Towards More Accurate Static Analysis For Taint-style Bug Detection in Linux Kernel

aka: **The Hitchhiker's Guide to Program Analysis, Part II: Deep Thoughts by LLMs**

Haonan Li[1], Hang Zhang[2], Kexin Pei[3], Zhiyun Qian[1]

[1]UC Riverside & [2]IU Bloomington & [3]UChicago

ABSTRACT:

- We propose **BugLens**, an LLM-based post-refinement tool for static analysis
- BugLens improves the **precision** of Suture (an advanced taint-style bug detector) from **0.1 to 0.72**
- LLM makes static analysis better!

Background: Taint-style bugs

```c
int main(){
    char buf[16];
    int sz = get_user_input();

    for(int i = 0; i < sz; i++){
        use(buf[i]);
    }
}
```

- **A typical taint-style bug:**
  - "sz" is **tainted** (user-controlled)
  - May cause out-of-bound (OOB) access
- **Static analysis 101:**
  - Define **sources** (e.g., get_user_input())
  - Define **sinks** (e.g., array access buf[i])
  - Run taint propagation → easy to detect

Challenge 1: Complex Sink

```c
int use(char value){
    if (value == '\0'){
        return -1;
    }
    return 0;
}

int main(){
    char buf[16];
    buf[15] = '\0';
    int res;
    int sz = get_user_input();

    for(int i = 0; i < sz; i++){
        res = use(buf[i]);
        if (res == -1) break;
    }
}
```

- **Sink:** array access buf[I] 🤔
- **Static analysis view:**
  - "buf[i] may be tainted"
  - "Loop depends on tainted sz"
  - Possible OOB access ⚠️
- **But safe:**
  - buf[15] = '\0' ensures early termination
  - use() stops loop when '\0' seen

Challenge 2: Complex Sanitization

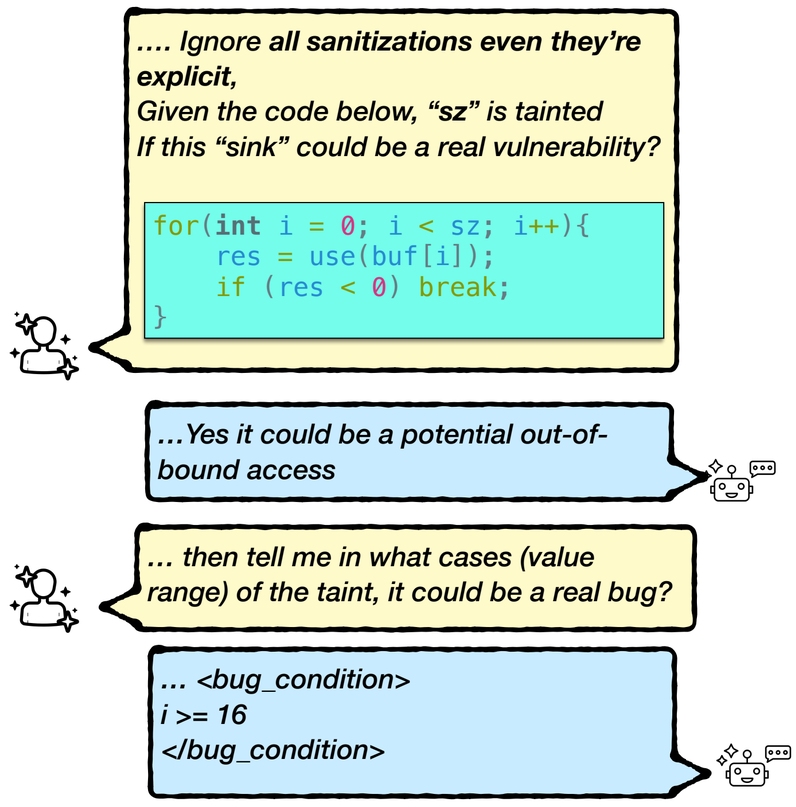
```c
int main(){
    char buf[16];
    int sz = get_user_input();
    sz = sz % 16;

    for(int i = 0; i < sz; i++){
        use(buf[i]);
    }
}
```

