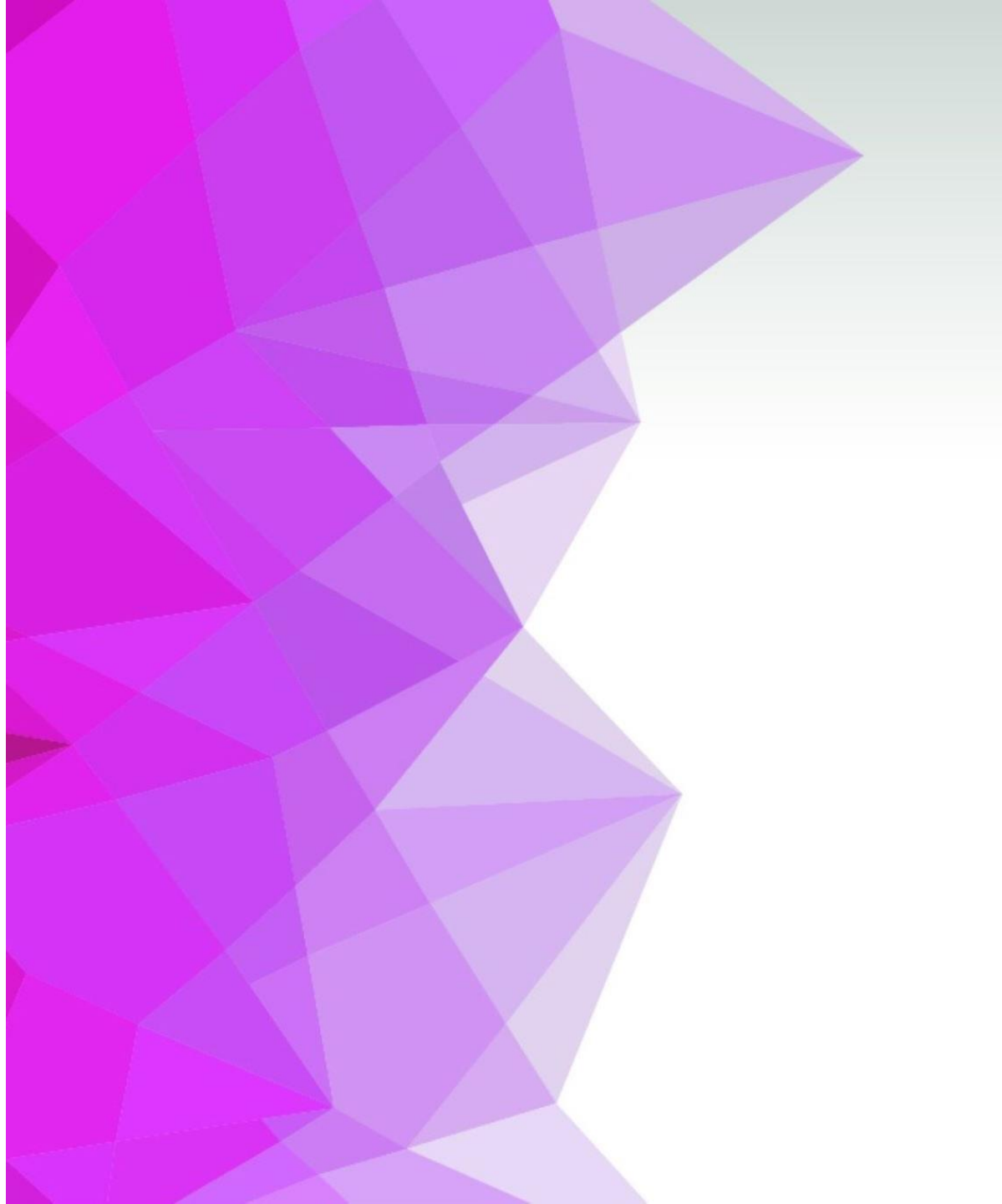
TOWARDS MORE ACCURATE STATIC ANALYSIS FOR TAINT-STYLE BUG DETECTION IN LINUX KERNEL

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

[1]University of California, Riverside

[2]Indiana university Bloomington

[3]The University of Chicago



THE HITCHHIKER'S GUIDE TO PROGRAM ANALYSIS (PART II)

