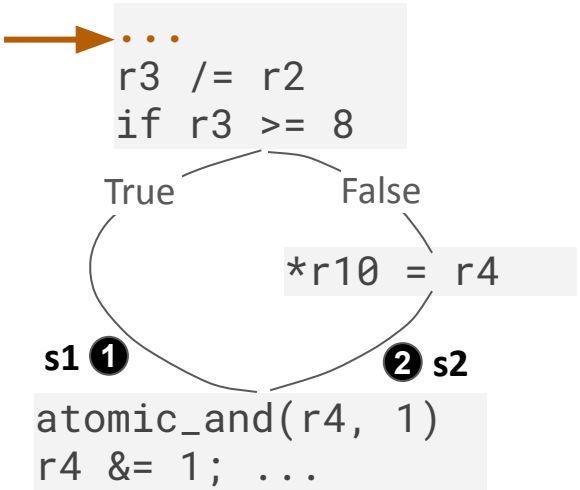
Verifier Internals & Root Causes

verifier = path enumeration + abstract interpretation + safety checks



Verifier Internals & Root Causes

verifier = path enumeration + abstract interpretation + safety checks



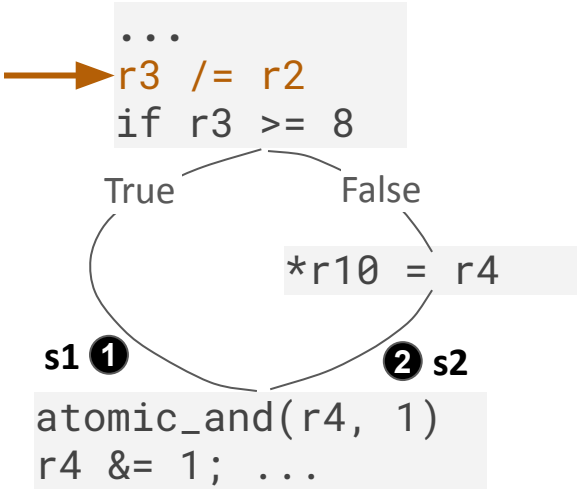
abstract state

r10: stack pointer, r4: a pointer

r2: [100, 255], r3: [200,255]

Verifier Internals & Root Causes

verifier = path enumeration + abstract interpretation + safety checks



over-approximated state

abstract state

r10: stack pointer, r4: a pointer

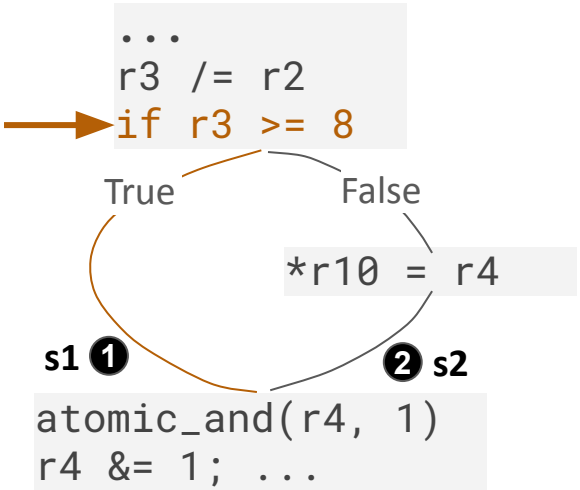
r2: [100, 255], r3: [0, 2^{64}-1]

r3: [0,2] VS [0, 2^{64}-1]

