Leveraging LLMs for Program Verification

Adharsh Kamath[1,3] Nausheen Mohammed[1,3] Aditya Senthilnathan[1,4]
Saikat Chakraborty[1] Pantazis Deligiannis[1] Shuvendu Lahiri[1]
Akash Lal[1] Aseem Rastogi[1] Subhajit Roy[2] Rahul Sharma[1]

[1]Microsoft Research

2

3

4

a,*b* : work done while at *a*, currently at *b*

Verifying safety

- Proving that control does not reach an "unsafe" program state

- Encoded as assertion(s) in a C program

- Inductive loop invariants are required for each loop. Such invariants are:

(i) true at the beginning of the loop

(ii) preserved by the loop body

(iii)* imply the assertion in question

Verifying safety

```
void main()
{
    int n = 0;
    int k = unknown_int();
    if (k < 0) return;
    while (n < k)
    {
        n++;
    }
    assert (n == k);
}
```

Verifying safety

Annotation →

```
void main()
{
    int n = 0;
    int k = unknown_int();
    if (k < 0) return;
    //@ invariant 0 <= n <= k
    while (n < k)
    {
        n++;
    }
    assert (n == k);
}
```

Verifying termination

- Proving that a loop terminates
- Requires a "ranking function" (variant), an expression that:
    (i) is non-negative at the beginning of every loop iteration
    (ii) strictly decreases with every iteration
- Could be templates: lexicographic variants, multi-phase variants
- Often require supporting inductive invariants

Verifying termination

```
void main()
{
    int n = 0;
    int k = unknown_int();
    if (k < 0) return;
    //@ invariant 0 <= n <= k
    while (n < k)
    {
        n++;
    }
}
```

Verifying termination

Annotation →

```
void main()
{
    int n = 0;
    int k = unknown_int();
    if (k < 0) return;
    //@ invariant 0 <= n <= k
    //@ variant k - n
    while (n < k)
    {
        n++;
    }
}
```

Pre-conditions, Post-conditions

- Desribing the behavior of a method

- Formula involving the input and output values of the method

- Loops in the method body require loop invariants

Pre-conditions, Post-conditions

```
int sum(int n, int m) {
    if (n == 0) {
        return m;
    } else {
        return sum(n - 1, m + 1);
    }
}
```

Pre-conditions, Post-conditions

Annotation →

```
//@ requires n >= 0 && m >= 0;
//@ ensures \result == n + m;
int sum(int n, int m) {
    if (n == 0) {
      return m;
    } else {
      return sum(n - 1, m + 1);
    }
}
```

Program verification tasks

- Broken down to:
  - Annotation inference: requires ingenuity
  - Automated verification: automatable (SMT solvers!)

Program verification tasks

- Broken down to:
  - Annotation inference: requires ingenuity
  - Automated verification: automatable (SMT solvers!)

What if we use LLMs for annotation inference, and SMT solvers for automated verification?

Loopy

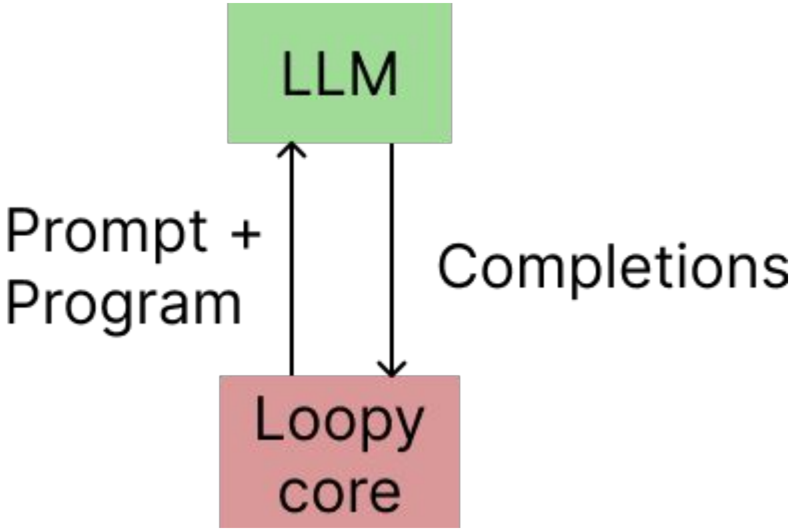
- Loopy is a toolchain that uses LLMs and classical (symbolic) tools in a guess-and-check setting

- Can be instantiated with different LLMs, different checkers

- Uses *Houdini* – finds the largest subset of inductive invariants

- Can be used for different tasks – loop invariants, ranking functions, pre/post-conditions

Loopy$_{safe}$



Loopy$_{safe}$: prompt

- Prompt contains
    - Loop invariant definition

```
Instructions:
- Make a note of the pre-conditions or variable assignments in the program.
- Analyze the loop body and make a note of the loop condition.
- Output loop invariants that are true
(i) before the loop execution,
(ii) in every iteration of the loop and
(iii) after the loop termination,
such that the loop invariants imply the post condition.
- If a loop invariant is a conjunction, split it into its parts.
```

Loopy$_{safe}$: prompt

- Prompt contains
  - Loop invariant definition
  - Output syntax

````
- Output all the loop invariants in one code block. For example:
```
/*@
    loop invariant i1;
    loop invariant i2;
*/
```
````

Loopy$_{safe}$: prompt

- Prompt contains
    - Loop invariant definition
    - Output syntax
    - Additional "rules" for generating invariants

```
Rules:
- **Do not use variables or functions that are not declared in the program.**
- **Do not make any assumptions about functions whose definitions are not given.**
```

$\text{Loopy}_{\text{safe}}$: prompt

- Prompt contains
    - Loop invariant definition
    - Output syntax
    - Additional "rules" for generating invariants

