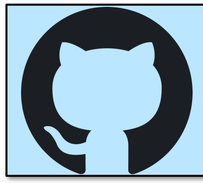
Challenging Bug Prediction and Repair Models with Synthetic Bugs

Ali Reza Ibrahimzada[1], Yang Chen[1], Ryan Rong[2], Reyhaneh Jabbarvand[1]

[1]University of Illinois Urbana-Champaign

[2]Stanford University

In Proceedings of The 25th IEEE International Conference on Source Code Analysis & Manipulation (SCAM 2025)
September 7-12, 2025, Auckland, New Zealand



github.com/Intelligent-CAT-Lab/BugFarm

Software Bugs

A **software bug** is an error that makes a program deviate from its intended requirements

Buggy Code

```
def average(nums):
    total = sum(nums)
    return total / len(nums)
```

```
assert average([1, 2, 3]) == 2.0  # pass ✅
assert average([]) == 0.0         # fail ❌
```

Correct Code

```
def average(nums):
    total = sum(nums)
    return total / len(nums) if nums else 0.0
```

```
assert average([1, 2, 3]) == 2.0  # pass ✅
assert average([]) == 0.0         # pass ✅
```

Software Bugs

Debugging in Software Engineering is the process of detecting, localizing and repairing Software Bugs

Buggy Code

```python
def average(nums):
    total = sum(nums)
    return total / len(nums)
```

```python
assert average([1, 2, 3]) == 2.0  # pass ✅
assert average([]) == 0.0         # fail ❌
```

Correct Code

```python
def average(nums):
    total = sum(nums)
    return total / len(nums) if nums else 0.0
```

```python
assert average([1, 2, 3]) == 2.0  # pass ✅
assert average([]) == 0.0         # pass ✅
```

Debugging Techniques

- ❖ Manual Debugging
  - ➢ Peer-programming
  - ➢ Logging traces
- ❖ Automated Debugging
  - ➢ Testing
  - ➢ Program Analysis
- ❖ Learning-based Debugging
  - ➢ Language Models

```python
def average(nums):
    print("DEBUG >> nums: ", nums)
    total = sum(nums)
    return total / len(nums)
```

```python
assert average([1, 2, 3]) == 2.0  # pass ✅
assert average([]) == 0.0         # fail ❌
```

Are these techniques evaluated effectively? Can we challenge these techniques?

Limitations of Existing Bug Generation Techniques

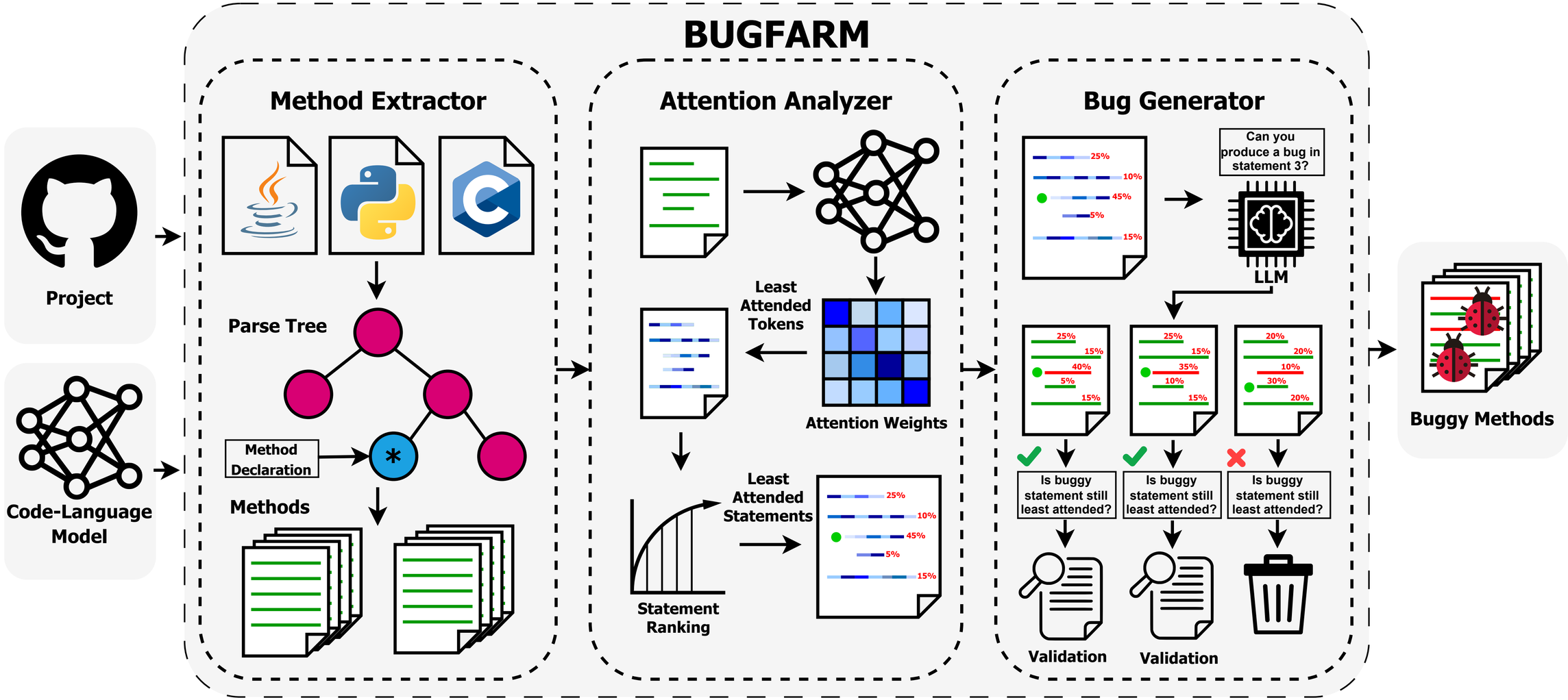
- Saturated
  - Defects4J: 11 years old
  - Data contamination
- Simple
  - Rule-based mutation operators
  - Easy for bug detection techniques
- Single-line
  - Bugs are only in one location
  - Easy for bug repair techniques

Can we generate *hard-to-detect* and *hard-to-repair* bugs?

