Superion: Grammar-Aware Greybox Fuzzing

2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Junjie Wang∗, Bihuan Chen†, Lei Wei∗, Yang Liu∗‡

∗School of Computer Science and Engineering, Nanyang Technological University, Singapore

†School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China

‡College of Information Science, Zhejiang Sci-Tech University, China

Shared by @mrdrivingduck

Programs that Process Structured Inputs

Feature - process inputs in stages:

- Syntax parsing
- Semantic checking
- Application execution

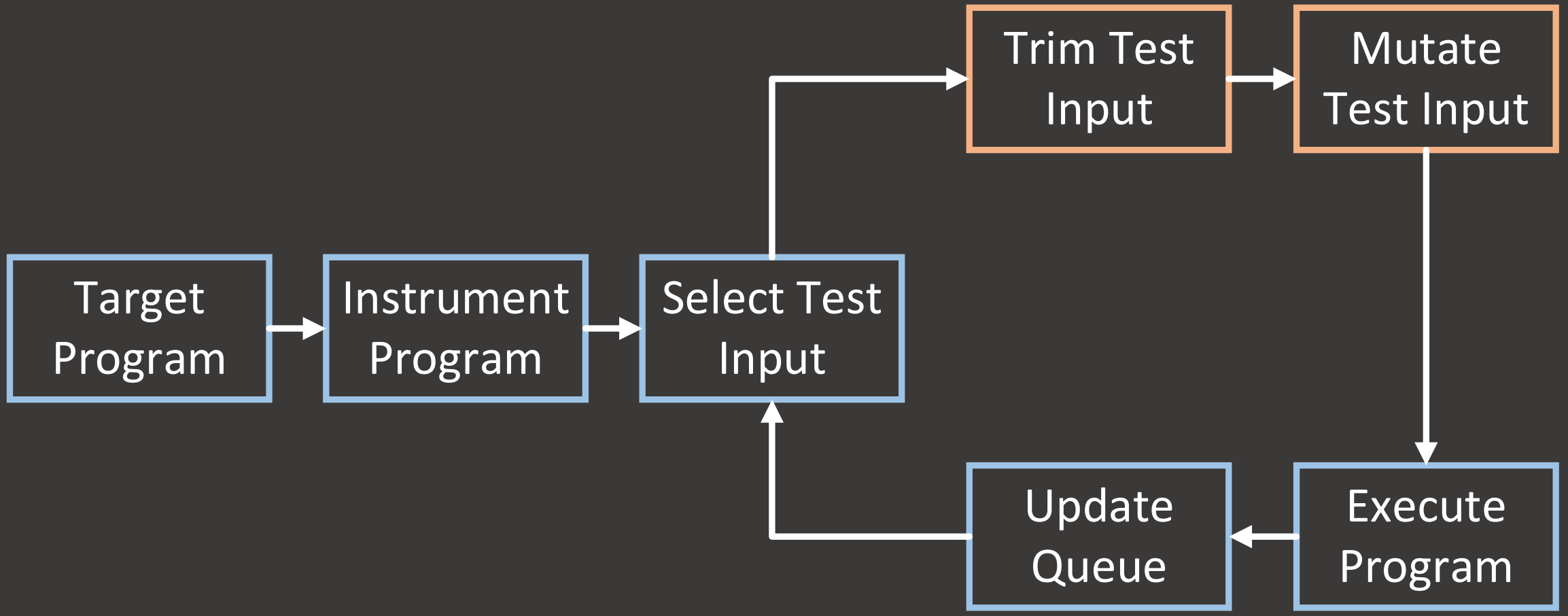
e.g.

compilers, language interpreter (JavaScript),

browsers (CSS, XML), databases (SQL) …

AFL ?

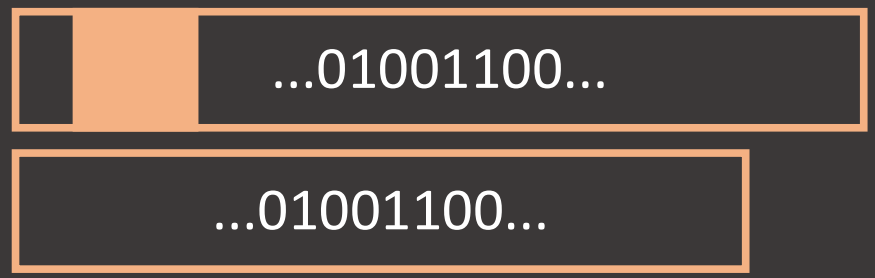
The General Workflow of AFL



Trimming

- Less search space for mutation

- For unstructured binary input



Same Coverage

- For highly-structured input

Syntax Error !!!