- Sanitizer: sz = sz % 16
- Sanitization can be various in real world

Takeaway: Static Analysis Needs LLMs

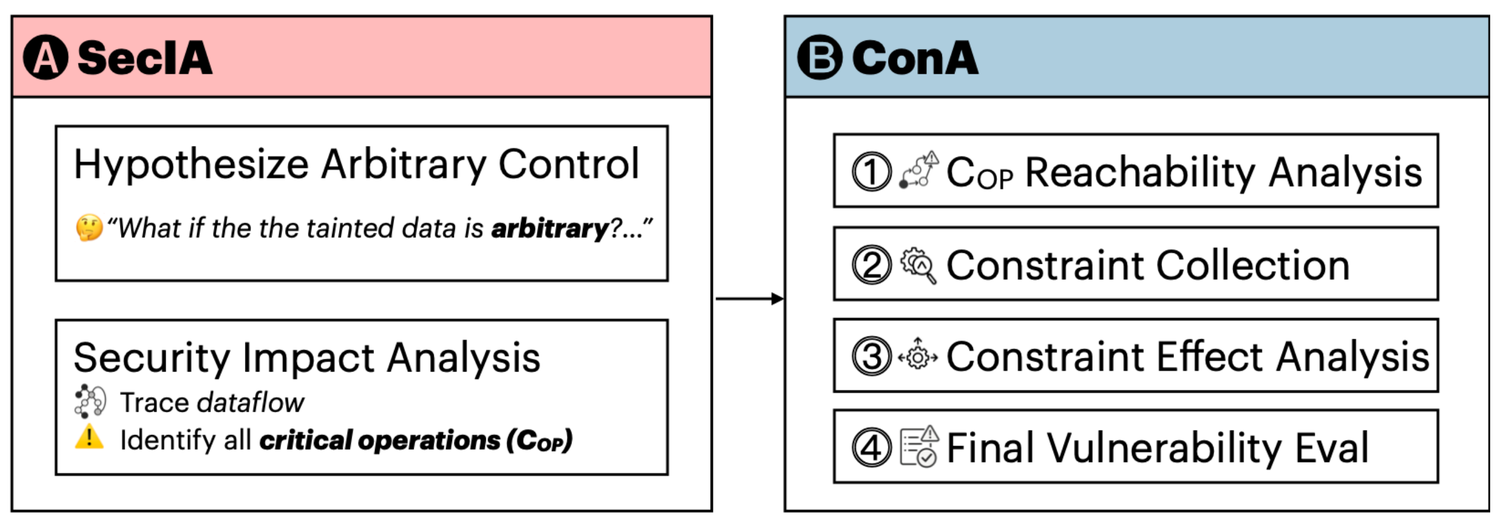
- Real-world programs are semantically complex and diverse
  - infinitely many patterns, corner cases, and project-specific idioms
- Accurate modeling of sinks, sanitization, and control/data-flow
  - hard to hand-craft and often a long-tail problem
- Static analyzers rely on **rigid, rule-based models**
- LLMs can understand code semantics and intent

