Advances in AI-powered Code Security: Next-Level Bug Detection

Chengpeng Wang
Purdue University

https://chengpeng-wang.github.io
Email: wang6590@purdue.edu

Outline

- Static Bug Detection
- LLM-driven Data-flow Bug Detection
  - LLMSAN
  - LLMDFA
  - RepoAudit
- Configurable LLM-Agent for Static Analysis
  - LLMSA
  - Neuro-Symbolic CodeQL

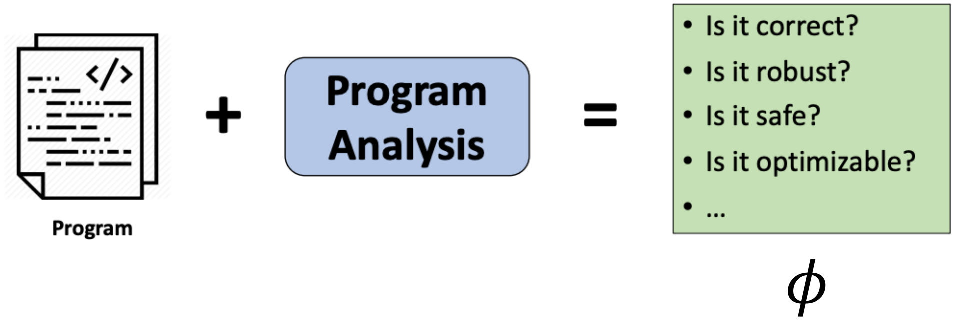
Programming in the AI Era

- *Copy-and-paste* the code from intelligent search engine
- *Prompt-and-paste* the code from LLM bots
- *Comment-and-select* the code recommended by LLM-powered IDE

A Nature Shift of Programming

- Shift from writing code to validating correctness
  - Writing code is cheap
  - Validating correctness is expensive
  - Critical for software reliability

- Static analysis: Reason the program statically without execution
  - Determine whether a specific property ϕ holds for any inputs



Example: Divide-by-Zero (DBZ) Bug Detection

- Target property ϕ: All the divisors are not equal to 0
- Rule-based symbolic analysis discovers a data-flow path from a source to a sink
  - Sources: Faulty values, Sinks: Operands of dangerous operations

```
1  public static void bar(int x, int y){
2      if (x != 0)
3          return (y * 1.0 / x);
4      else
5          return (x * 1.0 / y);
6  }
7  public static void main(){
8      int a = 0;
9      int b = parseInt("123");
10     System.out.println(bar(b, a));
11     String arg = args[0];
12     int c = parseInt(arg);
13     System.out.println(bar(a, c));
14     c = b;
15     System.out.println(bar(a, c));
16 }
```

sink

source

Does ϕ always hold? → No

Counterexample: args[0] = "0" and $y@\ell_5 = 0$

Mainstream Static Bug Detectors

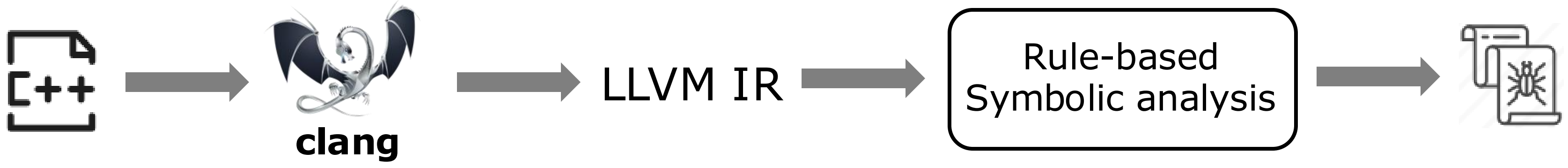
- Existing effort: Make it more precise, more efficient, and more scalable

- Limitations
  - Compilation reliance
  - Customization obstacle
  - Specification burden

SonarQube
Coverity
Cppcheck
PVS-Studio
Facebook Infer
Polyspace
Cppdepend
Helix QAC
CodeQL
Sourcebrella PINPOINT

