Semantic Bug Seeding: A Learning-Based Approach for Creating Realistic Bugs

Michael Pradel

Software Lab – University of Stuttgart

Joint work with Jibesh Patra

Why Seed Bugs?

Large set of known, realistic bugs

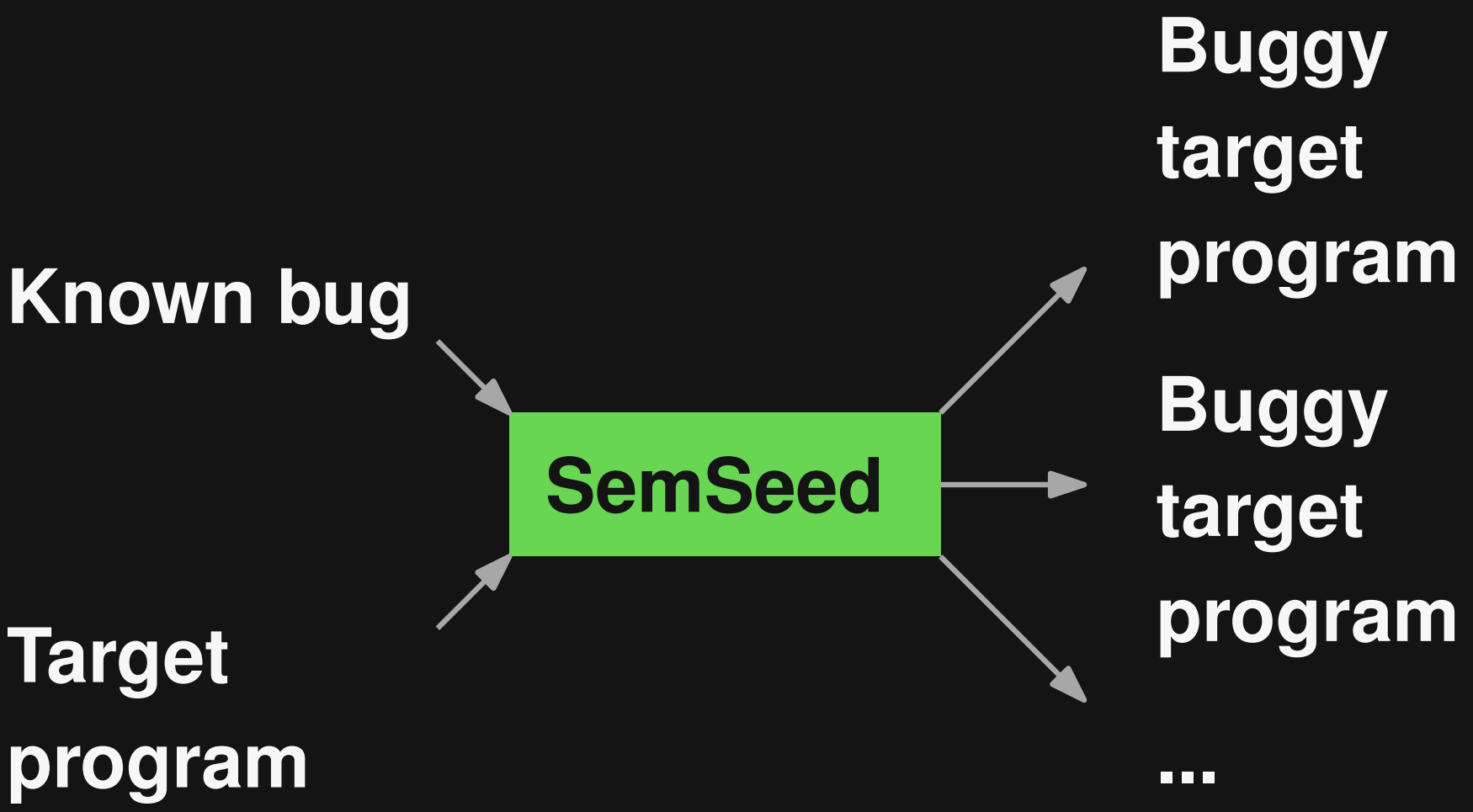
Benchmark for

- Test suites
- Bug detectors
- Repair tools

Training data for

- Learning-based bug detectors
- Learning-based repair tools

Idea: Imitate a Known Bug



Example and Challenges

```
if (process.platform === "darwin")
   ...
```

Known bug fix

```
if (process.platform !== "win32")
   ...
```

Example and Challenges

```
if (process.platform === "darwin")
  ...
```


```
if (process.platform !== "win32")
  ...
```

Example and Challenges

```
if (process.platform === "darwin")
  ...
```


```
if (process.platform !== "win32")
  ...
```

Challenge 1: Where in the target program to seed this kind of bug?

Example and Challenges

```
if (process.platform === "darwin")
   ...
```


```
if (process.platform !== "win32")
   ...
```

Challenge 2: How to adapt the bug to the target program?

Example and Challenges

```
if (process.platform === "darwin")
   ...
```


```
if (process.platform !== "win32")
   ...
```

Challenge 3: How to handle “unbound” tokens?

Step 1: Abstraction to Bug Pattern

```
if (process.platform === "darwin")
  ...
```


```
if (process.platform !== "win32")
  ...
```

- **Reduce** to smallest AST subtree that contains all changed tokens
- **Abstract** identifiers and literals

Step 1: Abstraction to Bug Pattern

```
process.platform === "darwin"
```


```
process.platform !== "win32"
```

- **Reduce** to smallest AST subtree that contains all changed tokens
- **Abstract** identifiers and literals

Step 1: Abstraction to Bug Pattern

```
id1.id2 === lit1
```


```
id1.id2 !== lit2
```

- **Reduce** to smallest AST subtree that contains all changed tokens
- **Abstract** identifiers and literals