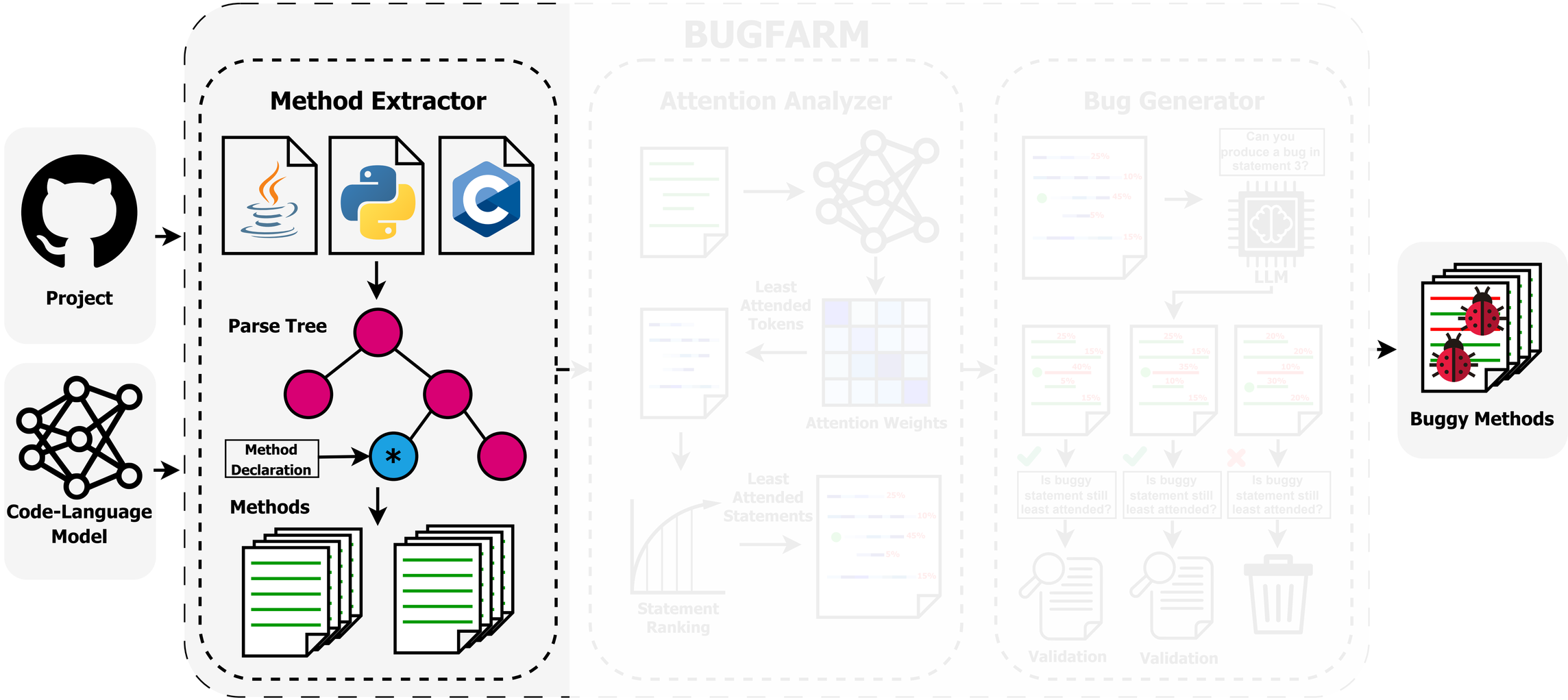
BugFarm

- A framework for end-to-end synthetic bug generation
  - Analyze the attention mechanism of the bug detector and repair model
  - Identify least attended locations to inject bugs
  - Generate multiple bugs for those locations
- BugFarm generates hard-to-detect and hard-to-repair bugs
  - Different unseen bug patterns from training data
  - Similar representation to models as correct code
- Compared against state-of-the-art bug generator techniques
  - LEAM
  - µBERT

