





Automated Lemma Discovery in Agentic Program Verification

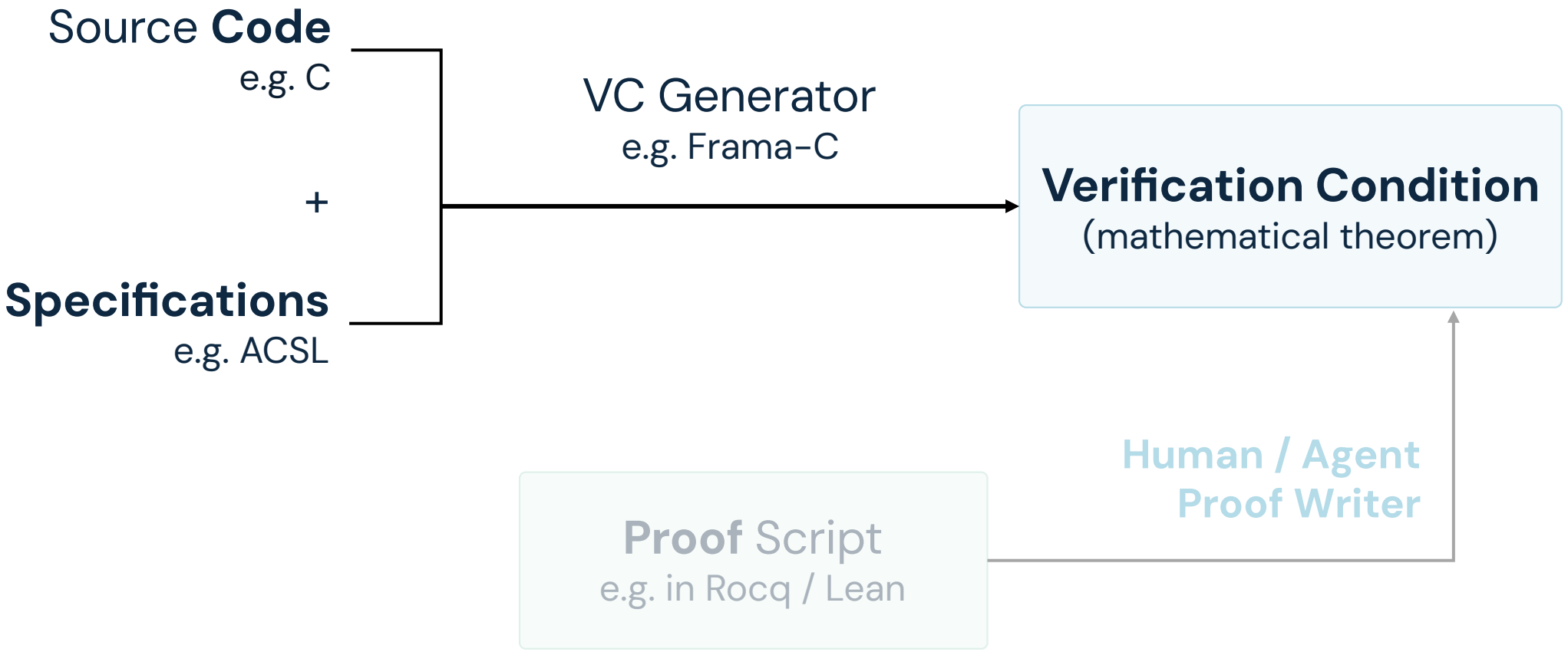
Huan Zhao*, Haoxin Tu*, Zhengyao Liu,
Martin C. Rinard, Abhik Roychoudhury

“

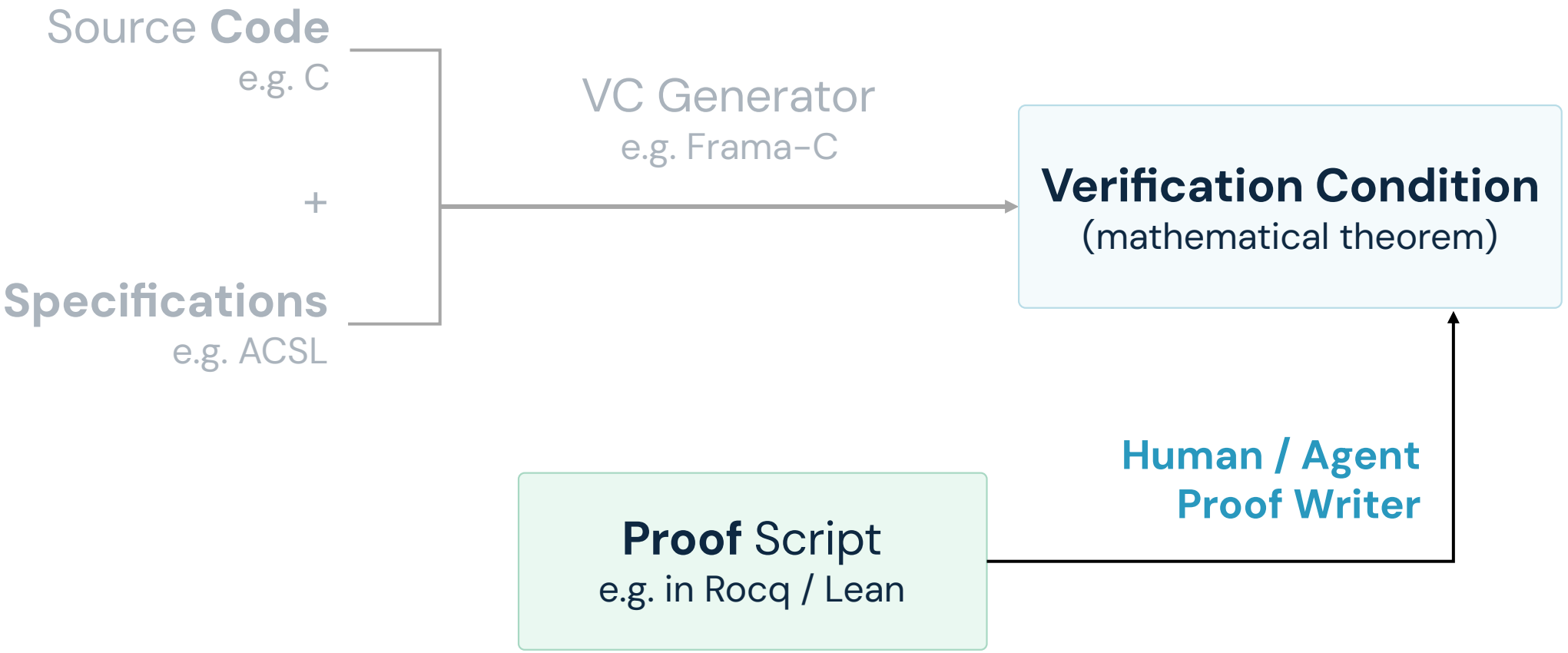
Verification is the *only* way to establish that software is correct and bug-free.