Based on failure cases we added more "nudges" to hint at likely invariants

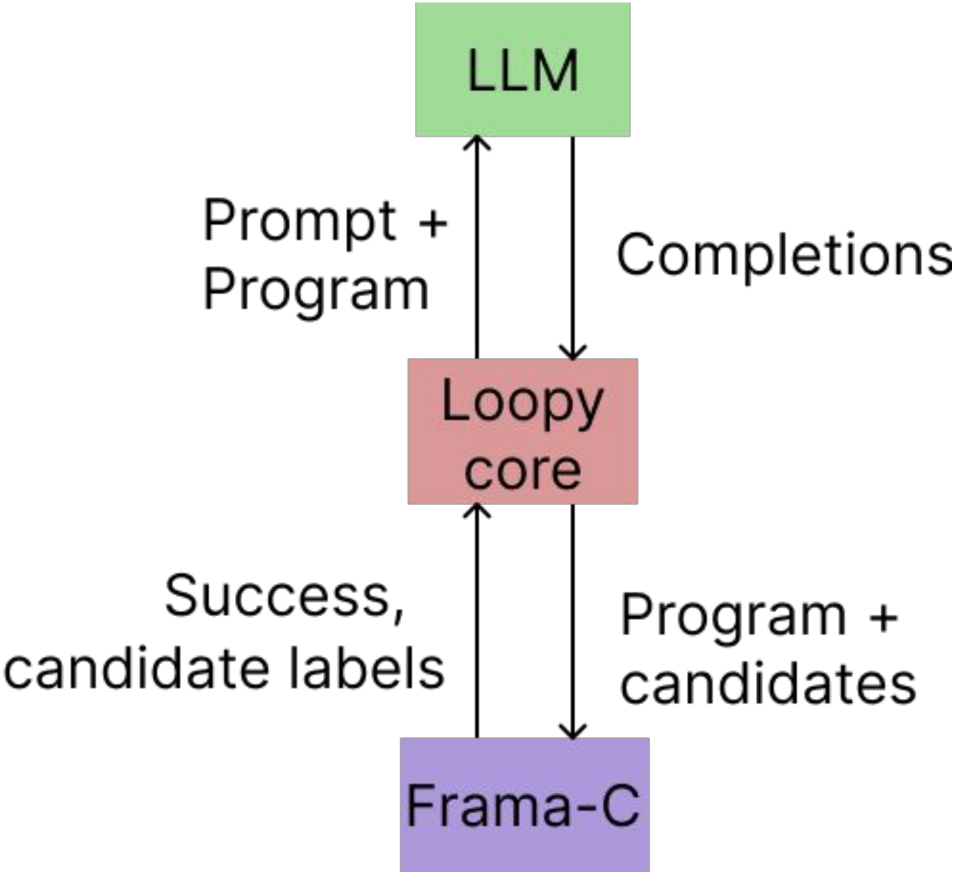
$\text{Loopy}_{\text{safe}}$: prompt

- Prompt contains
    - Loop invariant definition
    - Output syntax
    - Additional "rules" for generating invariants
    - "Nudges"

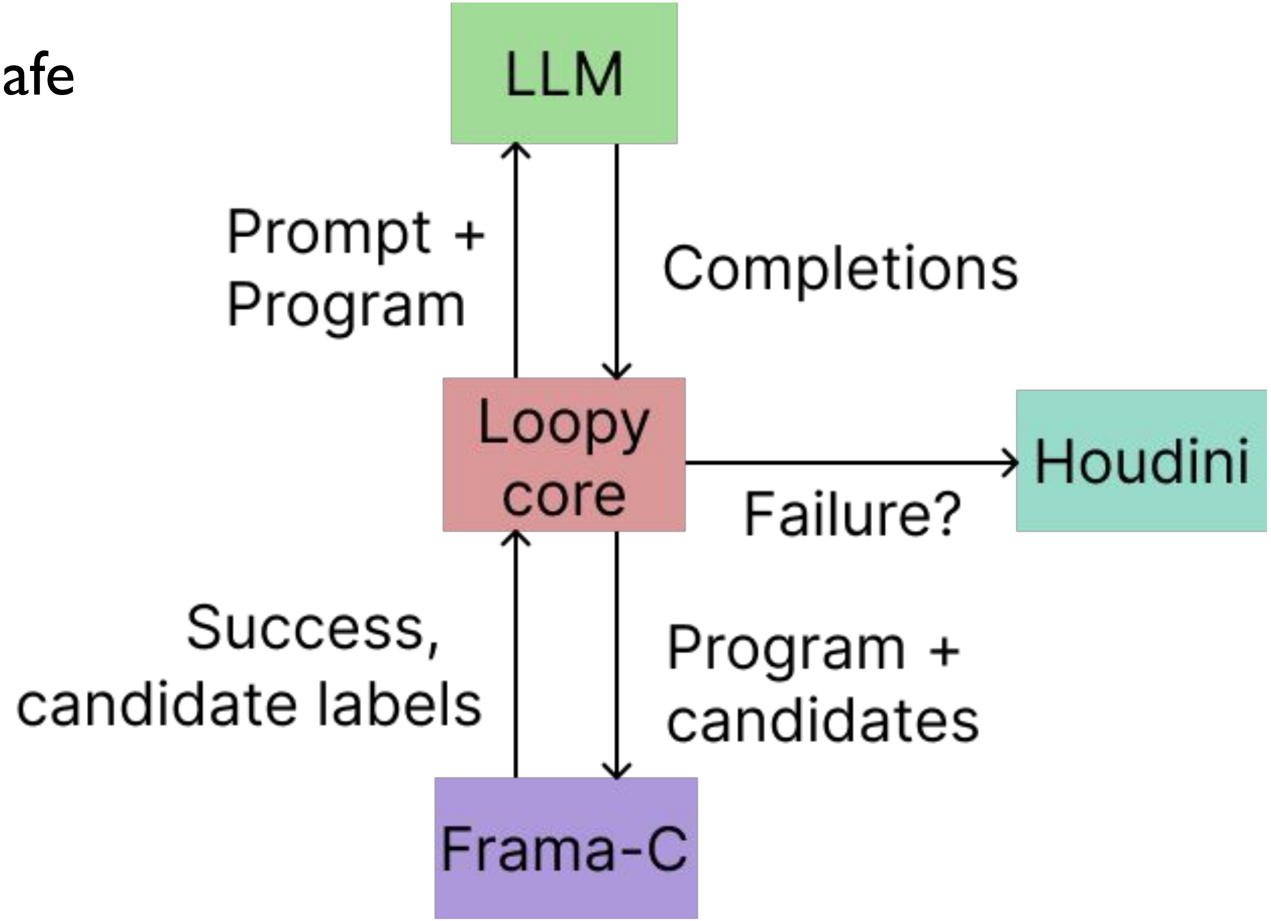
```
For all variables, add conjunctions that bound the maximum and
minimum values that they can take, if such bounds exist.

If a variable is always equal to or smaller or larger than
another variable, add a conjunction for their relation.
```