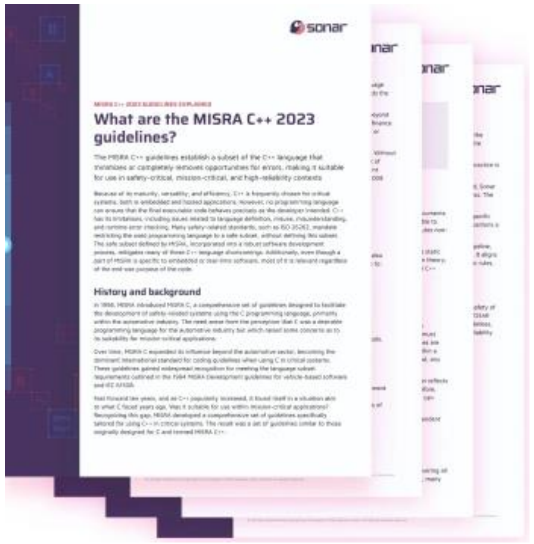
Limitation I: Compilation Reliance

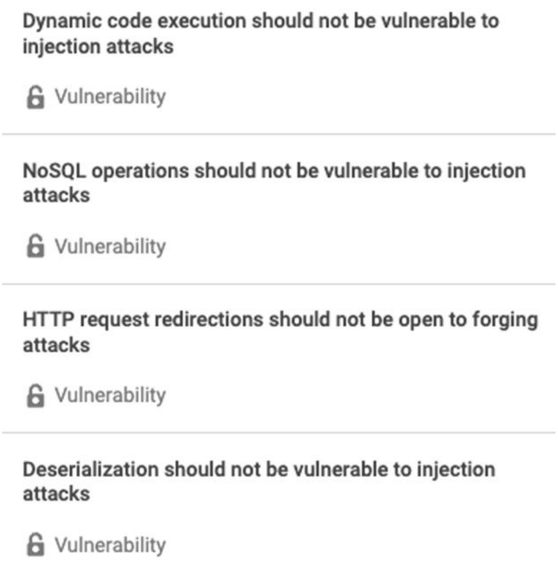
- Existing static bug detectors/analysis platforms require intermediate representations (IRs) of programs generated by successful compilation
- Fail to discover security vulnerabilities in the incomplete code
  - Incomplete code: Program under the development, code snippets generated by AI



Limitation II: Customization Obstacle

- Existing static bug detectors/analysis platforms only target specific bug types and can not support the user-friendly customization
- Require the expert knowledge on hacking the compiler infrastructure





Limitation III: Specification Burden

- Existing static bug detectors/analysis platforms require manually specified specifications, such as library semantic specifications
  - Example: Library APIs that may return zero values for the Divide-by-Zero detection
- Require laborious manual effort

```
public static void main(){
 int a = 0;
 int b = parseInt("123");
 System.out.println(bar(b, a));
 String arg = args[0];
 int c = parseInt(arg);
 System.out.println(bar(a, c));
 c = b;
 System.out.println(bar(a, c));
}
```

Our Goal

- Build a *compilation-free, customizable, and intelligent* static analysis
  - No reliance on build/compilation
  - No reliance on expertise
  - No reliance on labor work

Outline

- Static Bug Detection
- LLM-driven Data-flow Bug Detection
  - LLMSAN
  - LLMDFA
  - RepoAudit
- Configurable LLM-Agent for Static Analysis
  - LLMSA
  - Neuro-Symbolic CodeQL

Chengpeng Wang, et al. Sanitizing Large Language Models in Bug Detection with Data-Flow. EMNLP Findings 2024

Chengpeng Wang, et al. LLMDFA: Analyzing dataflow in code with large language model. NeurIPS 2024

Jinyao Guo, et al. REPOAUDIT: An Autonomous LLM-Agent for Repository-Level Code Auditing. [ongoing]

Reshaping Static Analysis with LLMs

Please detect divide-by-zero bug step by step. Hint: If a zero value flows to the second operand of / or %, a divide-by-zero bug can arise. Please make your answer brief and informative.

J public class Demo {

4. **Call to `foo` Method:**

```
x = foo(y, z); // foo(1, 0) -> returns 0
```

5. **Second Potential Divide-by-Zero:**

```
System.out.println(x / x); // 0 / 0 -> Divide-by-zero error
```

Summary of Divide-by-Zero Bugs:

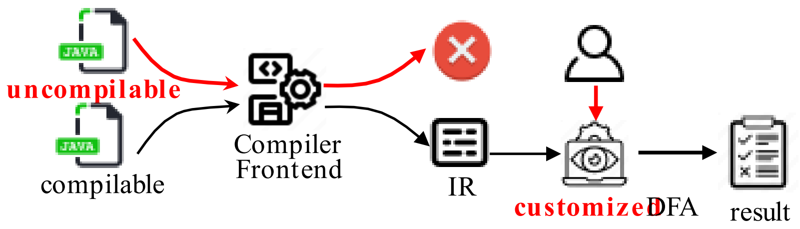
1. `System.out.println(y / z);` results in `1 / 0` -> Divide-by-zero error.
2. `System.out.println(x / x);` results in `0 / 0` -> Divide-by-zero error.

J public class Demo { Untitled-1

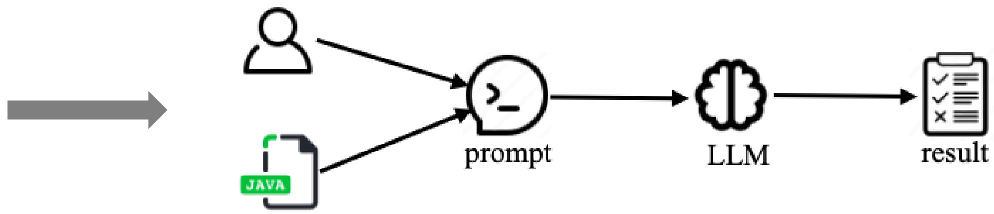
```java
public class Demo {

    public static int foo(int a, int b) {
        if (Math.abs(b) > 1)
            System.out.println(a / b);
        return b;
    }

    public static void main(String[] args) {
        int x = 1;
        int y = 10 / x;
        x = 0;
        y = x + 1;
        int z = x / y;
        z = x;
        System.out.println(y / z);

        x = foo(y, z);
        System.out.println(x / x);
    }
}
```