– ***Edsger W. Dijkstra***

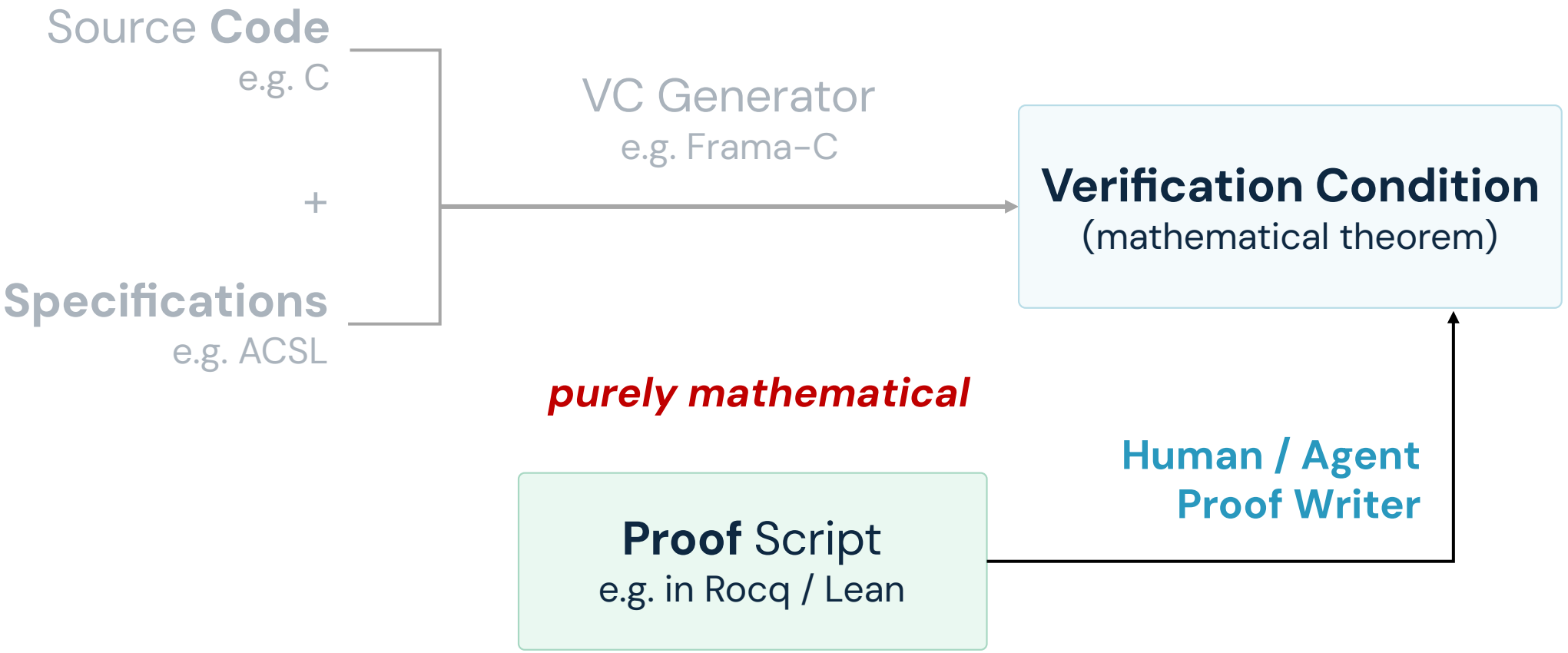
Background: Verification as theorem proving



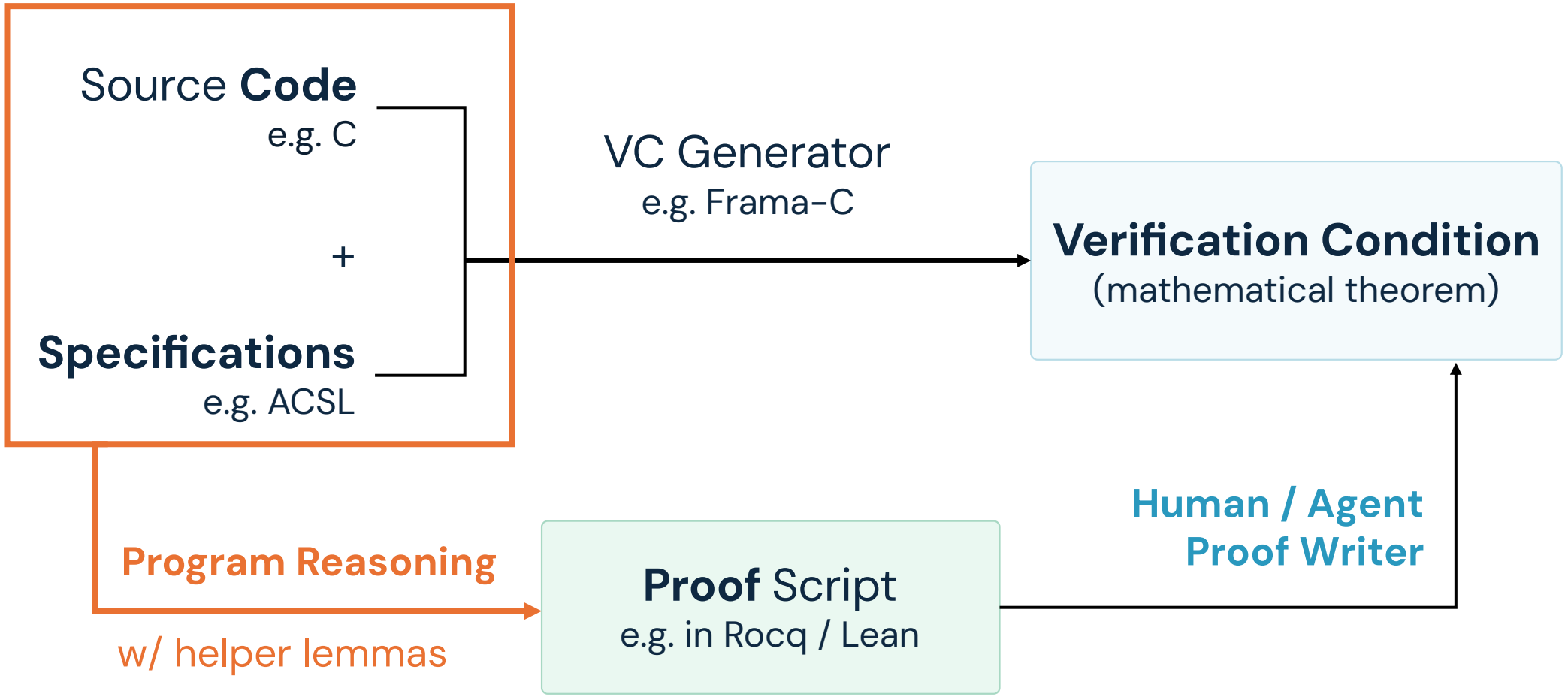
Background: Verification as theorem proving



Background: Verification as theorem proving



This work: proof via **program reasoning**



This work: helper lemmas

WHAT THEY ARE?

Small proved statements to simplify the proof of a main theorem

e.g., helper lemma `x << 1 == x * 2`
theorem `x << y == x * 2^y`

WHAT THEY DO?