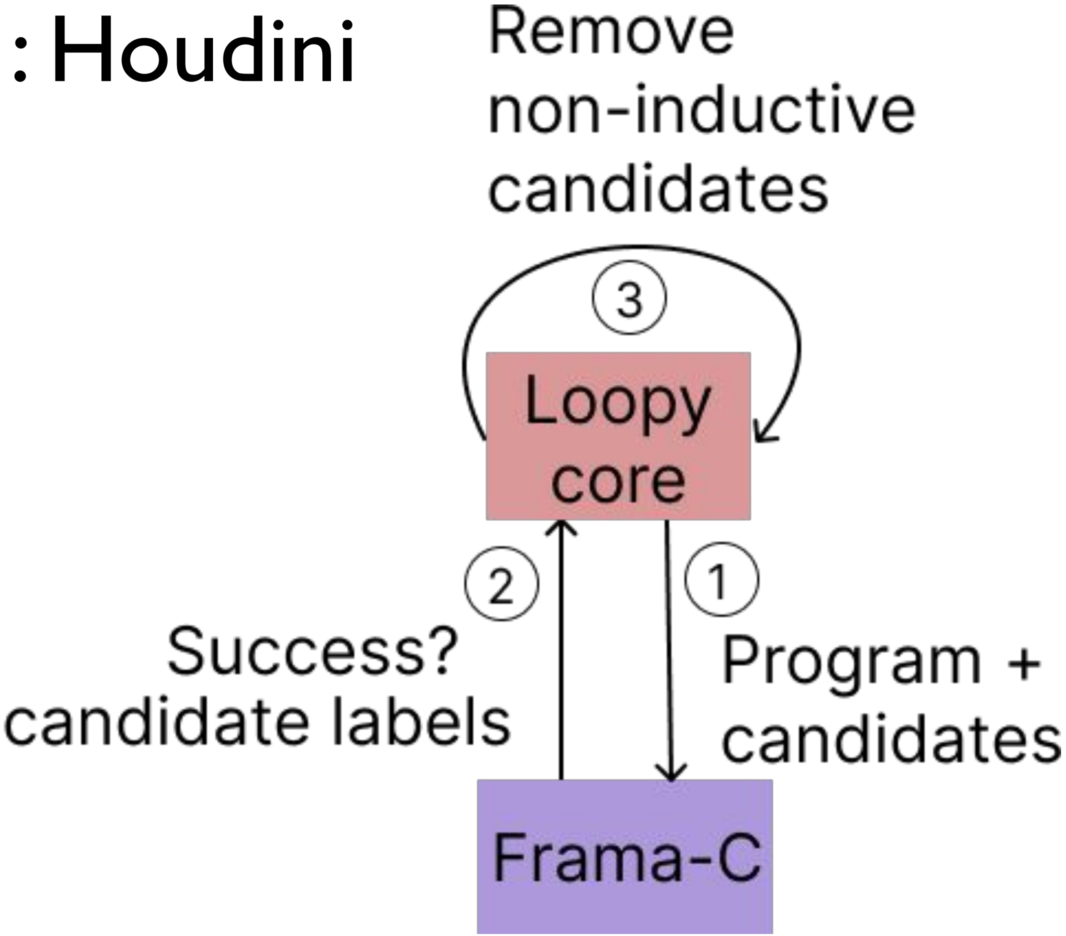
$Loopy_{safe}$


LLM
Prompt +
Program
Completions
Loopy
core
Success,
candidate labels
Program +
candidates
Frama-C

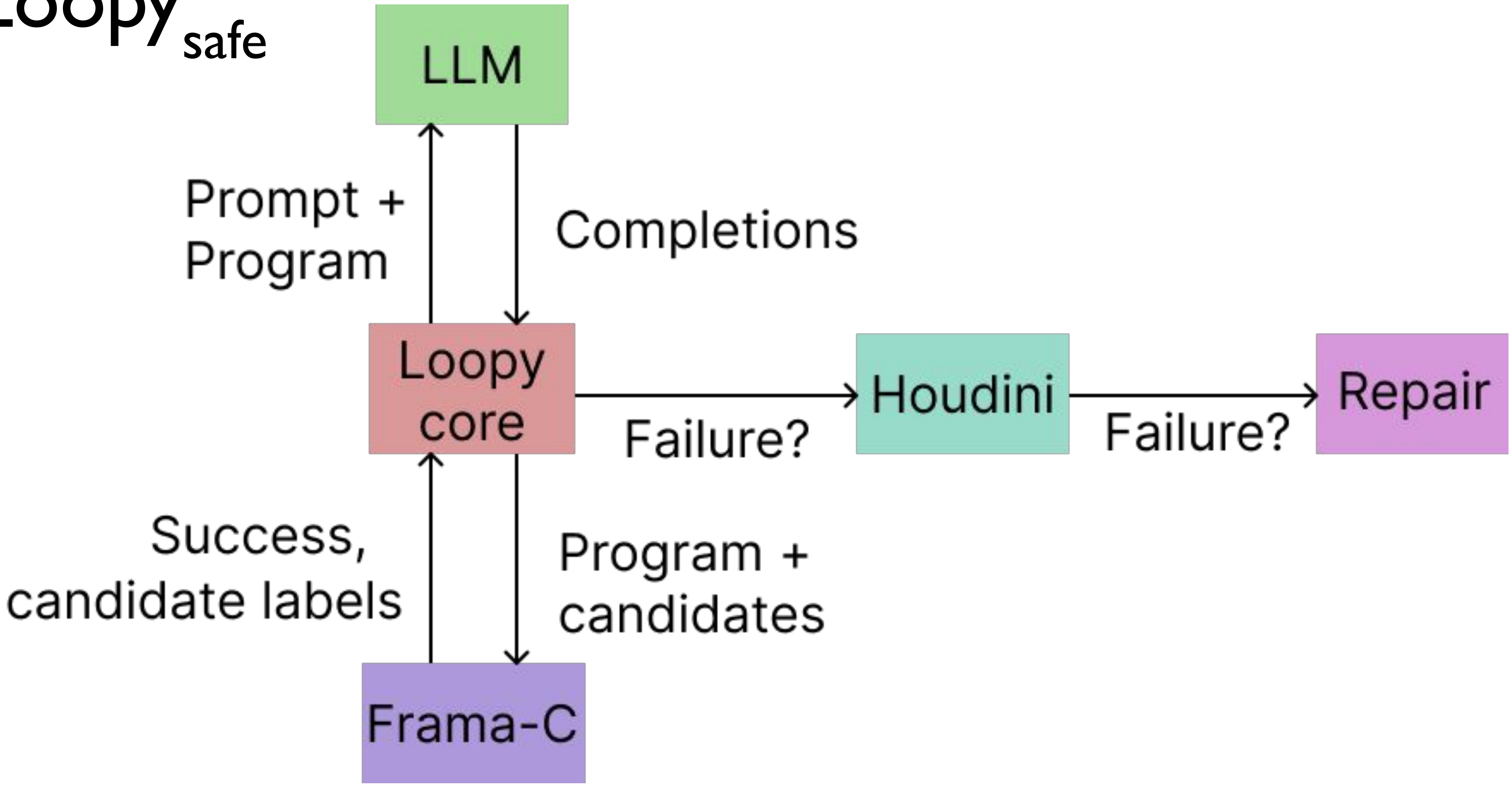
Loopy$_{safe}$


LLM
Prompt +
Program
Completions
Loopy
core
Houdini
Failure?
Success,
candidate labels
Program +
candidates
Frama-C