- This work = Part II
  - *"The Hitchhiker's Guide to Program Analysis, Part II: Deep Thoughts by LLMs" (arXiv)*
- Part I: *"Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach"* (OOPSLA'24)
- *Research Goal:* **How can we make program analysis (for bug detection) more reliable and practical?**
- **Research evolution:**
  - From *simple bugs → complex*
  - From *new bugs → precision*
  - From *specific → general analysis frameworks*

BACKGROUND: TAINT-STYLE BUGS

- **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
- **But…?**
  - Real world is complex

```
12
13  int main(){
14      char buf[16];
15      int sz = get_user_input();
16
17      for(int i = 0; i < sz; i++){
18          use(value: buf[i]);
19      }
20  }
```

CHALLENGE 1: COMPLEX SINKS

- **Sink:** still array access, **with return check**
- **Static analysis view:**
  - "buf[i] may be tainted"
  - "Loop depends on tainted sz"
  - **Possible OOB access** ⚠
- **But safe:**
  - buf[15] = '\0' ensures early termination
  - Won't cause OOB

```
 9  int use(char value){
10      if (value == '\0'){
11          return -1;
12      }
13      return 0;
14  }
15
16  int main(){
17      char buf[16];
18      buf[15] = '\0';
19      int sz = get_user_input();
20
21      for(int i = 0; i < sz; i++){
22          int res = use(value: buf[i]);
23          if (res == -1){
24              break;
25          }
26      }
27  }
```

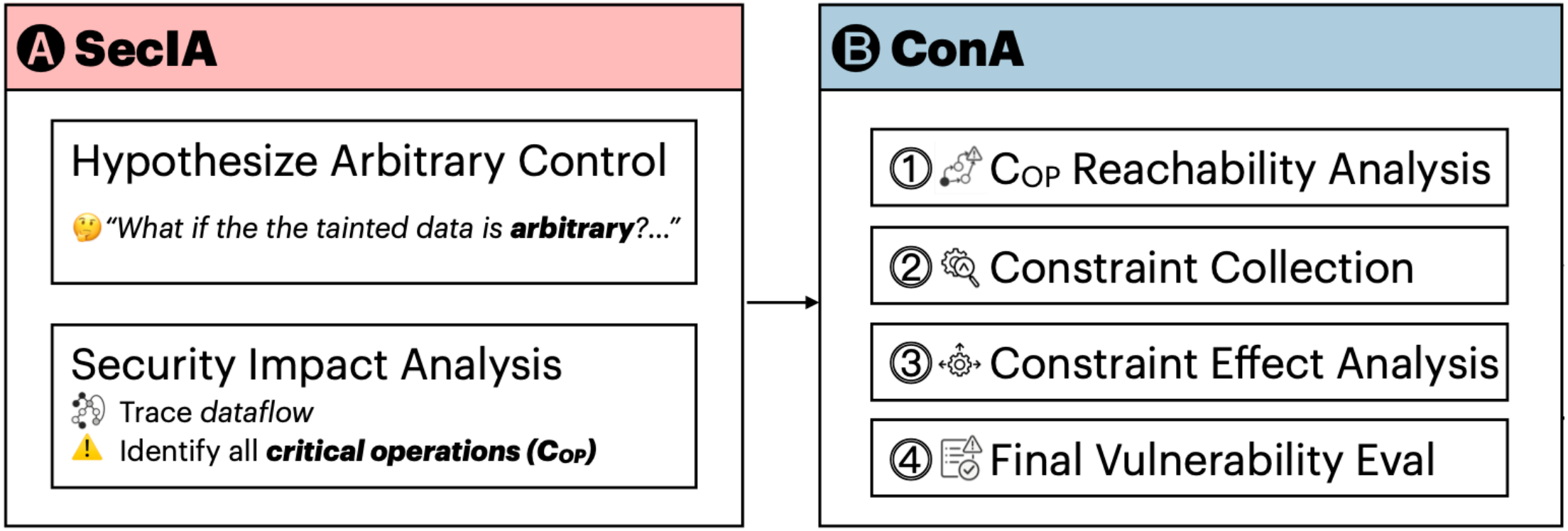
CHALLENGE 2: COMPLEX SANITIZATION

- Sanitizer: sz = sz % 16
  - Making sure sz ∈ [0,15]
- Needs to understand mod operator
- Besides,
  - Conditional sanitization
  - Derived-value sanitization (*sanitization* vs. *validation* discussed in paper)
  - …

```
16  int main(){
17      char buf[16];
18
19      int sz = get_user_input();
20      sz = sz % 16;
21
22      for(int i = 0; i < sz; i++){
23          use(value: buf[i]);
24      }
25  }
```