- Capture key intermediate results **via program reasoning**
- Help with proof structuring **for agentic VC proving**

Intuitive proofs via **program** reasoning

```
/*@ requires ...; ensures ... */
int hex2bin(u8 *dst, const char *src, size_t count)
{
    //@ ghost size_t ocount = count;
    //@ ghost char *osrc = src;
    /*@
        loop invariant 0 <= count <= ocount;
        loop invariant osrc <= src;
        loop invariant \forall char *p; ...
     */
    while (count--) {
        int hi = hex_to_bin(*src++);
        int lo = hex_to_bin(*src++);
        if ((hi < 0) || (lo < 0))
            return -1;
        //@ assert 0 <= ((hi << 4) | lo) <= 255;
        *dst++ = (hi << 4) | lo;
    }
    //@ assert count == ((size_t)-1);
    return 0;
}
```

hex2bin.c from the Linux kernel with **ACSL specifications**

Loop invariant to be proved

"src is always greater or equal to osrc"

Intuitive proofs via **program** reasoning

```
/*@ requires ...; ensures ... */
int hex2bin(u8 *dst, const char *src, size_t count)
{
    //@ ghost size_t ocount = count;
    //@ ghost char *osrc = src;
    /*@
        loop invariant 0 <= count <= ocount;
        loop invariant osrc <= src;
        loop invariant \forall char *p; ...
     */
    while (count--) {
        int hi = hex_to_bin(*src++);
        int lo = hex_to_bin(*src++);
        if ((hi < 0) || (lo < 0))
            return -1;
        //@ assert 0 <= ((hi << 4) | lo) <= 255;
        *dst++ = (hi << 4) | lo;
    }
    //@ assert count == ((size_t)-1);
    return 0;
}
```

hex2bin.c from the Linux kernel with **ACSL specifications**

Loop invariant to be proved

"src is always greater or equal to osrc"

A Proof, informally

"if a variable is incremented exactly by two each round, it does not decrease."

Intuitive proofs via **program** reasoning

```
/*@ requires ...; ensures ... */
int hex2bin(u8 *dst, const char *src, size_t count)
{
    //@ ghost size_t ocount = count;
    //@ ghost char *osrc = src;
    /*@
        loop invariant 0 <= count <= ocount;
        loop invariant osrc <= src;
        loop invariant \forall char *p; ...
     */
    while (count--) {
        int hi = hex_to_bin(*src++);
        int lo = hex_to_bin(*src++);
        if ((hi < 0) || (lo < 0))
            return -1;
        //@ assert 0 <= ((hi << 4) | lo) <= 255;
        *dst++ = (hi << 4) | lo;
    }
    //@ assert count == ((size_t)-1);
    return 0;
}
```

hex2bin.c from the Linux kernel with **ACSL specifications**

Loop invariant to be proved

"src is always greater or equal to osrc"

```
(* Semantics-aware Verificaion Condition *)
Lemma goal_osrc_le_src_preserved :
  forall osrc src :  addr,addr_le_model osrc src →
    addr_le_model osrc (shift src 2).
Proof.
  intros osrc src [Hb Hoff]. split.
  (* base preserved *)
  - unfold shift, base in *;  simpl in *;  exact Hb.
  (* offset incremented *)
  - unfold shift, offset in *;  simpl in *;  lia.
Qed.
```

ψ_A

VC proving via **proof** reasoning

The **theorem** to be proved in Rocq is *much larger than* the code!

Tool-specific **memory model** for C

Safety checks (e.g. overflow)

Pointer arithmetic

Quantifiers

ψ_C

```
Definition is_sint32 (x:Numbers.BinNums.Z) : Prop := ...

Theorem wp_goal :
  forall (t:Z → Z) (t1:addr → Z) (t2:Z → Z)
  (a:addr) (i:Z) (i1:Z) (i2:Z) (a1:addr) (a2:addr),
  let x := (16 * i) in let x1 := lor i1 x in let x2 := offset a2 in
  let x3 := offset a1 in let x4 := t1 a2 in let x5 := t2 (to_uint8 x4) in
  let x6 := t1 (shift a2 1) in let x7 := t2 (to_uint8 x6) in let x8 :=
  land 68 x5 in let x9 := land 68 x7 in
  ~ (i2 = 0) → (x = (lsl i 4)) → ((i1 + x) = x1) →
  (( Z.quot (x2 + ((-1) * x3)) 2) = 0) →
  (( Z.rem (x3 + ((-1) * x2)) 2) = 0) →
  (0 <= i2) → (0 <= i1) → (0 <= i) →
  (( region (base a1)) <= 0) → ((region (base a)) <= 0) →
  (0 <= x1) → ((to_uint64 ((-1) + i2)) <= i2) → (i1 <= 15) → (i <= 15) →
  (x1 <= 255) → IsArray_uint8 t2 → linked t → sconst t1 →
  is_sint32 i1 → is_sint32 i → is_uint64 i2 → addr_le a1 a2 →
  addr_le a1 a1 → addr_le a a → is_sint8 x4 → is_uint8 x5 →
  is_sint8 x6 → valid_rw t (shift a 0) (1 + i2) →
  valid_rd t (shift a1 0) (1 + (2 * i2)) → is_uint8 x7 →
  (( x8 = 0) → (i = (-1))) → (~ (x8 = 0) → ((L_hex_to_bin x4) = i)) →
  (( x9 = 0) → (i1 = (-1))) → (~ (x9 = 0) → ((L_hex_to_bin x6) = i1)) →
  (forall (a3:addr), addr_lt a3 a2 → addr_le a1 a3 →
  ~ (( land 68 (t2 (to_uint8 (t1 a3)))) = 0)) →
  (forall (a3:addr), addr_lt a3 a1 → addr_le a1 a3 →
  ~ (( land 68 (t2 (to_uint8 (t1 a3)))) = 0)) →
  addr_le a1 (shift a2 2).
```

VC proving via **proof** reasoning

The **theorem** to be proved in Rocq is *much larger than* the code!

Proving ψ_C is difficult due to verbose encodings that obscure program structure

Tool-specific **memory model** for C

Safety checks (e.g. overflow)

