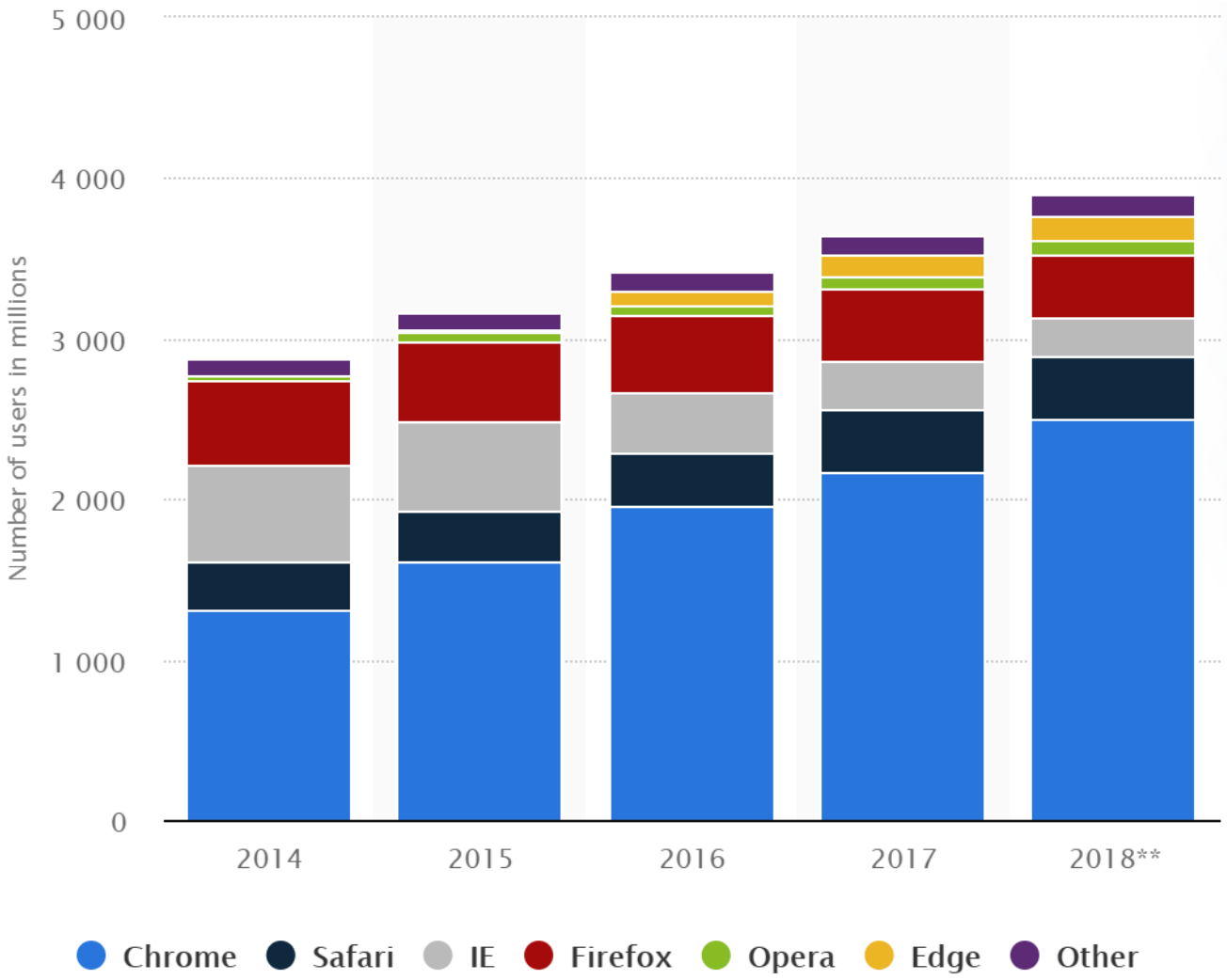
Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer

Suyoung Lee, HyungSeok Han, Sang Kil Cha, **Sooel Son**

Graduate School of Information Security (GSIS) , KAIST

KAIST

Popularity of Web Browsers

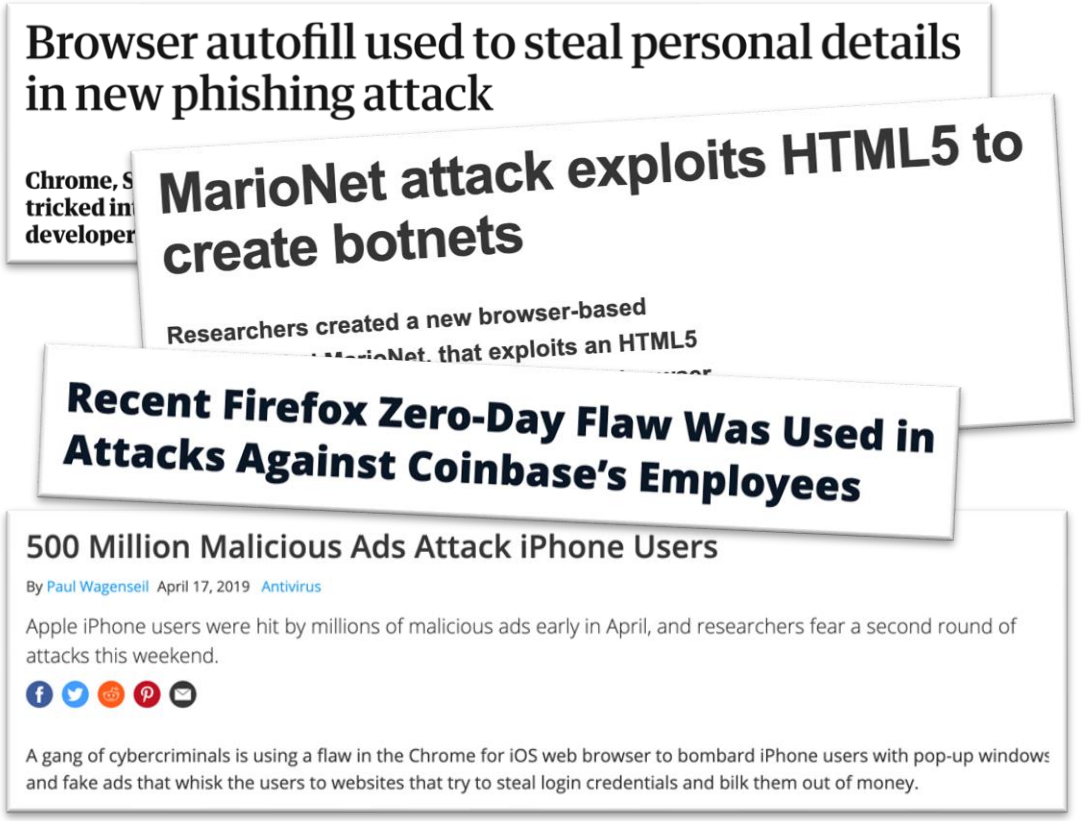


4 billion users

Source: https://www.statista.com/statistics/543218/worldwide-internet-users-by-browser/