E.g., Out of bound access (OOB) if use r3 as an index for 20 byte array

Verifier Internals & Root Causes

verifier = path enumeration + abstract interpretation + safety checks



abstract state

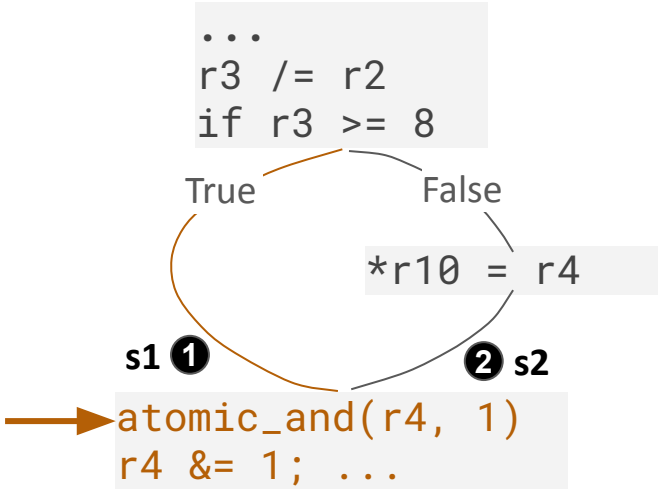
r10: stack pointer, r4: a pointer

r2: [100, 255], r3: [8, 2^64-1]

over-approximated state

Verifier Internals & Root Causes

verifier = path enumeration + abstract interpretation + safety checks



abstract state

r10: stack pointer, r4: a pointer

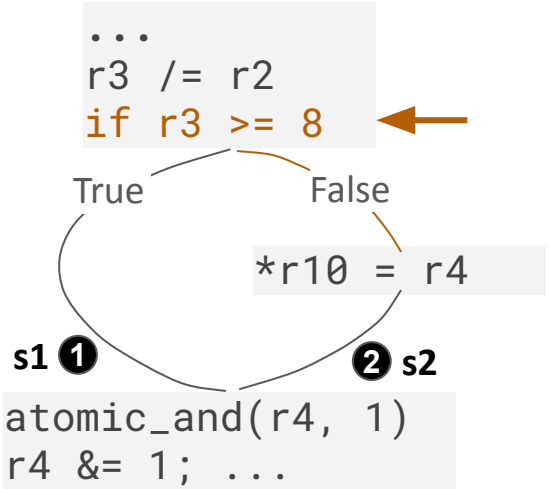
r2: [100, 255], r3: [8, 2^64-1]

over-approximated state

incorrect checks

Verifier Internals & Root Causes

verifier = path enumeration + abstract interpretation + safety checks



abstract state

r10: stack pointer, r4: a pointer

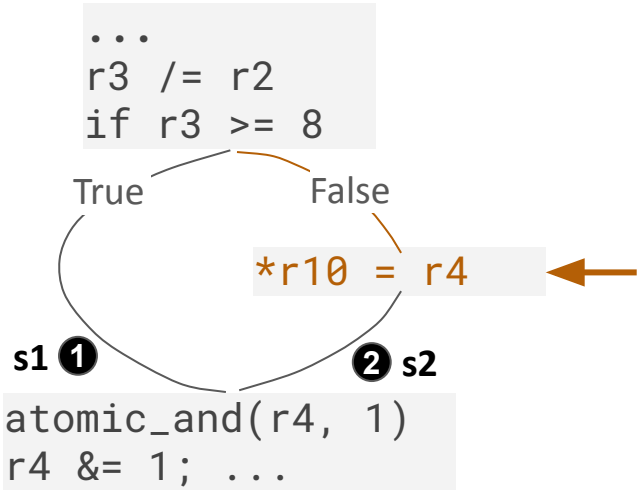
r2: [100, 255], r3: [0, 8)

over-approximated state

incorrect checks

Verifier Internals & Root Causes

verifier = path enumeration + abstract interpretation + safety checks



abstract state

r10: stack pointer, r4: a pointer

r2: [100, 255], r3: [0, 8)