Our Vision

- LLMs for static analysis
  - Analyze code as specific kind of textual data
    - No need to build the code
  - Easy to customize the analysis
    - Analyze code via prompting
  - Serve as knowledge base of specifications
    - Pretrained upon huge data



Existing Static Analyzers



Our Proposal

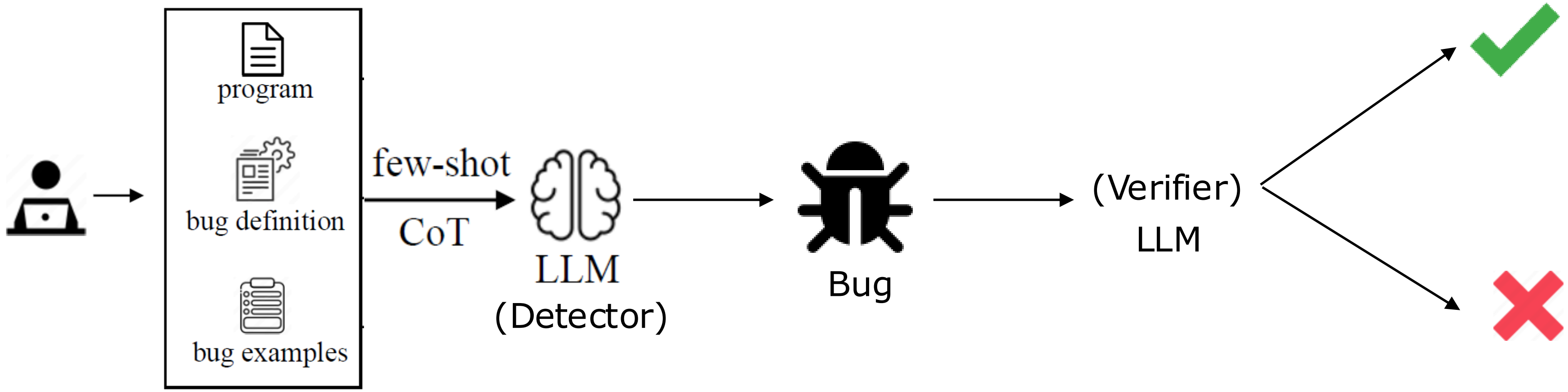
LLMs are NOT Silver Bullets for Static Analysis

- Hallucinations introduce FPs/FNs
- Empirical study [arXiv 2024]
    - Select 100 buggy functions
    - Before and after fixes
    - Bug type + location + root cause: precision ~30%
- Even worse in repo-level detection

Steenhoek B, et al. A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection, arXiv 2024.

Few-shot CoT Prompting-based Bug Detection

- Check the detection results
  - Simple solution: Using LLMs as verifiers while still hallucinate



Key Idea: Data-flow Paths as Verifiable Bug Proofs

- Generate bug proofs: Data-flow paths from sources to sinks
- Sanitize data-flow paths via divide-and-conquer
  - Data sanitization: Whether start/end values conform to the forms of sources/sinks
  - Flow sanitization: Whether the faulty value can be propagated along the path in each step

```
1 public static void bar(int x, int y){
2   if (x != 0)
3     return (y * 1.0 / x);
4   else
5     return (x * 1.0 / y); //bug
6 }
7 public static void main(){
8   int a = 0; //zero
9   int b = parseInt("123");
10  System.out.println(bar(b, a));
11    String arg = args[0];
12    int c = parseInt(arg); //potential zero
13  System.out.println(bar(a, c));
14  c = b;
15  System.out.println(bar(a, c));
16 }
```

Program: [Example Program with a DBZ bug]
Explanation: v is assigned with 0 at line 3. Then u is initialized with v at line 7 and used as a divisor at line 9, causing a DBZ bug.
Dataflow path: [(v, ℓ_3), (u, ℓ_7), (u, ℓ_9)]

Few-shot Example

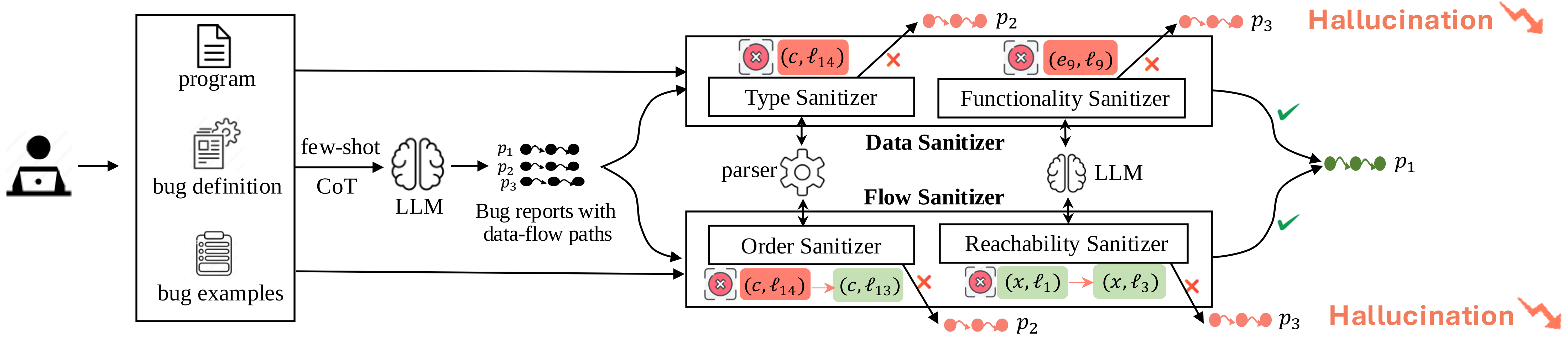
p_1: $(e_{12}, \ell_{12}) \rightarrow (c, \ell_{12}) \rightarrow (c, \ell_{13}) \rightarrow (y, \ell_1) \rightarrow (y, \ell_5)$ ✔ valid