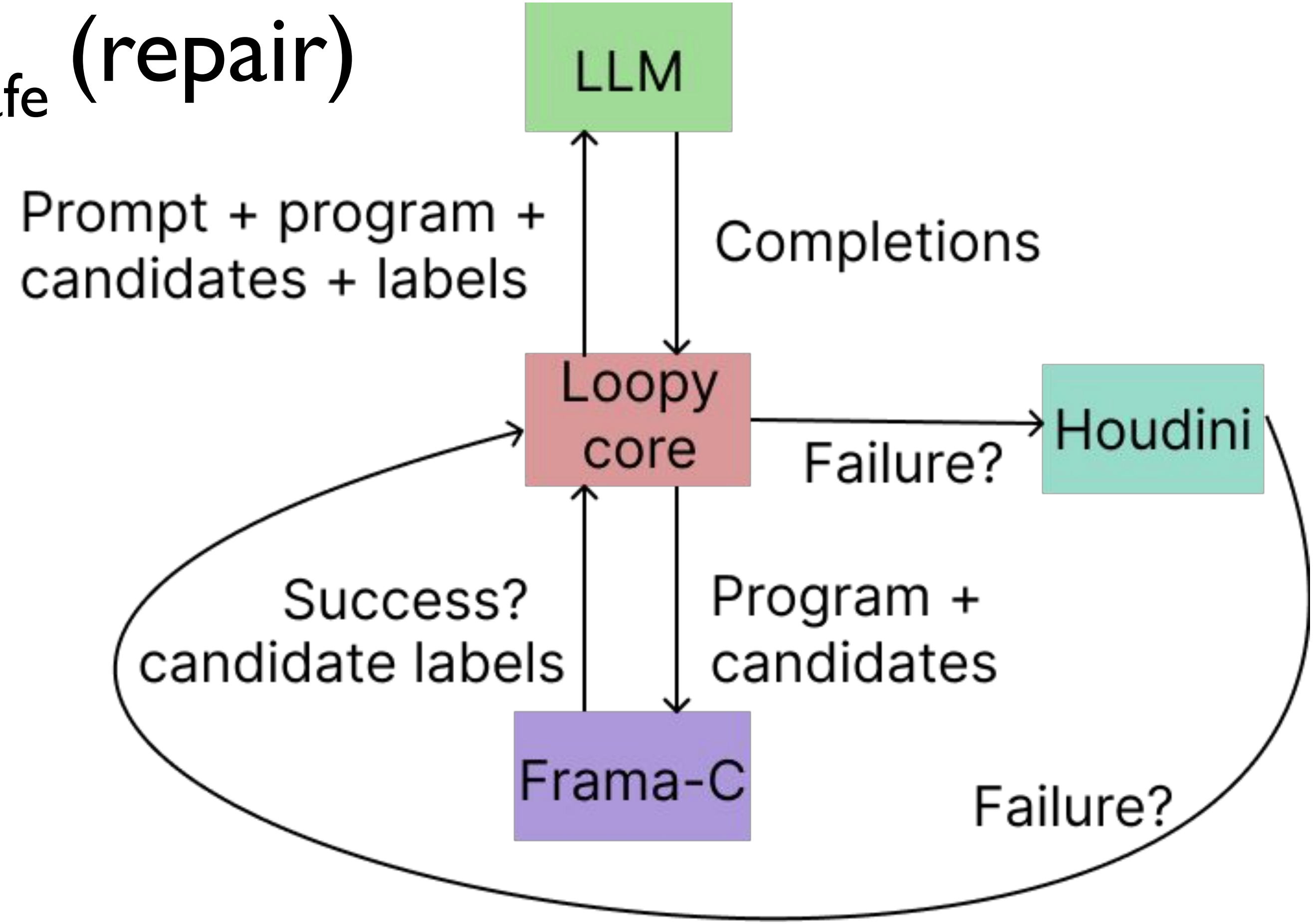
Loopy$_{safe}$: Houdini



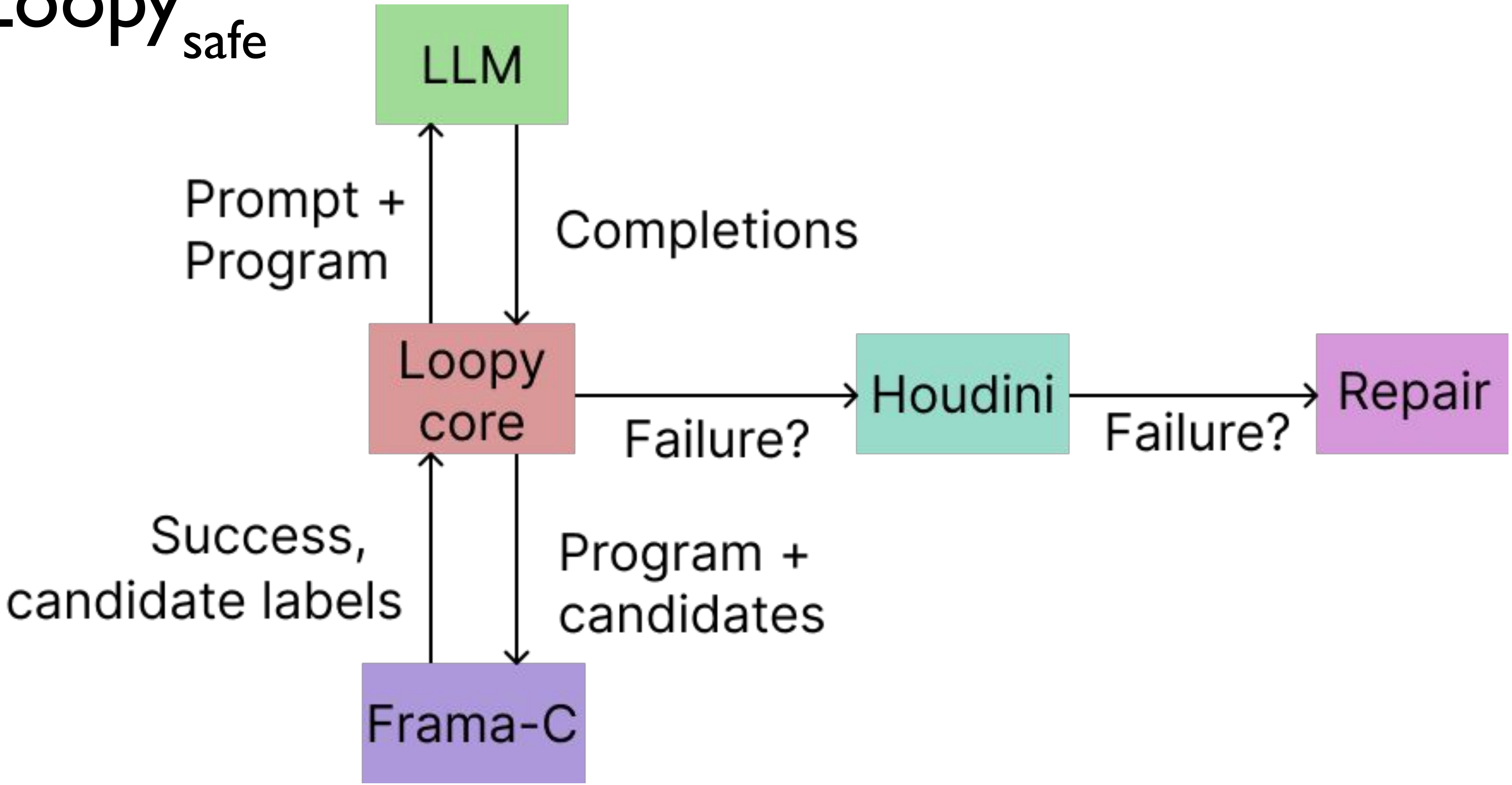
