

After the Hitchhiker's Guide

Toward Principled LLM Assistance for Static Analysis

Haonan Li

UC Riverside · SPLASH Doctoral Symposium

Part I · 2023

Why use LLMs?

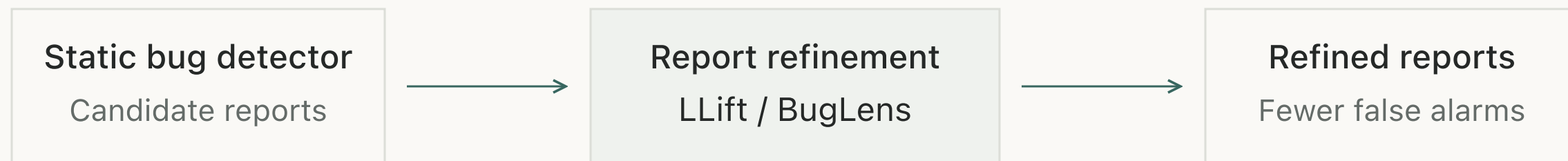
LLift · BugLens

Part II · 2026

Why not just use an LLM?

Toward principled assistance

Report Refinement



LLift · OOPSLA 2024

UBI (use before initialization)

BugLens · ASE 2025

Taint-style bugs

Uninitialized Variables

```
unsigned int a, b, c, d;  
int n;  
  
if (sscanf(str, "%u.%u.%u.%u%n",  
          &a, &b, &c, &d, &n) >= 4) {  
    // use a, b, c, d  
}
```

Reported warning

a, b, c, d may be used before initialization.

The fact we need

For this format, a return value ≥ 4 means all four were initialized.

Why this is hard

The analyzer must connect initialization inside `vsscanf` to its return value.

LLift · OOPSLA 2024

UBITect reports a use without an established `must_init`.

01 Input

User (use)

The variable, its use, and relevant code

02 LLM reasoning

Initializer + guards

When this use is reached, must initialization have happened?
Here: `sscanf` with `r ≥ 4`.

03 Output

`must_init` / `may_init`

Must-init: dismiss the warning.
Otherwise: keep it.

The LLM determines whether initialization must happen before the use.

BugLens · ASE 2025

01 · Security Impact Assessor

Is this sink actually dangerous?

Could arbitrary input here cause memory corruption or DoS?

Example: timer arithmetic can wrap intentionally.

02 · Constraint Assessor

When do the checks work?

```
if (config && num > ARR_SIZE)
    num = ARR_SIZE;
```

Precondition	Postcondition
<code>config != 0</code>	<code>num ≤ ARR_SIZE</code>
<code>config == 0</code>	<code>num unchanged</code>

The LLM summarizes pre/postconditions and solves the combined constraints.

Published Results

LLift · Linux bug discovery

4

Community-confirmed bugs

From the extension to 1,000
undecided cases.

BugLens · Precision

0.44 → 0.72

Higher precision

Suture refinement → BugLens.

BugLens · False positives

25 → 9

Fewer false alarms

Same 251 Suture warnings.

Our earlier work shows that LLMs can make static bug detectors more precise.

Part II · 2026

Why not just use an LLM?

When it says “no bug,”
what supports that answer?

A Missed Bug

Warning: out-of-bounds write

```
int items[16];  
if (ucontrol->value[0] > 16)  
    return -EINVAL;  
int n = ucontrol->value[0];  
for (int i = 1; i < n; ++i)  
    items[i] = ucontrol->value[i];
```

No-bug explanation · paraphrased

The guard bounds the loop.

So the array write appears safe.

But what happens at

`n = ucontrol->value[0];`

long to int

```
int n          =      ucontrol->value[0];
```

32-bit destination

Truncates here

64-bit source

Before "=" · long · 64 bits

~~0xFFFFFFFF~~00000011

The upper 32 bits are discarded.

$\text{value}[0] = -(2^{32} - 17)$

$\text{value}[0] > 16$ is false.

The guard does not return.

After "=" · int · 32 bits

0x00000011

The low 32 bits become positive 17.

$n = 17$

At $i = 16$, $i < n$ is true.

The write `items[16]` is out of bounds.

Justifying “No Bug”

Finding a bug

One failing input is enough.

`value[0] = -(232 - 17)`

It reaches `items[16]`.

Dismissing a warning

All allowed inputs matter.

We need to establish that none can reach `items[16]`.

Taking the guard, conversion, and loop into account.

We need a more rigorous way to decide whether this warning can be dismissed.

The “new” direction

Let program analysis
analyze the program.

Give the task of checking the warning
back to the analyzer.

Inform the Analyzer

LLM output · sscanf

If $r \geq 4$,
a, b, c, d are initialized.

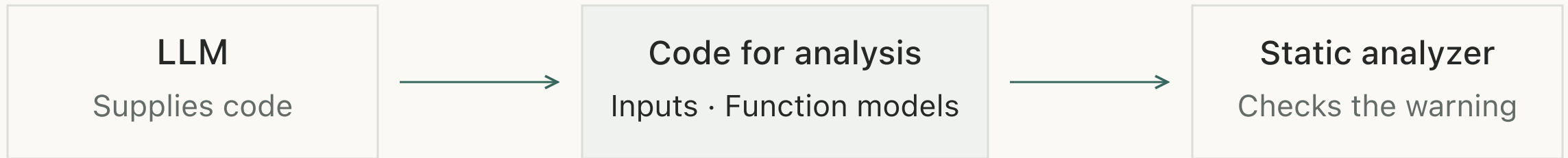
The analyzer

“Not a bug” for
use(a, b, c, d).

The analyzer uses the LLM output
to reach this conclusion.

New problem: how do we pass this information to the analyzer?

Code as a Bridge



The code makes the LLM's rationale explicit.

Checking the supplied code

Are we still analyzing the same program?

Explicit does not mean correct.

We still need to check the code against the original.

Input Restrictions

The original driver

Negative inputs are allowed.

`value[0] = -(232 - 17)`

It passes the guard, becomes `n = 17`, and reaches `items[16]`.

Suppose the LLM supplies · pseudocode

```
value[0] = choose(0, 16);  
call_driver(value);
```

choose: any value in the given range.

The LLM has added an input restriction.

We can reject this restriction with a clear rule.

Left = Right?

The original code

```
r = sscanf(str, "%u.%u.%u.%u%n",
           &a, &b, &c, &d, &n);
if (r >= 4)
    use(a, b, c, d);
```

The warning concerns `a`, `b`, `c`, `d`.

LLM-proposed summary · pseudocode

```
on return r >= 4:
    must_init(&a) && must_init(&b)
    && must_init(&c) && must_init(&d)
    && may_init(&n)
```

`must_init`: always initializes.
`may_init`: may initialize.

Checking the equivalence is still hard.

Roadmap

Published **LLift and BugLens refine static-analysis reports.**
They reduce false alarms and help find real bugs.

Implemented **The analyzer checks the warning.**
The LLM supplies input setup and models of code.

Next **Does the model match the original code?**
For a function or a code fragment.

Let the analyzer check the warning.
Check what the LLM supplies.

Thank you

I'd welcome your questions and thoughts.

Website

haonan.me

Email

hli333@ucr.edu