Lazy Abstraction Refinement with Proof

Hao Sun, Zhendong Su
ETH Zurich

ETH zürich

In-Kernel Verification

Goal-1: always proves the property if it holds (completeness)

Goal-2: never proves the property if it does not hold (soundness)

A sound and complete verifier

Goal-1: always proves the property if it holds (completeness)

Goal-2: never proves the property if it does not hold (soundness)

Decidability

A sound and complete verifier

Always proves the property if it holds

The Verifier

- Verify programs with abstract interpretation-based techniques
- Tracks the program state, e.g., range, in interval and tnum
- Equivalence classes of values detection using identity tracking
- Tacking the stack states for register/value spilling and filling
- Heuristically pruning via comparing with known safe states
- Fix point computation with loop contract

Imprecisions

The Interval Domain

Operation**:**

Addition: **If** $x \in [a_1, b_1]$ **and** $y \in [a_2, b_2]$, **then** $x + y \in [a_1 + a_2, b_1 + b_2]$

Subtraction: $x - y \in [a_1 - b_2, b_1 - a_2]$

Information Loss:

- Minimal Loss: For addition and subtraction, the interval domain provides a tight approximation.

Example:

- If $x \in [0,1]$ and $y \in [0,1]$, then $x + y \in [0,2]$, precisely covering all possible sums of x and y
- However, $x - y$ is unbounded in the unsigned domain

Significant Loss: under/overflow leads to unbound ranges

The Interval Domain

Operation:

Multiplication: For $x \in [a_1, b_1]$ and $y \in [a_2, b_2]$, compute all products of interval endpoints:

$$P = \{a_1a_2,\ a_1b_2,\ b_1a_2,\ b_1b_2\}$$

Then $x \times y \in [\min(P), \max(P)]$

Information Loss:

- **Significant Loss**: cross positive and negative values.
- **Significant Loss:** potential overflow, e.g., greater than U16/U32_MAX

The Interval Domain

AND: over-approximate the higher bound, obtain the lower bound from tnum

OR: over-approximate the lower bound, obtain the higher bound from tnum

XOR: obtain the bounds from tnum

LSH: shift bound if top bit not shifted out, otherwise unbound/tnum

RSH: lose all sign information

ARSH: lose all unsigned information, obtain from tnum

For all shift operations, losing everything with variable operand

The Tnum Domain

Bitwise Operations:

- performed per bit using extended truth tables that handle the unknown state (⊤).

Information Loss:

- Minimal Loss: highly precise for bitwise operations, as it tracks each bit individually

Example:

- Let $x = [1,0, \top ,1]$
- Let $y = [\top,1,0,1]$
- Compute z = x & y

Truth table for AND

1	⊤	⊤
0	1	0
⊤	0	0
1	1	1

The Tnum Domain

Addition:

- bit-level computations with carries.
- unknown bits and carries propagate uncertainty.

Information Loss:

- **Significant Loss**: Even a single unknown bit can cause multiple bits in the result to become unknown due to carry propagation.

Example:

- Let $x = [1,0,1,1]$

  $y = [0,1, \top ,0]$
- Compute z = x + y
- Result: z = [⊤, ⊤, ⊤, 1]

Bit Position	x_i	y_i	Carry-In	Sum	Result Bit	Carry-Out
LSB (3)	1	0	0	1	1	0
2	1	⊤	0	⊤	⊤	⊤
1	0	1	⊤	⊤	⊤	⊤
MSB (0)	1	0	⊤	⊤	⊤	⊤

The Tnum Domain

Subtraction:

- bit-level computations with borrows.
- unknown bits and borrows propagate uncertainty.

Information Loss:

- **Significant Loss**: Similar to addition, uncertainty in bits and borrows leads to multiple unknown bits in the result.

Example:

- Let $x = [1, \top, 1, 0]$
  $y = [0, 1, 0, 1]$
- Compute z = x - y
- Result: z = [⊤, ⊤, 0, 1]

Bit Position	x_i	y_i	Borrow-In	Difference	Result Bit	Borrow-Out
LSB (3)	0	1	0	-1	1	1
2	1	0	1	0	0	0
1	⊤	1	0	⊤	⊤	⊤
MSB (0)	1	0	⊤	⊤	⊤	⊤

The Tnum Domain

Multiplication:

- computed through partial products, shift, and addition.
- unknown bits in operands lead to many unknown bits in the result.

Information Loss:

- **Significant Loss**: unknown bits cause entire rows in the multiplication table to be uncertain.

Example:

- Let $x = [0,\top]$ and $y = [1,1]$
- Compute z = x * y
  - Partial product
    - $x_1 \times y = 0 \times [1,1] = [0,0]$
    - $x_0 \times y = \top \times [1,1] = [\top, \top]$
  - Shift and add
    - Shifted $x_1 \times y = [0,0]$
  - Result: $[\top, \top]$

ETH zürich

Variable Relationship

Interval and tnum treat variables independently, **losing** relationship information.

- r0 and r1 contain the same input source
- (r1 >> 1) <= 4 implies r0 <= 9

```
1: r0 &= 255                ; R0=scalar(umin=0,umax=255,var_off=(0x0; 0xff))
2: r1 = r0                  ; R0=scalar(id=1…) R1_w=scalar(id=1…)
…
6: r1 >>= 1                 ; R1=scalar(umin=0,umax=127,var_off=(0x0; 0x7f))
7: if r1 > 0x4 goto pc+2    ; R1=scalar(umin=0,umax=4,var_off=(0x0; 0x7))
8: r2 += r0
9: r3 = *(u8 *)(r2 +0)
invalid variable-offset read from stack R2 var_off=(0x0; 0xff) off=-16 size=1
```

Variable Relationship

Interval and tnum treat variables independently, **losing** relationship information.

- r0 ∈ [0,15], r4 = 15 - r0
- *(u8*) (r1 + r0 + r4) => *(u8*) (r1 + 15)

```
; R1 = fp(off=-16)
1: r0 &= 0xf          ; R0_w=scalar(umin=0,umax=15)
2: r1 += r0           ; R1_w=fp(off=-16,umax=15)
3: r4 = 0xf           ; R4_w=15
4: r4 -= r0           ; R4_w=scalar(umin=0,umax=15)
; the offset is r0+(15-r0)
5: r1 += r4           ; R1_w=fp(off=-16,smax=30)
6: r0 = *(u8 *)(r1 +0)
invalid variable-offset read from stack R1
```

```
struct {
   __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
   __uint(max_entries, 1);
   __type(key, __u32);
   __type(value, __u64[MAX_STACK_RAWTP]);
   __type(value, __u64[2* MAX_STACK_RAWTP]);

} rawdata_map SEC(".maps");

SEC("raw_tracepoint/sys_enter")
int bpf_prog1(void *ctx)
{
…

   max_len = MAX_STACK_RAWTP * sizeof(__u64);
   /* write both kernel and user stacks to the same buffer */
   raw_data = bpf_map_lookup_elem(&rawdata_map, &key);
   if (!raw_data)
      return 0;

   usize = bpf_get_stack(ctx, raw_data, max_len, BPF_F_USER_STACK);
   if (usize < 0)
      return 0;

   ksize = bpf_get_stack(ctx, raw_data + usize, max_len - usize, 0);
   if (ksize < 0)
      return 0;

…
}
```

Existing Efforts

Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers
Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, Santosh Nagarakatte
Proceedings of CGO '22

Fixing Latent Unsound Abstract Operators in the eBPF Verifier of the Linux Kernel
Matan Shachnai, Harishankar Vishwanathan, Srinivas Narayana, Santosh Nagarakatte

Simple and precise static analysis of untrusted Linux kernel extensions
Gershuni Elazar, et. al.
PLDI '19