Pointer arithmetic

Quantifiers

ψ_C

```
Definition is_sint32 (x:Numbers.BinNums.Z) : Prop := ...

Theorem wp_goal :
  forall (t:Z → Z) (t1:addr → Z) (t2:Z → Z)
  (a:addr) (i:Z) (i1:Z) (i2:Z) (a1:addr) (a2:addr),
  let x := (16 * i) in let x1 := lor i1 x in let x2 := offset a2 in
  let x3 := offset a1 in let x4 := t1 a2 in let x5 := t2 (to_uint8 x4) in
  let x6 := t1 (shift a2 1) in let x7 := t2 (to_uint8 x6) in let x8 :=
  land 68 x5 in let x9 := land 68 x7 in
  ~ (i2 = 0) → (x = (lsl i 4)) → ((i1 + x) = x1) →
  (( Z.quot (x2 + ((-1) * x3)) 2) = 0) →
  (( Z.rem (x3 + ((-1) * x2)) 2) = 0) →
  (0 <= i2) → (0 <= i1) → (0 <= i) →
  (( region (base a1)) <= 0) → ((region (base a)) <= 0) →
  (0 <= x1) → ((to_uint64 ((-1) + i2)) <= i2) → (i1 <= 15) → (i <= 15) →
  (x1 <= 255) → IsArray_uint8 t2 → linked t → sconst t1 →
  is_sint32 i1 → is_sint32 i → is_uint64 i2 → addr_le a1 a2 →
  addr_le a1 a1 → addr_le a a → is_sint8 x4 → is_uint8 x5 →
  is_sint8 x6 → valid_rw t (shift a 0) (1 + i2) →
  valid_rd t (shift a1 0) (1 + (2 * i2)) → is_uint8 x7 →
  (( x8 = 0) → (i = (-1))) → (~ (x8 = 0) → ((L_hex_to_bin x4) = i)) →
  (( x9 = 0) → (i1 = (-1))) → (~ (x9 = 0) → ((L_hex_to_bin x6) = i1)) →
  (forall (a3:addr), addr_lt a3 a2 → addr_le a1 a3 →
  ~ (( land 68 (t2 (to_uint8 (t1 a3)))) = 0)) →
  (forall (a3:addr), addr_lt a3 a1 → addr_le a1 a3 →
  ~ (( land 68 (t2 (to_uint8 (t1 a3)))) = 0)) →
  addr_le a1 (shift a2 2).
```

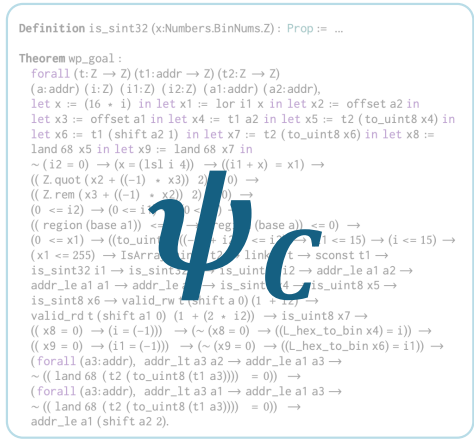
Recap: proof vs program reasoning

Program Reasoning

```
/*@ requires ...; ensures ... */
int hex2bin(u8 *dst, const char *src, size_t count)
{
    //@ ghost size_t ocount = count;
    //@ ghost char *osrc = src;
    /*@
        loop invariant 0 <= count <= ocount;
        loop invariant osrc <= src;
        loop invariant \forall char *p; ...
     */
    while (count--) {
        int hi = hex_to_bin(*src++);
        int lo = hex_to_bin(*src++);
        if ((hi < 0) || (lo < 0))
            return -1;
        //@ assert 0 <= ((hi << 4) | lo) <= 255;
        *dst++ = (hi << 4) | lo;
    }
    //@ assert count == ((size_t)-1);
    return 0;
}
```

Simple Program

produces



ψ_C

Huge Theorems

HARD TO PROVE!

A Proof, informally

"if a variable is incremented exactly by two each round, it does not decrease."

Intuitive encoding

gives

```
(* Custom model of address comparison *)
Definition addr_le_model (p q:addr) : Prop :=
  base p = base q ∧ offset p <= offset q.

(* Semantics-aware Verificaion Condition *)
Lemma goal_osrc_le_src_preserved :
  forall osrc s
    addr_le_model osrc src →
    addr_le_mo
Proof.
  intros osrc src
  (* base preserved *)
  - unfold shift, base in *; simpl in *; exact Hb.
  (* offset incremented *)
  - unfold shift, offset in *; simpl in *; lia.
Qed.
```

ψ_A

Straightforward Proofs

NO GUARANTEES!

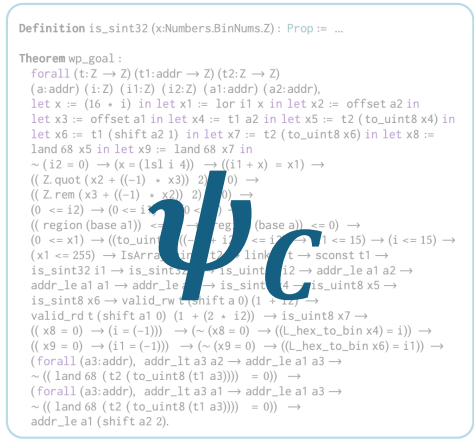
Insight: bridging the gap with **helper lemmas**

Program Reasoning

```
/*@ requires ...; ensures ... */
int hex2bin(u8 *dst, const char *src, size_t count)
{
    //@ ghost size_t ocount = count;
    //@ ghost char *osrc = src;
    /*@
        loop invariant 0 <= count <= ocount;
        loop invariant osrc <= src;
        loop invariant \forall char *p; ...
     */
    while (count--) {
        int hi = hex_to_bin(*src++);
        int lo = hex_to_bin(*src++);
        if ((hi < 0) || (lo < 0))
            return -1;
        //@ assert 0 <= ((hi << 4) | lo) <= 255;
        *dst++ = (hi << 4) | lo;
    }
    //@ assert count == ((size_t)-1);
    return 0;
}
```

Simple Program

produces



Huge Theorems

ψ_C

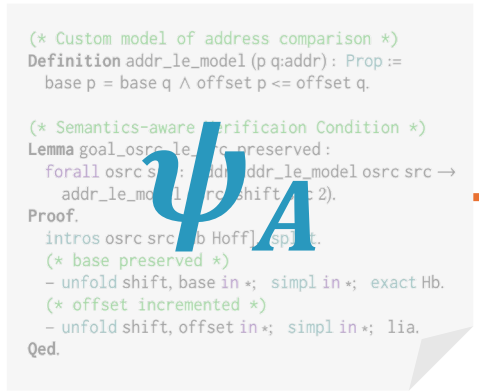
Guidance w/ ***helper lemmas***

A Proof, informally

"if a variable is incremented exactly by two each round, it does not decrease."

Intuitive encoding

gives



ψ_A

Straightforward Proofs

Bridging proof structures with **helper lemmas**

A STRAIGHTFORWARD PROOF

ψ_A

```
Definition addr_le_model (p q:addr) : Prop :=
  base p = base q ∧ offset p <= offset q.

(* Semantics-aware Verificaion Condition *)
Lemma goal_osrc_le_src_preserved :
  forall osrc src : addr,addr_le_model osrc src →
    addr_le_model osrc (shift src 2).
Proof.
  intros osrc src [Hb Hoff]. split.
  (* base preserved *)
  - unfold shift, base in *; simpl in *; exact Hb.
  (* offset incremented *)
  - unfold shift, offset in *; simpl in *; lia.
Qed.
```