Step 2: Semantic Matching

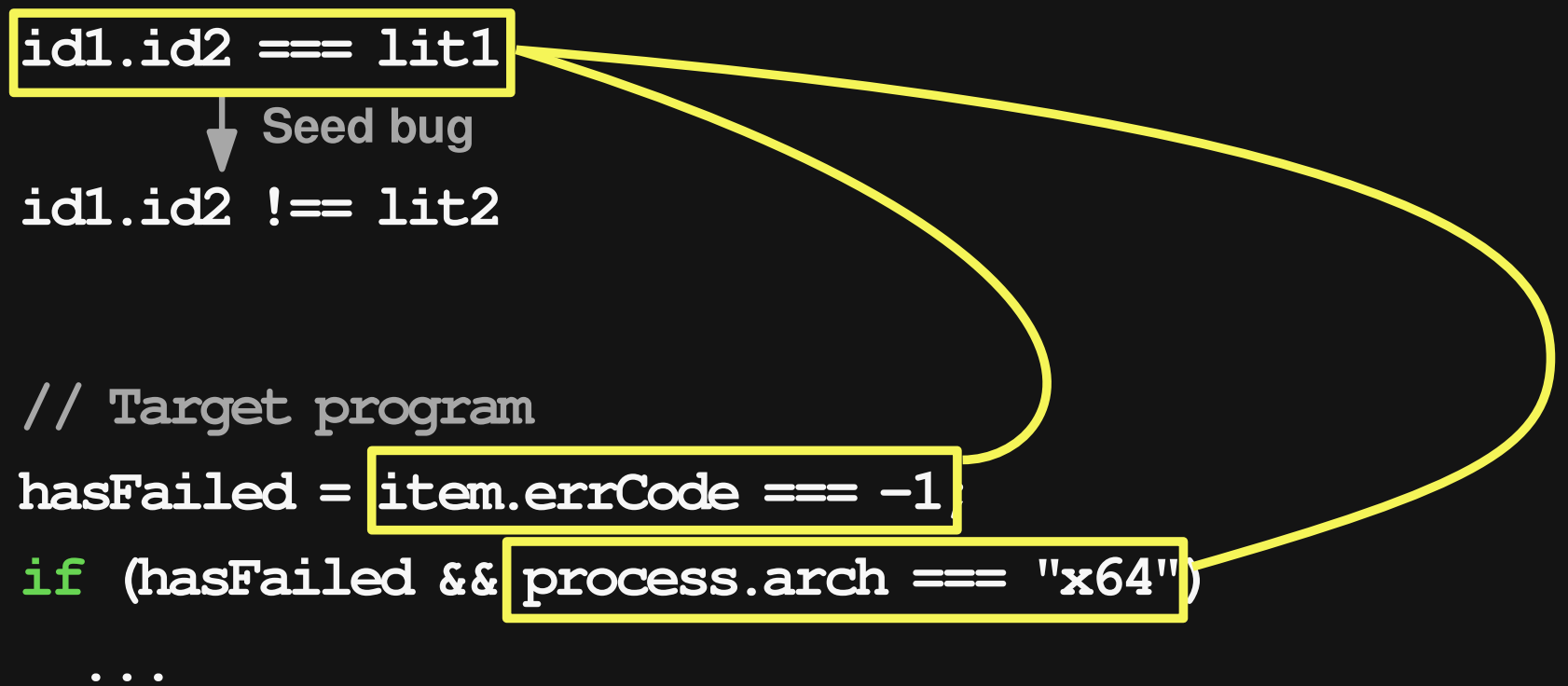
```
id1.id2 === lit1
```


```
id1.id2 !== lit2
```

```
// Target program
hasFailed = item.errCode === -1;
if (hasFailed && process.arch === "x64")
  ...
```

- **Syntactic** matching
- **Semantic** matching based on **learned token embeddings**

Step 2: Semantic Matching



- **Syntactic** matching
- **Semantic** matching based on **learned token embeddings**

Step 2: Semantic Matching

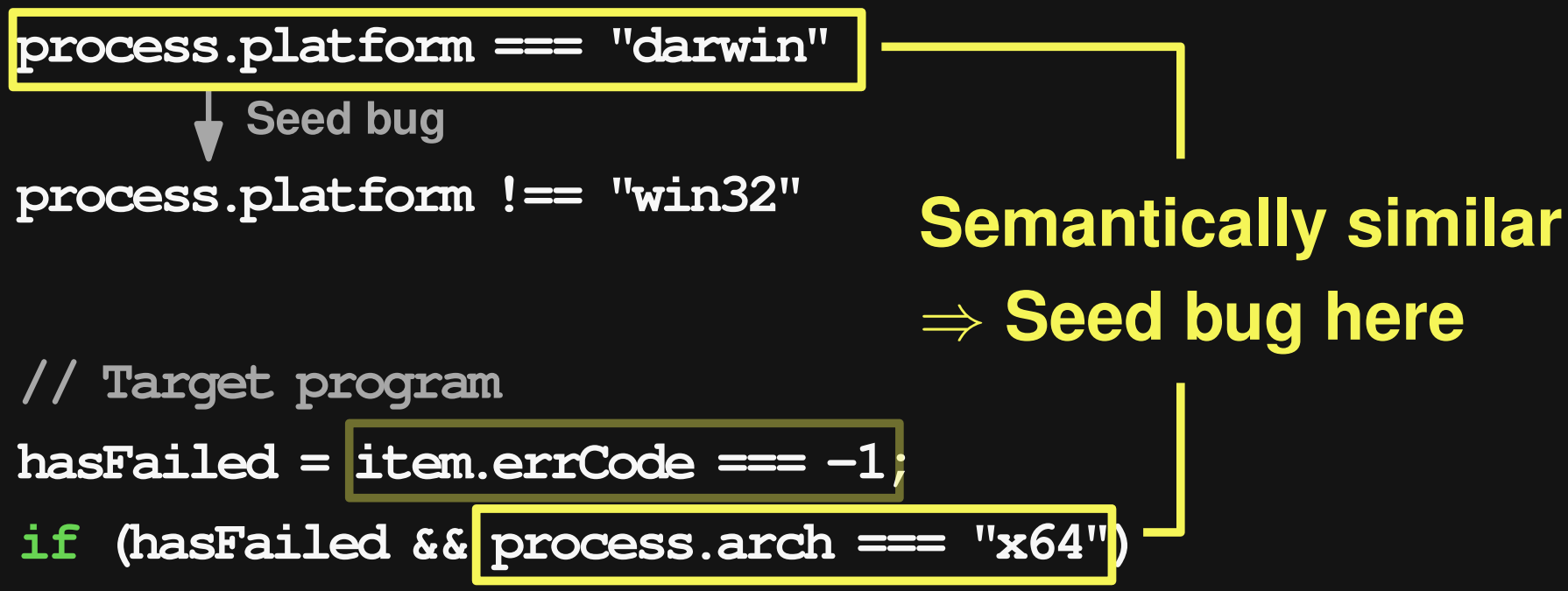
```
process.platform === "darwin"
```


```
process.platform !== "win32"


// Target program
hasFailed = item.errCode === -1;
if (hasFailed && process.arch === "x64")
  ...
```

- **Syntactic** matching
- **Semantic** matching based on **learned token embeddings**

Step 2: Semantic Matching



- **Syntactic** matching
- **Semantic** matching based on **learned token embeddings**

Step 3: Apply Pattern

```
process.platform === "darwin"
```


```
process.platform !== "win32"
```

```
hasFailed = item.errCode === -1;
if (hasFailed && process.arch === "x64")
  ..
```


```
hasFailed = item.errCode === -1;
if (hasFailed && process.arch !== ???)
  ...
```

Step 3: Apply Pattern

```
process.platform === "darwin"
```

Seed bug

```
process.platform !== "win32"
```

What literal to use?