p_2: $(c, \ell_{14}) \rightarrow (c, \ell_{13}) \rightarrow (y, \ell_1) \rightarrow (y, \ell_5)$ ✘ spurious

p_3: $(e_9, \ell_9) \rightarrow (b, \ell_9) \rightarrow (b, \ell_{10}) \rightarrow (x, \ell_1) \rightarrow (x, \ell_3)$ ✘ spurious

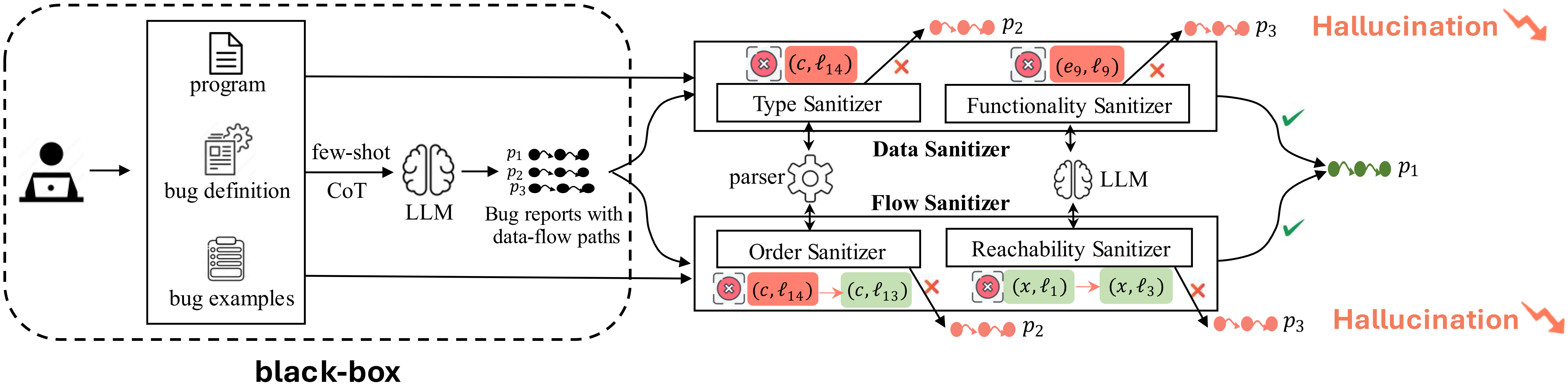
LLMSAN: LLM-driven Bug Detection with Sanitization

- Sanitize data-flow paths emitted by few-shot CoT prompting
- Four sanitizers powered by parsers and LLMs
  - Decompose the validation of syntactic and semantic properties
  - Syntactic properties can be perfectly validated by parsing-based sanitizers



LLMSAN: Discussion

- End-to-end prompting in the detection phase
  - Pro: Compilation-free and easy to customize
  - Con: Low recall, e.g., GPT-3.5 powered LLMSAN misses all the DBZ bugs
- How to improve: Avoid black-box detection



Key Idea: Summary-based Static Analysis

- Example: DBZ detection
- Problem decomposition
  - Source/sink extraction
  - Dataflow summarization
  - Path feasibility validation

```java
public class Demo {
 public static int foo(int a, int b){
   if (Math.abs(b) > 1)
     System.out.println(a / b);
   return b;
 }
 public static void main(String[] args){
   int x = 1;
   x = 0;
   int y = x + 1;
   int z = x / y;
   z = x;
   System.out.println(y / z);
   x = foo(y, z);
   System.out.println(x / x);
 }
}
```