BugFarm Framework



BugFarm Framework

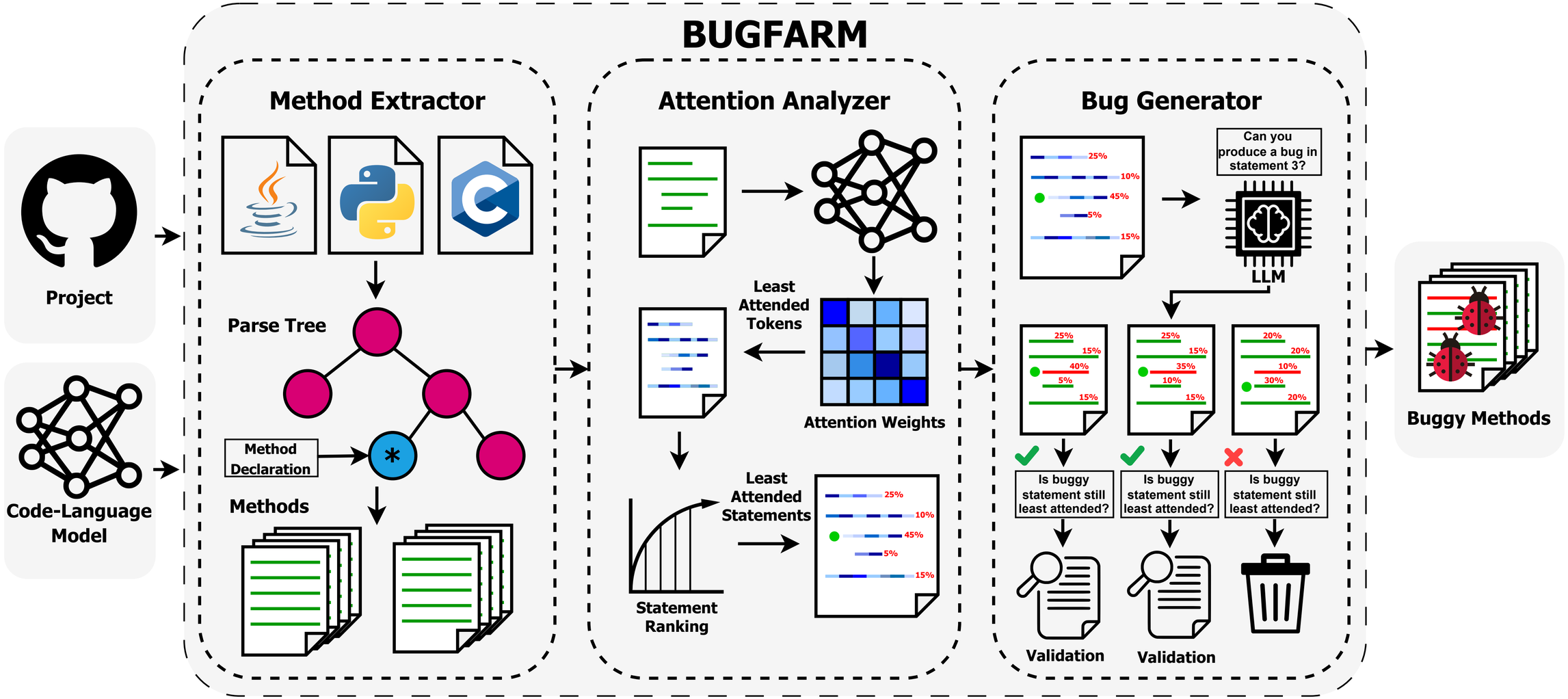


Method Extractor

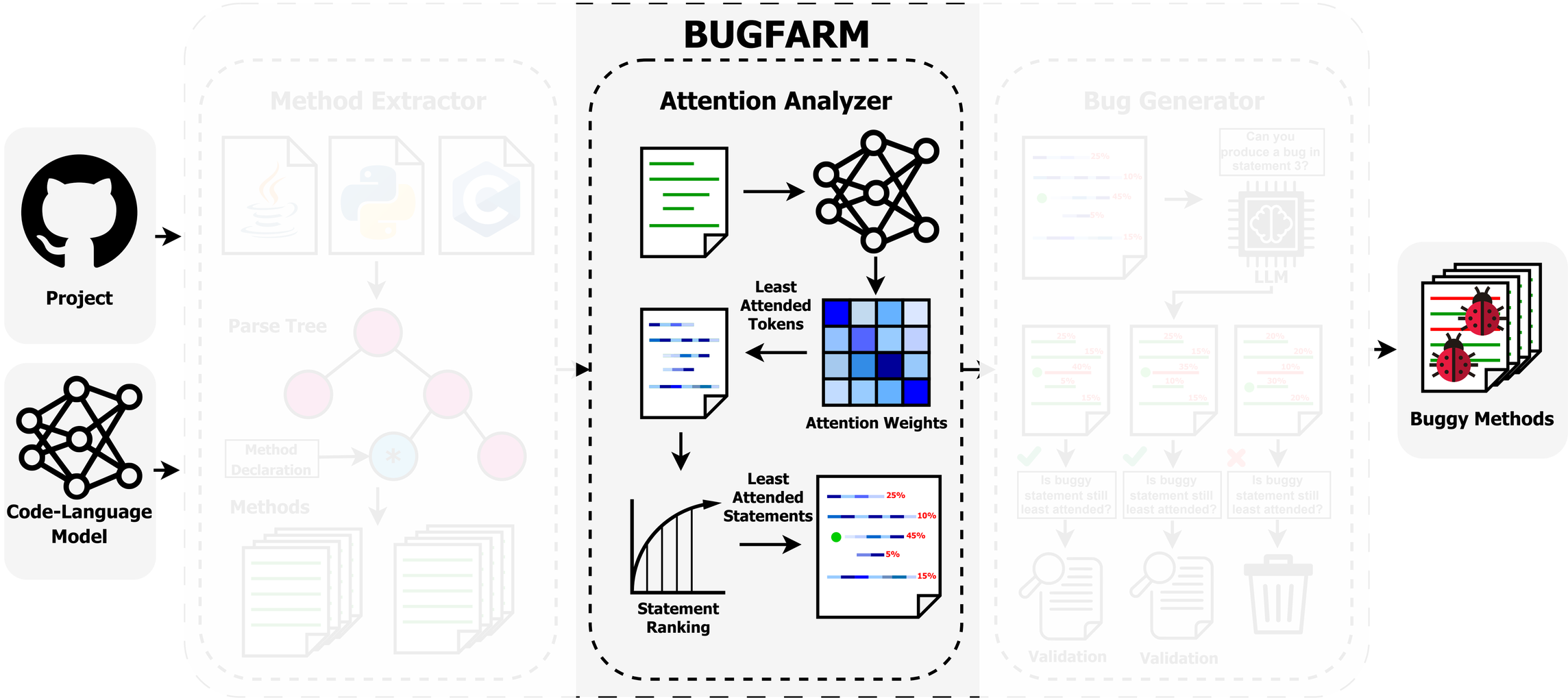
- Use language specific parser to extract methods and constructors
- Javalang[1] parser for Java programming language
- Create parse tree of each Java file
- Locate method declaration
- Extract methods and their signatures

[1]https://github.com/jose/javalang.git

BugFarm Framework



BugFarm Framework



Attention Analyzer

- Given a method and a model
- Extract token attention weights
- Extract least-attended tokens (LATs)
- Rank statements based on LATs
- Extract least-attended statements (LASs)
- Threshold “K” is used to get top-K LATs and LASs
- K = 10% in all experiments

Algorithm 1: Attention Analyzer

Inputs: Method $method$, Threshold k, Transformer-based model M
Output: Least attended statements LAS

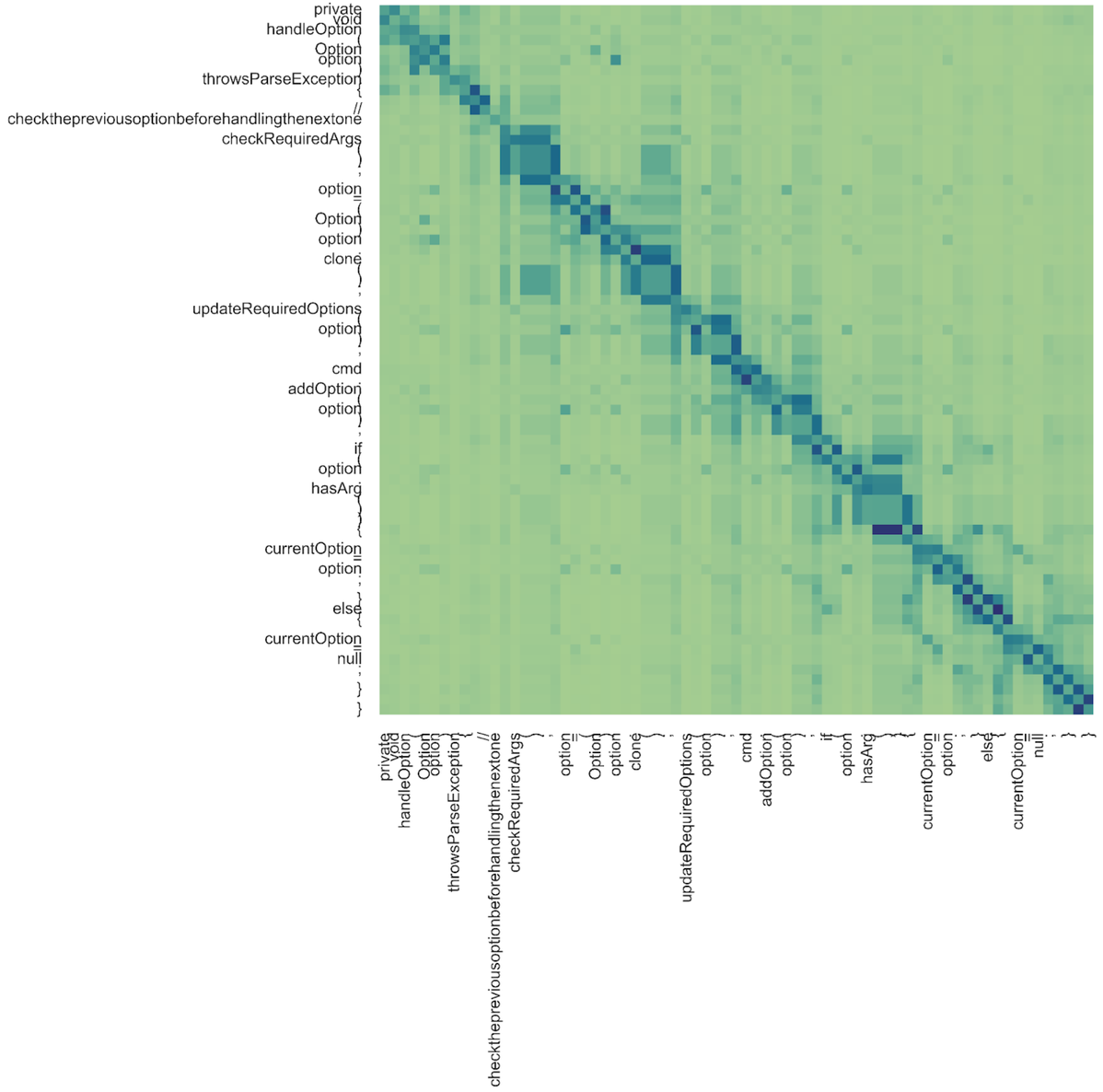
$\overrightarrow{Tkn} \leftarrow getTokens(method)$;
$\overrightarrow{Smt} \leftarrow getStatements(method)$;
$LAT, LAS \leftarrow \emptyset$;
$SelfAttnW \leftarrow getSelfAttnW(M, \overrightarrow{Tkn})$;
$\overrightarrow{TknAttnW} \leftarrow getTknAttnW(SelfAttnW)$;
$\overrightarrow{SortedTknInd} \leftarrow getSortedTknIndices(TknAttnW)$;
$i \leftarrow 0$;
while $i < \lceil (k/100) * SortedTknInd.length \rceil$ **do**
    $LAT \leftarrow LAT \cup Tkn[SortedTknInd[i]]$;
    $i \leftarrow i + 1$;
$SmtScore \leftarrow \emptyset$;
foreach $s_i \in Smt$ **do**
    $score \leftarrow |s_i \cap LAT| / s_i.length$;
    $SmtScore \leftarrow SmtScore \cup \langle s_i, score \rangle$;
$SortedSmtInd \leftarrow getSortedSmtIndices(SmtScore)$;
$i \leftarrow 0$;
while $i < \lceil (k/100) * SortedSmtInd.length \rceil$ **do**
    $LAS \leftarrow LAS \cup Smt[SortedSmtInd[i]]$;
    $i \leftarrow i + 1$;
return LAS