$\text{Loopy}_{\text{safe}}$



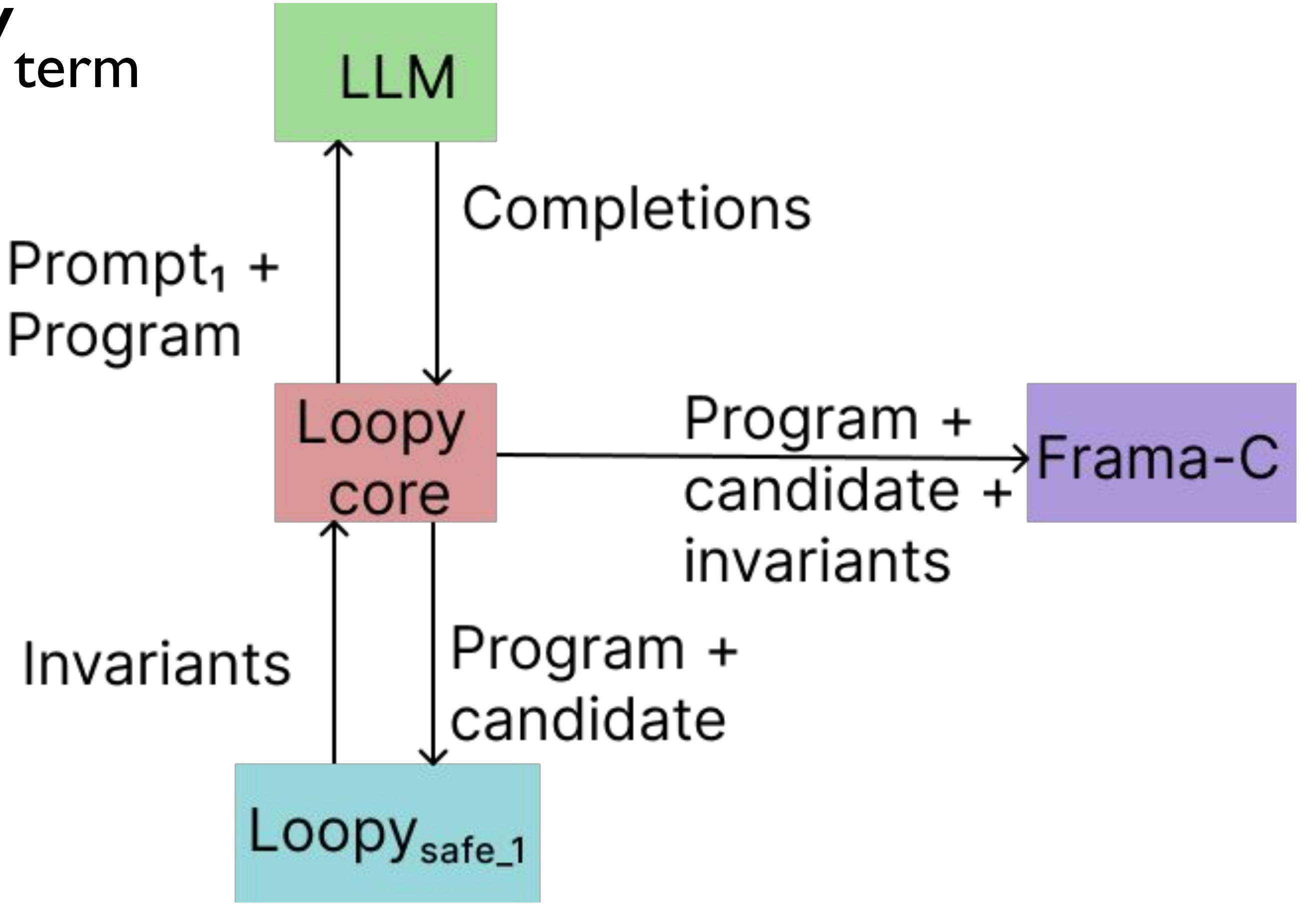
Loopy$_{safe}$ (repair)