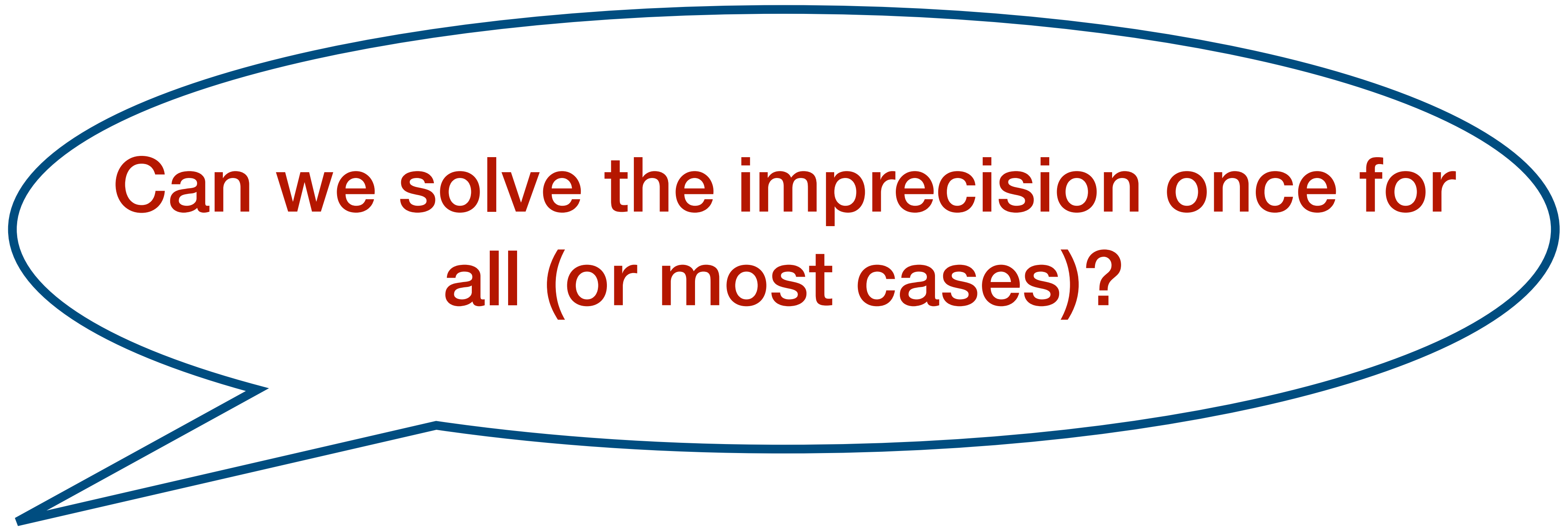
Automatic Discovery of Linear Constraints among Variables of a Program
Patrick Cousot and Nicolas Halbwachs.
POPL '78

The Octagon Abstract Domain
Antoine Miné
Higher-Order Symb Comput

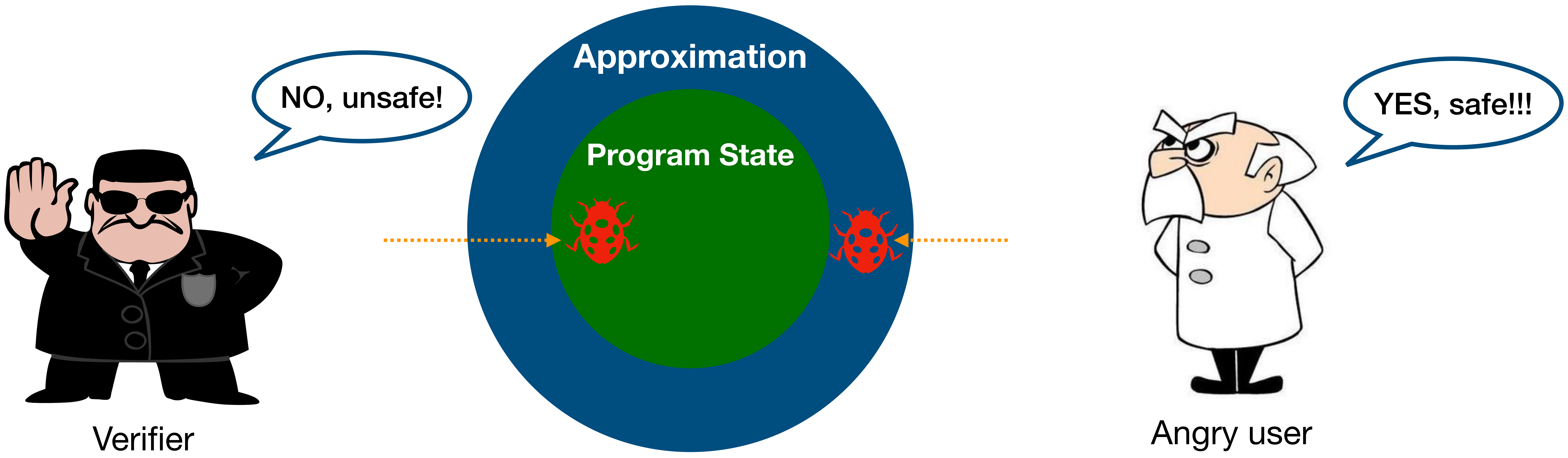
BPF register bounds logic and testing improvements
Andrii Nakryiko

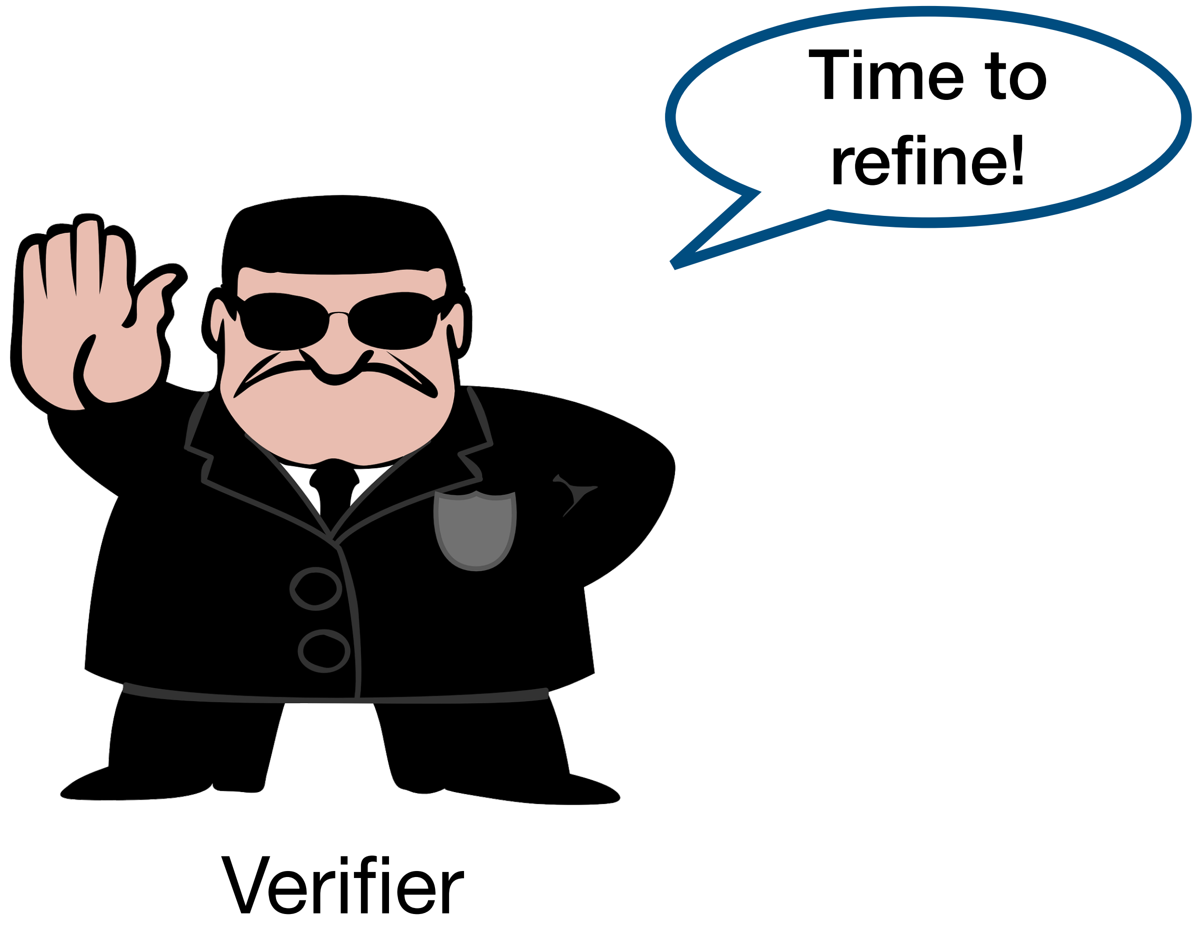
bpf: Track equal scalars history on per-instruction level
Eduard Zingerman