[1]https://github.com/jose/javalang.git

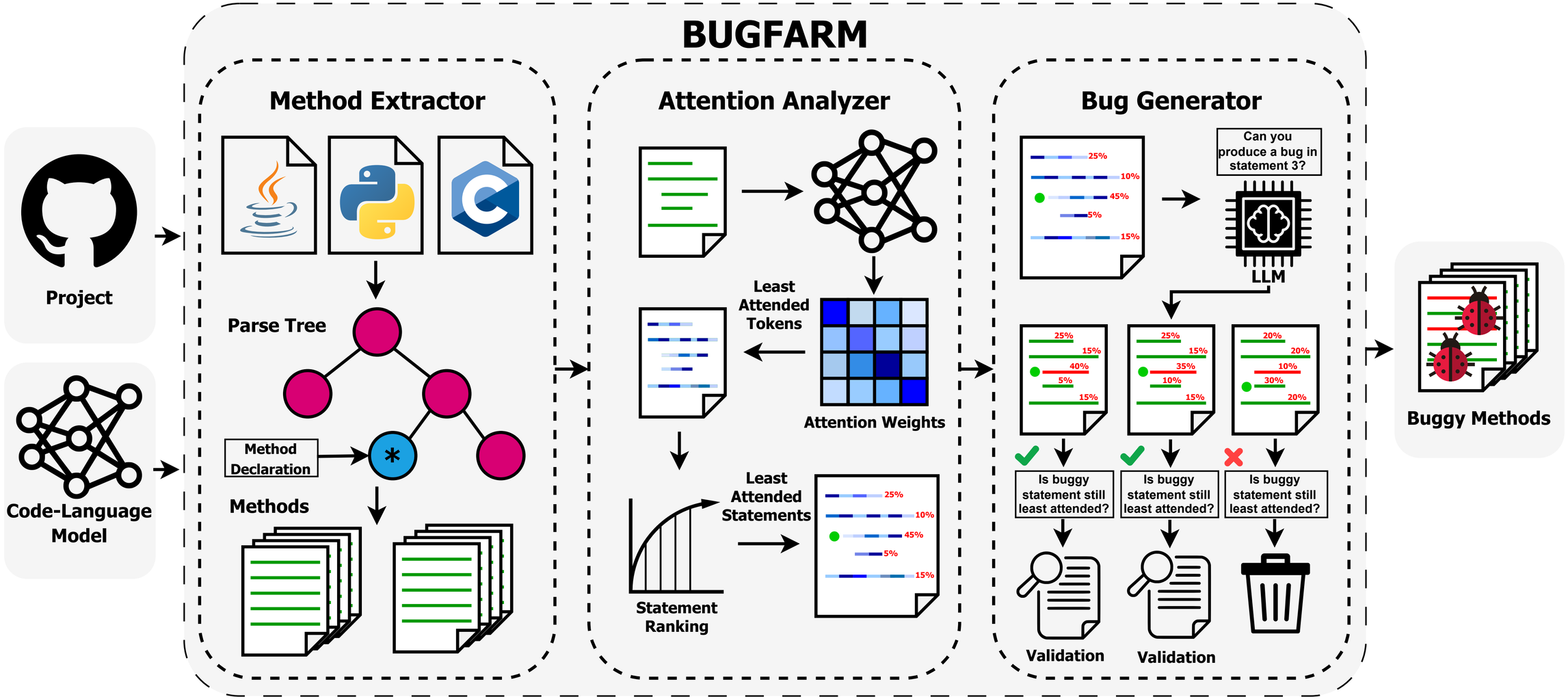
Attention Analysis - Example

```
private void handleOption(Option option) throws ParseException {
    // check the previous option before handling the next one
    checkRequiredArgs();
    option = (Option) option.clone();
    updateRequiredOptions(option);
    cmd.addOption(option);
    if (option.hasArg()) {
        currentOption = option;
    } else {
        currentOption = null;
    }
}
```