sink

source

sink

sink

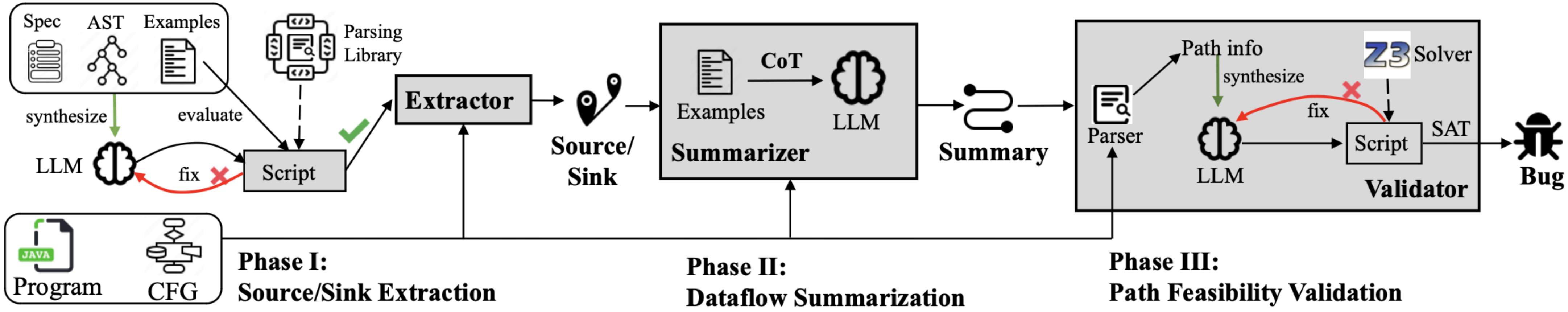
sink

1 source -> argument

2 parameter -> return value

3 output value -> sink

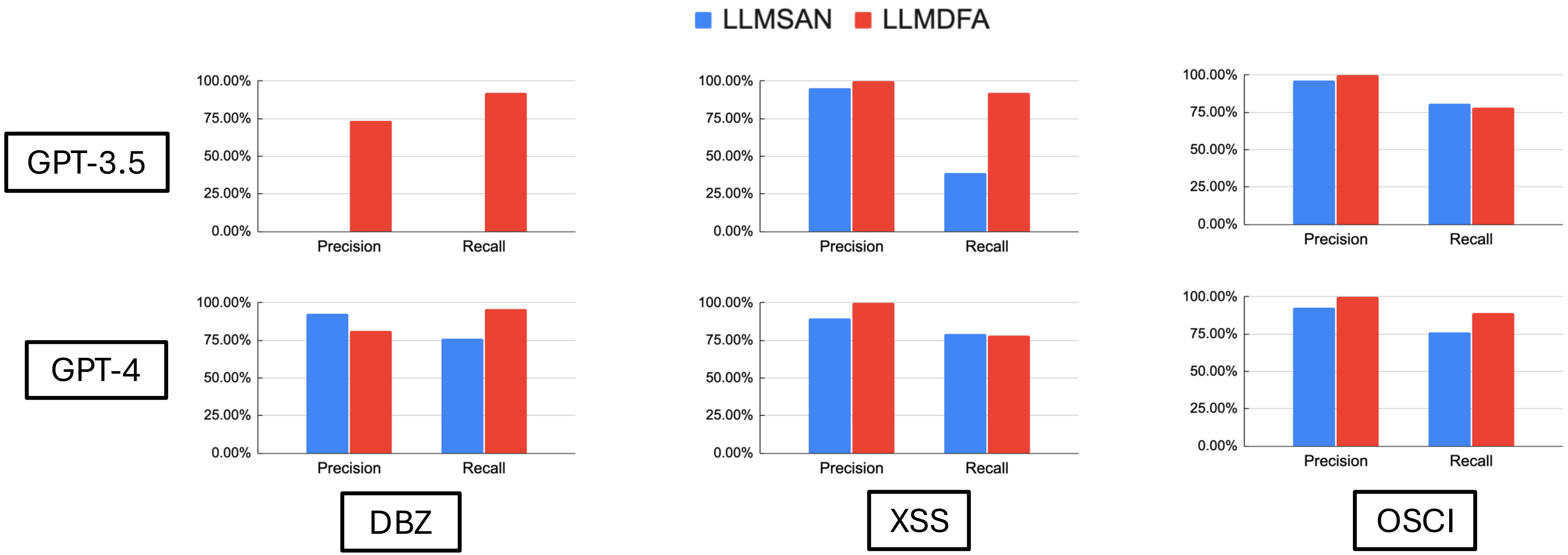
LLMDFA: Analyzing Data-flow with LLMs



- Problem Decomposition
- Tool Synthesis

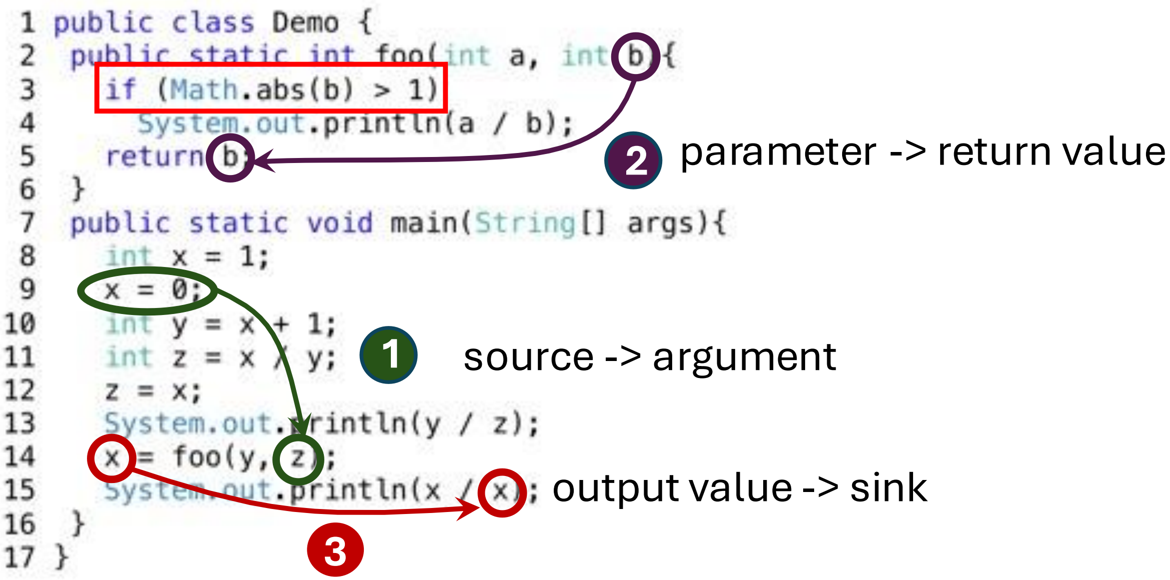
Comparison with LLMSAN

- LLMSAN vs LLMDFA powered by GPT-3.5 and GPT-4



LLMDFA: Discussion

- High cost of LLMDFA due to
  - The large numbers of sources/sinks
  - The large numbers of functions
  - The large numbers of function calls



RepoAudit: Repository-level Bug Detection

- Enhanced version of LLMDFA: Autonomous LLM-agent
  - Memory
    - Summarize data-flow facts along different paths via prompting LLMs with single functions
  - Tool using

Tool Domain	Tool Name	Usage Pattern
Retrieval	Value Retriever	`SrcRetrieve(Prog)`
		`SinkRetrieve(Prog)`
	Statement Retriever	`StmtRetriever(Func)`
	Function Retriever	`FuncRetriever`$(\texttt{Func}, [s_{b_j}]_{j=0}^{m})$
Interpretation	Function Interpreter	`FuncInterpret`$(\texttt{Func}, v@s_i, S)$
Validation	Order Validator	`OrdValidate`$(v@s_i, [s_{b_j}]_{j=0}^{m})$

  - Planning
    - Start from functions containing sources
    - Search for sinks by exploring callers and callees on demand

Outline

- Static Bug Detection
- LLM-driven Data-flow Bug Detection
  - LLMSAN
  - LLMDFA
  - RepoAudit
- Configurable LLM-Agent for Static Analysis
  - LLMSA
  - Neuro-Symbolic CodeQL

Chengpeng Wang, et al. LLMSA: A Compositional Neuro-Symbolic Approach to Compilation-free and Customizable Static Analysis. arXiv 2024.

Towards More Customizable Analysis

- LLMSAN & LLMDFA & RepoAudit