safety checks

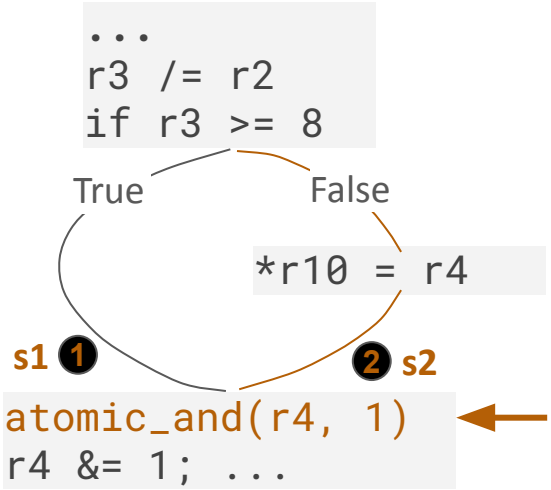
within-bound mem access

over-approximated state

incorrect checks

Verifier Internals & Root Causes

verifier = path enumeration + abstract interpretation + safety checks



abstract state

r10: stack pointer, r4: a pointer

r2: [100, 255], r3: [0, 8)

safety checks

within-bound mem access

path pruning

if s2 ⊆ s1, skip last code block *in* ❷

over-approximated state

incorrect checks

mis-implementations

Remedies to the Verifier's Issues

	Fuzzing with heuristics-based checker (e.g., KASAN)	
Over-approximated state	✗	
Incorrect safety checks	✗	
Mis-implementations	✓	

Remedies to the Verifier's Issues

	Fuzzing with heuristics-based checker (e.g., KASAN)	Partial verification (i.e., Agni)
Over-approximated state	✗	✓
Incorrect safety checks	✗	✗
Mis-implementations	✓	✓

They all have limited capability to address these issues!

Our Goal: Heuristic ➡ Systematic

CORE QUESTION

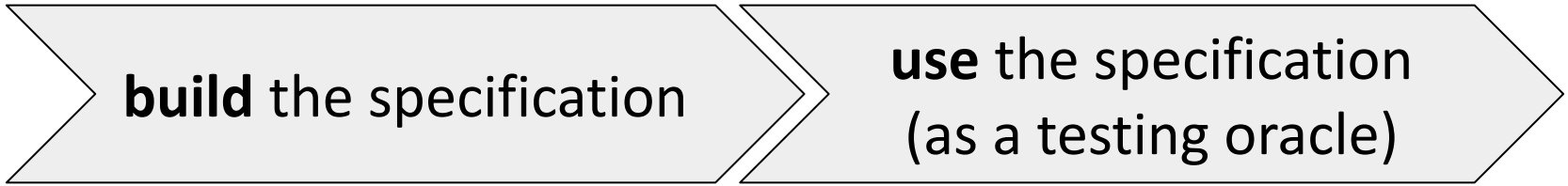
What is the **correctness** of the eBPF verifier?

SPECIFICATION

=

instruction semantics **+** safety constraints

Veritas Roadmap



What the Specification Looks Like?

- Specification = per-instruction semantics + safety constraints

```
function insn(s: State, insn: Instruction) : State

// safety constraints as preconditions
requires ...
{
  // semantics as a function body


}
```

What the Specification Looks Like?

- Specification = per-instruction semantics + safety constraints

```
function neg32(s: State, insn: Instruction) : State



{



}
```

What the Specification Looks Like?

- Specification = per-instruction semantics + **safety constraints**

```
function neg32(s: State, insn: Instruction) : State

// safety constraints as preconditions
requires insn.dst != R10 && ...
{



}
```

What the Specification Looks Like?

- Specification = per-instruction **semantics** + safety constraints

```
function neg32(s: State, insn: Instruction) : State

// safety constraints as preconditions
requires insn.dst != R10 && ...
{
  // semantics as a function body
  var new_val := bvnot32(arith_val(s, insn.dst));

}
```

Augmenting Semantics with Dynamic Types

- Prevents value-semantics mismatches
  - E.g., using an integer instead of a pointer to access memory

Augmenting Semantics with Dynamic Types

- Prevents value-semantics mismatches
  - E.g., using an integer instead of a pointer to access memory
- Each register and memory byte is associated with a dynamic type

```
datatype TYPEDVAL =
  | Uninit
  | Scalar(...)
  | PtrType(...)
  | PtrOrNullType(...)
```