Mutation

- For unstructured binary input (byte flip)

...00**01101010**00...

...00**10010101**00...

- For highly-structured input

<a**>**emmm

<a**?**emmm

Syntax Error !!!

Motivation

AFL: **Grammar-unaware**

Superion: **Grammar-aware**

- Grammar-aware trimming

  Minimize input while keeping syntax correct

- Grammar-aware mutation

  Trigger new coverage while keeping syntax correct

Grammar-Aware Trimming Strategy

AFL: split the input into **length/n** chunks

- n from 16 to 1024

libplist (an XML engine)

- Same coverage
- Destroyed grammar

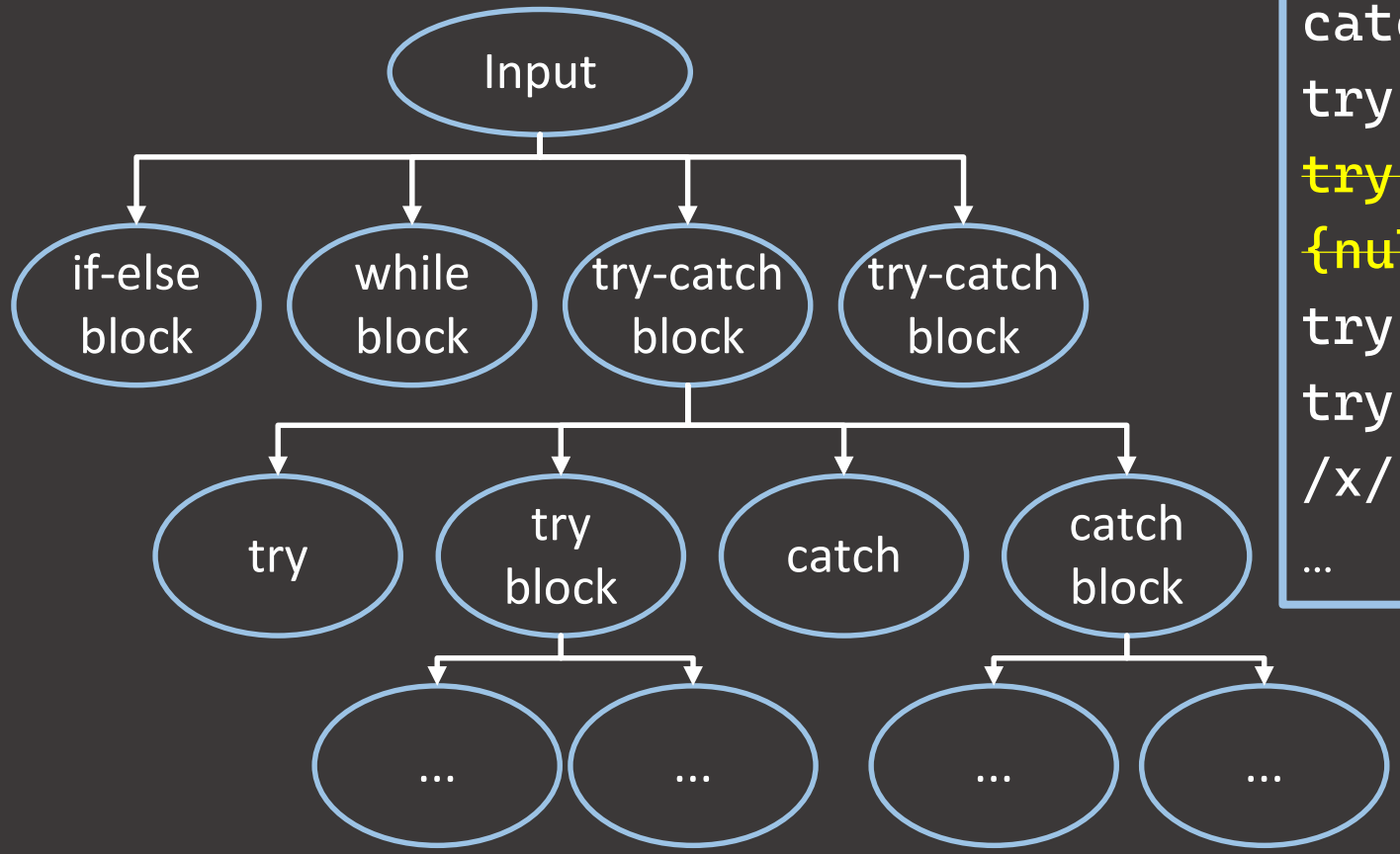
```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
    <key>Some ASCII string</key>
    <string></string>
    <data>
    </data>
</dict>
</plist>
```