  - Agent-centric solutions: More precise, more complete, and more scalable
  - No planning: Fixed action space
  - Determine *whether two program values are data-flow reachable or not*

- How to address more diverse analysis demands
  - Program slicing
  - Implicit flow analysis
  - Control-flow integrity analysis

- Solution: Build a configurable agent

Key Idea I: Bridging Syntactic & Semantic Properties

- Analyzing non-trivial semantic properties is undecidable while syntactic analysis is decidable
  - Semantic: Data dependency, points-to relation
  - Syntactic: control flow order, control dependency

- Static analysis agent = *Syntactic* analysis + *Semantic* analysis
  - Parsing-based analyzers for syntactic analysis
  - LLMs for semantic analysis

Key Idea II: Datalog as Analysis Policy Language

- Represent syntactic/semantic properties as symbolic/neural relations
- Derive new properties based on Datalog rules
- Example: intra-procedural program slicing (backward slicing)
  - Neural relation: DataDep

$$\mathrm{Slice}(e1, l) \leftarrow \mathrm{SliceExpr}(e1, e2), \mathrm{ExprLoc}(e2, l) \quad (1)$$

$$\mathrm{SliceExpr}(e1, e2) \leftarrow \mathrm{ExprName}(e1, \text{“userCity”}), \mathrm{ExprLoc}(e1, 26), \mathrm{DataDep}(e2, e1) \quad (2)$$

$$\mathrm{SliceExpr}(e1, e2) \leftarrow \mathrm{ExprName}(e1, \text{“userCity”}), \mathrm{ExprLoc}(e1, 26), \mathrm{CtrlDep}(e2, e1) \quad (3)$$

$$\mathrm{SliceExpr}(e1, e3) \leftarrow \mathrm{SliceExpr}(e1, e2), \mathrm{DataDep}(e3, e2) \quad (4)$$

$$\mathrm{SliceExpr}(e1, e3) \leftarrow \mathrm{SliceExpr}(e1, e2), \mathrm{CtrlDep}(e3, e2) \quad (5)$$