Vulnerable Web Browsers

- Browser-based cyber threats



Browser autofill used to steal personal details in new phishing attack

Chrome, S… tricked in… developer…

MarioNet attack exploits HTML5 to create botnets

Researchers created a new browser-based … that exploits an HTML5 …

Recent Firefox Zero-Day Flaw Was Used in Attacks Against Coinbase's Employees

500 Million Malicious Ads Attack iPhone Users

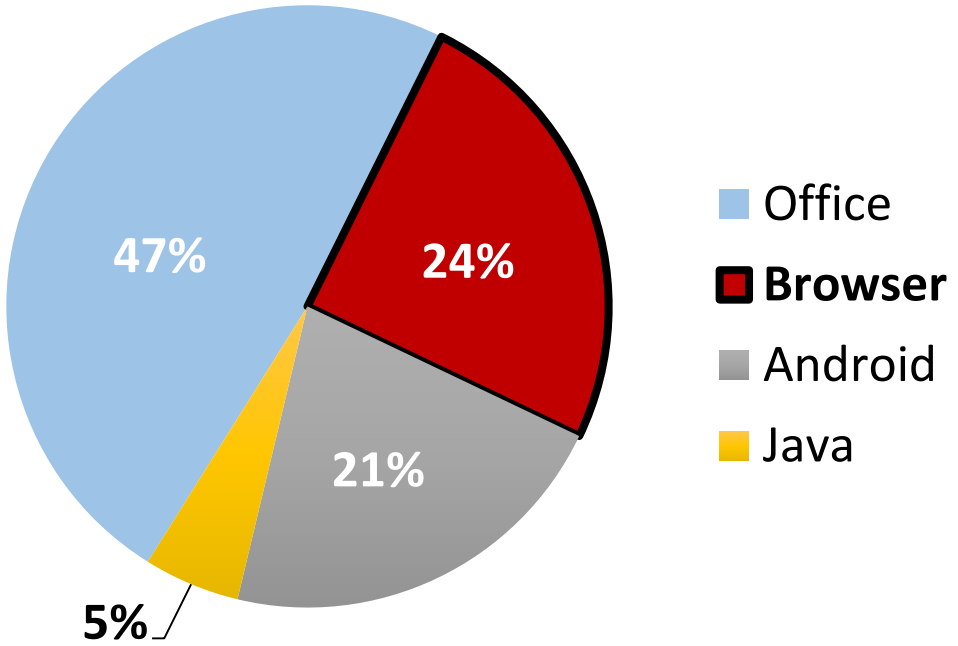
By Paul Wagenseil April 17, 2019 Antivirus

Apple iPhone users were hit by millions of malicious ads early in April, and researchers fear a second round of attacks this weekend.

A gang of cybercriminals is using a flaw in the Chrome for iOS web browser to bombard iPhone users with pop-up windows and fake ads that whisk the users to websites that try to steal login credentials and bilk them out of money.

https://www.theguardian.com/technology/2017/jan/10/browser-autofill-used-to-steal-personal-details-in-new-phising-attack-chrome-safari
https://searchsecurity.techtarget.com/news/252458522/MarioNet-attack-exploits-HTML5-to-create-botnets
https://cointelegraph.com/news/recent-firefoxs-zero-day-flaw-was-used-in-attacks-against-coinbases-employees
https://www.tomsguide.com/us/ios-malvertising-barrage,news-29880.html

- Most exploited applications in 2018



https://www.kaspersky.com/about/press-releases/2018_microsoft-office-exploits

KAIST

JS Engine Vulnerabilities

JS engine vulnerabilities pose **critical security threat!**

JS Engine Fuzzing

- **JS Engines**

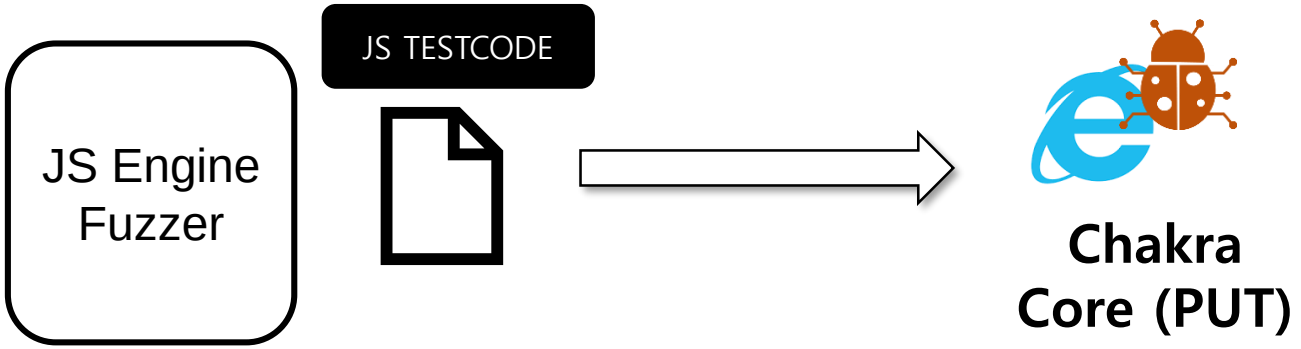
V8

Spider Monkey

Chakra Core

JavaScript Core

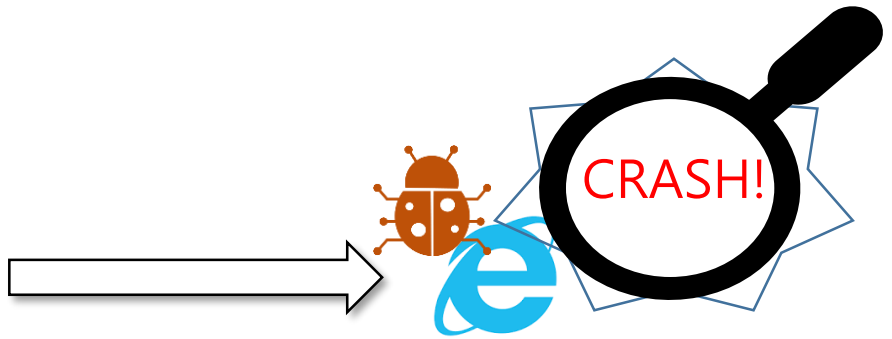
- **Fuzzing (Fuzz Testing)**
  - An automated software testing that involve providing invalid or unexpected input to a program under testing (PUT).