Grammar-Aware Trimming Strategy

Superion

- Input + Grammar → AST (if error, use AFL trimming)
- Trim a subtree n from tree
- Test coverage
- Trimming until no sub-tree can be trimmed

Grammar-Aware Trimming Strategy

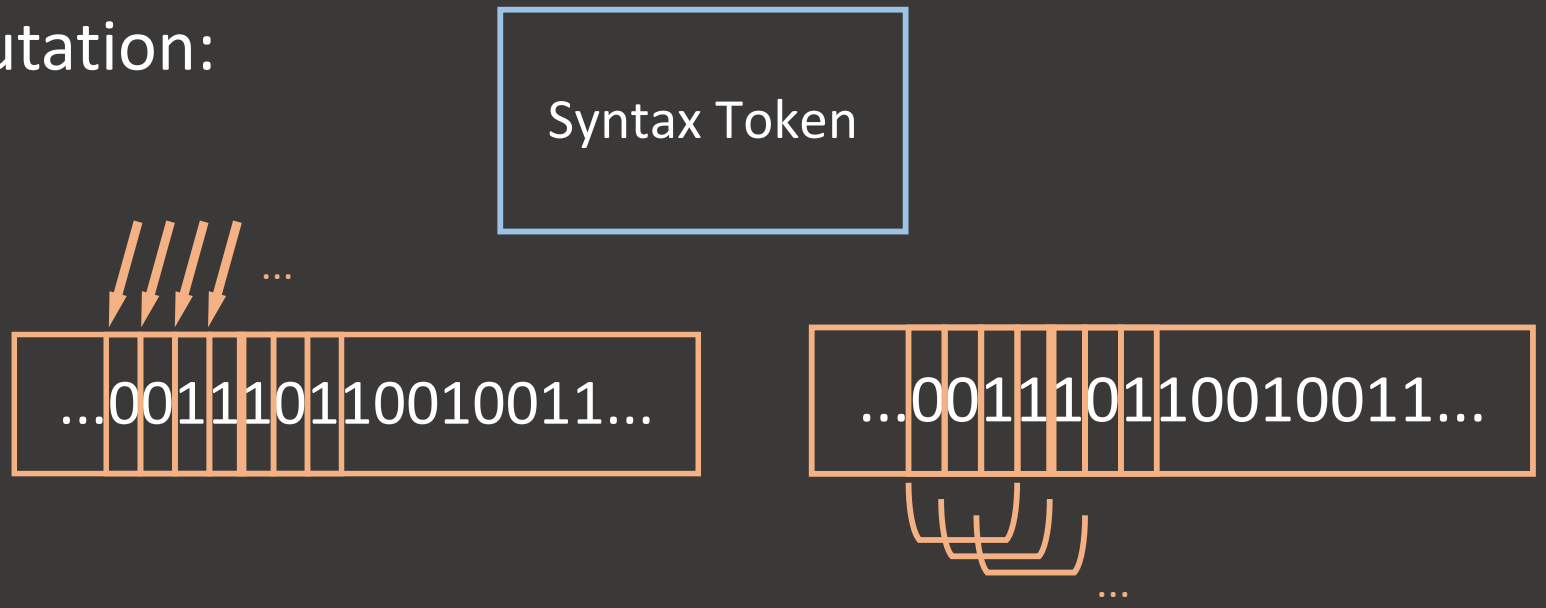



```
…
try{eval("\nbreak M;\n");}
catch(ex){}
try{…}catch(ex){}
try{eval("if(\"\"){}else if(x4)
{null;}");} catch(ex){}
try{eval("{}");} catch(ex){}
try{eval("for(var x = x in x –
/x/ ){}");} catch(ex){}
…
```