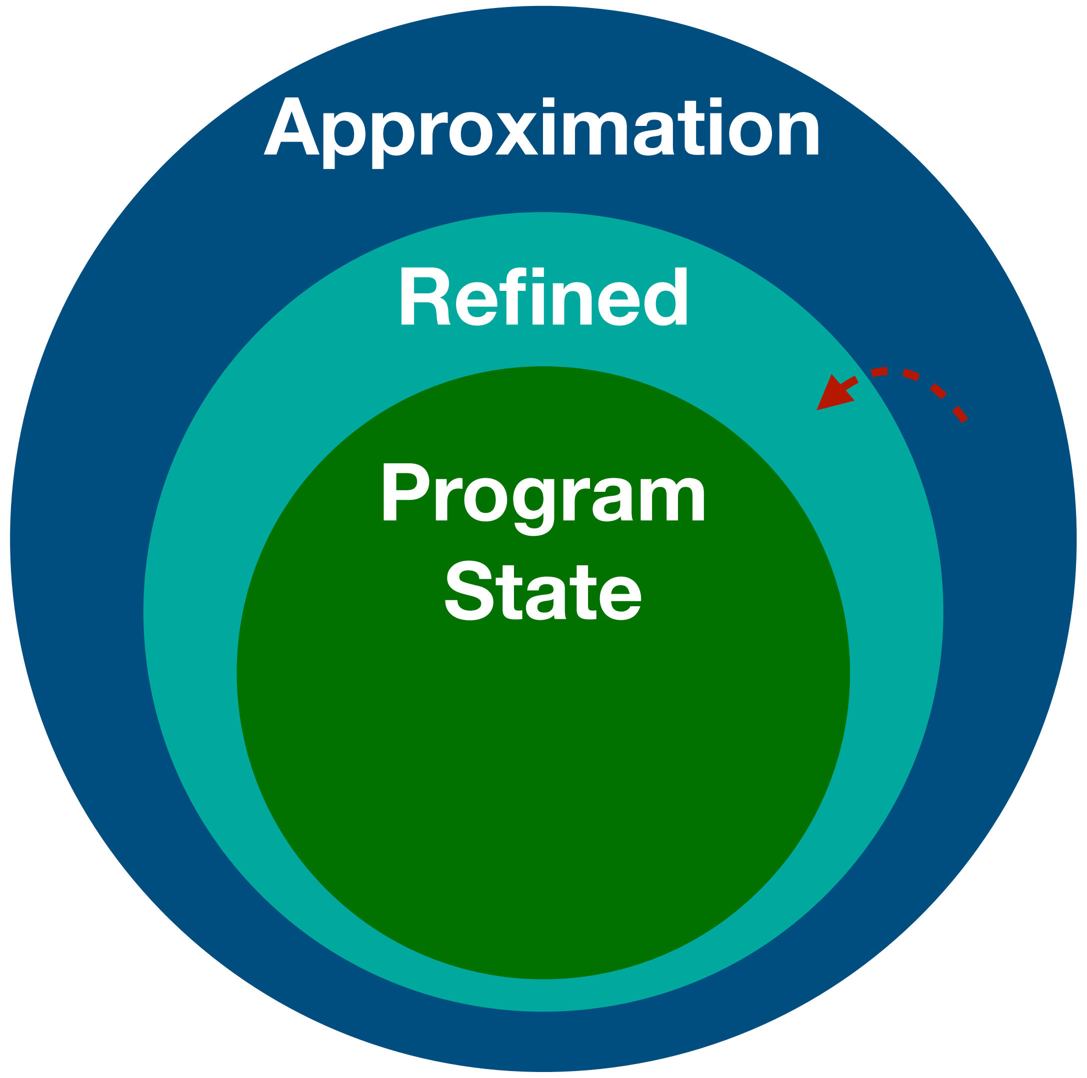
bpf: Track delta between "linked" registers
Alexei Starovoitov