How Can We (Fuzzer) Generate Test Input?

Proof of Concept (PoC)
for CVE-2017-8586

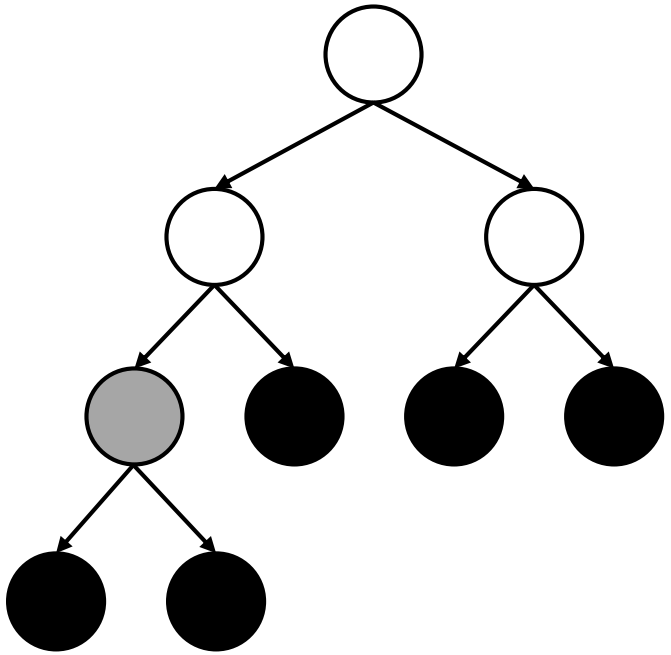
```
var v1 = {
    'a': function () {}
}
var v2 = 'a';
(function () {
    try {
    } catch ([v0 = (v1[v2].__proto__(1, 'b'))]) {
        var v0 = 4;
    }
    v0++;
})();
```

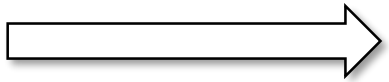


Chakra
Core (PUT)

Previous Work

Abstract Syntax Tree (AST)

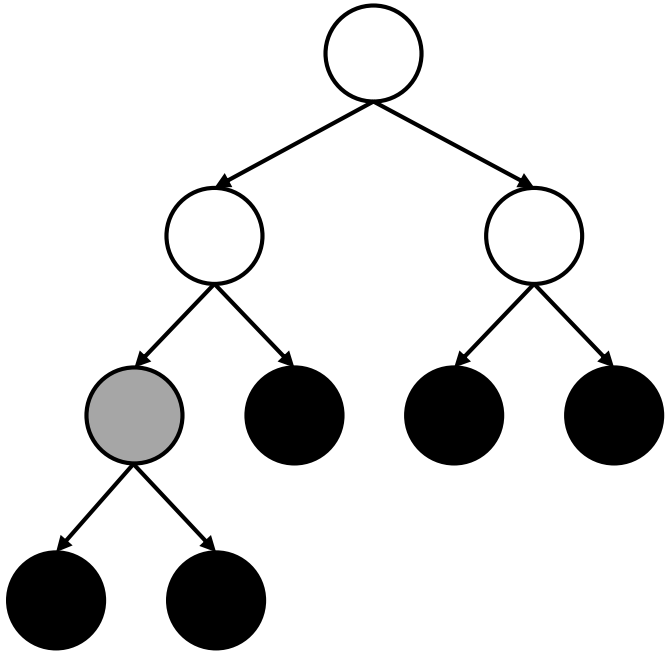




```
var v1 = {
    'a': function () {}
}
var v2 = 'a';
(function () {
    try {
    } catch ([v0 = (v1[v2].__proto__(1, 'b'))]) {
        var v0 = 4;
    }
    v0++;
})();
```

Previous Work

Abstract Syntax Tree (AST)



1. Mutation-based fuzzers

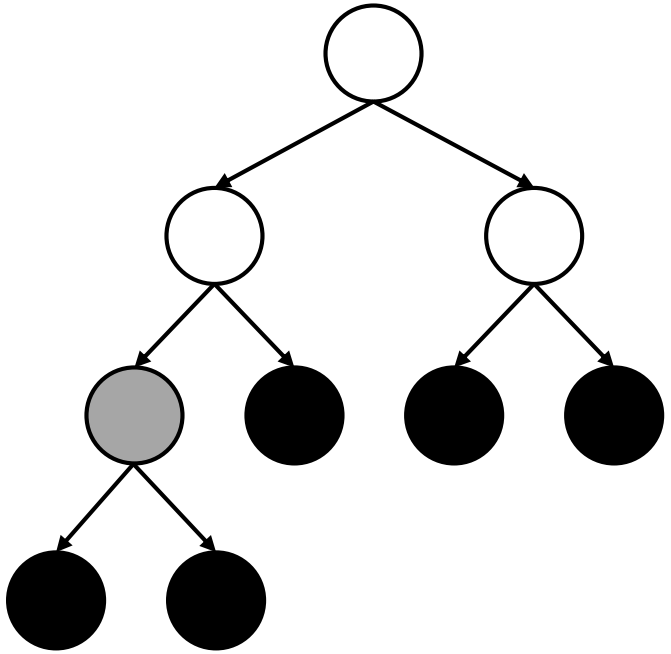
- LangFuzz, IFuzzer, and GramFuzz
- Combining **AST subtrees** extracted from JS seeds

2. Generation-based fuzzers

- jsfunfuzz
- Applying **JS grammar rules** from scratch

Previous Work

Abstract Syntax Tree (AST)