Grammar-Aware Mutation Strategies: Enhanced Dictionary-Based Mutation

AFL's dictionary-based mutation:

- Insert
- Overwrite



Grammar-Aware Mutation Strategies:
Enhanced Dictionary-Based Mutation

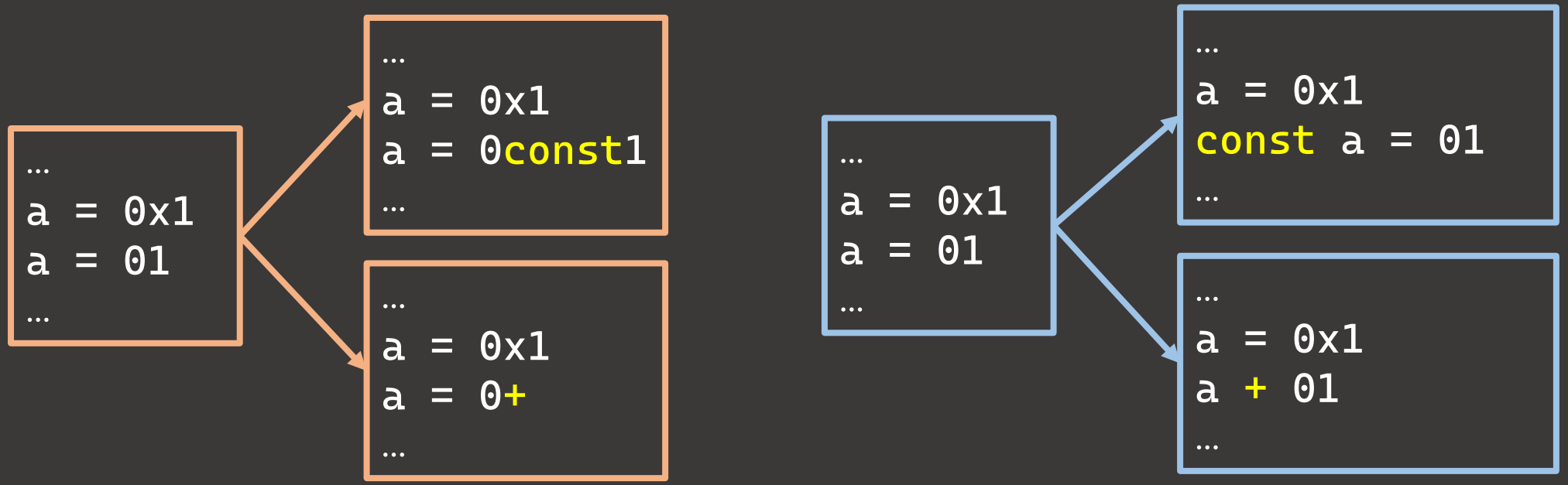
Superion’s dictionary-based mutation:

- Locate the token boundaries

  Alphabets + digits (variable/function names, reserved keywords)
- Insert only to each boundaries
- Overwrite only tokens between two boundaries

Grammar-Aware Mutation Strategies:
Enhanced Dictionary-Based Mutation

- Greatly decreases the number of token insertions/overwrites
- Maintains the structure of mutated test inputs
- Effectiveness + efficiency