Analysis Policy

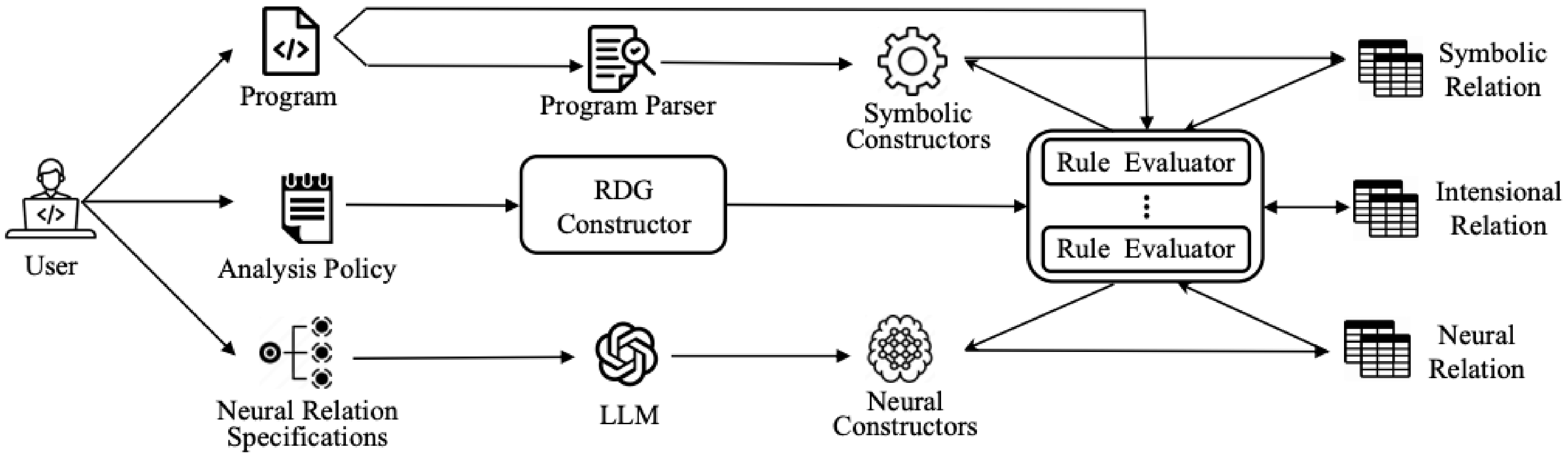
Definition: List all the expression pairs (e1, e2) if the value of the expression e1 is affected by the value of the expression e2 during program execution.

Examples: In the following program {code}, we can obtain all the expression pairs with data dependency: {a list of expression pairs}. Here is {explanation}.

Neural Relation Spec

LLMSA: Configurable LLM-Agent for Static Analysis

- **Program**: The targeted program
- **Analysis policy**: Neuro-symbolic Datalog program for defining the LLM-agent
- **Neural relation spec**: Define the prompts for neural relation generation



Lazy, incremental, and parallel prompting

LLMDFA as an Instance of LLMSA

- Absolute Path Traversal (APT) Detection
  - Provide Source/Sink examples to define APTSrcNeural/APTSinkNeural

$$
\begin{aligned}
\mathrm{APTBug}(e1, e2) &\leftarrow \mathrm{APTSrc}(e1), \mathrm{FunctionSummary}(e1, e2), \mathrm{APTSink}(e2) \\
\mathrm{FunctionSummary}(e1, e2) &\leftarrow \mathrm{SummaryStartExpr}(e1), \mathrm{TaintProp}(e1, e2), \mathrm{SummaryEndExpr}(e2) \\
\mathrm{FunctionSummary}(e1, e2) &\leftarrow \mathrm{FunctionSummary}(e1, e3), \mathrm{OutRet}(e4, e3), \mathrm{FunctionSummary}(e4, e2) \\
\mathrm{FunctionSummary}(e1, e2) &\leftarrow \mathrm{FunctionSummary}(e1, e3), \mathrm{ArgPara}(e3, e4), \mathrm{FunctionSummary}(e4, e2) \\
\mathrm{SummaryStartExpr}(e1) &\leftarrow \mathrm{APTSrc}(e1) \\
\mathrm{SummaryStartExpr}(e1) &\leftarrow \mathrm{Paras}(e1) \\
\mathrm{SummaryStartExpr}(e1) &\leftarrow \mathrm{Outs}(e1) \\
\mathrm{SummaryEndExpr}(e1) &\leftarrow \mathrm{APTSink}(e1) \\
\mathrm{SummaryEndExpr}(e1) &\leftarrow \mathrm{Rets}(e1) \\
\mathrm{SummaryEndExpr}(e1) &\leftarrow \mathrm{Args}(e1) \\
\mathrm{APTSrc}(e1) &\leftarrow \mathrm{APTSrcNeural}(e1), \mathrm{Outs}(e1) \\
\mathrm{APTSink}(e1) &\leftarrow \mathrm{APTSinkNeural}(e1), \mathrm{Args}(e1)
\end{aligned}
$$

Outline

- Static Bug Detection
- LLM-driven Data-flow Bug Detection
    - LLMSAN
    - LLMDFA
    - RepoAudit
- Configurable LLM-Agent for Static Analysis
    - LLMSA
    - Neural-Symbolic CodeQL

From LLMSA to Neuro-Symbolic CodeQL

- Build the next-generation static analysis platform
  - Neural analysis: LLMs
  - Symbolic analysis: CodeQL also supports semantic analysis
- Natural advantages: Compilation-free and multi-lingual supports
- More attractive features
  - ???
  - ???