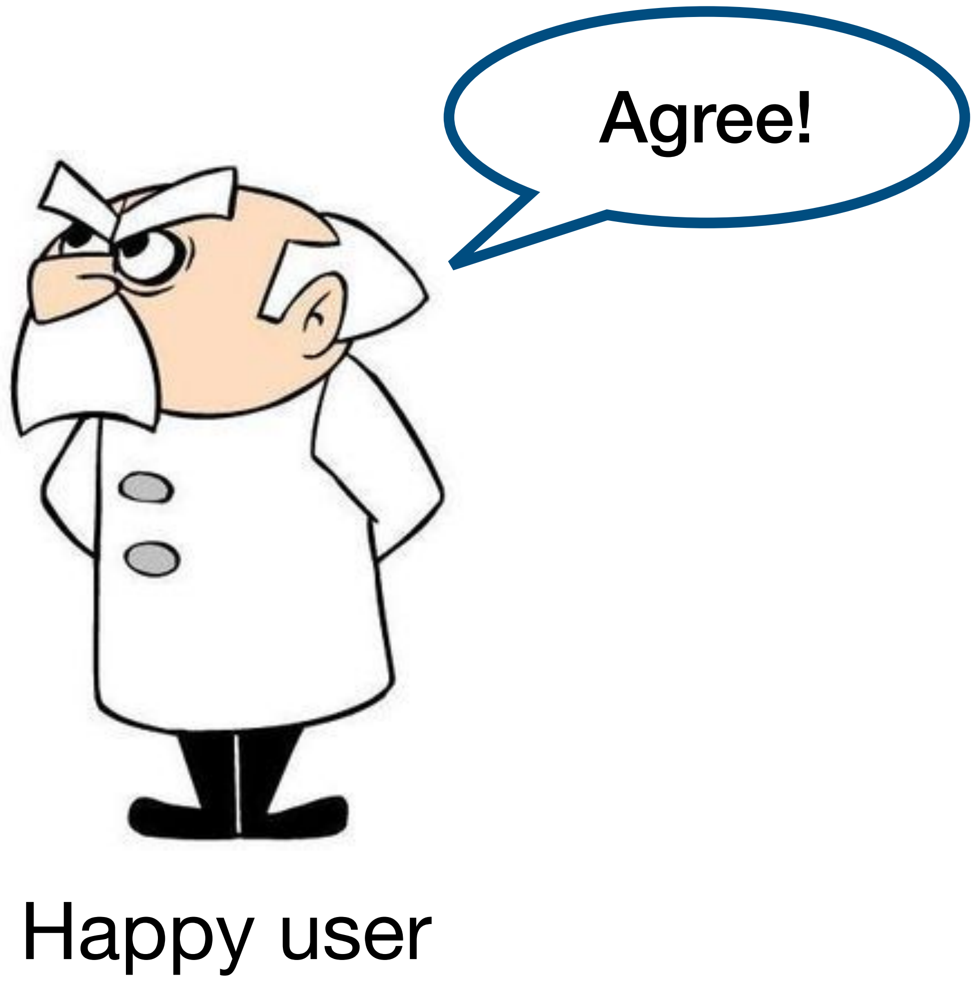


Lazy Abstraction Refinement with Proof

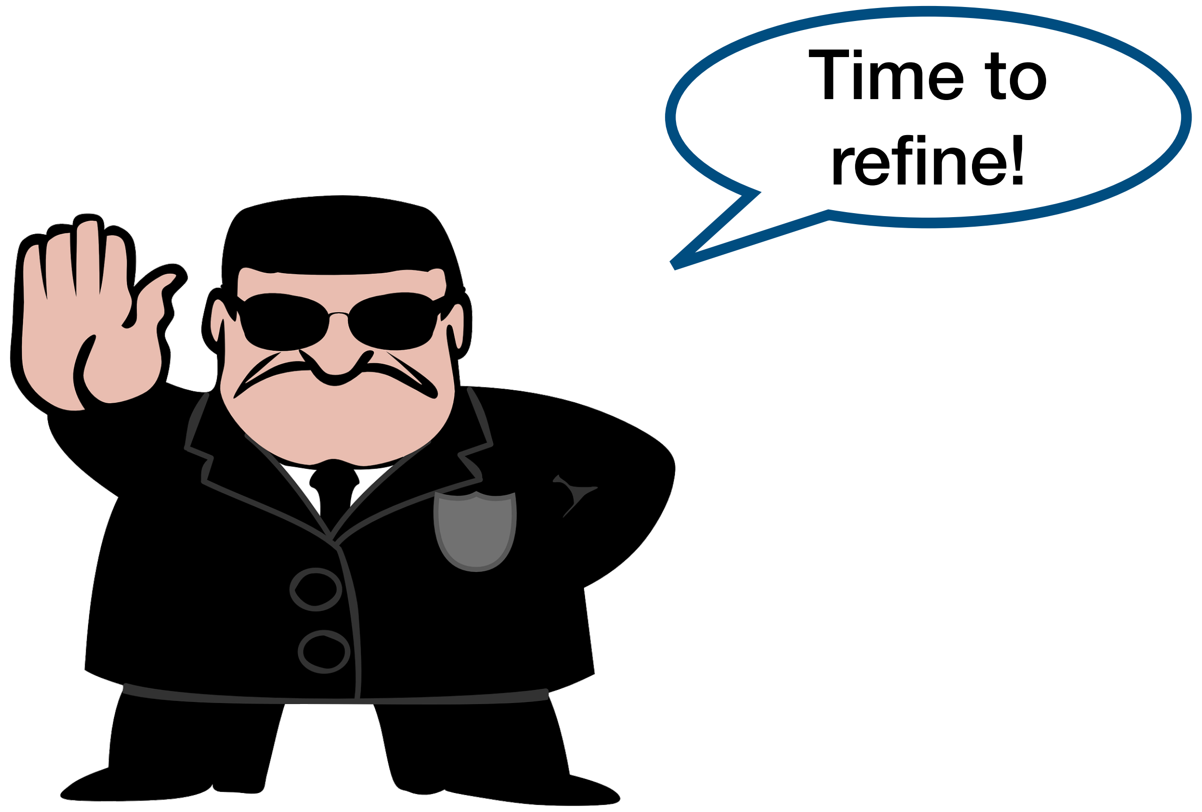

NO, unsafe!
Verifier
Approximation
Program State
YES, safe!!!
Angry user



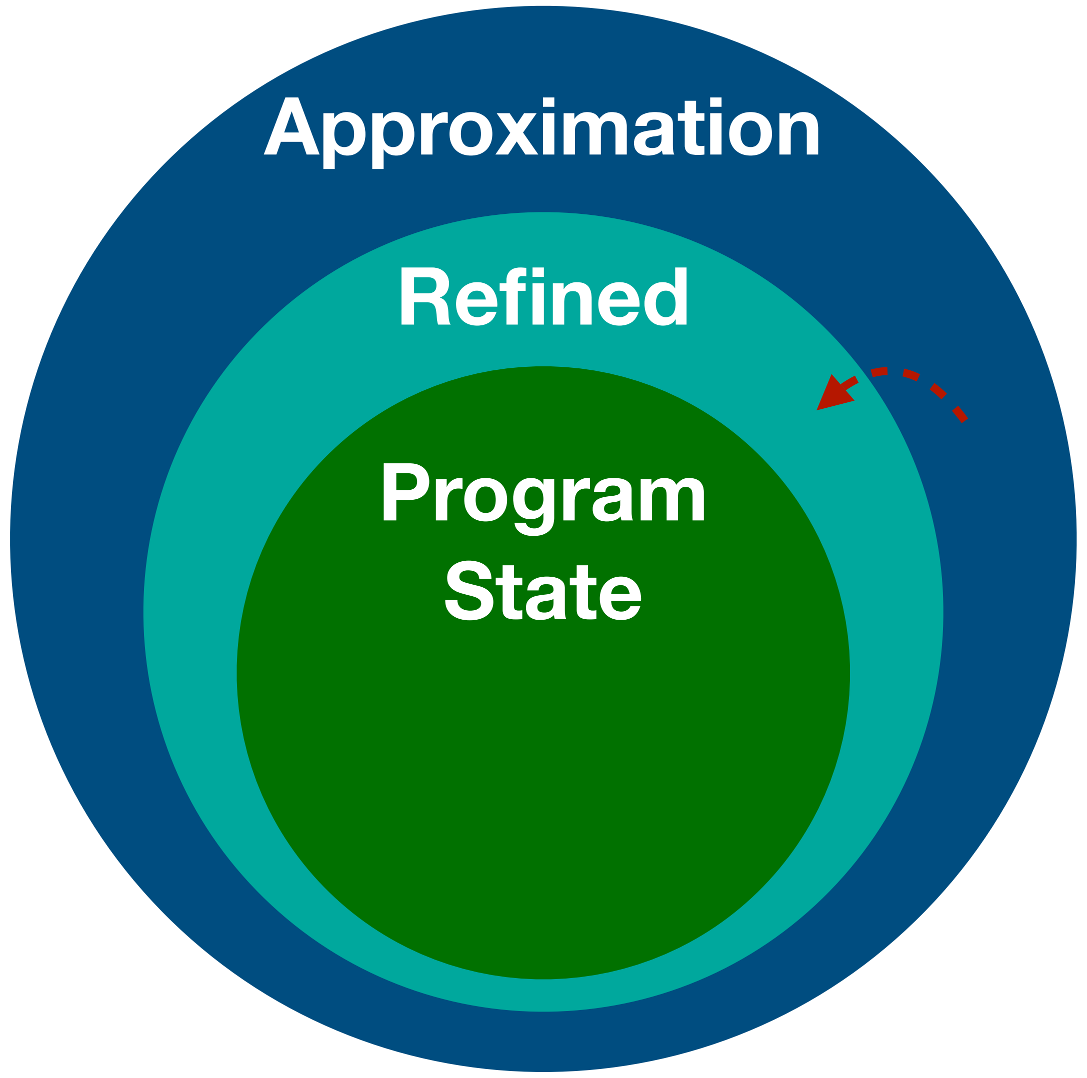


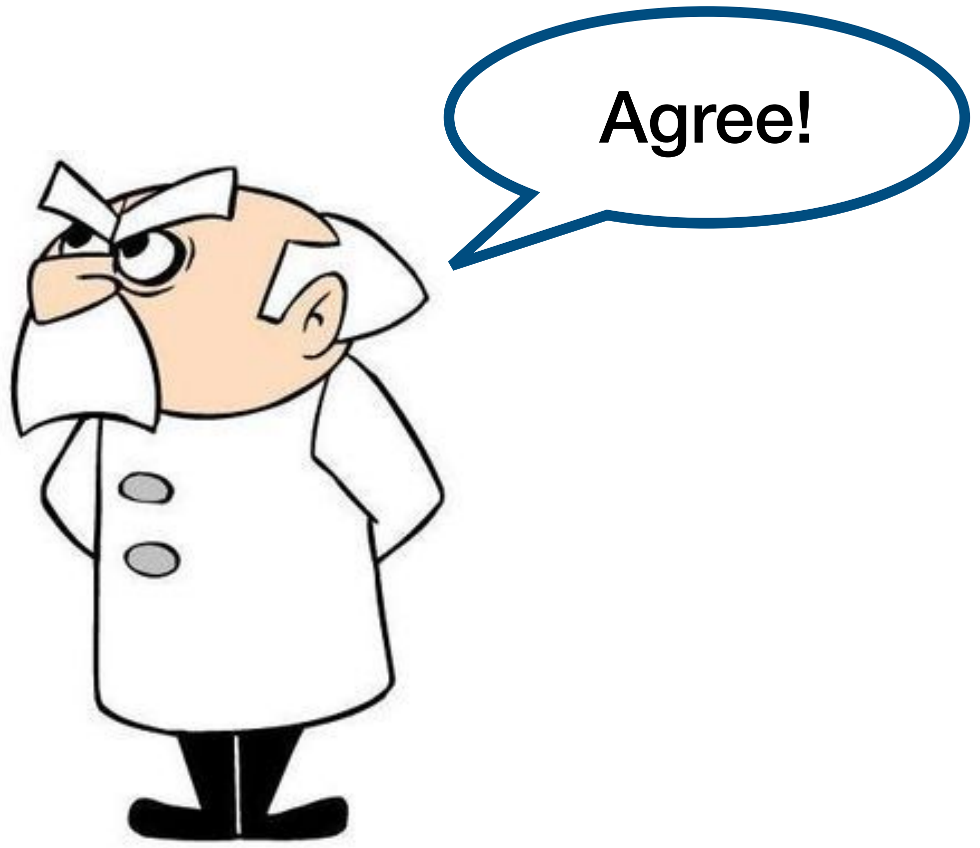


Laziness-1: only refine when the verifier cannot continue



Verifier

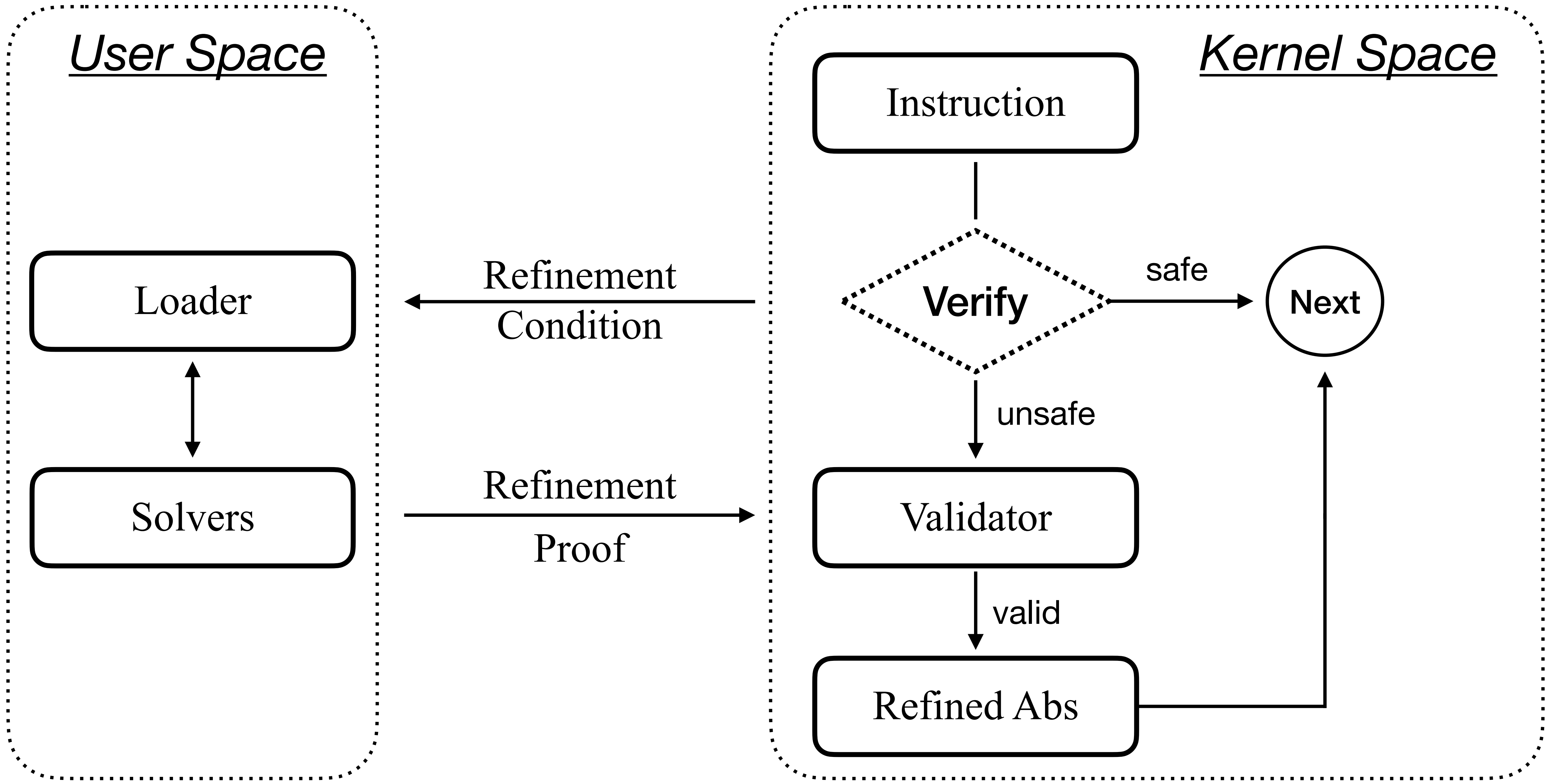




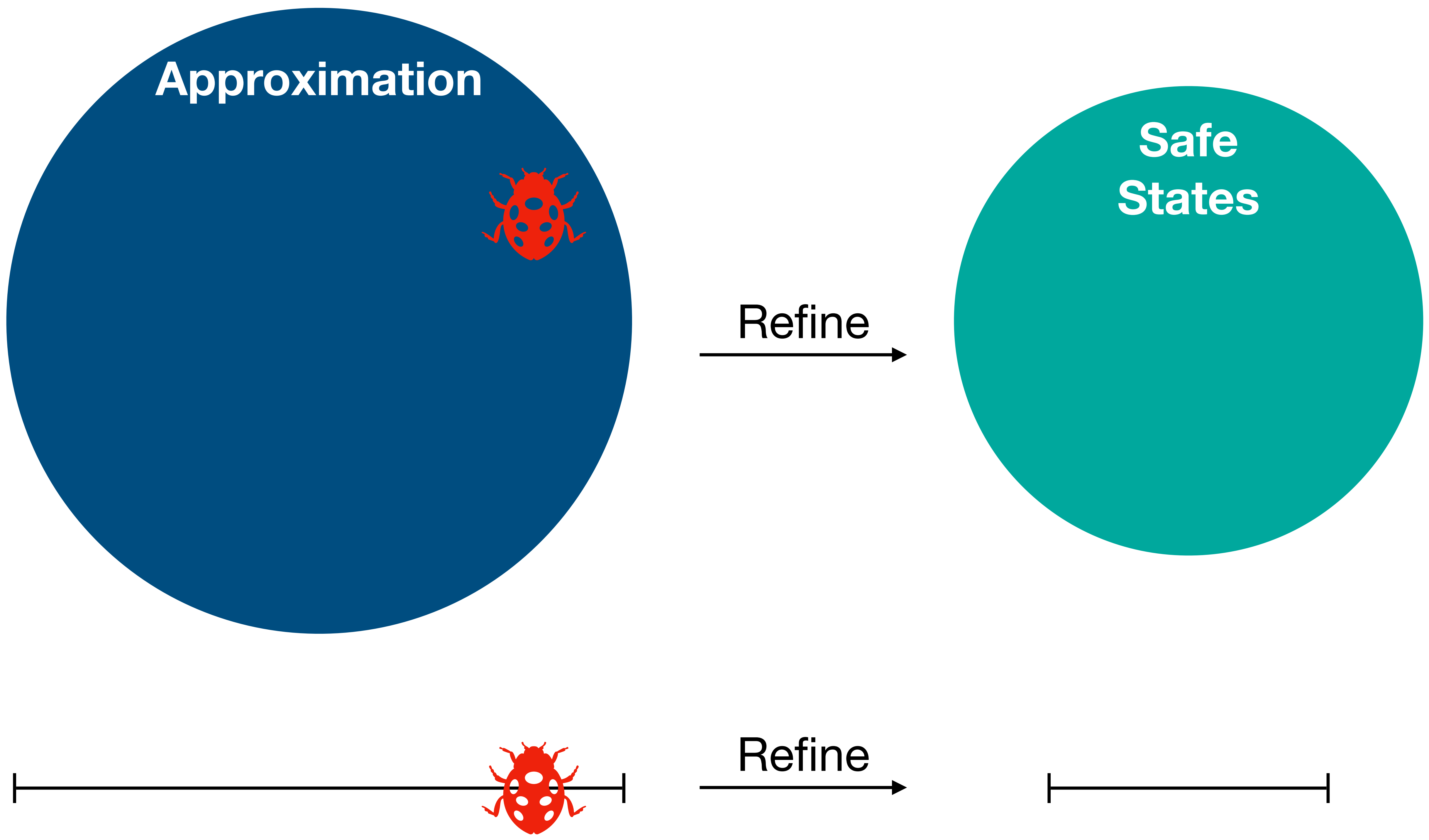
Happy user

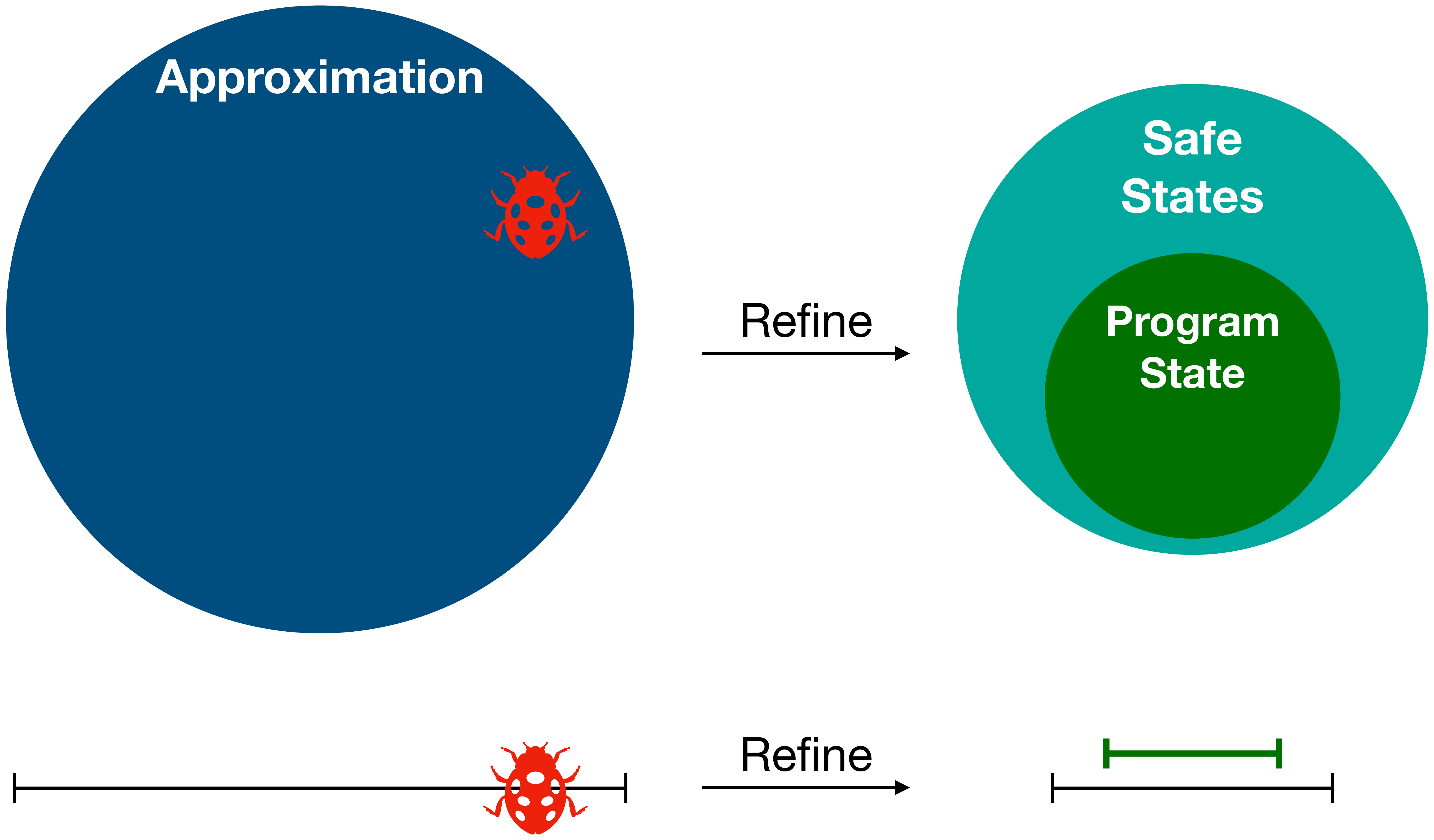
Laziness-2: only refine to just enough to continue

Lazy Abstraction Refinement with Proof



Proof generated in user space and validated in kernel space ensures minimal overhead while achieving a high precision.


Approximation
Refine
Safe
States
Refine



Refinement condition: applicable only if all possible values is within the safe bound

Symbolic "Domain"

- Using the most precise domain to encode all possible values of a variable
  - Make every input as symbolic value
  - Represent computation as symbolic expressions
  - Record the path condition after each jmp

	Symbolic	Interval	Tristate
Precision	Exact Semantics	Over-approximation	Over-approximation
Arithmetic	Exact	Low	Low