Augmenting Semantics with Dynamic Types

- Prevents value-semantics mismatches
  - E.g., using an integer instead of a pointer to access memory
- Each register and memory byte is associated with a dynamic type
- Each instruction corresponds to a type rule = $\frac{\textit{required types of src operands}}{\textit{produced type of dst operands}}$

```
datatype TYPEDVAL =
  | Uninit
  | Scalar(...)
  | PtrType(...)
  | PtrOrNullType(...)
```

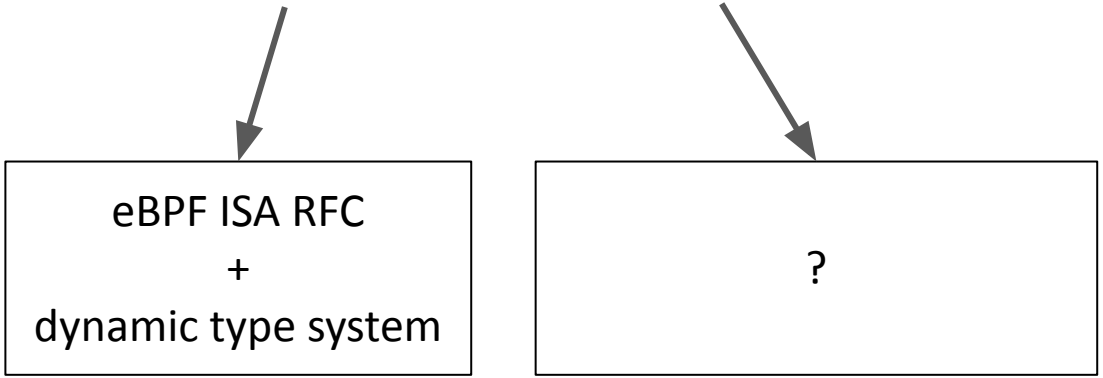
Augmenting Semantics with Dynamic Types

- Specification = (**augmented**) per-instruction semantics + safety constraints

```
function neg32(s: State, insn: Instruction) : State
requires get_reg_tv(s, insn.dst) != Uninit
// safety constraints as preconditions
requires insn.dst != R10
{
  // semantics as a function body
  var new_val := bvnot32(arith_val(s, insn.dst));
  new_state(s, insn.dst, Scalar(Normal, new_val))
}
```

Mid-talk Recap

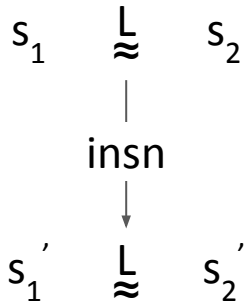
- Specification = per-instruction semantics + safety constraints



Where the Safety Constraints Come From?

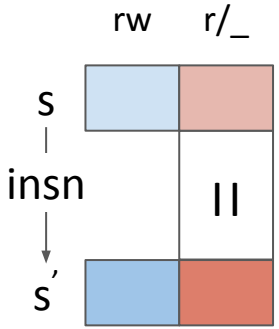
CIA triad security model for the OS kernel

Confidentiality



high-security data ⇏ low

Integrity



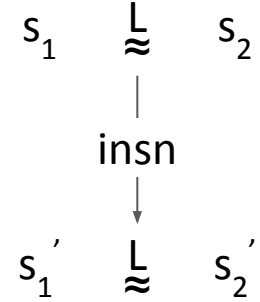
Non-writable data is unmodified

Availability

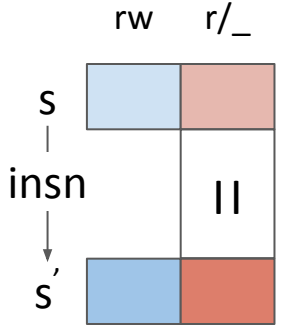
- Termination
- No resource exhaustion
- No memory crashes