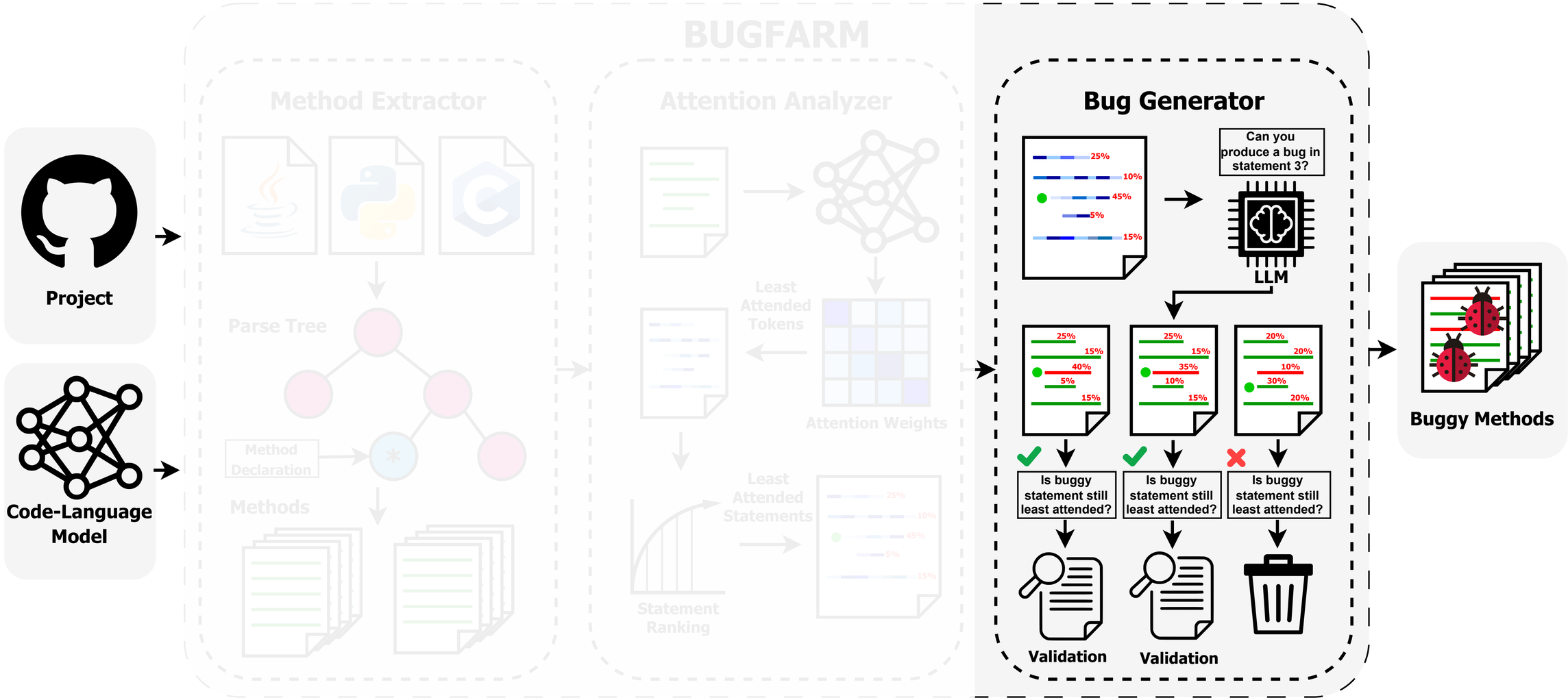
Least Attended Statements



BugFarm Framework



BugFarm Framework



Bug Generator

- Given a method and LASs
- Prompt a strong LLM to generate N different buggy versions
- Validate the generated bugs are parseable
- Ensure changed statements are still LASs
- N = 3 in all our experiments and gpt-3.5-turbo is used as strong LLM

Algorithm 2: Bug Generator

Inputs: Method $method$, Least attended statements LAS, Number of bugs N, Transformer-based model M
Output: Buggy methods $Bugs$

```
1  Bugs ← ∅;
2  LASInd ← getLASIndices(method, LAS);
3  method ← addIndices(method);
4  Prompt ← "Inject $N bugs in the following method by changing
     only the statements at locations $LASInd: $method";
5  Responses ← queryLLM(Prompt);
6  foreach Response ∈ Responses do
7      if !isParseable(Response) then
8          continue;
9      newLAS ← getLAS(Response, M);
10     check ← true;
11     foreach stmt ∈ getDiff(Response, method) do
12         if stmt ∉ newLAS then
13             check ← false;
14             break;
15     if check then
16         Bugs ← Bugs ∪ Response;
```

[1]https://github.com/jose/javalang.git

Bug Generator - Example

```java
private void handleOption(Option option) throws ParseException {
    checkRequiredArgs();
    Option newOp = (Option) option.clone();
    updateRequiredOptions(newOp);
    cmd.addOption(newOp);
    if (newOp.hasArg() && newOp.args >= 3)) {
        currentOption = newOp;
    } else {
        currentOption = option;
    }
}
```

(a) Original Method

```java
private void handleOption(Option option) throws ParseException {
    checkRequiredArgs();
    Option newOp = (String) option.clone();
    updateRequiredOptions(newOp);
    cmd.addOption(newOp);
    if (newOp.hasArg() && newOp.arg >= 3) {
        currentOption = newOp;
    } else {
        currentOption = option;
    }
}
```

(b) Bug generated by LEAM [52]

```java
private void handleOption(Option option) throws ParseException {
    checkRequiredArgs();
    Option newOp = (Option) option.clone();
    updateRequiredOptions(cmd);
    cmd.addOption(newOp);
    if (newOp.hasArg() && newOp.arg >= 3) {
        currentOption = newOp;
    } else {
        currentOption = option;
    }
}
```

(c) Bug generated by μBERT [33]

```java
private void handleOption(Option option) throws ParseException {
    checkRequiredArgs();
    Option newOp = (Option) option.clone();
    updateRequiredOptions(newOp);
    cmd.addOption(newOp);
    if (newOp.hasArg() && newOp.arg >= 3) {
        currentOption = option;
    } else {
        currentOption = newOp;
    }
}
```

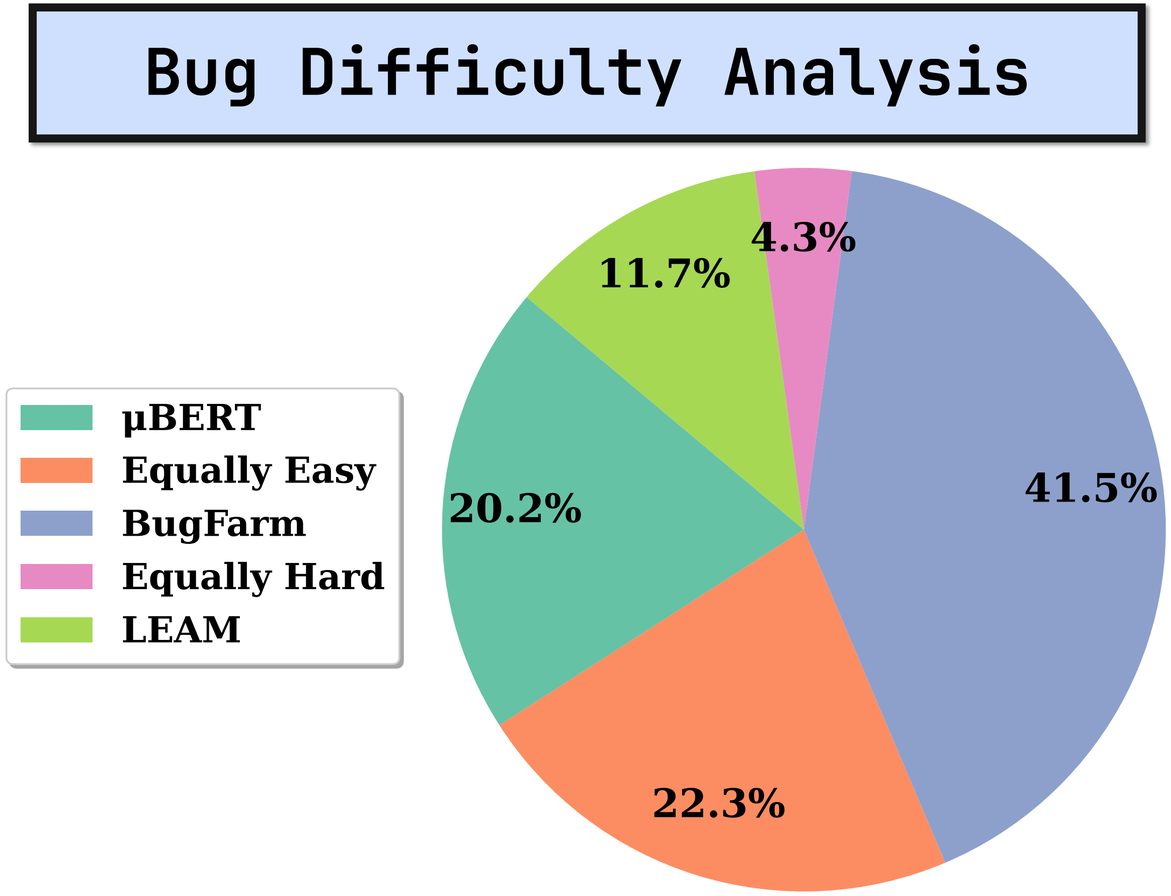
(d) Bug generated by BugFarm

Experimental Setup

- Mutant Generation Techniques:
  - LEAM: Use deep learning to learn to mutate code from large set of examples
  - µBERT: MASKs AST nodes and uses language models to predict them
- Automated Bug Confirmation:
  - 1,908,566 mutants | 699,575 syntactically correct | 434,215 confirmed bugs
- Models:
  - Bug Prediction (CodeBERT, NatGen, CodeT5), Automated Program Repair (CodeT5-large)
- Subject Projects:
  - 15 Java projects from Defects4J