HELPER LEMMAS

```
Lemma base_shift :
  forall p k, base (shift p k) = base p. Proof... Qed.

Lemma offset_shift :
  forall p k, offset (shift p k) = (offset p + k)%Z. Proof... Qed.

(* Helper Lemma 1 *)
Lemma HL1_addr_le_shift_same_base :
  forall p q k, base p = base q → addr_le p q
    → (0 <= k)%Z → addr_le p (shift q k).
Proof.
  rewrite base_shift. rewrite offset_shift... (* steps omitted *)
Qed.

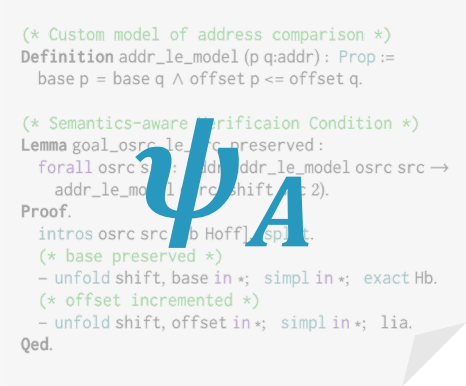
(* Helper Lemma 2 *)
Lemma HL2_addr_le_same_base :
  forall p q, addr_le p q → base p = base q. Proof... Qed.
```

FINAL PROOF OF ψ_C

```
Proof.
  intros t t1 i t2 a i1 i2 a1 a2; cbn. ...
  apply (HL1_addr_le_shift_same_base a1 a2 2).
  apply (HL2_addr_le_same_base _ _ Hle_a1a2).
  all: try (exact Hle_a1a2); try lia.
Qed.
```

Follow proof structure of ψ_A
to write the final proof of ψ_C

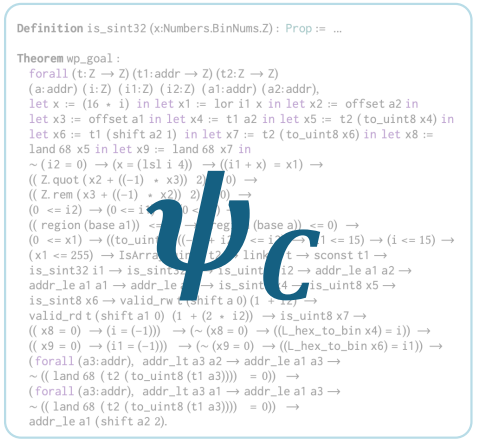
Challenges in applying helper lemmas



ψ_A

Straightforward Proofs

Helper Lemmas



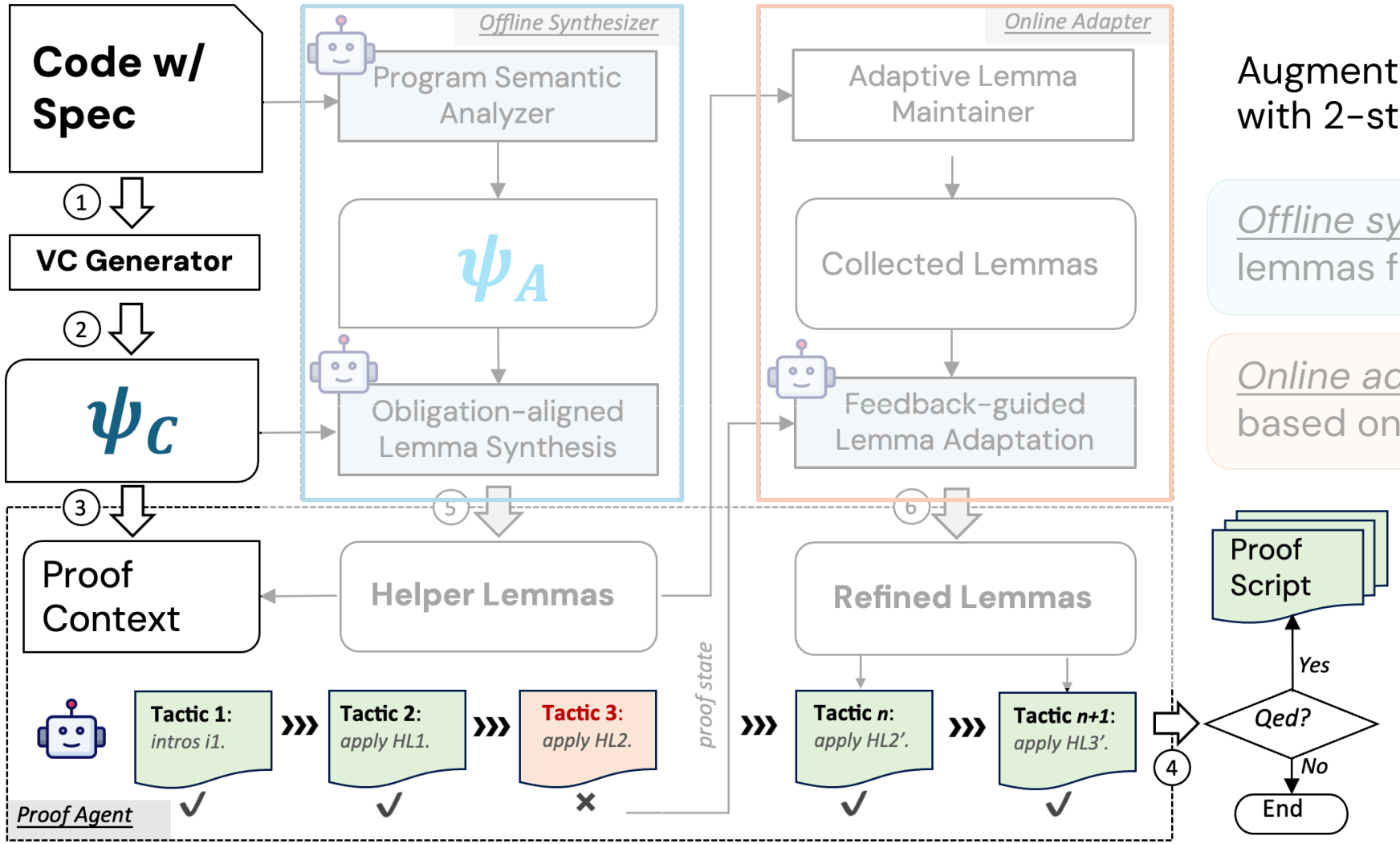
ψ_C

Huge Theorems

Challenge 1. How to generate useful helper lemmas?

Challenge 2. How to apply them as proof unfolds?