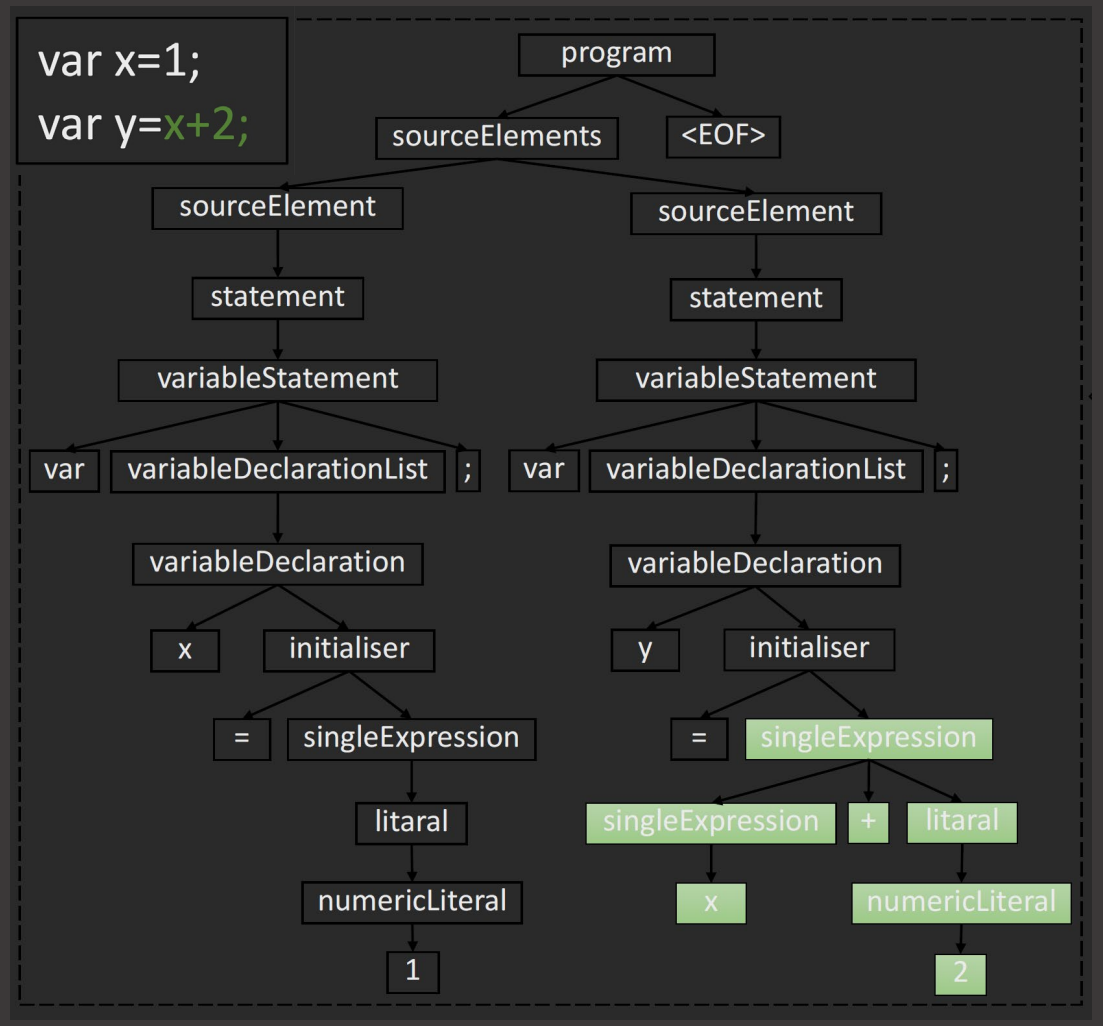
Grammar-Aware Mutation Strategies:
Tree-Based Mutation