Bitwise	Exact	Low	High
Variable Relation	Exact	Not Captured	Not Captured
Path Condition	Exact	Over-approximation	Over-approximation
Maintaining Cost	Low	Low	Low
Reasoning Cost	Complex	Low	Low

Example:

- Let r0 = sym0, r1 = 1
  - For: r2 = 2*r0 + r1
  - Result: r2 = 2*sym0 + 1

  - For: if r2 > r1 goto +off
  - Remember: 2*sym0 + 1 > 1

Symbolic "Domain"

- Represent all possible input sources as symbolic value
- Represent computation as symbolic expressions
- Record the path condition after each jmp
- Essentially, every reg/slot is an identifier binded to some immutable value

```
; R0 = sym0, R1 = fp(-16)
1: r0 &= 0xf          ; R0 = sym0&0xf
2: r1 += r0           ; R1=fp(-16 + (sym0&0xf))
3: r4 = 0xf           ; R4=15
4: r4 -= r0           ; R4=15 - (sym0&0xf)
5: r1 += r4           ; R1=fp(-16 + (sym0&0xf) + (15-(sym0&0xf)))
6: r0 = *(u8 *)(r1 +0)

; off = -16 + (sym0&0xf) + (15-(sym0&0xf))
```

Refinement Condition

- Refine the abstraction to just enough to continue, i.e., safe bound
- Assert the symbolic state is within this safe bound, i.e., refinement condition
- The refinement is accepted if the condition holds

```
; R0 = sym0, R1 = fp(-16)