Problem: Specification Burden

- CodeQL: The symbolic analyzer that reasons code with specifications via Datalog rules
  - All the program facts are derived from code and specified specifications

- Tricky issues:
  - What if bugs cannot be formulated, e.g., performance bug?
  - What if library APIs can not be comprehensively enumerated?
  - What if the rule-based analysis reports false positives due to low-quality of specifications?

Future Work I: Multi-modal Static Analysis

- Proposal I: LLMs retrieve specs for CodeQL as neural relations

  R1(e1, e2) <-- R2(e1, e2), R3(e1, e2), ... Rn(e1, e2), NeuralRelation(e1)

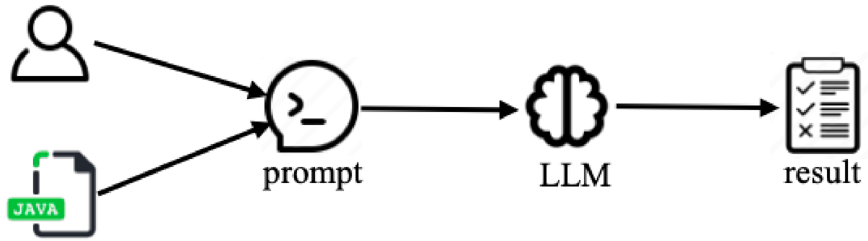
  - Library spec
  - Bug spec

- Proposal II: LLMs examine the data provenance witness of CodeQL
  - Example: Check the data-flow paths in data-flow analysis with LLMs
  - Similar to sanitization phase in LLMSAN

- Parametric design: Multi-modal knowledge base + RAG

Problem: Customization Obstacle

- Ideal mode of static bug detection: Few-shot CoT prompting



- LLMDFA supports prompting-based customization
  - Limitation: Only target data-flow bugs

- LLMSA customizes the agent for static analysis
  - Limitation: Difficult to specify the analysis policy

Future Work II: Autonomous Static Analysis

- <u>Proposal: Synthesize neuro-symbolic static analyzers from multi-modal data</u>
  - Example: Patches = Buggy/Non-buggy code + Bug description
  - Previous trial: Synthesize queries based on pos/neg examples for code search [ECOOP 2023]
  - Synthesize neuro-symbolic CodeQL queries to detect diverse types of bugs
    - Step 1: Search caller functions of *dev_kfree_skb* via CodeQL relations
    - Step 2: Utilize LLMs to check the code in each caller function

```
struct net_device_ops rionet_netdev_ops = {
  /* Interface: function pointer*/
  .ndo_start_xmit = rionet_start_xmit,
};
int rionet_start_xmit(struct sk_buff *skb) {
 - dev_kfree_skb(skb);
  ndev->stats.tx_packets++;
  ndev->stats.tx_bytes += skb->len;  /* Use-after-free on skb */
 + dev_kfree_skb(skb);     /* Uniform API invocation */
}
```

```
BugCase(e) <- R11(e)
R11 <-- R12, R13, … R1n
R21 <-- R22, R23, … R2m
```

```
BugCase(e) <- R11(e)
R11 <-- R12,  … R1n, NeuralRelation
R21 <-- R22,  … R2m, NeuralRelation
```

Chengpeng Wang, et al. Synthesizing Conjunctive Queries for Code Search. ECOOP 2023.

Hallucination Mitigation in Neuro-Symbolic Analysis

- Tool using: Use the rules in CodeQL as many as possible
- Tool synthesis: Synthesize tools to populate the neural relations
- LLM sanitization: Post-verify the outputs of LLMs before utilizing them in the rule evaluation

Three Paradigms of Static Analysis

Symbolic Static Analysis

- Rule-based expert system
  - Explainable
  - Deterministic
  - Limited applicability
  - Restricted usability
  - Single modality (code only)

KLEE

making your software safer with Facebook Infer

Neural Static Analysis

- Data-driven black box
  - Extensive applicability
  - Flexible usability
  - Multi-modality (code, doc, etc)
  - Unexplainable
  - Nondeterministic
  - Hallucinatory

Claude

OpenAI o1

Neuro-Symbolic Static Analysis

- LLM agent
  - (Relatively) Explainable
  - (Relatively) Deterministic
  - Extensive applicability
  - Flexible usability
  - Multi-modality

LLMSAN LLMDFA

RepoAudit LLMSA

Conclusion

- Static bug detection is critical for software reliability in the AI era.
  - Conventional symbolic static analysis cannot well support AI-generated code.
- LLMs can reshape static analysis but are not silver bullets due to inherent hallucinations.
- Developing neuro-symbolic static bug detection techniques holds great potential.
  - Compilation-free and Customizable
  - Multi-modal and Autonomous

Q&A

- Chengpeng Wang
- Homepage: https://chengpeng-wang.github.io/
- Email: wang6590@purdue.edu