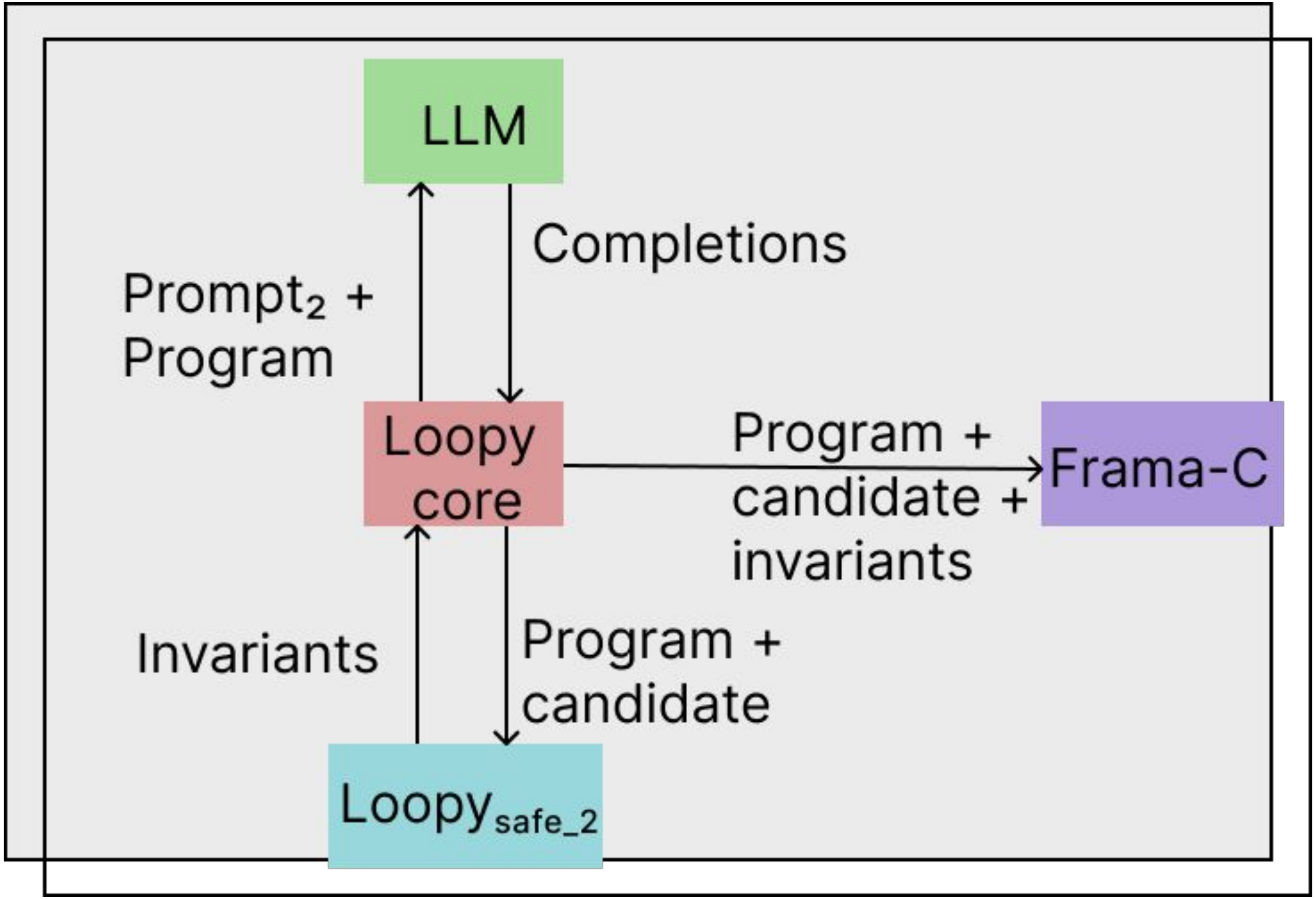
Loopy$_{safe}$


LLM
Prompt +
Program
Completions
Loopy
core
Failure?
Houdini
Failure?
Repair
Success,
candidate labels
Program +
candidates
Frama-C