1: r0 &= 0xf            ; R0 = sym0&0xf
2: r1 += r0             ; R1=fp(-16 + (sym0&0xf))
3: r4 = 0xf             ; R4=15
4: r4 -= r0             ; R4=15 - (sym0&0xf)
5: r1 += r4             ; R1=fp(-16 + (sym0&0xf) + (15-(sym0&0xf)))
6: r0 = *(u8 *)(r1 +0)

; off = -16 + (sym0&0xf) + (15-(sym0&0xf))
; Refinement condition: -16 <= off < 0
```

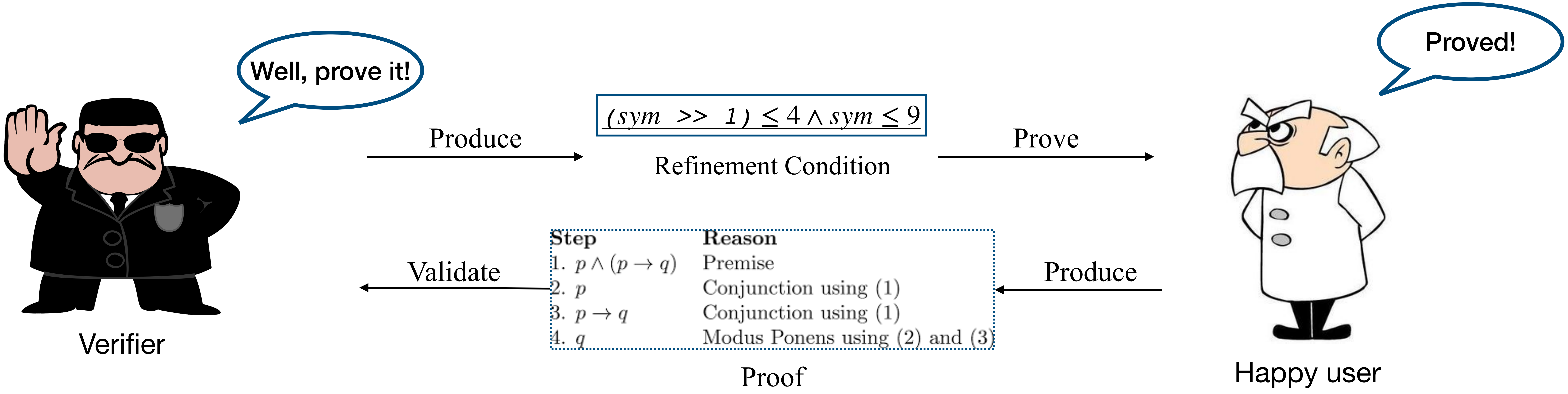
Refinement Condition

- Refine the abstraction to just enough to continue, i.e., safe bound
- Assert the symbolic state is ***within the safe bound***, i.e., refinement condition
- The refinement is accepted if the condition holds

```
R1: off = -16 + (sym0&0xf) + (15-(sym0&0xf))
Refinement condition: -16 <= off < 0
```

```
R1: fp(off=-16,smin=0, smax=30)
OOB Condition: smin+off < -16 or smin+off>-1 or smax+off+sz > 0
```

```
smax+off+sz<=0
Hence: smax refined to -(off+sz) = -(-16+1), i.e., 15
R1_w=fp(off=-16,smax=15)
```


Well, prove it!
Verifier
Produce
(sym >> 1) ≤ 4 ∧ sym ≤ 9
Refinement Condition
Prove
Proved!
Happy user
Produce
Step | Reason
1. p ∧ (p → q) | Premise
2. p | Conjunction using (1)
3. p → q | Conjunction using (1)
4. q | Modus Ponens using (2) and (3)
Proof
Validate