WHY PROGRAM ANALYSIS NEEDS LLM?

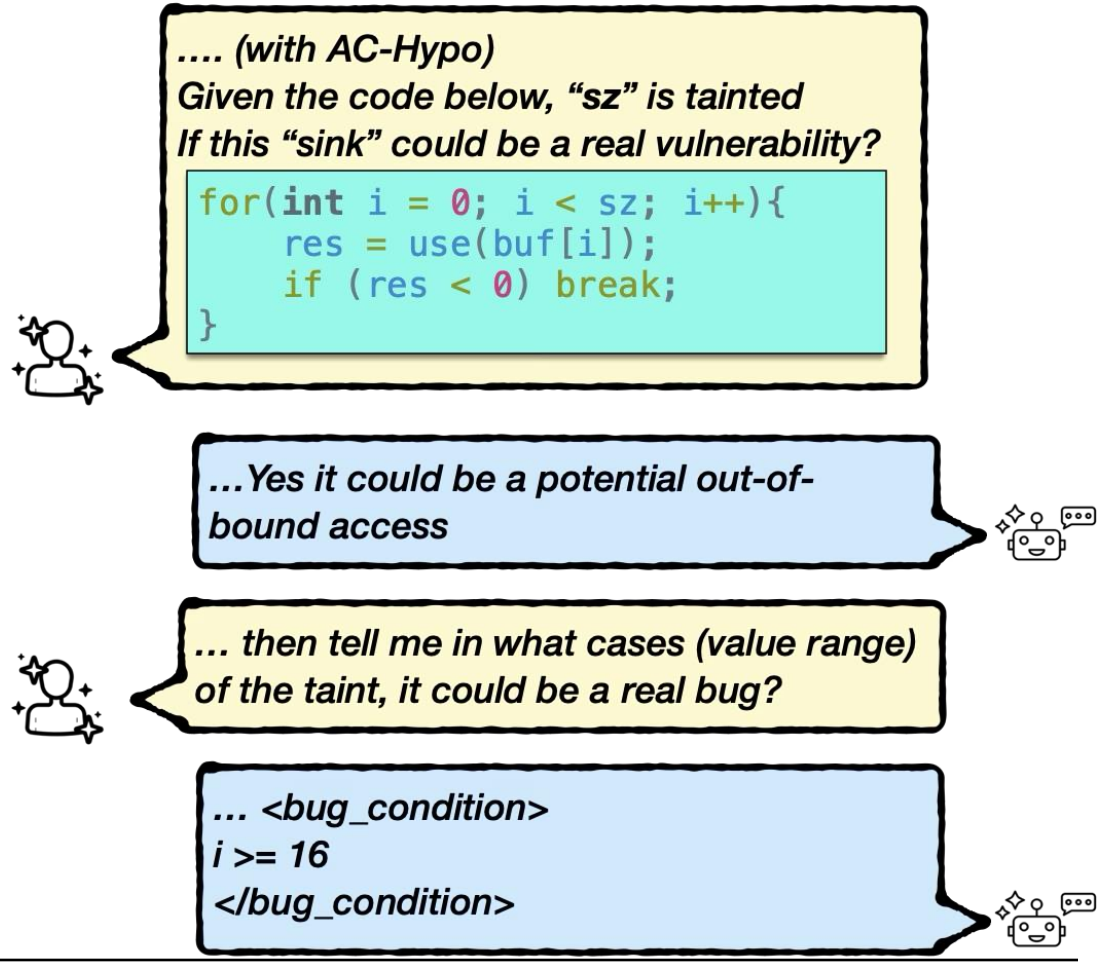
- Real-world programs are **semantically complex and diverse**
  - infinitely many patterns, corner cases, and project-specific idioms
- Accurate modeling of **sinks, sanitization, and control/data-flow**
  - is **hard to hand-craft** and often a **long-tail problem**
- LLMs can understand **code semantics and intent**
  - *"symbolic AI" vs. "neural AI",* we are doing *"neuron-symbolic" program analysis*
  - Help **reduce false positives** by reasoning about likely-safe paths