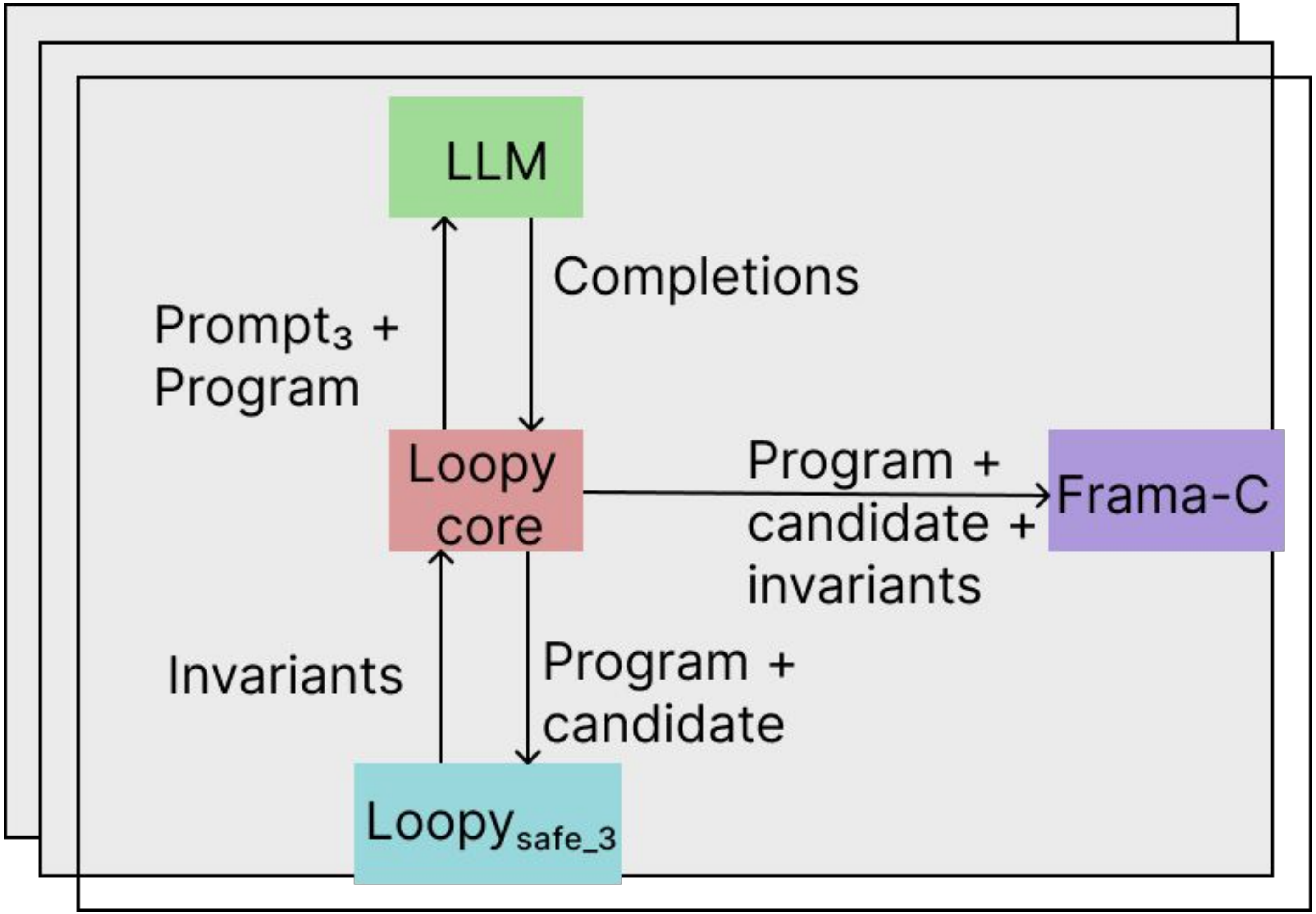
$\text{Loopy}_{\text{term}}$



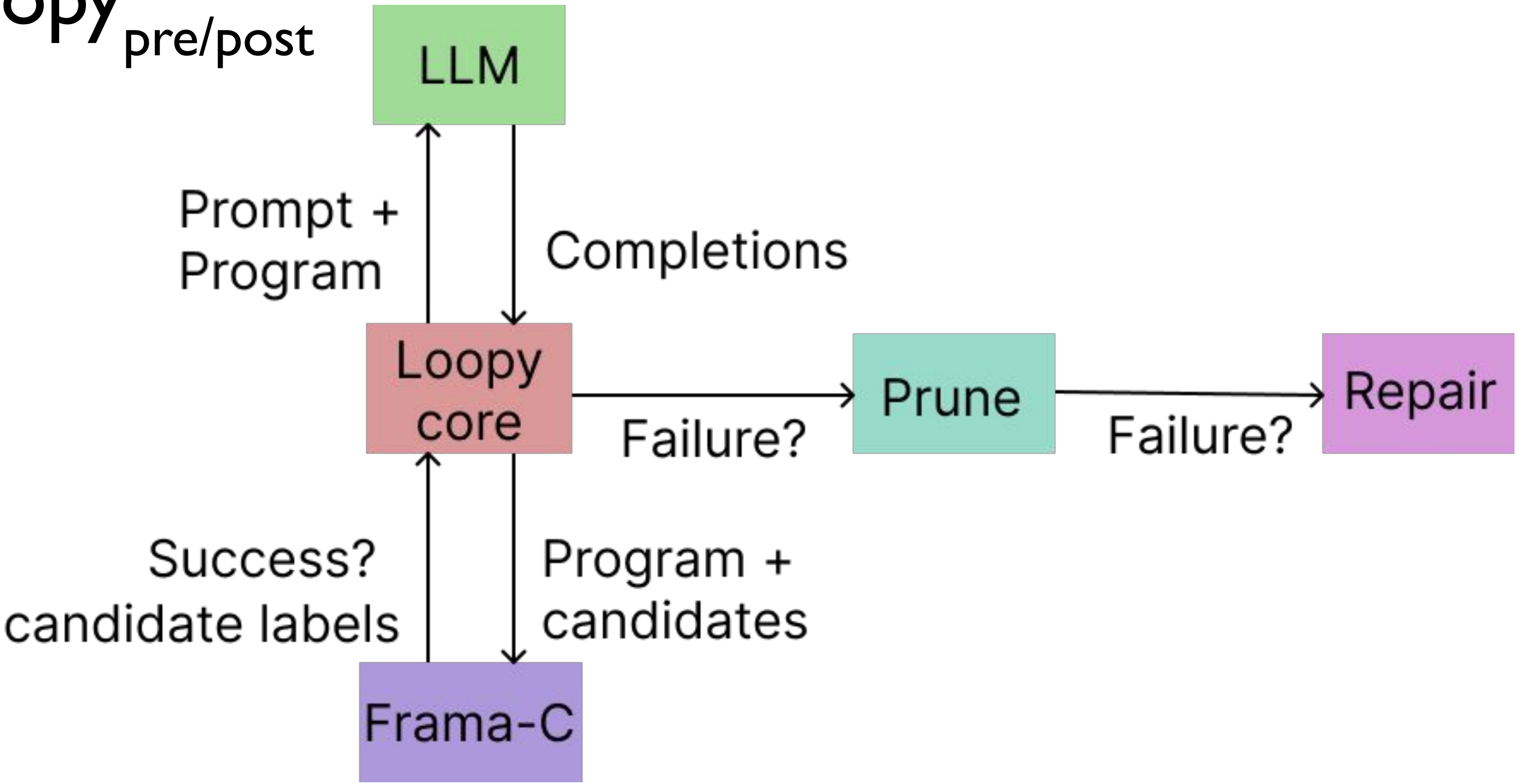
$\text{Loopy}_{\text{term}}$



Loopy_{term}



$\text{Loopy}_{\text{pre/post}}$



Benchmarks

Name	Size	Features	Sources
Scalar loops	469	one loop, one method, no arrays	SVCOMP, Code2Inv, etc.
Array loops	169	≥ one loop, one method, ≥ one array	Diffy
Termination	281	one loop, one method, no arrays	SVCOMP, TermComp
Recursive	32	no loops, ≥ one recursive method	SVCOMP

Experiments

Compare Loopy instantiated with different LLMs

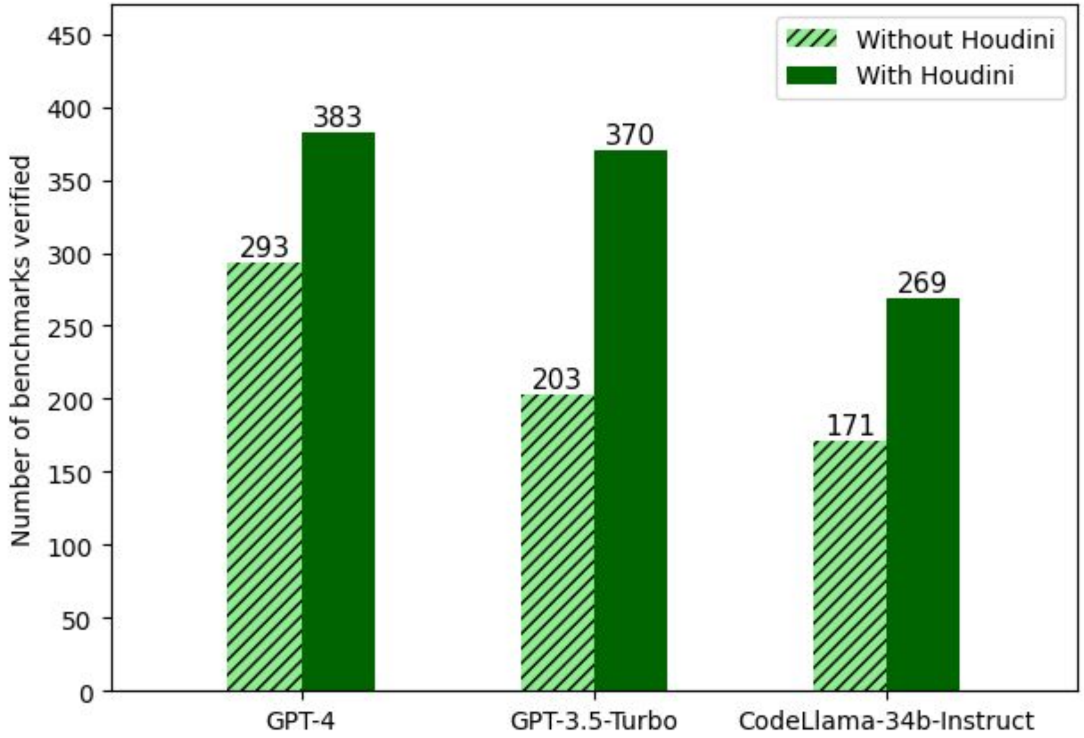
Compare Loopy with and without Houdini in each case