LEMMANET: Agentic VC Proving through lemma discovery

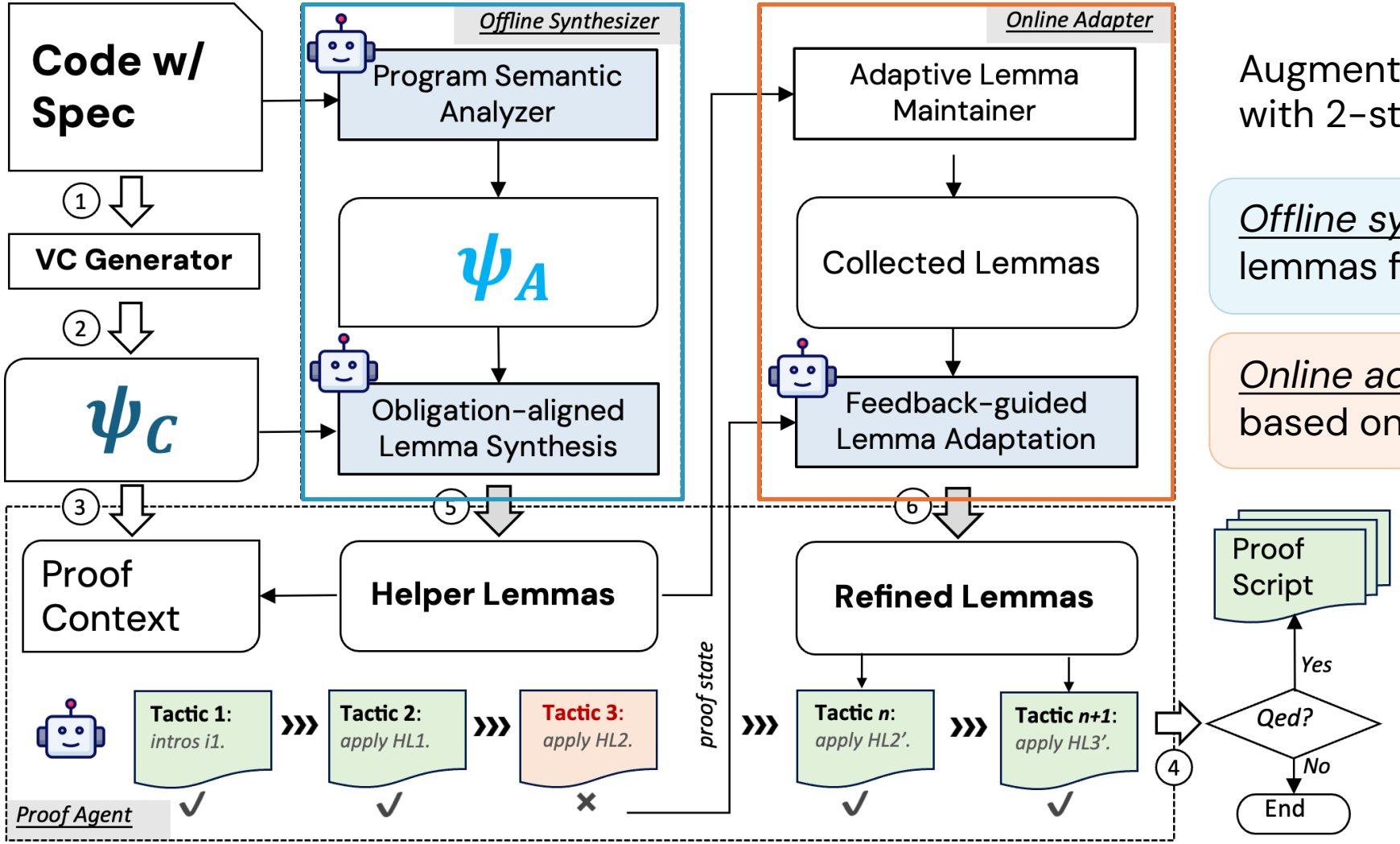


Augment proof agent (①›②›③›④) with 2-stage helper lemma discovery

Offline synthesizer that discovers lemmas from program reasoning

Online adapter that refines lemmas based on evolving proof states

LEMMANET: Agentic VC Proving through lemma discovery



Augment proof agent (①›②›③›④) with 2-stage helper lemma discovery

Offline synthesizer that discovers lemmas from program reasoning

Online adapter that refines lemmas based on evolving proof states

Evaluation Setup

Comparison tools

3
baselines

Copra [1]
LLM proof agent

AutoRocq [2]
LLM proof agent; implementation baseline

CoqHammer [3]
Popular automated prover

Setup: 10 min per VC

Benchmarks

941
VCs

SV-COMP [4] 641 verification conditions

From 131 small C programs

NTP4VC [5] 300 verification conditions

Larger real-world programs, including
- Linux kernel code (scheduler, memory, string),
- C++ libraries, Contiki OS, X.509 parser, etc.

[1] Thakur, Amitayush, *et al*. "An in-context learning agent for formal theorem-proving." *COLM*, 2024.
[2] Tu, Haoxin, *et al*. "Agentic Verification of Software Systems." *FSE*, 2026.
[3] Czajka, Łukasz, *et al*. "Hammer for Coq: Automation for dependent type theory." *Journal of automated reasoning*, 2018.
[4] Beyer, Dirk, *et al*. "Improvements in Software Verification and Witness Validation: SV-COMP 2025." *TACAS*, 2025.
[5] Xu, Qiyuan, *et al*. Neural Theorem Proving for Verification Conditions: A Real-World Benchmark." *ICLR*, 2026.