Manual Bug Confirmation

- 300 samples from confirmed bugs
- 97 participants & 3 participant / sample
- Labels: Yes (bug), No (not bug), Maybe
- 290 person-hours
- 143 samples confirmed as bugs by majority voting:
  - BugFarm: 72 (50.35%)
  - µBERT: 53 (37.06%)
  - LEAM: 18 (12.59%)



Research Questions

- RQ1: Characteristics of generated bugs
- RQ2: Effectiveness in generating hard-to-detect bugs
- RQ3: Effectiveness in generating hard-to-repair bugs
- RQ4: Performance

Characteristics of Generated Bugs

❖ BugFarm changes more statements to generate bugs than LEAM and µBERT

Subjects	M	BUGFARM							LEAM					µBERT				
		SCM	CB	SI	LD	ED	OL	OM	SCM	CB	SI	LD	ED	SCM	CB	SI	LD	ED
cli	276	295	266	2.9	0.0%	33.2	⟨13.9%,0.6⟩	⟨8.6%,0.7⟩	2126	1388	2.6	13.69%	26.1	2053	1532	2.0	0.07%	19.3
codec	901	544	360	3.1	0.0%	31.6	⟨3.1%,0.7⟩	⟨5.5%,0.9⟩	4607	2130	2.8	7.84%	22.4	14738	9551	2.0	0.0%	17.9
collections	4440	3775	3175	2.9	0.0%	29.7	⟨11.1%,0.6⟩	⟨5.1%,0.7⟩	21379	11266	2.8	7.77%	20.9	34038	22196	2.0	0.0%	20.0
compress	4123	573	468	3.0	0.0%	29.4	⟨4.4%,0.7⟩	⟨2.4%,0.8⟩	20401	9373	2.6	12.55%	24.5	57815	23602	2.0	0.0%	24.2
csv	248	282	252	2.7	0.0%	24.9	⟨11.9%,0.7⟩	⟨1.5%,0.7⟩	1803	1189	2.8	14.55%	33.5	131	109	2.0	0.0%	7.0
jxpath	1672	1152	948	3.3	0.0%	28.0	⟨8.0%,0.7⟩	⟨6.1%,0.8⟩	11511	6169	2.9	15.24%	26.2	21335	10907	2.0	0.0%	25.8
lang	3810	4421	3791	3.0	0.0%	30.1	⟨7.5%,0.7⟩	⟨3.1%,0.8⟩	26365	16316	2.7	9.69%	21.4	23312	15732	2.0	0.02%	21.1
math	5796	6466	6207	3.4	0.0%	34.2	⟨4.4%,0.7⟩	⟨2.3%,0.8⟩	43090	42262	2.7	8.43%	17.0	85393	83512	2.0	0.01%	18.3
gson	985	631	545	3.3	0.0%	38.2	⟨7.3%,0.7⟩	⟨1.6%,0.7⟩	4896	2992	2.6	17.95%	27.9	4173	2067	2.0	0.05%	19.8
jackson-core	2626	2281	1732	4.0	0.0%	43.1	⟨2.9%,0.7⟩	⟨2.3%,0.7⟩	21472	9963	2.5	22.58%	25.8	19118	9310	2.1	0.0%	21.6
jackson-db	8076	4828	3965	3.9	0.0%	41.9	⟨7.2%,0.7⟩	⟨6.9%,0.7⟩	36155	18454	2.8	12.17%	28.7	51032	22454	2.0	0.0%	22.3
jackson-xml	586	312	258	3.9	0.0%	44.0	⟨6.3%,0.7⟩	⟨5.9%,0.7⟩	3096	1580	2.7	12.85%	31.1	2576	1226	2.0	0.08%	14.7
jfreechart	8602	4051	3186	3.1	0.0%	27.1	⟨7.4%,0.7⟩	⟨1.1%,0.8⟩	40313	17987	2.5	10.64%	19.8	18924	7045	2.0	0.07%	28.2
joda-time	4279	4188	3734	2.6	0.0%	22.1	⟨8.6%,0.6⟩	⟨3.6%,0.7⟩	26224	15641	2.9	6.29%	24.7	49907	28694	2.0	0.0%	21.1
jsoup	1642	1561	1356	2.8	0.0%	28.5	⟨9.4%,0.7⟩	⟨4.7%,0.7⟩	9534	5456	2.5	14.15%	23.6	6699	4013	2.2	0.05%	21.0
Total	**48062**	**35359**	**30242**	-	-	-	-	-	**272972**	**162166**	-	-	-	**391244**	**241950**	-	-	-
Average	**3204**	**2357**	**2016**	**3.2**	**0.0%**	**32.4**	**⟨7.6%,0.7⟩**	**⟨4.0%,0.7⟩**	**18198**	**10811**	**2.7**	**12.43%**	**24.9**	**26083**	**16130**	**2.0**	**0.02%**	**20.2**

Characteristics of Generated Bugs

❖ BugFarm changes are bigger, yet less noticeable than LEAM and µBERT

Subjects	M	BUGFARM							LEAM					µBERT				
		SCM	CB	SI	LD	ED	OL	OM	SCM	CB	SI	LD	ED	SCM	CB	SI	LD	ED
cli	276	295	266	2.9	0.0%	33.2	⟨13.9%,0.6⟩	⟨8.6%,0.7⟩	2126	1388	2.6	13.69%	26.1	2053	1532	2.0	0.07%	19.3
codec	901	544	360	3.1	0.0%	31.6	⟨3.1%,0.7⟩	⟨5.5%,0.9⟩	4607	2130	2.8	7.84%	22.4	14738	9551	2.0	0.0%	17.9
collections	4440	3775	3175	2.9	0.0%	29.7	⟨11.1%,0.6⟩	⟨5.1%,0.7⟩	21379	11266	2.8	7.77%	20.9	34038	22196	2.0	0.0%	20.0
compress	4123	573	468	3.0	0.0%	29.4	⟨4.4%,0.7⟩	⟨2.4%,0.8⟩	20401	9373	2.6	12.55%	24.5	57815	23602	2.0	0.0%	24.2
csv	248	282	252	2.7	0.0%	24.9	⟨11.9%,0.7⟩	⟨1.5%,0.7⟩	1803	1189	2.8	14.55%	33.5	131	109	2.0	0.0%	7.0
jxpath	1672	1152	948	3.3	0.0%	28.0	⟨8.0%,0.7⟩	⟨6.1%,0.8⟩	11511	6169	2.9	15.24%	26.2	21335	10907	2.0	0.0%	25.8
lang	3810	4421	3791	3.0	0.0%	30.1	⟨7.5%,0.7⟩	⟨3.1%,0.8⟩	26365	16316	2.7	9.69%	21.4	23312	15732	2.0	0.02%	21.1
math	5796	6466	6207	3.4	0.0%	34.2	⟨4.4%,0.7⟩	⟨2.3%,0.8⟩	43090	42262	2.7	8.43%	17.0	85393	83512	2.0	0.01%	18.3
gson	985	631	545	3.3	0.0%	38.2	⟨7.3%,0.7⟩	⟨1.6%,0.7⟩	4896	2992	2.6	17.95%	27.9	4173	2067	2.0	0.05%	19.8
jackson-core	2626	2281	1732	4.0	0.0%	43.1	⟨2.9%,0.7⟩	⟨2.3%,0.7⟩	21472	9963	2.5	22.58%	25.8	19118	9310	2.1	0.0%	21.6
jackson-db	8076	4828	3965	3.9	0.0%	41.9	⟨7.2%,0.7⟩	⟨6.9%,0.7⟩	36155	18454	2.8	12.17%	28.7	51032	22454	2.0	0.0%	22.3
jackson-xml	586	312	258	3.9	0.0%	44.0	⟨6.3%,0.7⟩	⟨5.9%,0.7⟩	3096	1580	2.7	12.85%	31.1	2576	1226	2.0	0.08%	14.7
jfreechart	8602	4051	3186	3.1	0.0%	27.1	⟨7.4%,0.7⟩	⟨1.1%,0.8⟩	40313	17987	2.5	10.64%	19.8	18924	7045	2.0	0.07%	28.2
joda-time	4279	4188	3734	2.6	0.0%	22.1	⟨8.6%,0.6⟩	⟨3.6%,0.7⟩	26224	15641	2.9	6.29%	24.7	49907	28694	2.0	0.0%	21.1
jsoup	1642	1561	1356	2.8	0.0%	28.5	⟨9.4%,0.7⟩	⟨4.7%,0.7⟩	9534	5456	2.5	14.15%	23.6	6699	4013	2.2	0.05%	21.0
Total	**48062**	**35359**	**30242**	-	-	-	-	-	**272972**	**162166**	-	-	-	**391244**	**241950**	-	-	-
Average	**3204**	**2357**	**2016**	**3.2**	**0.0%**	**32.4**	**⟨7.6%,0.7⟩**	**⟨4.0%,0.7⟩**	**18198**	**10811**	**2.7**	**12.43%**	**24.9**	**26083**	**16130**	**2.0**	**0.02%**	**20.2**