ETH zürich

Proof

- Prove the condition representing the refinement
  - Each possible value **<u>contained</u>** in the symbolic expression satisfies the condition
- Essentially, the problem of satisfiability (SMT, a well established field)
- Proof contains a sequence of steps, each step is a small, simple proof rule
- A proof checker only accepts **<u>well-formed</u>** proof, and the proof is done by **<u>contradiction</u>**

```
Proof ::= Step+
Step  ::= Rule Premise* Arg*
Rule  ::= Resolution | Modus Ponens | …
```

Prove $\phi(x)$:

$$\text{Assume } \neg\, \phi(x)$$
$$\neg\phi(x) \implies \phi_j(x) \quad (rule_k)$$
$$\phi_j(x) \implies false$$

Boolean – Modus Ponens

$$\frac{F_1, (F_1 \rightarrow F_2) \mid -}{F_2}$$

Equality – Transitivity

$$\frac{t_1 = t_2, \ldots, t_{n-1} = t_n \mid -}{t_1 = t_n}$$

Equality – Reflexivity

$$\frac{- \mid t}{t = t)}$$

Resolution:

$$\frac{(A \vee l) \quad (B \vee \neg l)}{A \vee B}$$

Proof

- Converting the refinement condition into SMT formula and querying the SMT solver

```
R1: off = -16 + (sym0&0xf) + (15-(sym0&0xf))
Refinement condition: -16 <= off < 0
```

Proof ::= Step+

Step ::= Rule Premise* Arg*

Rule ::= Resolution | Modus Ponens | …

Prove $\phi(x)$:

Assume $\neg\ \phi(x)$

$\neg\phi(x) \implies \phi_j(x)$ $(rule_k)$

$\phi_j(x) \implies false$

```
off = -16 + (sym0&0xf) + (15-(sym0&0xf))
Prove: off >= -16

Proof:
  Assume off < -16
  Rewrite -16 + (sym0&0xf) + 15 - (sym0&0xf)
        = -16 + 15 = -1
  Trans off = -1
  Refl  -16 = -16
  Cong  (off < -16) = (-1 < -16)
  Rewrite (-1 < -16) = false
  Trans (off < -16) = false
Q.E.D.
```

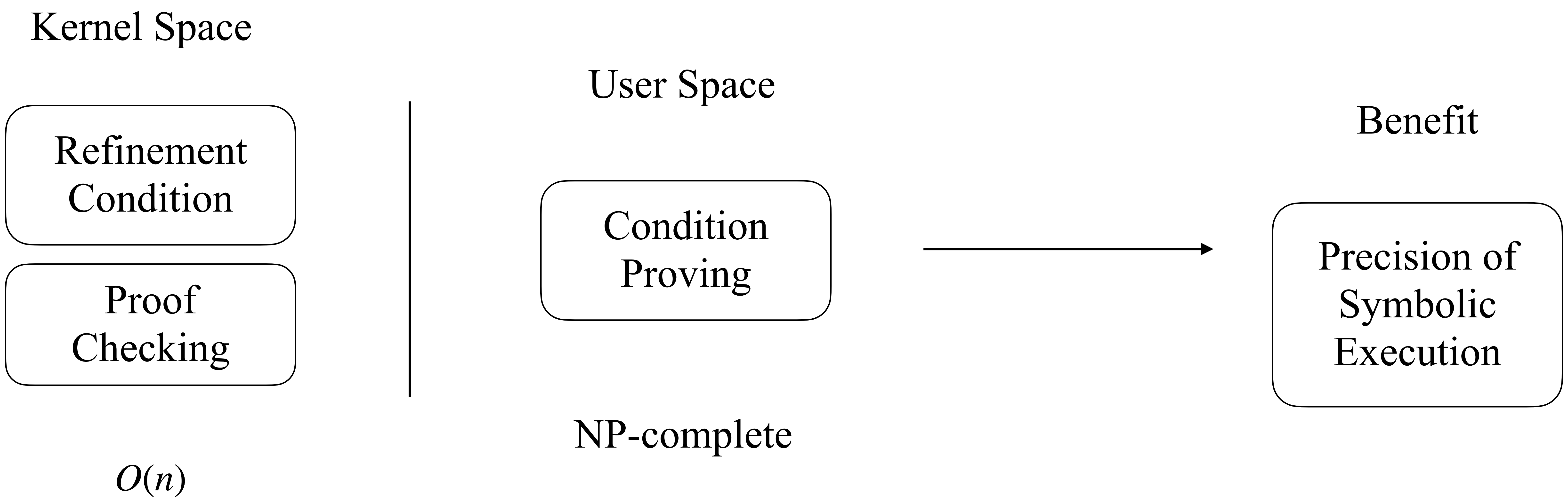
```
; note: in practice we use QF_BV
(set-logic ALL)
(set-option :produce-proofs true)

(declare-const sym0 Int)
(assert (and (>= sym0 0) (<= sym0 15)
       (< (+ -16 (- 15 sym0) sym0) -16)))

(check-sat)
(get-proof)
```

```
(define @t1 () (- 15 sym0))
(define @t2 () (+ -16 @t1 sym0))
(define @t3 () (< @t2 -16))
(define @t4 () (+ 15 (* -1 sym0)))
(assume @p1 (and (<= sym0 0) (<= sym0 15) @t3))
(step @p2 :rule trust :premises () :args ((= (not true) false)))
(step @p3 :rule trust :premises () :args ((= (>= -1 -16) true)))
(step @p4 :rule refl :args (-16))                                  ; p4: -16 = -16
(step @p5 :rule trust :premises () :args ((= (+ -16 @t4 sym0) -1)))  ; p5: -16 + t4 + sym0 = -1
(step @p6 :rule refl :args (sym0))                                 ; p6: sym0 = sym0
(step @p7 :rule trust :premises () :args ((= @t1 @t4)))            ; p7: t1 = t4
(step @p8 :rule nary_cong :premises (@p4 @p7 @p6) :args (+))       ; p8: -16+t1+sym0 = -16 + t4 + sym0
(step @p9 :rule trans :premises (@p8 @p5))                         ; p9: -16+t1+sym0 = -1
(step @p10 :rule cong :premises (@p9 @p4) :args (>=))              ; p10: (-16+t1+sym0 >= -16) = (-1 >= -16)
(step @p11 :rule trans :premises (@p10 @p3))                       ; p11: (-16+t1+sym0 >= -16) = true
(step @p12 :rule cong :premises (@p11) :args (not))                ; p12: not (-16+t1+sym0 >= -16) = not true
(step @p13 :rule trans :premises (@p12 @p2))                       ; p13: not (-16+t1+sym0 >= -16) = false
(step @p14 :rule trust :premises () :args ((= @t3 (not (>= @t2 -16))))) ; p14: t3 = not (016+t1+sym0>=-16)
(step @p15 :rule trans :premises (@p14 @p13))                      ; p15: t3 = false
(step @p16 :rule and_elim :premises (@p1) :args (2))               ; p16: t3
(step @p17 false :rule eq_resolve :premises (@p16 @p15))           ;  p17: false
```