Specification: Safety Constraints

Confidentiality



Integrity



Availability

- Termination
- No resource exhaustion
- No memory crashes

Control flow safety

Memory safety

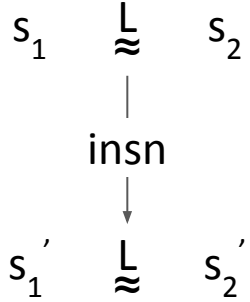
Resource safety

VM invariant

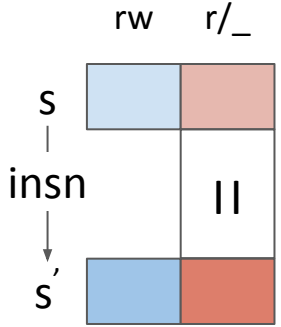
VM data safety

Specification: Safety Constraints

Confidentiality



Integrity



Availability

- Termination
- No resource exhaustion
- No memory crashes

Control flow safety | Memory safety | Resource safety

VM invariant | VM data safety

Please check out the detailed safety constraints in our paper

Specification Soundness and Completeness

Formal

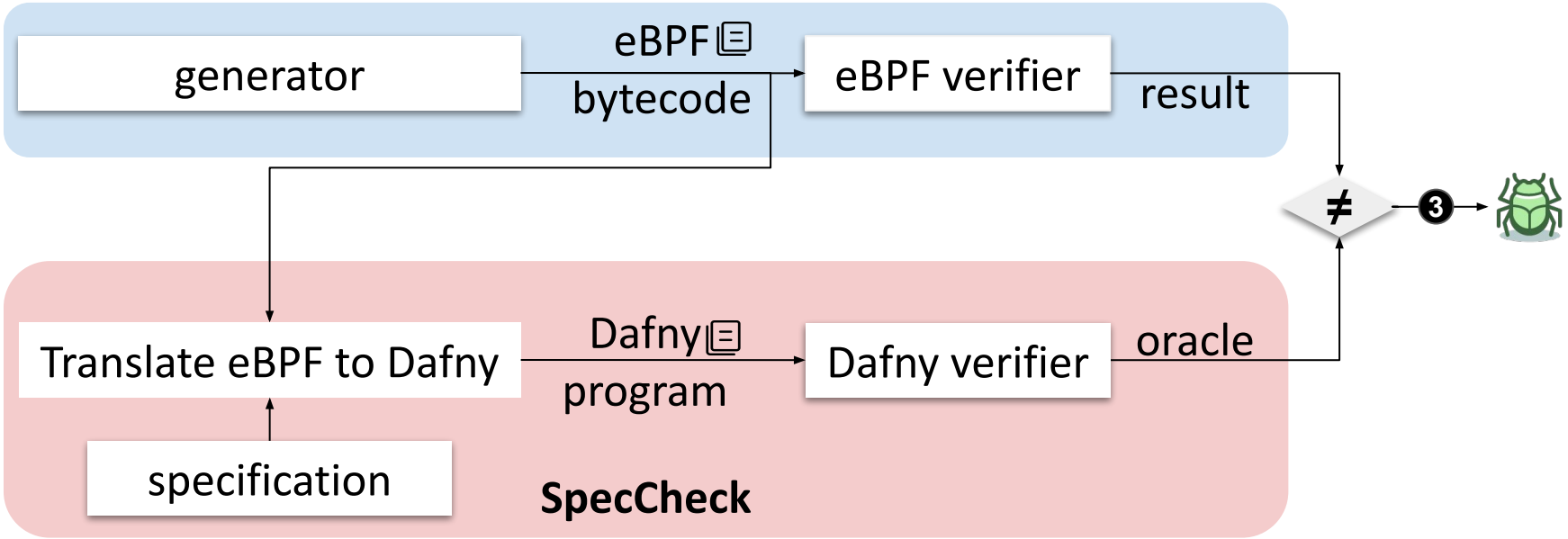
Confidentiality	**I**ntegrity	**A**vailability
Soundness ✓	Soundness ✓	