Effectiveness in Generating Hard-to-Detect Bugs

❖ Bug prediction models have a hard time predicting BugFarm bugs

	Fine-tune-Model-Test	Acc	Prec	Rec	F1	FNR
1	μBERT-CodeBERT-BugFarm	**71.88 (4.49% ↓)**	**89.94 (1.11% ↓)**	**49.26 (12.18% ↓)**	**63.66 (8.26% ↓)**	**50.74 (15.55% ↑)**
2	μBERT-CodeBERT-LEAM	75.26	90.95	56.09	69.39	43.91
3	LEAM-CodeBERT-BugFarm	**69.42 (10.76% ↓)**	**94.05 (1.48% ↓)**	**41.47 (28.93% ↓)**	**57.56 (20.53% ↓)**	**58.53 (40.53% ↑)**
4	LEAM-CodeBERT-μBERT	77.79	95.46	58.35	72.43	41.65
5	μBERT-CodeT5-BugFarm	**75.35 (0.91% ↓)**	**87.45 (0.35% ↓)**	**59.2 (2.18% ↓)**	**70.6 (1.45% ↓)**	**40.8 (3.34% ↑)**
6	μBERT-CodeT5-LEAM	76.04	87.76	60.52	71.64	39.48
7	LEAM-CodeT5-BugFarm	**71.27 (8.90% ↓)**	**89.29 (1.85% ↓)**	**48.35 (22.86% ↓)**	**62.73 (15.48% ↓)**	**51.65 (38.40% ↑)**
8	LEAM-CodeT5-μBERT	78.23	90.97	62.68	74.22	37.32
9	μBERT-NatGen-BugFarm	**74.42 (1.29% ↓)**	**85.07 (0.29% ↓)**	**59.24 (3.41% ↓)**	**69.84 (2.13% ↓)**	**40.76 (5.40% ↑)**
10	μBERT-NatGen-LEAM	75.39	85.32	61.33	71.36	38.67
11	LEAM-NatGen-BugFarm	**69.46 (9.33% ↓)**	**88.35 (2.14% ↓)**	**44.84 (24.80% ↓)**	**59.49 (17.17% ↓)**	**55.16 (36.64% ↑)**
12	LEAM-NatGen-μBERT	76.61	90.28	59.63	71.82	40.37
13	REAL-CodeBERT-BugFarm	**53.02 (3.84% ↓)**	**55.23 (5.20% ↓)**	**31.92 (11.92% ↓)**	**40.46 (9.44% ↓)**	**68.08 (6.78% ↑)**
14	REAL-CodeBERT-μBERT	55.14	58.26	36.24	44.68	63.76
15	REAL-CodeBERT-LEAM	57.62	61.38	41.12	49.25	58.88
16	REAL-CodeT5-BugFarm	**49.02 (4.72% ↓)**	**48.93 (5.03% ↓)**	**44.64 (9.18% ↓)**	**46.69 (7.20% ↓)**	**55.36 (8.87% ↑)**
17	REAL-CodeT5-μBERT	51.45	51.52	49.15	50.31	50.85
18	REAL-CodeT5-LEAM	53.81	53.79	54.05	53.92	45.95
19	REAL-NatGen-BugFarm	**50.75 (0.12% ↓)**	**50.62 (0.10% ↓)**	**61.07 (0.42% ↓)**	**55.36 (0.23% ↓)**	**38.93 (0.67% ↑)**
20	REAL-NatGen-μBERT	50.81	50.67	61.33	55.49	38.67
21	REAL-NatGen-LEAM	53.81	53.01	67.05	59.21	32.9

Effectiveness in Generating Hard-to-Detect Bugs

❖ Bug prediction models have a hard time predicting BugFarm bugs

	Fine-tune-Model-Test	Acc	Prec	Rec	F1	FNR
1	μBERT-CodeBERT-BugFarm	**71.88 (4.49% ↓)**	**89.94 (1.11% ↓)**	**49.26 (12.18% ↓)**	**63.66 (8.26% ↓)**	**50.74 (15.55% ↑)**
2	μBERT-CodeBERT-LEAM	75.26	90.95	56.09	69.39	43.91
3	LEAM-CodeBERT-BugFarm	**69.42 (10.76% ↓)**	**94.05 (1.48% ↓)**	**41.47 (28.93% ↓)**	**57.56 (20.53% ↓)**	**58.53 (40.53% ↑)**
4	LEAM-CodeBERT-μBERT	77.79	95.46	58.35	72.43	41.65
5	μBERT-CodeT5-BugFarm	**75.35 (0.91% ↓)**	**87.45 (0.35% ↓)**	**59.2 (2.18% ↓)**	**70.6 (1.45% ↓)**	**40.8 (3.34% ↑)**
6	μBERT-CodeT5-LEAM	76.04	87.76	60.52	71.64	39.48
7	LEAM-CodeT5-BugFarm	**71.27 (8.90% ↓)**	**89.29 (1.85% ↓)**	**48.35 (22.86% ↓)**	**62.73 (15.48% ↓)**	**51.65 (38.40% ↑)**
8	LEAM-CodeT5-μBERT	78.23	90.97	62.68	74.22	37.32
9	μBERT-NatGen-BugFarm	**74.42 (1.29% ↓)**	**85.07 (0.29% ↓)**	**59.24 (3.41% ↓)**	**69.84 (2.13% ↓)**	**40.76 (5.40% ↑)**
10	μBERT-NatGen-LEAM	75.39	85.32	61.33	71.36	38.67
11	LEAM-NatGen-BugFarm	**69.46 (9.33% ↓)**	**88.35 (2.14% ↓)**	**44.84 (24.80% ↓)**	**59.49 (17.17% ↓)**	**55.16 (36.64% ↑)**
12	LEAM-NatGen-μBERT	76.61	90.28	59.63	71.82	40.37
13	REAL-CodeBERT-BugFarm	**53.02 (3.84% ↓)**	**55.23 (5.20% ↓)**	**31.92 (11.92% ↓)**	**40.46 (9.44% ↓)**	**68.08 (6.78% ↑)**
14	REAL-CodeBERT-μBERT	55.14	58.26	36.24	44.68	63.76
15	REAL-CodeBERT-LEAM	57.62	61.38	41.12	49.25	58.88
16	REAL-CodeT5-BugFarm	**49.02 (4.72% ↓)**	**48.93 (5.03% ↓)**	**44.64 (9.18% ↓)**	**46.69 (7.20% ↓)**	**55.36 (8.87% ↑)**
17	REAL-CodeT5-μBERT	51.45	51.52	49.15	50.31	50.85
18	REAL-CodeT5-LEAM	53.81	53.79	54.05	53.92	45.95
19	REAL-NatGen-BugFarm	**50.75 (0.12% ↓)**	**50.62 (0.10% ↓)**	**61.07 (0.42% ↓)**	**55.36 (0.23% ↓)**	**38.93 (0.67% ↑)**
20	REAL-NatGen-μBERT	50.81	50.67	61.33	55.49	38.67
21	REAL-NatGen-LEAM	53.81	53.01	67.05	59.21	32.9