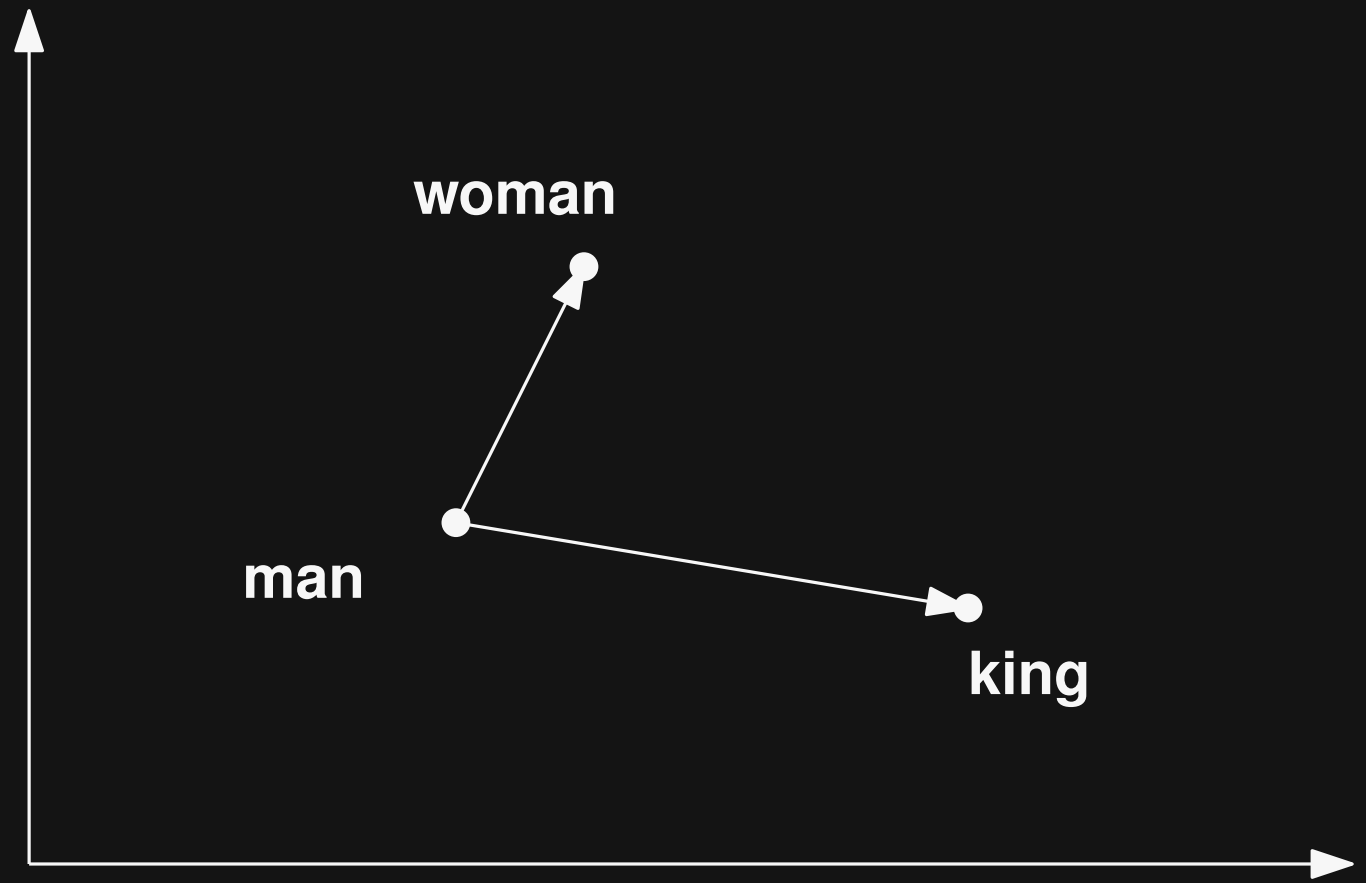
```
hasFailed = item.errCode === -1;
if (hasFailed && process.arch === "x64")
  ..
```

Seed bug

```
hasFailed = item.errCode === -1;
if (hasFailed && process.arch !== ???)
  ...
```

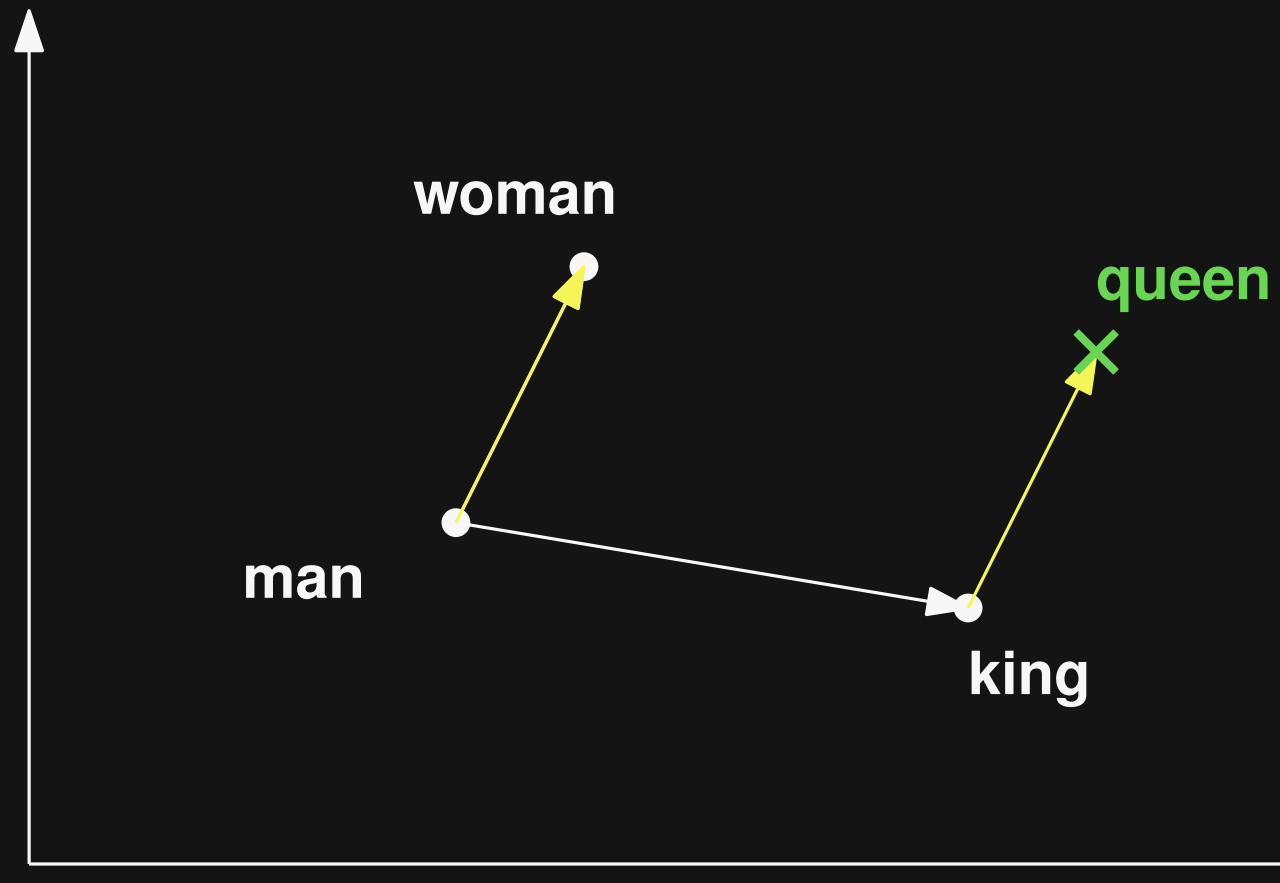
Step 3: Apply Pattern

Bind unbound tokens via analogy queries in token embedding space:



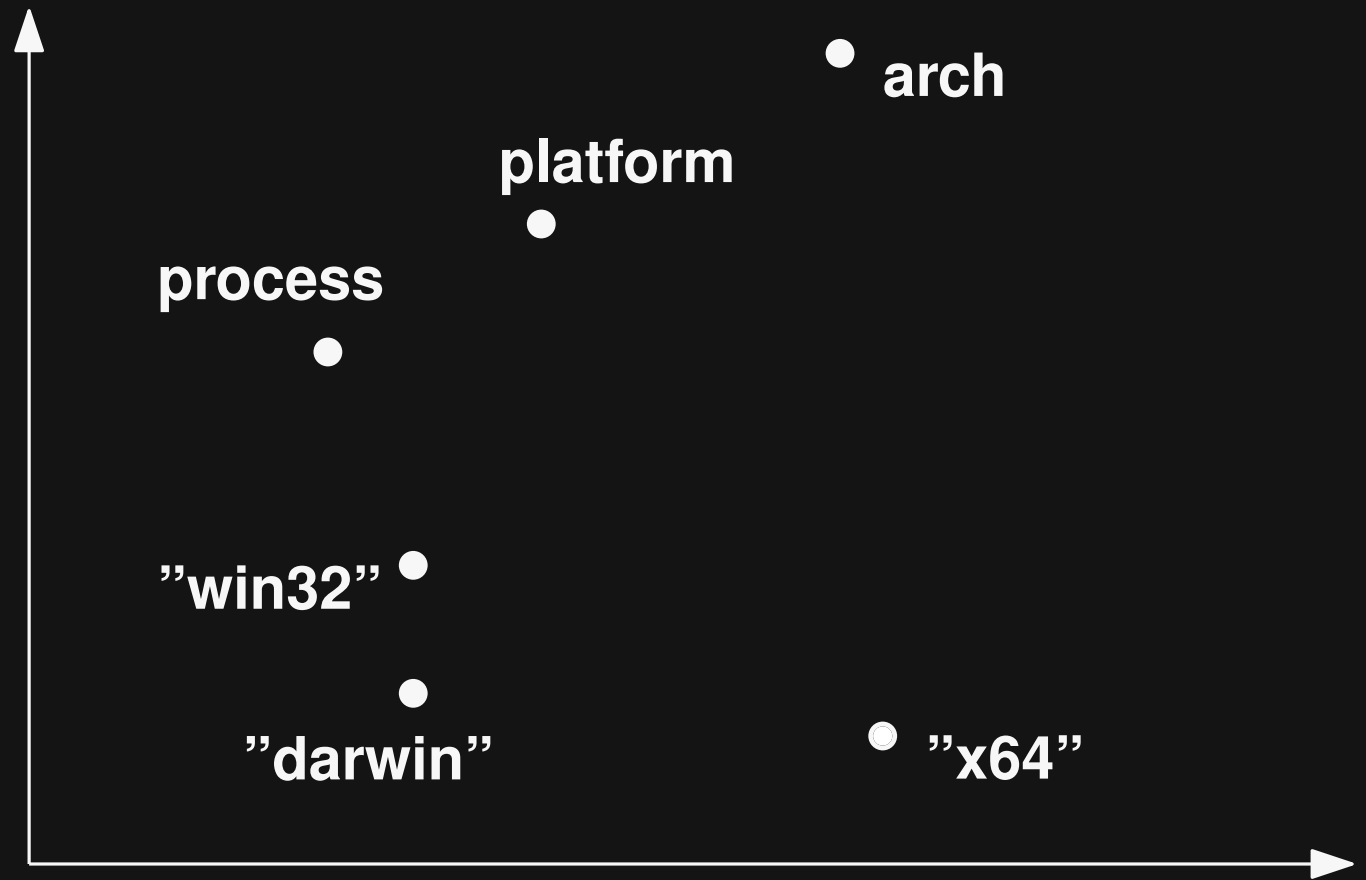
Step 3: Apply Pattern

Bind unbound tokens via analogy queries in token embedding space:



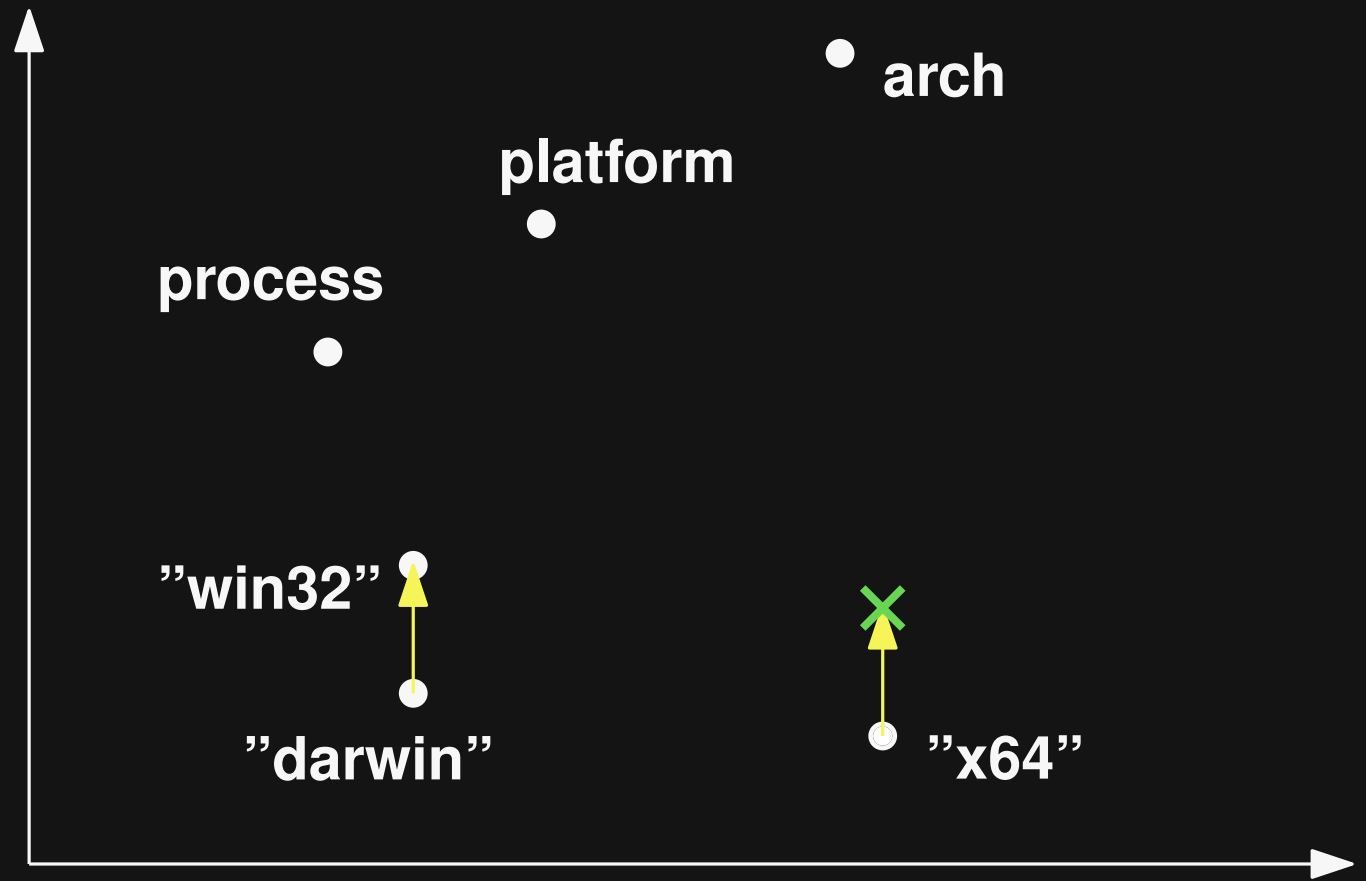
Step 3: Apply Pattern

Bind unbound tokens via analogy queries in token embedding space:



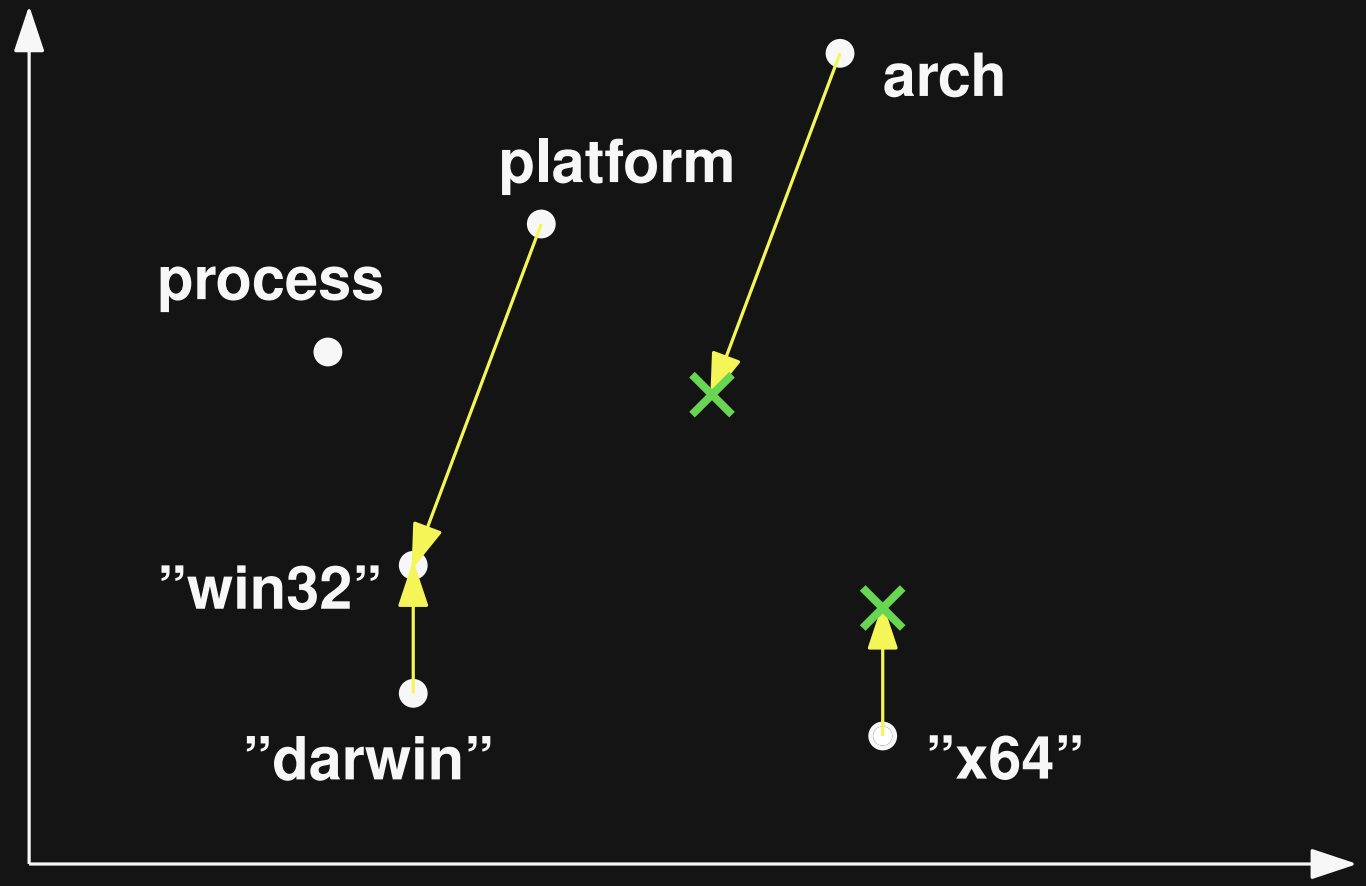
Step 3: Apply Pattern

Bind unbound tokens via analogy queries in token embedding space:



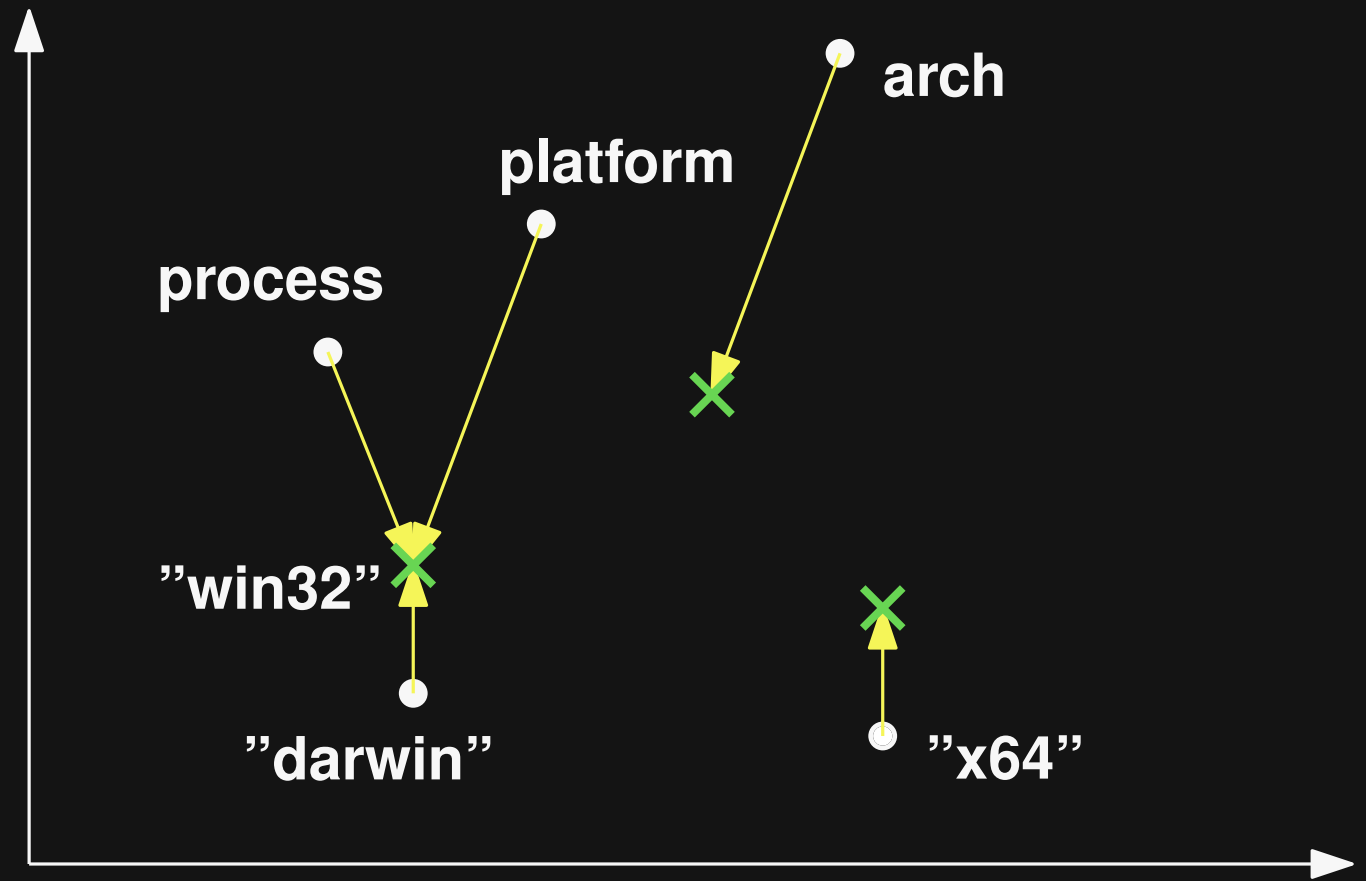
Step 3: Apply Pattern

Bind unbound tokens via analogy queries in token embedding space:



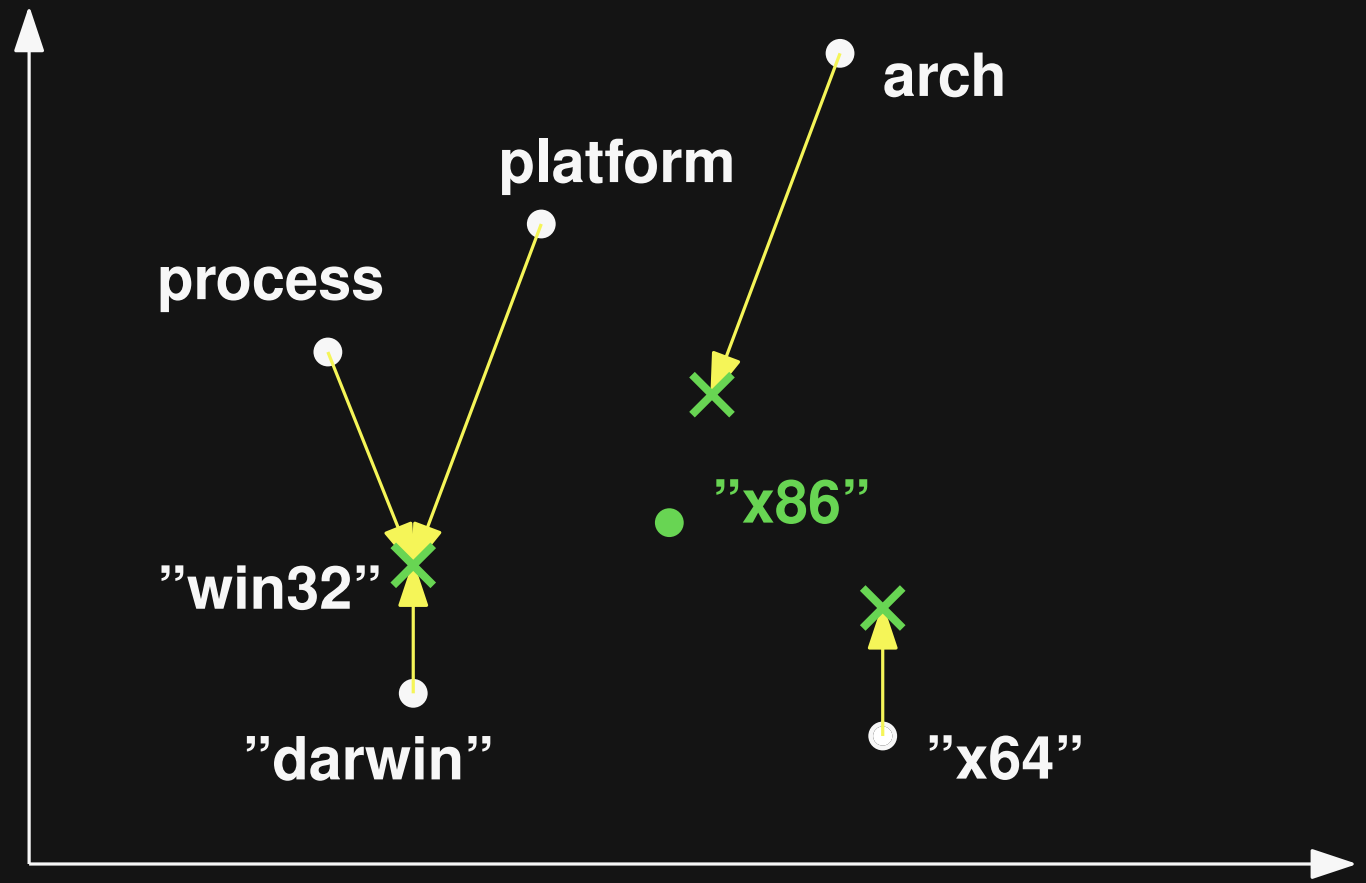
Step 3: Apply Pattern

Bind unbound tokens via analogy queries in token embedding space:



Step 3: Apply Pattern

Bind unbound tokens via analogy queries in token embedding space:



Step 3: Apply Pattern

```
process.platform === "darwin"
```

↓ Seed bug

```
process.platform !== "win32"
```

```
hasFailed = item.errCode === -1;
if (hasFailed && process.arch === "x64")
  ..
```