ETH zürich

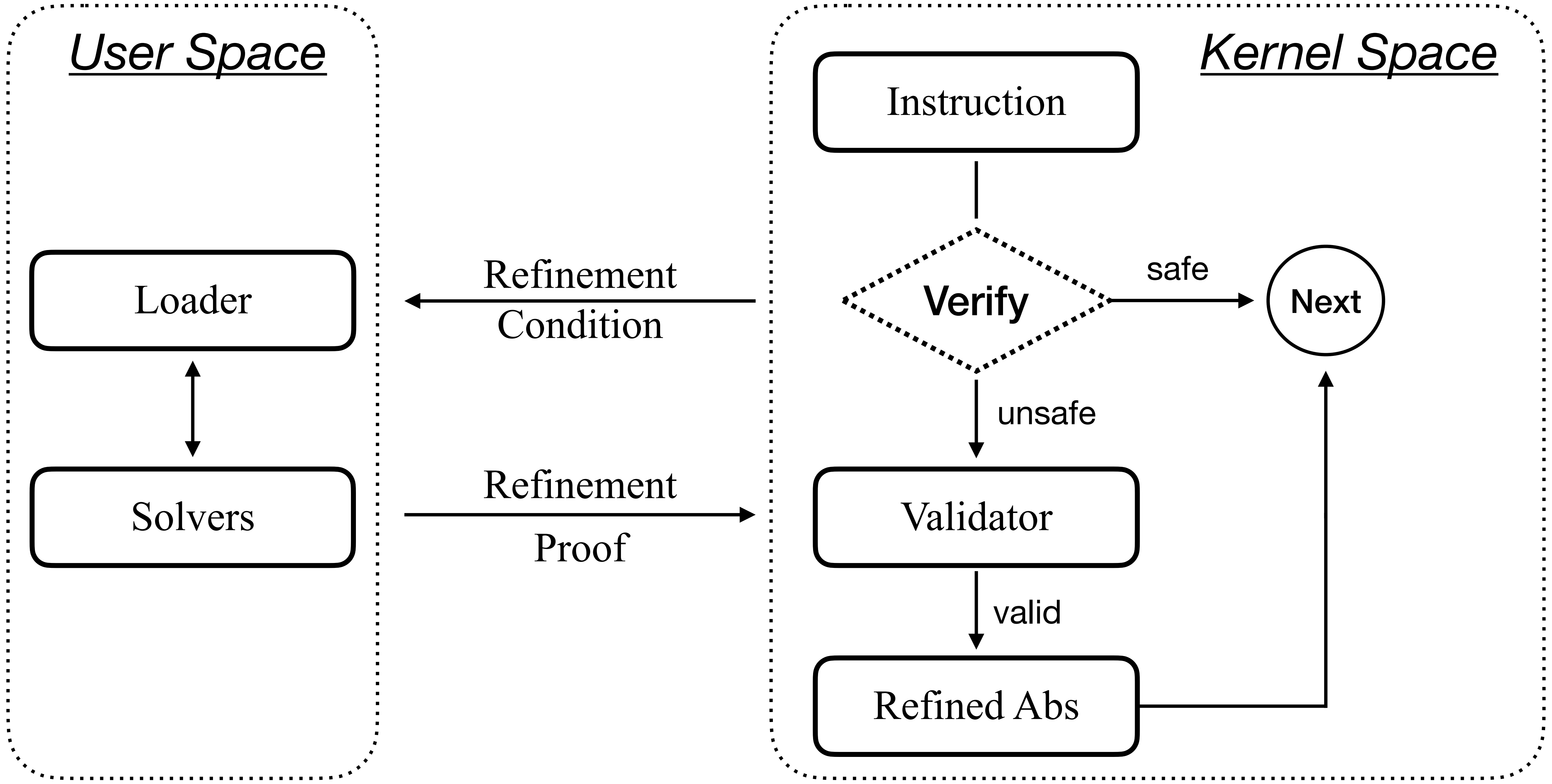
Complexity

- Essentially the satisfiability modulo theories (SMT) problem
- Reasoning unsatisfiable quantifier-free bitvec formula (QF_BV)
- QF_BV reduced to Boolean satisfiability (SAT)
- SAT is **NP-complete**, **but <u>decidable</u>**
- In practice, solvers can prove most formulas very **efficiently**
- Proof checking is a **<u>linear-time</u>** scan



Summary

- Collect symbolic representation and generate refinement condition in the kernel
- Prove the condition in user space, NP-complete but decidable and the solver is mostly efficient
- Accept the refinement only after a linear-time proof-checking
- Since symbolic “domain” is the most accurate domain, we can (hopefully) handle many cases once for all



Implementation

- BCF: BPF Certificate Format
- Provide a buffer for the refinement condition
- Set the flags and fd if proof is requested
- Prove in user space and provide the proof in the buf
- Set the proof size in `true_size` and the flag
- bcf_fd enables resuming the last check

```
diff --git a/include/uapi/linux/bpf.h b/include/uapi/linux/bpf.h

+enum {
+       BCF_F_PROOF_REQUESTED   = (1U << 0),
+       BCF_F_PROOF_PROVIDED    = (1U << 1),
+};
@@ -1569,6 +1577,12 @@ union bpf_attr {
                __s32           prog_token_fd;
+               /* output: bcf fd for loading proof if requested */
+               __u32           bcf_fd;
+               __aligned_u64   bcf_buf;
+               __u32           bcf_buf_size;
+               __u32           bcf_buf_true_size;
+               __u32           bcf_flags;
        };
```

Implementation

- Handle the BCF request in user space
- Convert the refinement condition into SMT formulas
- Query the solver and collect the proof
- Convert the proof into BCF format
- Provide the proof by setting the flag and bcf_fd

```c
static int bpf_prog_load(...)
{
	union bpf_attr attr = {
		.insns = (u64)(insns),
		.insn_cnt = insn_cnt,
		.bcf_buf = (u64)(bcf_buf),
		.bcf_buf_size = BCF_BUF_SIZE,
		...
	};

again:
	prog_fd = bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
	if (prog_fd < 0 && attr.bcf_flags & BCF_F_PROOF_REQUESTED) {

		unsat = prove_unsat(attr.bcf_buf, cond_idx);
		if (unsat) {
			copy_proof(attr.bcf_buf, ...);
			attr.bcf_flags = BCF_F_PROOF_PROVIDED;
			goto again;
		}
	}

	return prog_fd;
}
```

Implementation

- Handle the BCF request in user space
- Convert the refinement condition into SMT formulas
- Query the solver and collect the proof
- Convert the proof into BCF format
- Provide the proof by setting the flag and bcf_fd

```
static bool prove_unsat(...)
{
	cond = to_cvc5_term(&ctx, idx);
	cvc5_assert_formula(slv, cond);
	result = cvc5_check_sat(slv);

	if (cvc5_result_is_unsat(result)) {
		proof = cvc5_get_proof(slv, CVC5_PROOF_COMPONENT_FULL,
		                       &proof_size);
		if (!proof) {
			WARNF("failed to obtain proof");
			goto err_free;
		}
		cvc5_proof_to_bcf(proof, attr);
	}

	...
}
```

Implementation

- Start BCF tracking to collect symbolic state if requested
- do_check() is adapted to follow the current path only
- Symbolic state is collected mostly with bcf_alu()
- Preserve the env behind the bcf_fd
- Copy to user, set the flag and bcf_fd, and wait for the proof

```
ret = do_check(env);

if (ret < 0 && bcf_state->requested) {
        ...
        bcf_state->tracking = true;
        env->insn_idx = subprog_info(env, env->subprog)->start;
        err = init_subprog_state(env, env->subprog);
        if (err == 0)
                do_check(env);
        bcf_state->tracking = false;
        ...
}
```

Implementation

- Start BCF tracking to collect symbolic state if requested
- do_check() is adapted to follow the current path only
- Symbolic state is collected mostly with bcf_alu()
- Preserve the env behind the bcf_fd
- Copy to user, set the flag and bcf_fd, and wait for the proof

```
static int check_cond_jmp_op(...)
{
        ...
        err = 0;
        if (match_jmp_history(env, *insn_idx + insn->off + 1, *insn_idx)) {
                err = pop_stack(env, NULL, NULL, false);
                if (err == 0)
                        *insn_idx += insn->off;
        }
        return err;
 }
```

Implementation

- Start BCF tracking to collect symbolic state if requested
- do_check() is adapted to follow the current path only
- Symbolic state is collected mostly with bcf_alu()
- Preserve the env behind the bcf_fd
- Copy to user, set the flag and bcf_fd, and wait for the proof

```
static int adjust_scalar_min_max_vals(...)
{
        ...
        if (!tnum_is_const(dst_reg->var_off) || !tnum_is_const(src_reg.var_o
                ret = bcf_alu(env, insn);
                if (ret < 0)
                        return ret;
        }
        ...
```

Implementation

- Resume the last check if bcf_fd valid
- Validate the proof with an in-kernel proof check
- Continue with the refined state if the proof is accepted
- Check the rest of the program

```
int bpf_check(...)
{
	...
	resume = attr->bcf_flags & BCF_F_PROOF_PROVIDED;
	if (resume)
		goto verifier_check;
	...
verifier_check:
	if (resume) {
		ret = bcf_check_proof(env, attr, uattr);
		if (ret < 0)
			goto skip_full_check;
		ret = do_check_common(env);
	} else
		ret = do_check_main(env);
	ret = ret ?: do_check_subprogs(env);

	if (bcf_requested(env) &&
	    request_bcf(env, attr, uattr, uattr_size) == 0) {
		/* ... early exit, preserve all states */
		return ret;
	}
	...
}
```

Demo

Thank you!