(on Scalar Loops)

Result

Number of benchmarks verified:

(scalar loops)



Experiments

Compare Loopy with "vanilla LLMs" – no elaborate prompt, no Houdini, no repair

(on all benchmarks)

Results

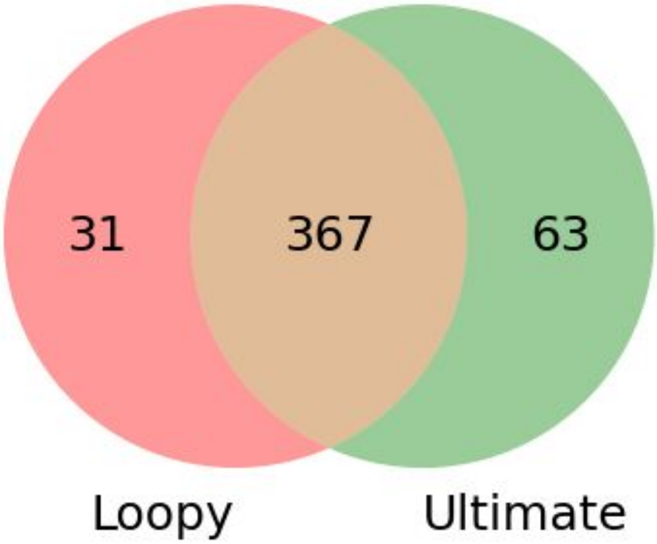
Name	Vanilla LLMs	Loopy
Scalar loops	51%	**85%**
Array loops	36%	**75%**
Termination	17%	**64%**
Recursive	45%	**52%**

Experiments

Compare Loopy with a symbolic tool – *Ultimate Automizer* across all the verification tasks

Results

Comparing Loopy with GPT-4 against Ultimate:
(on scalar loops)



Results

Name	Loopy	Ultimate	Loopy ∪ Ultimate
Scalar loops	85%	92%	**98%**
Array loops	75%	7%	**75%**
Termination	64%	84%	**91%**
Recursive	52%	65%	**74%**

Results

Loopy has been integrated into other tools and has shown value:

AutoVerus: Automated Proof Generation for Rust Code

(arxiv.org/abs/2409.13082)

Results

Lemur and Loopy:

(with equal LLM-query budget)

Benchmark	Lemur	Loopy
Code2Inv (133)	107	103
SVCOMP (50)	26	26

Extending Loopy

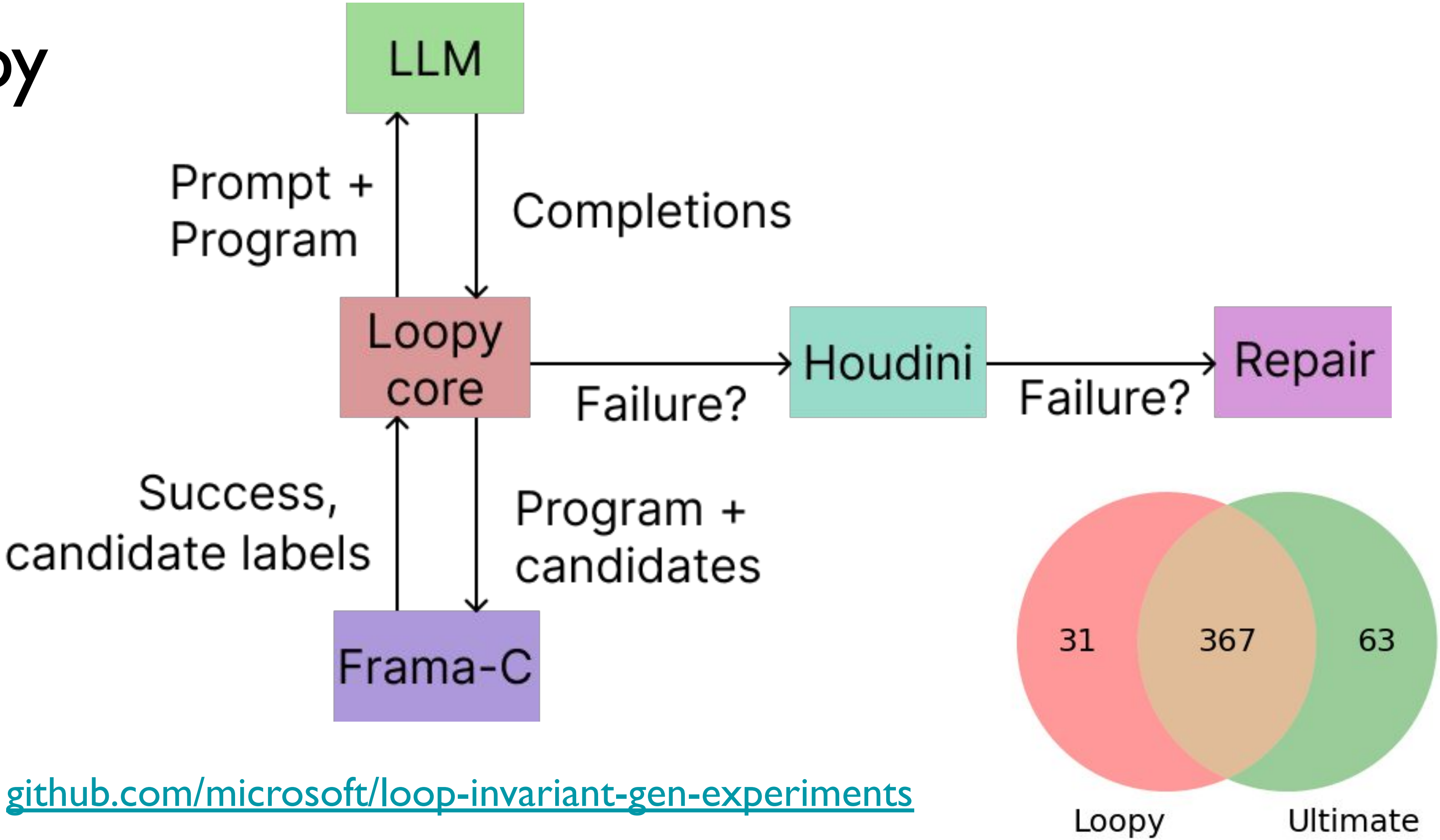
Fails to infer an inductive loop invariant here →

Common failure modes:

- Disjunctions in invariants
- Invariants with >3 terms

```
void main()
{
    int i, sn = 0;
    int SIZE = unknown_int();
    for (i = 1; i <= SIZE; i++)
    {
        sn = sn + 1;
    }
    assert(sn == SIZE * 1 ||
              sn == 0);
}
```

Loopy



github.com/microsoft/loop-invariant-gen-experiments