↓ Seed bug

```
hasFailed = item.errCode === -1;
if (hasFailed && process.arch !== "x86")
  ...
```

Evaluation

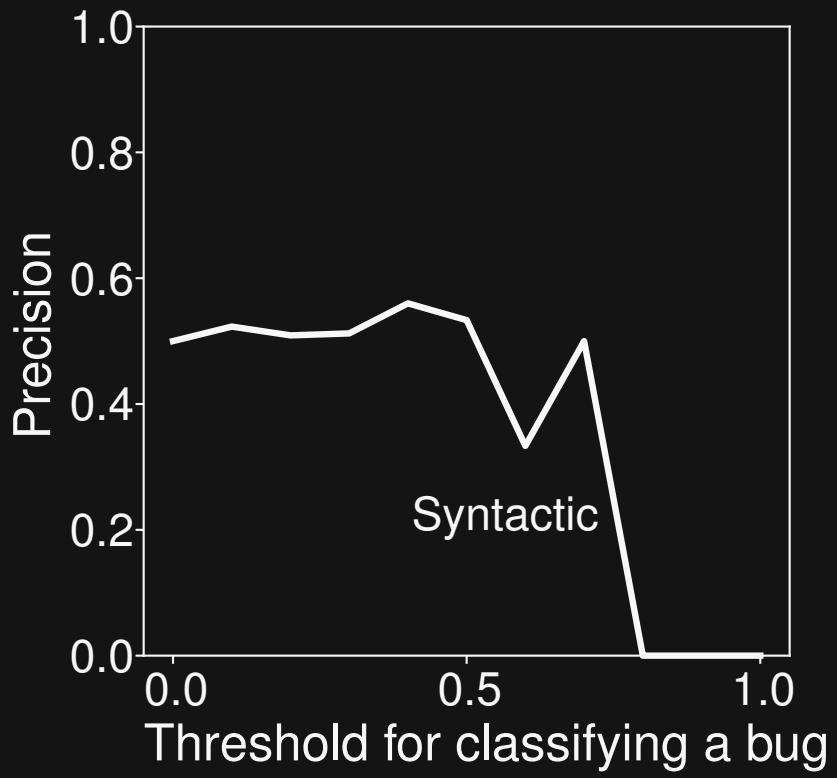
- **3,600 bug fixes from 100 popular JavaScript repositories**
  - Single-line changes with “bug”, “fix”, etc. in commit message
- **2,201 bug seeding patterns**
  - 62% have at least one unbound token

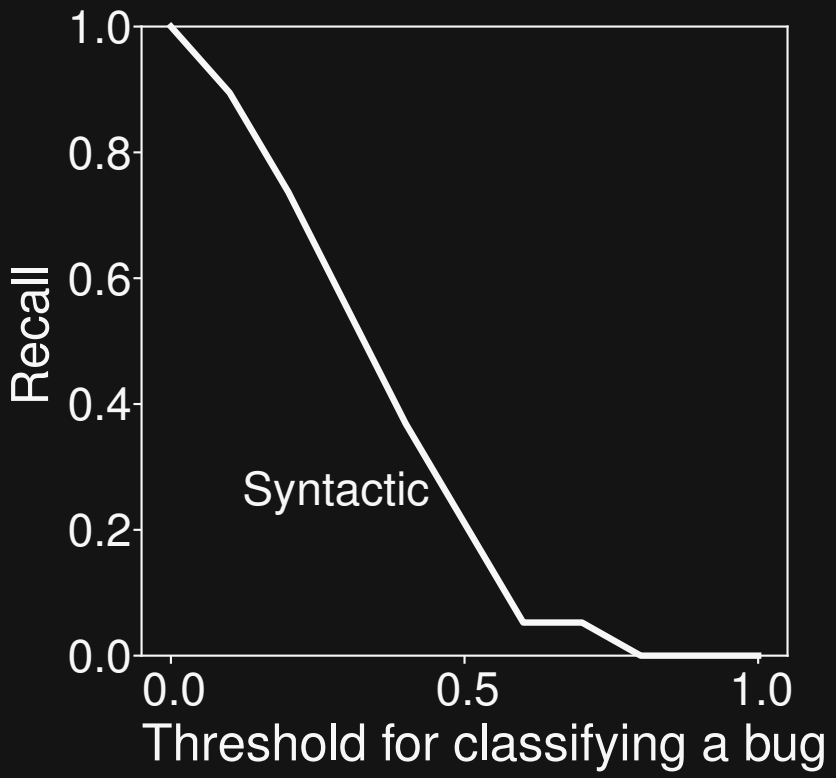
Reproducing Real Bugs

- **Seed 10 bugs per matching location**
- **Can reproduce held-out, real bugs?**
  - SemSeed reproduces 47/53 bugs
  - Syntactic baseline: 16/53 bugs
    - Main reason: Fails to guess unbound tokens

Learning Bug Detectors

Use seeded bugs as training data for learning bug detectors [DeepBugs, OOPSLA'18]