1. Mutation-based fuzzers

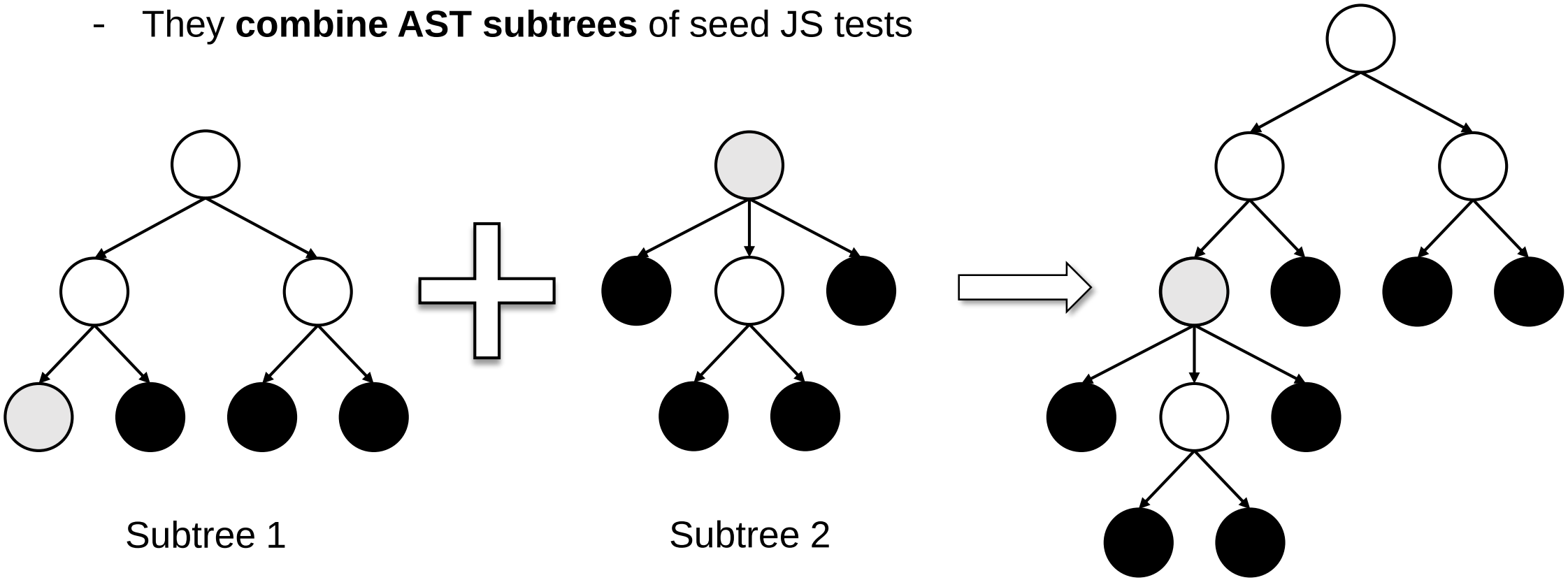
- LangFuzz, IFuzzer, and GramFuzz
- Combining **AST subtrees** extracted from JS seeds

2. Generation-based fuzzers

- jsfunfuzz
- Applying **JS grammar rules** from scratch

Previous Work: Mutation-based JS fuzzers

- **LangFuzz, IFuzzer, and GramFuzz**
  - They **combine AST subtrees** of seed JS tests



Relationship between Building Blocks

Current AST

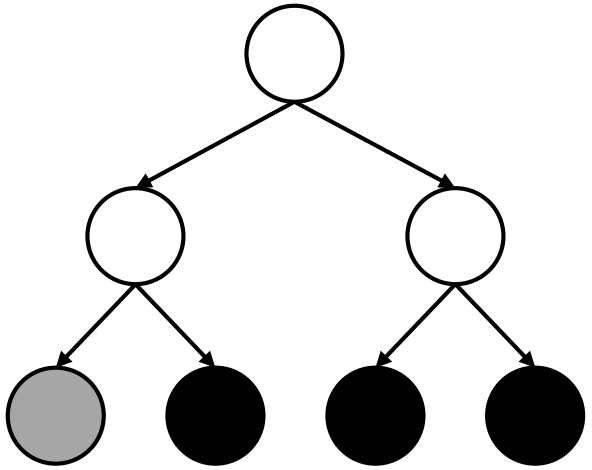
A set of applicable AST subtrees

None of the existing fuzzers consider **their relationships!**

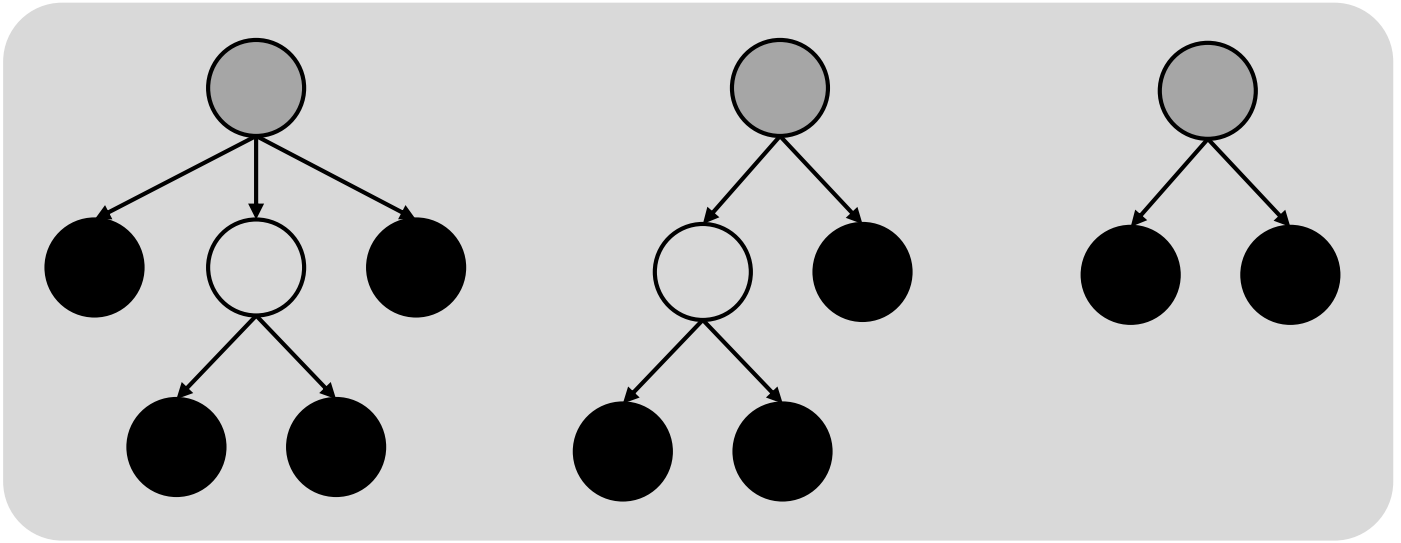
Which combination is **more likely to trigger** JS engine bugs?