Effectiveness in Generating Hard-to-Repair Bugs

- State-of-the-art FitRepair APR model with CodeT5-large
- Bugs with one, two, or more statements from BugFarm, LEAM, and µBERT
- Bugs are considered hard-to-repair if APR model cannot repair them

	LEAM	µBERT	BUGFARM-CODEBERT	BUGFARM-CODET5	BUGFARM-NATGEN
Total Bugs (SI=1,SI=2,SI>2)	200 (174,25,1)	200 (187,13,0)	200 (125,40,35)	200 (92,43,65)	200 (104,30,66)
Success Rate	36%	49%	29%	29%	28%

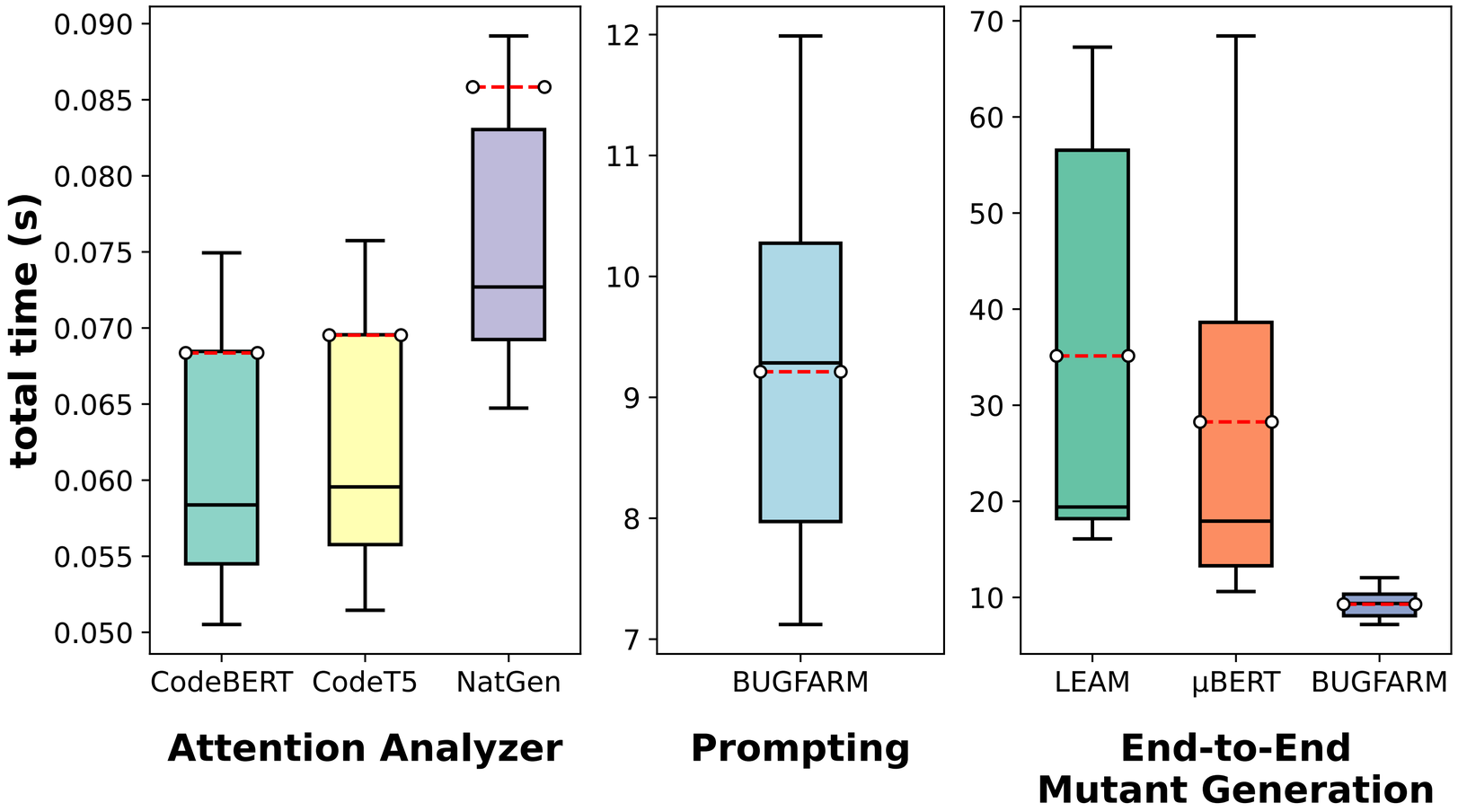
Effectiveness in Generating Hard-to-Repair Bugs

- State-of-the-art FitRepair APR model with CodeT5-large
- Bugs with one, two, or more statements from BugFarm, LEAM, and µBERT
- Bugs are considered hard-to-repair if APR model cannot repair them

	LEAM	µBERT	BUGFARM-CodeBERT	BUGFARM-CodeT5	BUGFARM-NATGEN
Total Bugs (SI=1,SI=2,SI>2)	200 (174,25,1)	200 (187,13,0)	200 (125,40,35)	200 (92,43,65)	200 (104,30,66)
Success Rate	36%	49%	29%	29%	28%

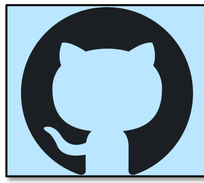
Performance

❖ BugFarm generates bugs in 9.2s compared to LEAM=35s and µBERT=28s



Concluding Remarks

- We introduce BugFarm, a language and model-agnostic tool for synthetic bug generation
- BugFarm bugs are unique and involves changes in multiple statements
- It generates hard-to-detect and hard-to-repair bugs
- BugFarm generates bugs **x3.80** and **x3.04** faster than LEAM and µBERT
- Future Work:
  - Integrate BugFarm with more recent reasoning LLMs
  - Measure the closeness with real bugs



github.com/Intelligent-CAT-Lab/BugFarm