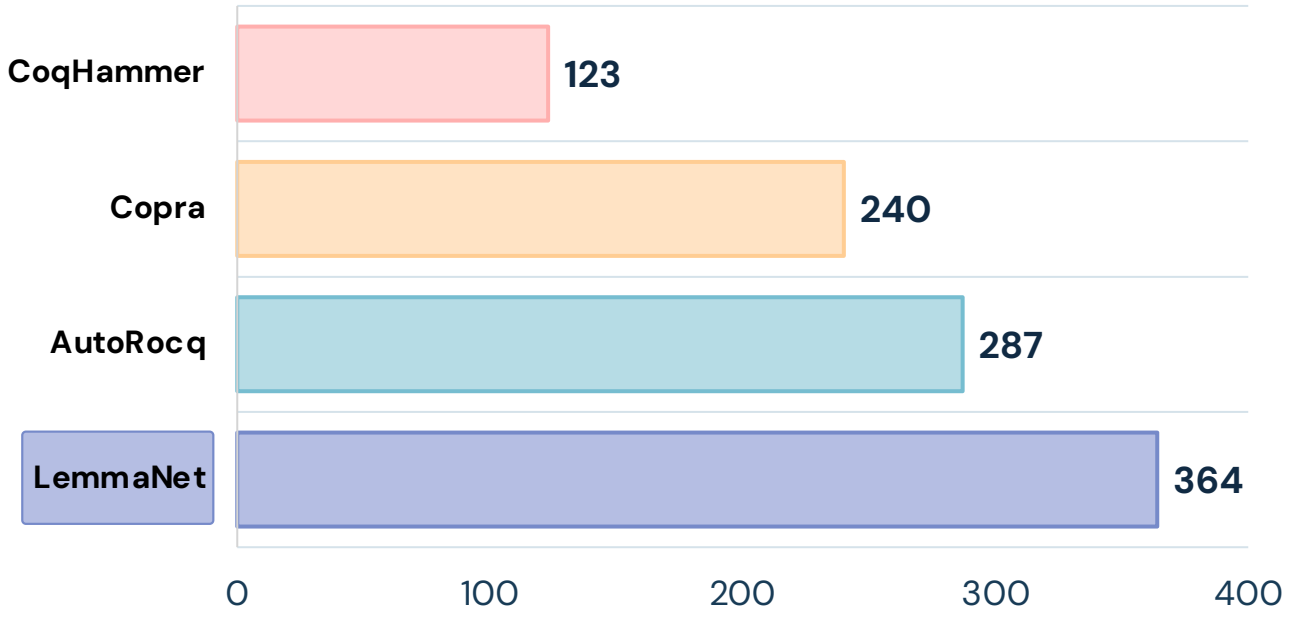
Efficacy in VC Proving

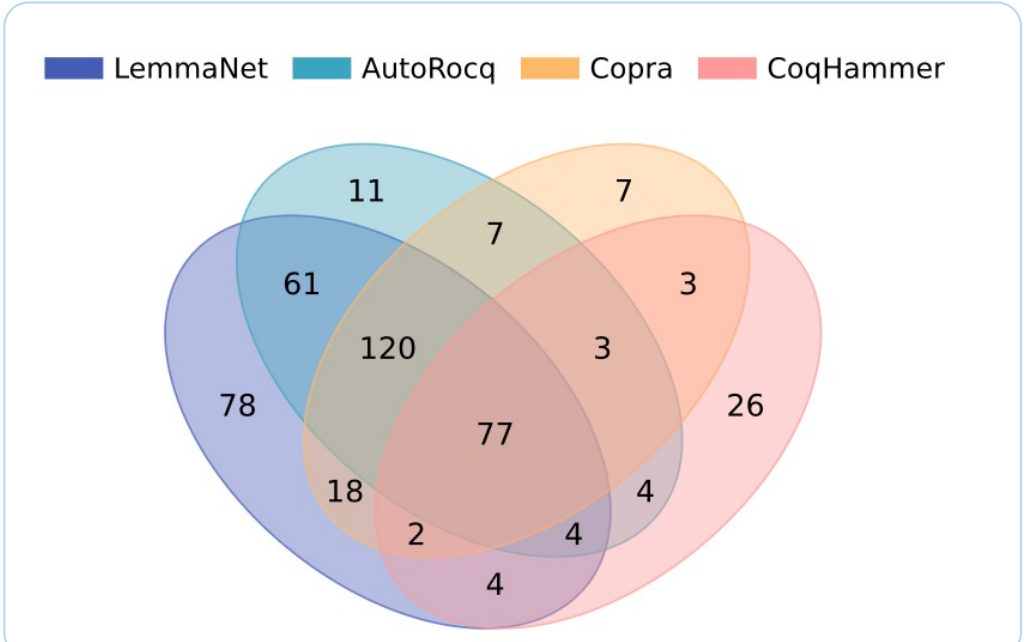
+26.8% VCs proved vs. AutoRocq across benchmarks

+65.0% VCs proved vs. AutoRocq on NTP4VC

78 VCs proved uniquely

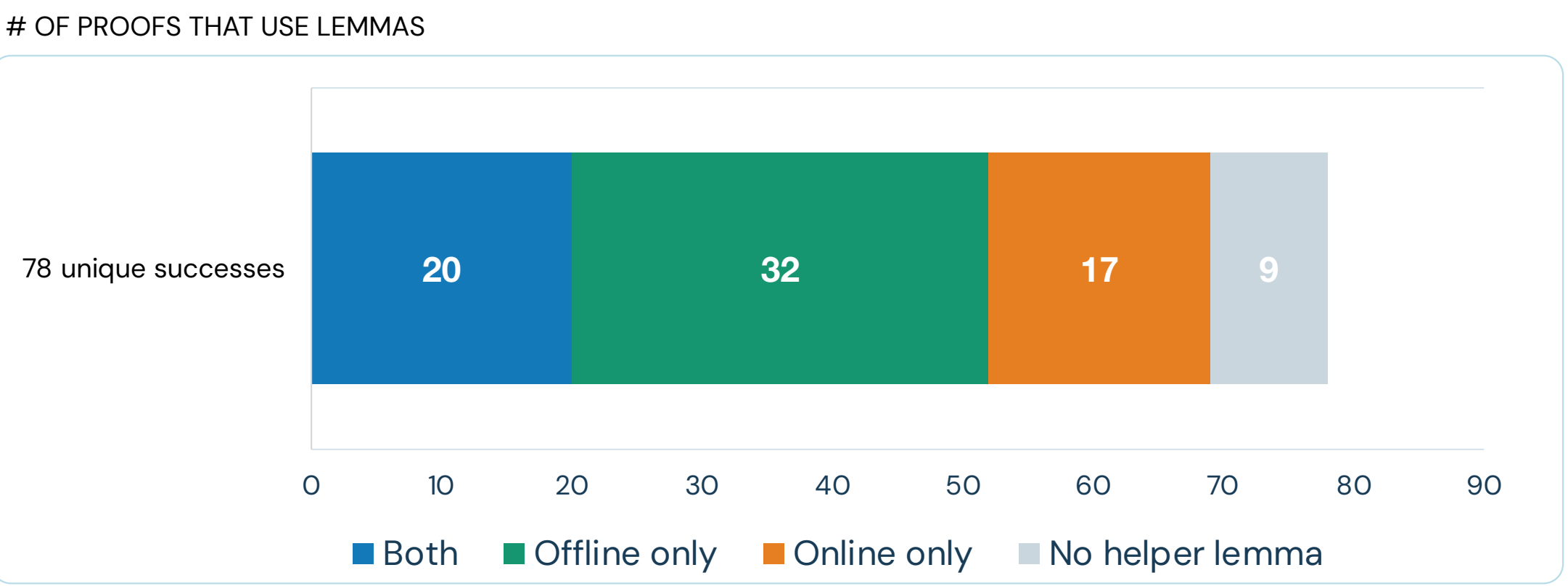
CoqHammer gives complementary results





Helper lemmas: do they help?

69 / 78 (**88.5%**) unique proofs by LemmaNet use a helper lemma

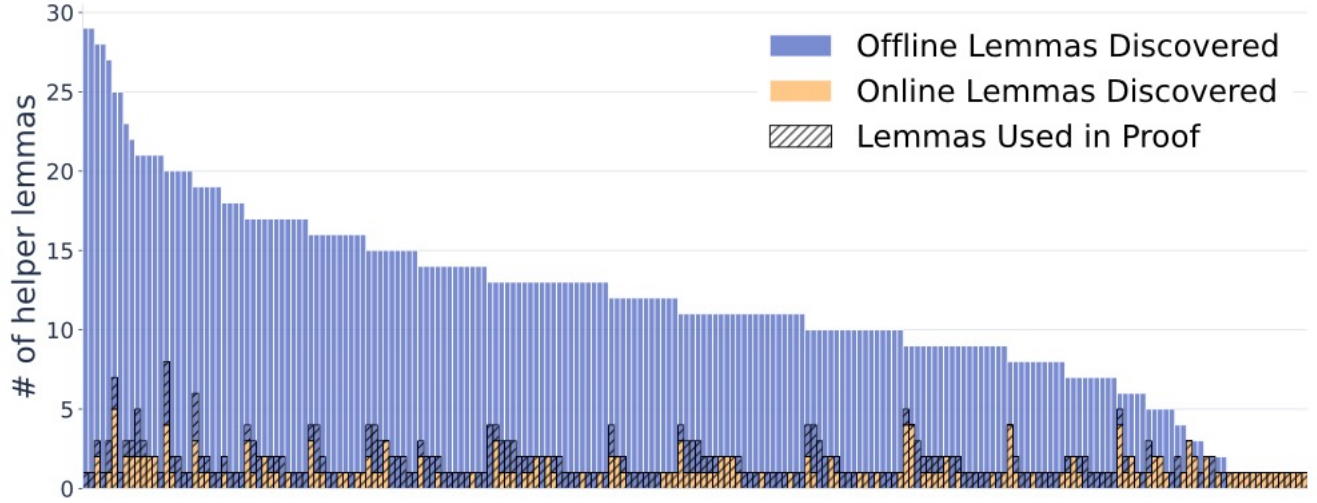


Helper lemmas: how many are needed?