Motivational Question

Current AST



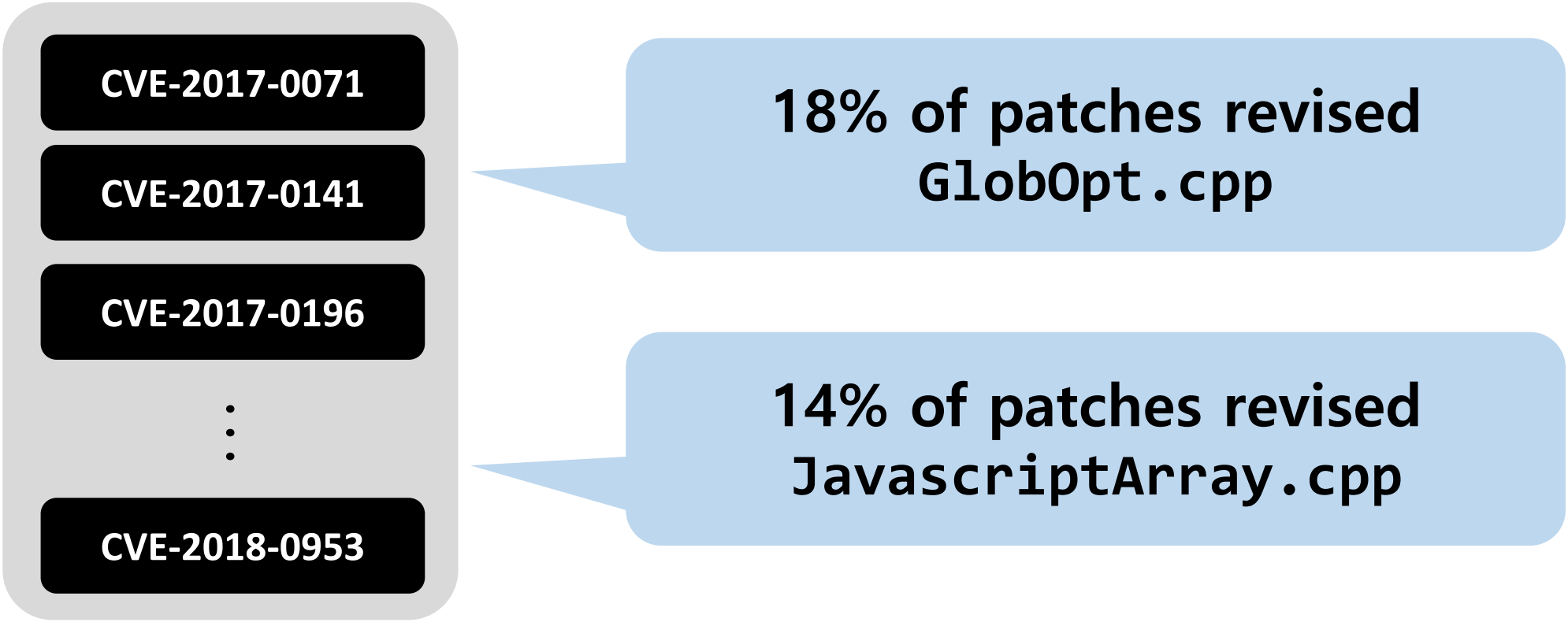
A set of applicable AST subtrees



Are there any **similar patterns** between bug-triggering JS code?