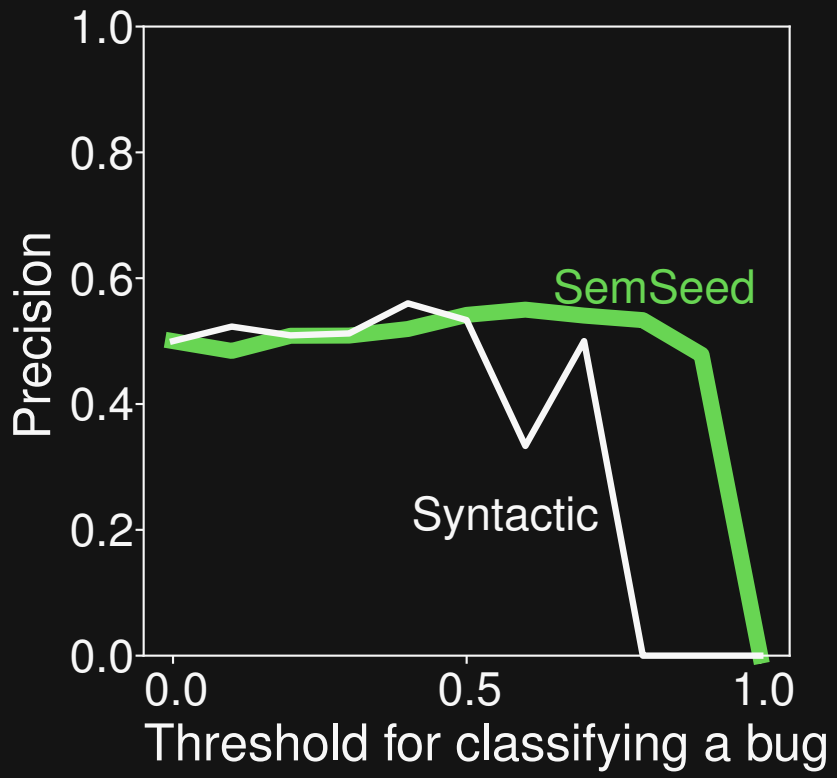


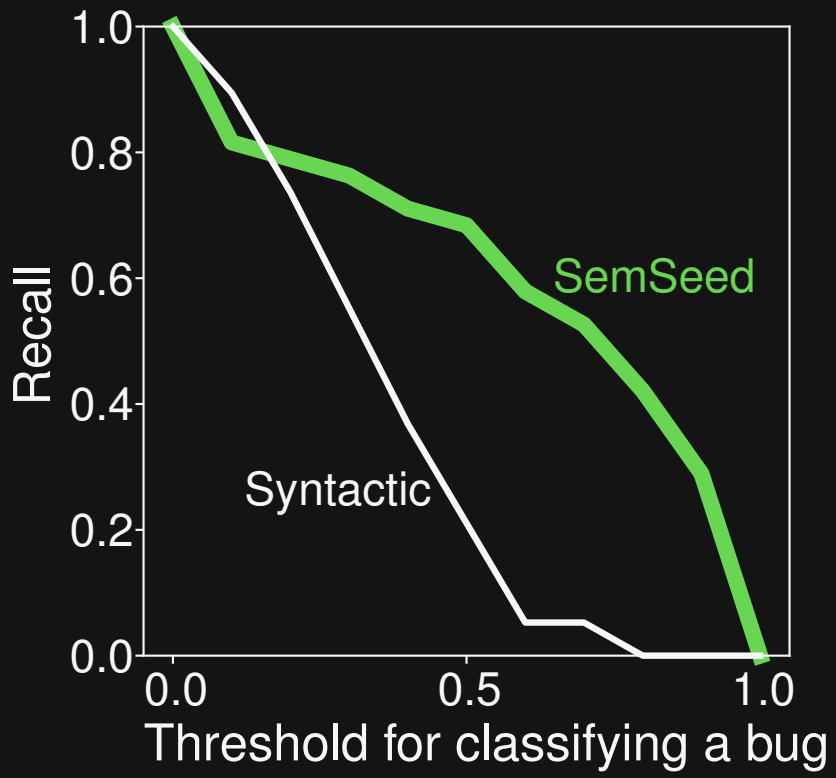
Incorrect assignment bugs, corpus of 120K files.

Artificial seeds 1.1M bugs, SemSeed seeds 248K bugs.

Learning Bug Detectors

Use seeded bugs as training data for learning bug detectors [DeepBugs, OOPSLA'18]

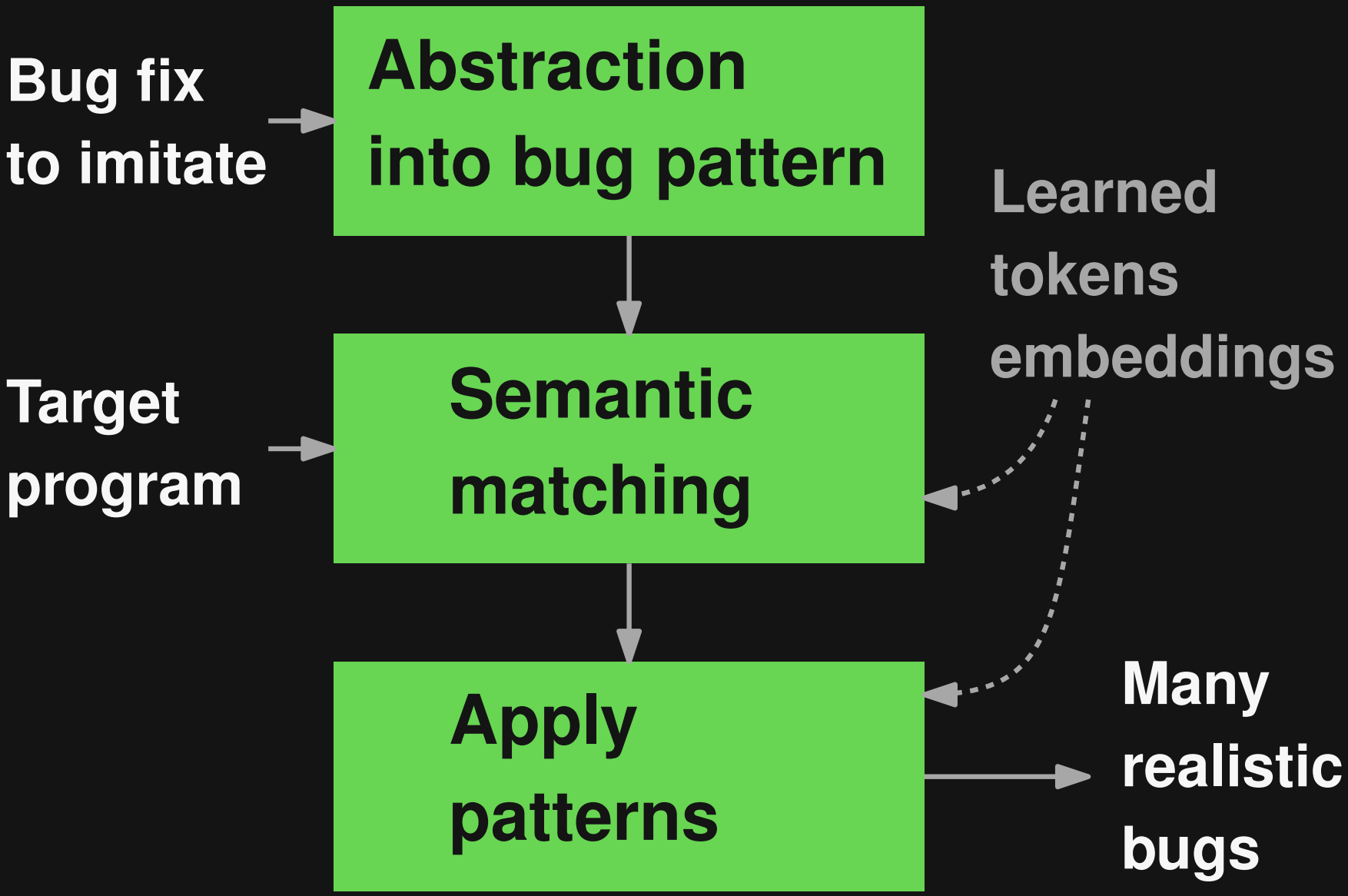




Incorrect assignment bugs, corpus of 120K files.

Artificial seeds 1.1M bugs, SemSeed seeds 248K bugs.

Summary


Bug fix
to imitate
Abstraction
into bug pattern
Learned
tokens
embeddings
Target
program
Semantic
matching
Apply
patterns
Many
realistic
bugs

Comparison: Mutation Operators

- **Comparison with 23 mutation operators in Mutandis** [ICST'13]
  - SemSeed supports 16/23 mutation operators
  - 98.2% of SemSeed-generated bugs go beyond the 23 operators
- **Complementary to traditional mutation operators**