[Offline Stage]
more helper lemmas are discovered

[Online Stage]
fewer are discovered;
targeted to live proof states and used

OF LEMMAS DISCOVERED & USED IN PROOF

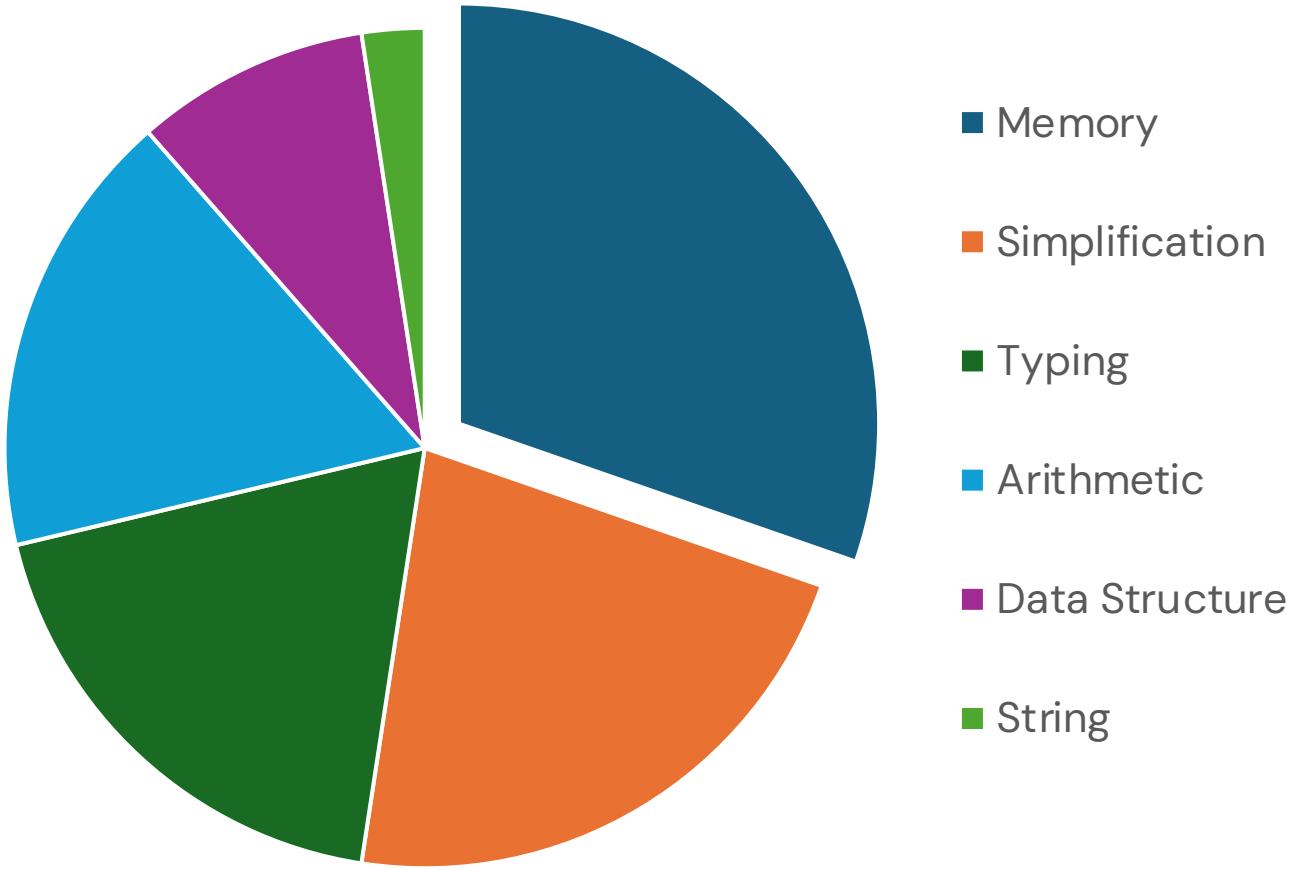


Helper lemmas: what do they do?

A wide range of lemmas can be discovered and leveraged

30.2% lemmas used are related to **memory** encoding

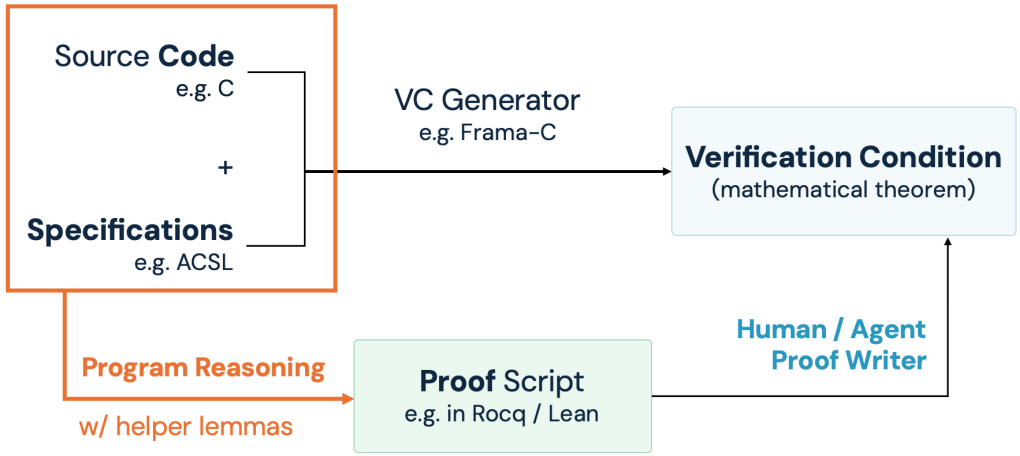
TYPE OF LEMMAS USED IN PROOF

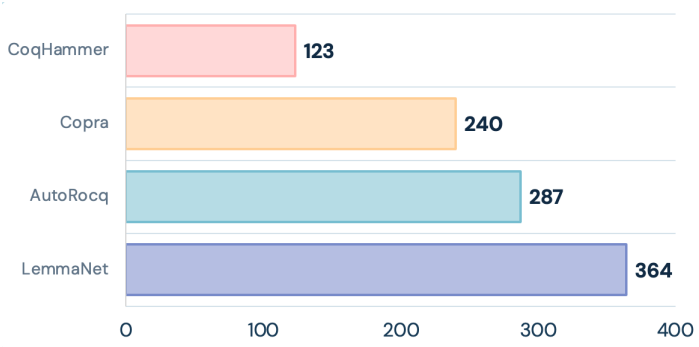


Summary of LEMMANET

Agentic verification with **helper lemmas**

- *Discovered* via program reasoning
- *Refined* based on evolving proof states





Paper

Code