Study on JS Engine Vulnerabilities

- **Analyzed patches of 50 CVEs assigned to ChakraCore**

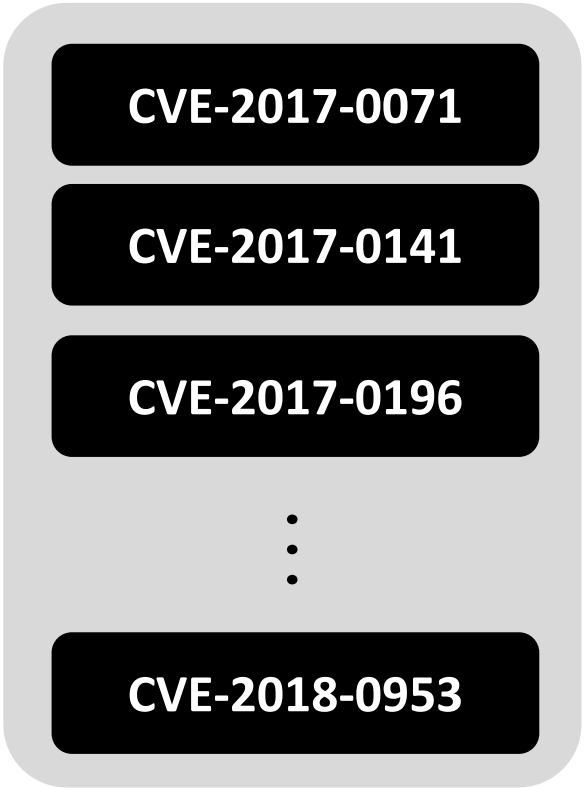


Patches of 50 CVEs

KAIST

Study on JS Engine Vulnerabilities

- **Analyzed patches of 50 CVEs assigned to ChakraCore**



Patches of 50 CVEs

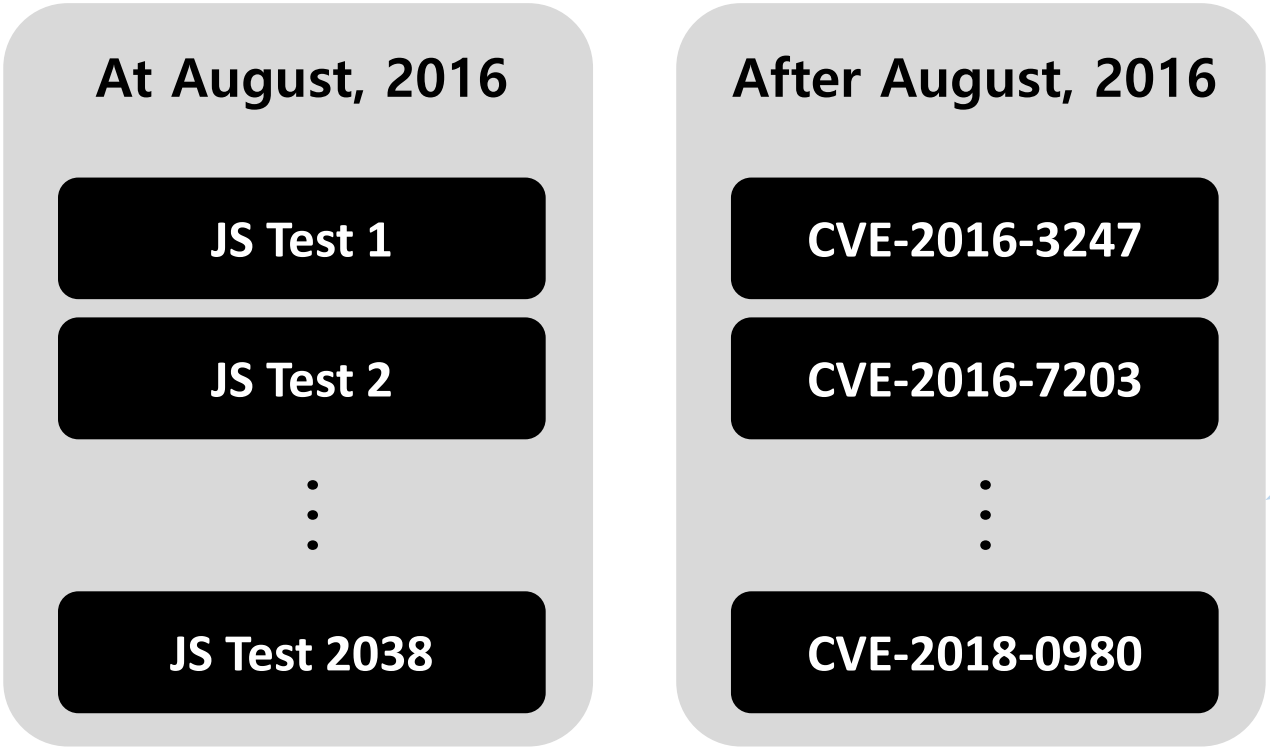
18% are related to global optimization

14% are related to JavaScript array

KAIST

Study on JS Engine Vulnerabilities

- **Compared AST subtrees from two sets**



2038 JS tests from ChakraCore repo

67 PoCs triggering ChakraCore CVEs

Over 95% subtrees from PoCs exist in regression tests

Our Goal

1. To leverage the **functionality** of JS regression tests

   - Mutation based approach

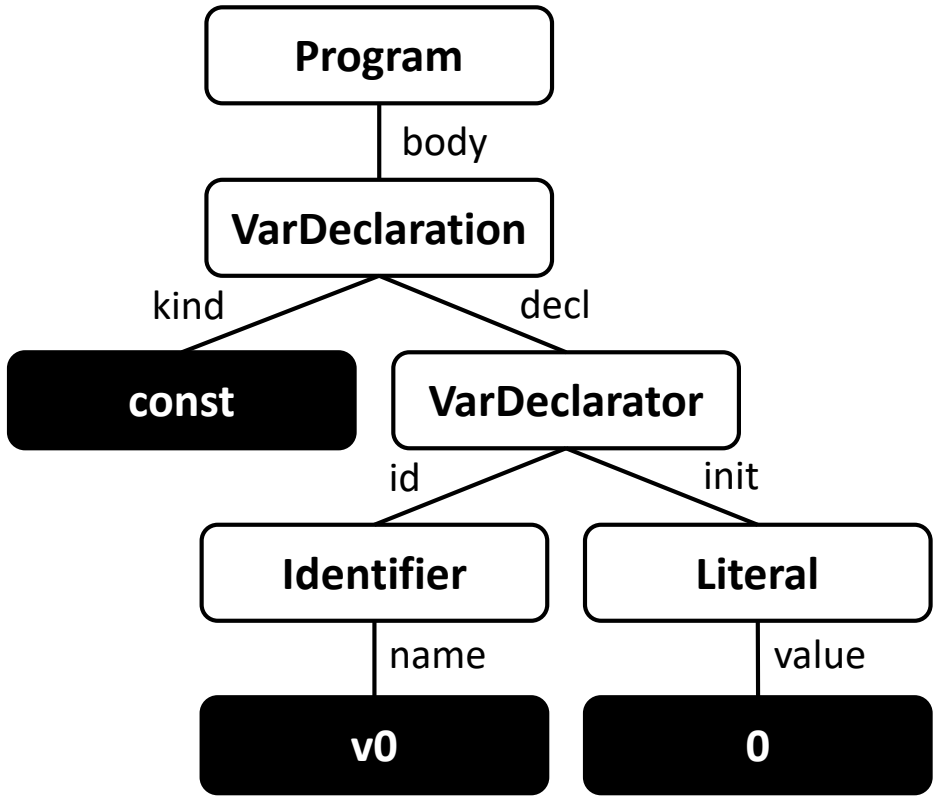
2. To learn the **relationship** of AST subtrees

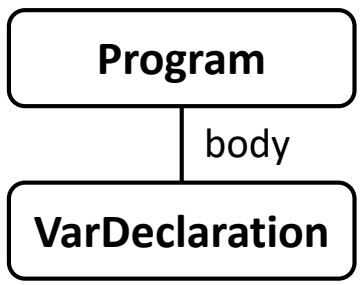
   - Modeling the relationship between AST subtrees