METHODOLOGY: BUGLENS

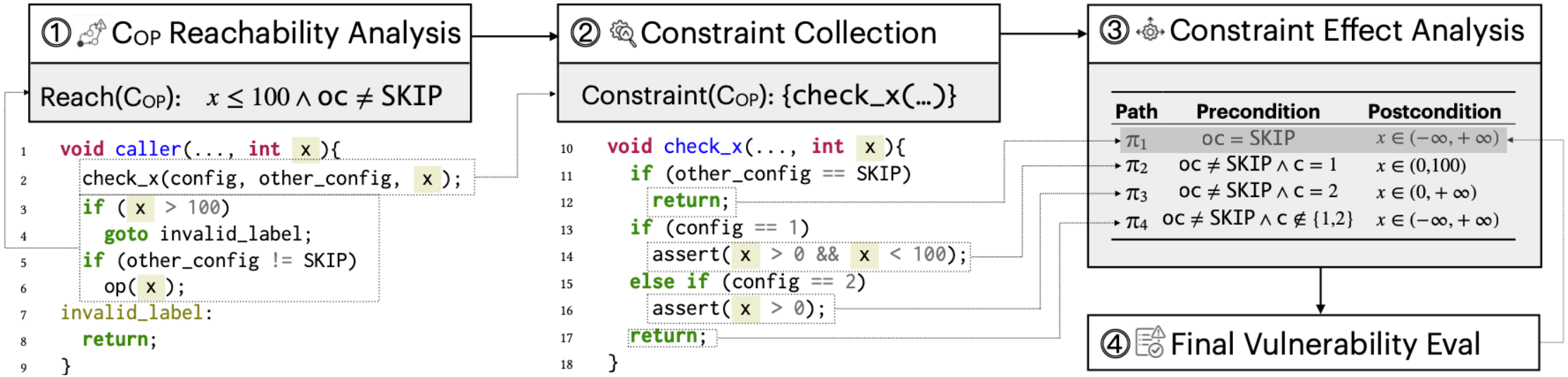
SECURITY IMPACT ACCESSOR (SECIA)

- Hypothesize on Arbitrary Control (AC-Hypo)
  - "*Ignore all sanitizations even they're **explicit**, if this operator cause any real security impact?*"
  - LLMs are "lazy", stop analysis if any "apparently" sanitizers



CONSTRAINT ASSESSOR (CONA)

- Core idea: any constraints (sanitizers) prevent the bug happen?



EXPERIMENTS

- RQ1: (Effectiveness) How effective is BUGLENS in identifying vulnerabilities?
- RQ2: (Component Contribution) How does the the prompt design affect the performance of BUGLENS?
- RQ3: (Model Versatility) How does the performance of BUGLENS vary across different LLMs

DATASET & EFFECTIVENESS

- $\text{Suture}_{\text{RP}}$: semi-automatic solution
- LLM outperforms human (in terms of FN)

Table I. Performance of BUGLENS on top of Suture and CodeQL-OOB.

Method	TP	TN	FP	FN	Prec	Rec	F_1
Suture	24	0	227	0	0.10	-	-
$\text{Suture}_{\text{RP}}$	20	202	25	4	0.44	0.83	0.58
$\text{Suture}_{\text{BUGLENS}}$	24	218	9	0	0.72	1.00	0.84
CodeQL-OOB	1	0	23	0	0.04	-	-
$\text{CodeQL-OOB}_{\text{BUGLENS}}$	1	22	2	0	0.33	1.00	0.5

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

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

RQ2 & RQ3

THANKS FOR YOUR LISTENING

- LLM makes static analysis better
- Open source: https://github.com/seclab-ucr/BugLens
- @Haonan Li: hli333@ucr.edu