SecIA: Hypo on Arbitrary Controls

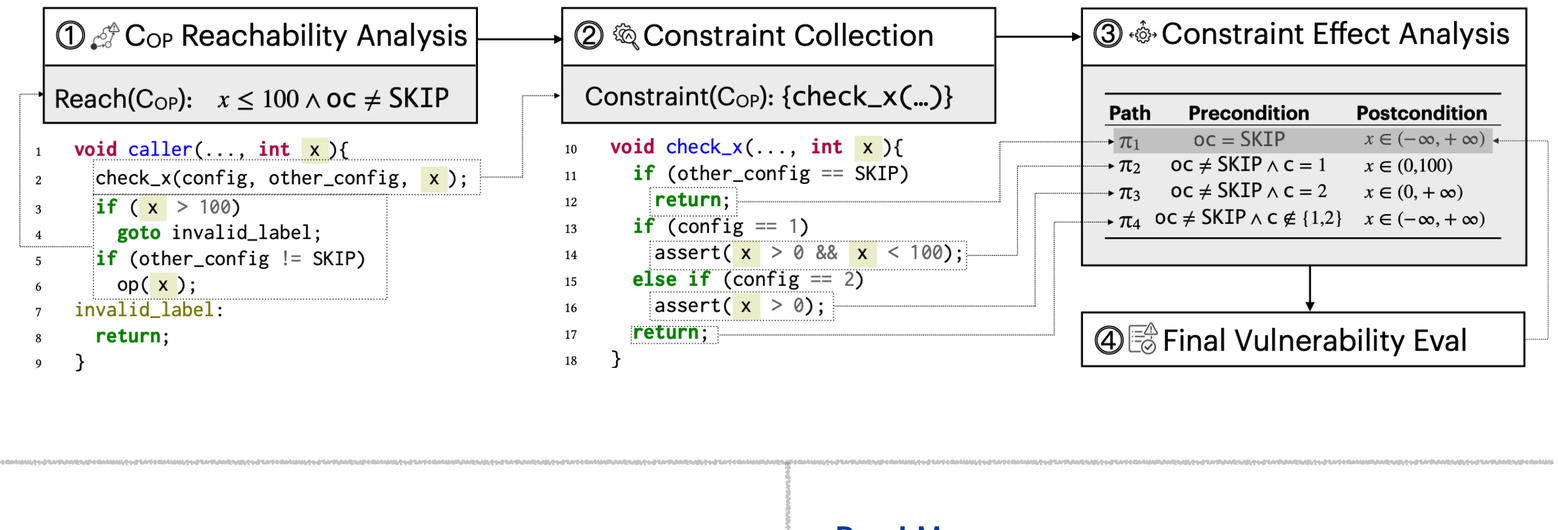


- Design rationale: LLMs are "lazy", they stop analysis if any "apparently" sanitation
- But not all "sanitation" always work
- Then "ConA": given the bug_condition, could it to be prevented by sanitization?

System Design: BugLens



ConA: Stepwise Constraint (Sanitization) Analysis



Experiments

Table II. Bug Analysis Performance Comparison Across LLMs and Design Approaches (Total Cases=120, Real Bugs=22)

Model	Full Design			w/o SAG			Simple Prompt		
	FN	FP	F_1	FN	FP	F_1	FN	FP	F_1
OpenAI o3-mini	0	3	**0.94**	10	1	0.67	18	7	0.24
OpenAI o1	3	6	0.81	8	6	0.67	18	6	0.25
OpenAI GPT-4.1	1	9	0.81	7	7	0.68	3	23	0.59
Gemini 2.5 Pro	12	3	0.57	14	4	0.47	6	24	0.52
Claude 3.7 Sonnet	13	2	0.54	17	2	0.34	1	51	0.44
DeepSeek R1	4	7	0.77	10	6	0.60	5	42	0.42

Table III. Performance of SecIA, with and without the Aribitary Control Hypothesis (AC-Hypo).

Model	w/o AC-Hypo				w/ AC-Hypo			
	FP	FN	Prec	Rec	FP	FN	Prec	Rec
OpenAI o3-mini	13	5	0.57	0.77	38	0	0.37	1.0
OpenAI o1	8	4	0.69	0.82	39	0	0.36	1.0
OpenAI GPT-4.1	15	2	0.57	0.91	36	1	0.69	0.95
Gemini 2.5 Pro	60	3	0.24	0.86	73	0	0.23	1.0
Claude 3.7 Sonnet	20	2	0.50	0.91	31	0	0.42	1.0
DeepSeek R1	25	12	0.29	0.45	71	4	0.18	0.82

Read More:

Email: hli333@ucr.edu (Haonan Li)

Full Paper

Opensource