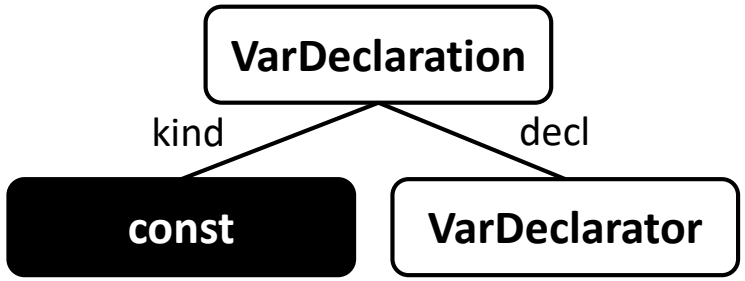
KAIST

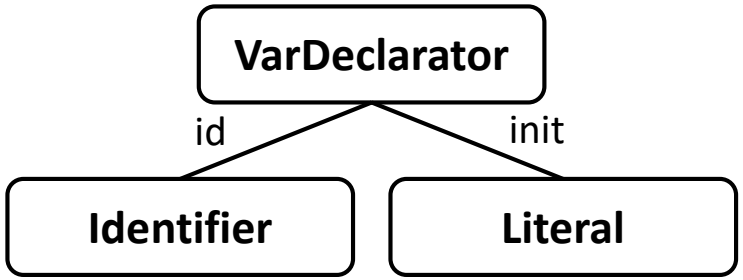
Our Building Block – Fragments

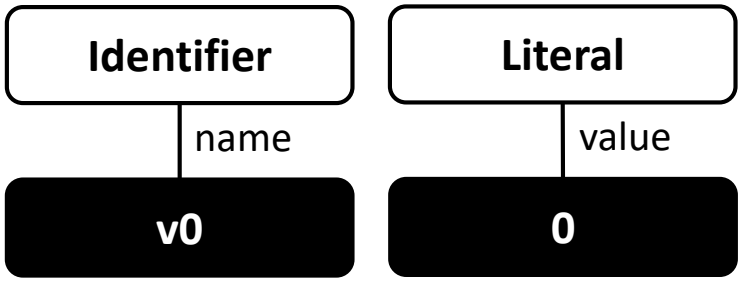
```
const v0 = 0;
```





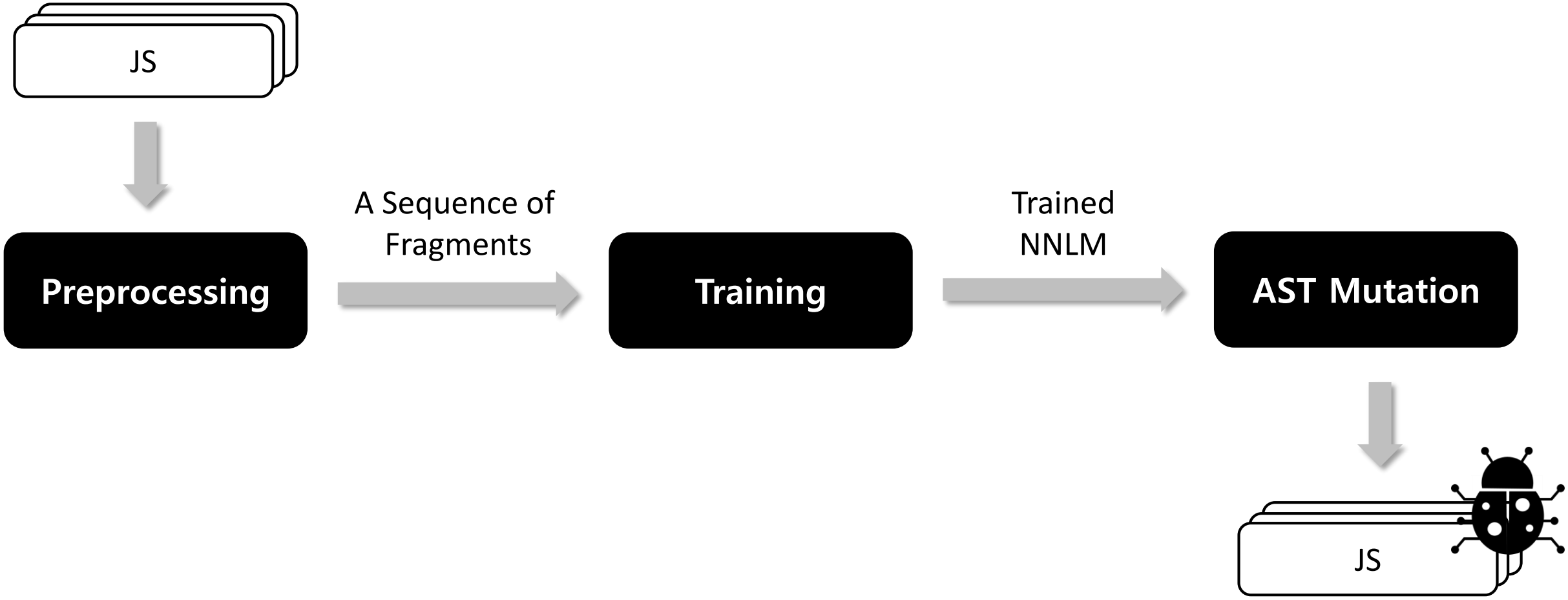






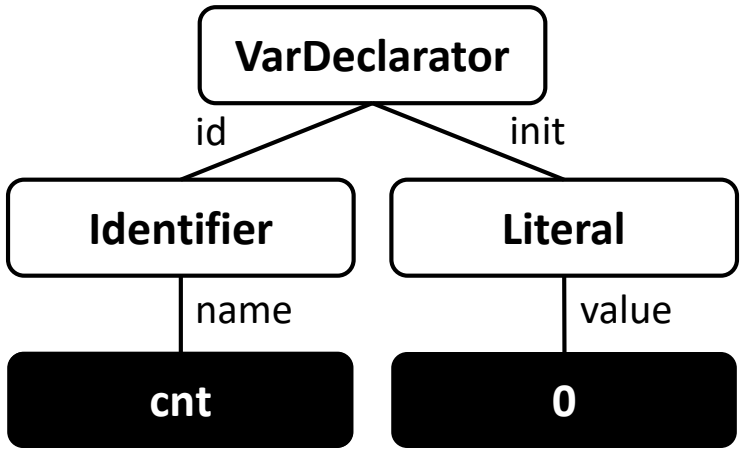
A fragment is a subtree of depth 1

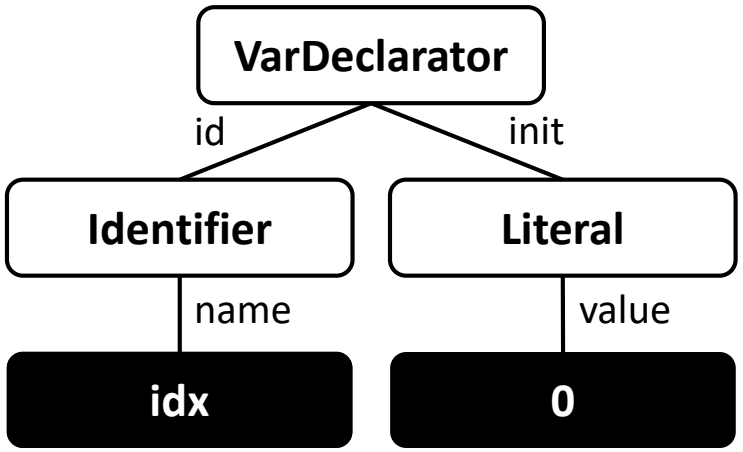
Montage Overview



Preprocessing – Normalization