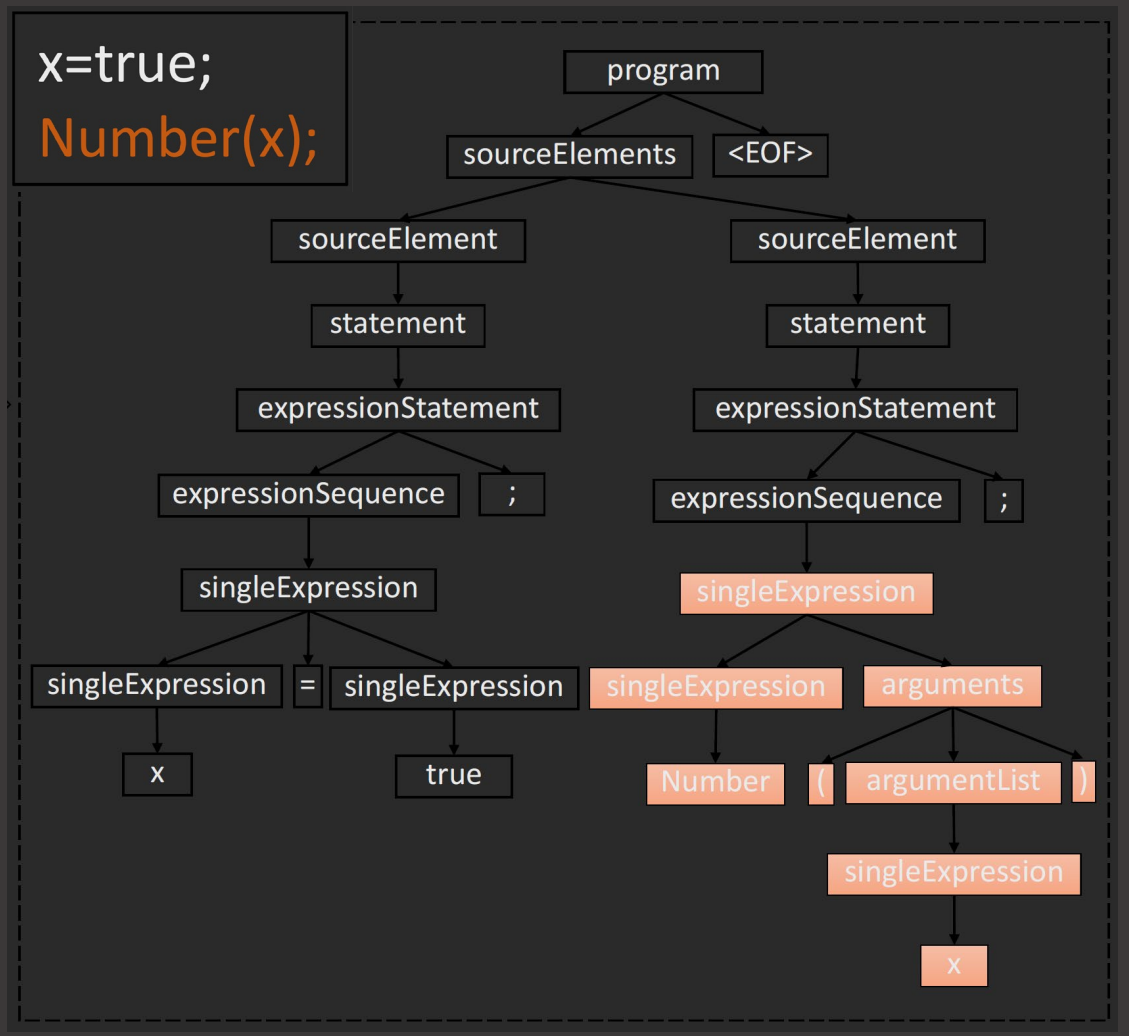
- Treeify the test input I into AST
- Treeify another random input into AST
- Extract all sub-tree into a set S
- Replace each sub-tree in I with each sub-tree in S

Grammar-Aware Mutation Strategies:

Tree-Based Mutation

```
var x=1;
var y=Number(x);
```





Evaluation

- Extending AFL with 3,372 lines of C/C++ code
- 1 open-source XML engine
- 3 open-source JavaScript engine

Evaluation - How is the bug-finding capability of Superion?

- 34 new bugs
- 22 new vulnerabilities with 19 CVE identifiers assigned

Evaluation - How is the code coverage of Superion?

- Improve 16.7% in line coverage
- Improve 8.8% in function coverage

Evaluation - How effective is our grammar-aware trimming?

- Relatively low trimming ratio
- Improve the grammar validity ratio after trimming

Evaluation - How effective is our grammar-aware mutation?

- Effective in generating test inputs that trigger new coverage

Evaluation - What is the performance overhead of Superion?

- Introduces additional overhead due to our grammar-aware tree-based mutation strategy
- Still acceptable considering the improved bug-finding capability and code coverage

Limitation

- Superion needs a user-provided grammar
- Proprietary grammars
- Undocumented extensions to standard grammars

Grimoire: Synthesizing Structure while Fuzzing (USENIX Security 2019)

- No program input specifications
- Slower than grammar-aware fuzzers

1. Input generalization
2. Input mutation

Grimoire: Synthesizing Structure while Fuzzing (USENIX Security 2019)

Input generalization

- Fragmentation
  - Byte chunks
  - Splitter
  - () [] {} <>
- General input + tokens

```
if(x>1) then x=3 end

if(x>1)□then □end

if(x>1)
then
end
```

Grimoire: Synthesizing Structure while Fuzzing (USENIX Security 2019)

Input Mutation

- Input extension
- Recursive replacement
- String replacement

```
□ if(x>1)□then □end □
□ pprint □

if(x>1) then pprint x end
if(key>1) then pprint key end
```