Empirical

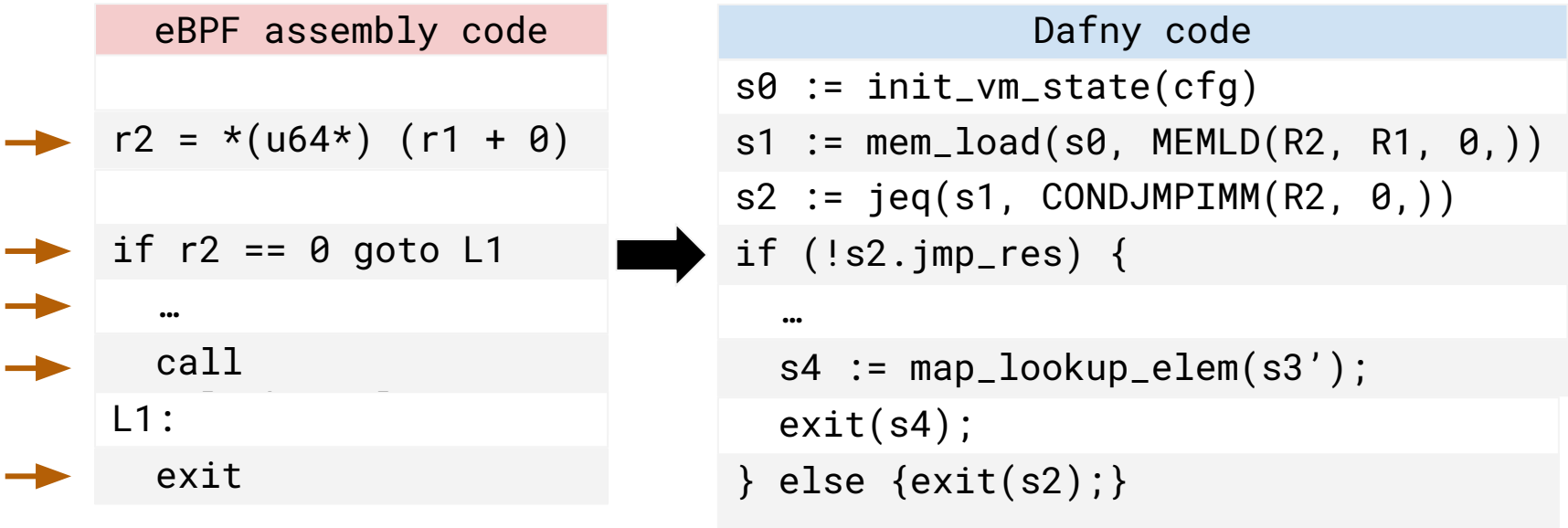
- No false alarms in the fuzzing campaign
- eBPF selftests: pass, or discrepancy with a known cause

Specification ➡ Improving the verifier's correctness

Specification as a Fuzzing Oracle



Translating eBPF Bytecode to Dafny



Evaluation

- Uncovered 13 new bugs
  - 3 security issues + 10 usability issues
- Comparison
  - SpecCheck detects all existing 14 bugs
  - Baselines cannot detect bugs found by SpecCheck
- Throughput
  - 23-25 tests / second
  - Trade throughput for better bug-detection capability

Conclusion

- **The Linux eBPF verifier is critical but vulnerable**
- **Fuzzing with a specification-based oracle**
  - Specification = semantics + safety constraints
  - eBPF bytecodes $\xrightarrow{spec}$ Dafny programs $\xrightarrow{Dafny\ verifier}$ oracles
- **Future use cases**
  - Proof-carrying code (PCC); verifying user-defined properties
- **Artifact**: https://github.com/rs3lab/veritas

Thank you!