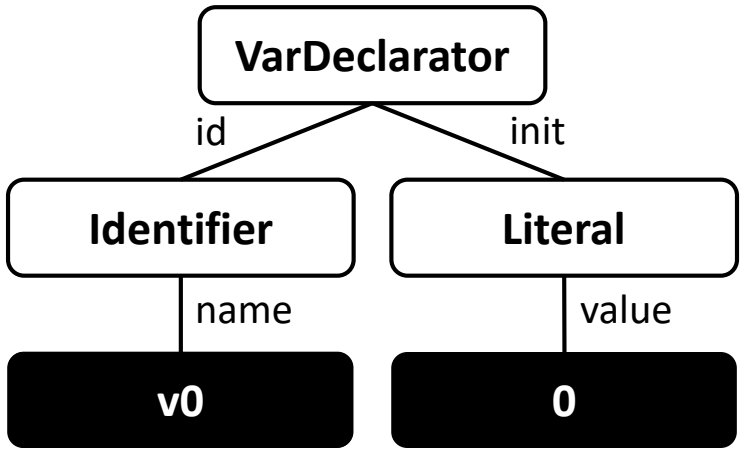
- **Normalizing IDs**
  - To decrease the # of unique fragments, **rename IDs**

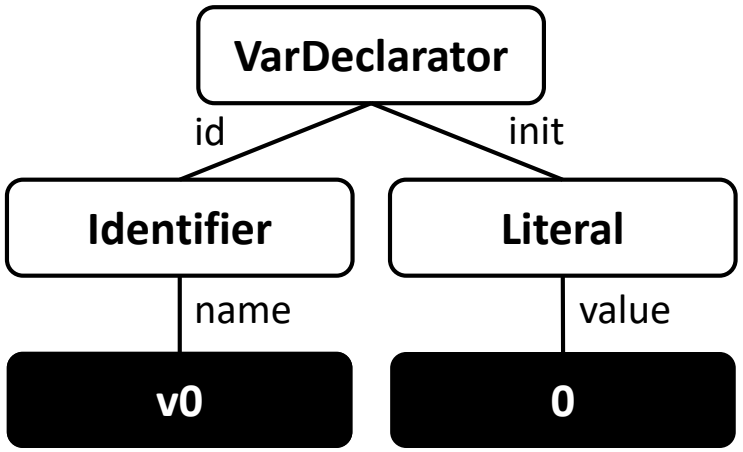




Preprocessing – Normalization

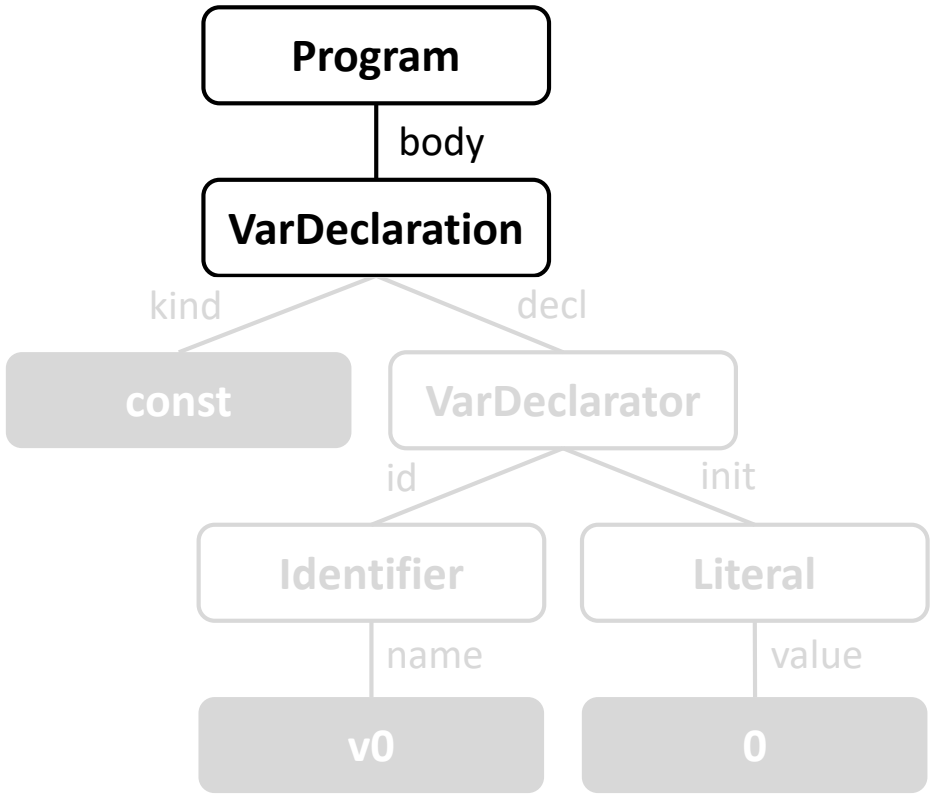
- **Normalizing IDs**
  - To decrease the # of unique fragments, **rename IDs**

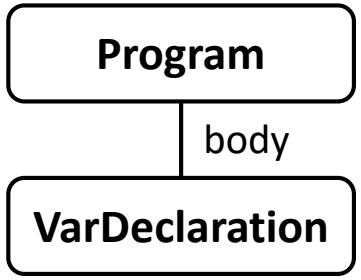




Preprocessing – Fragmentation

- **Fragmentation**



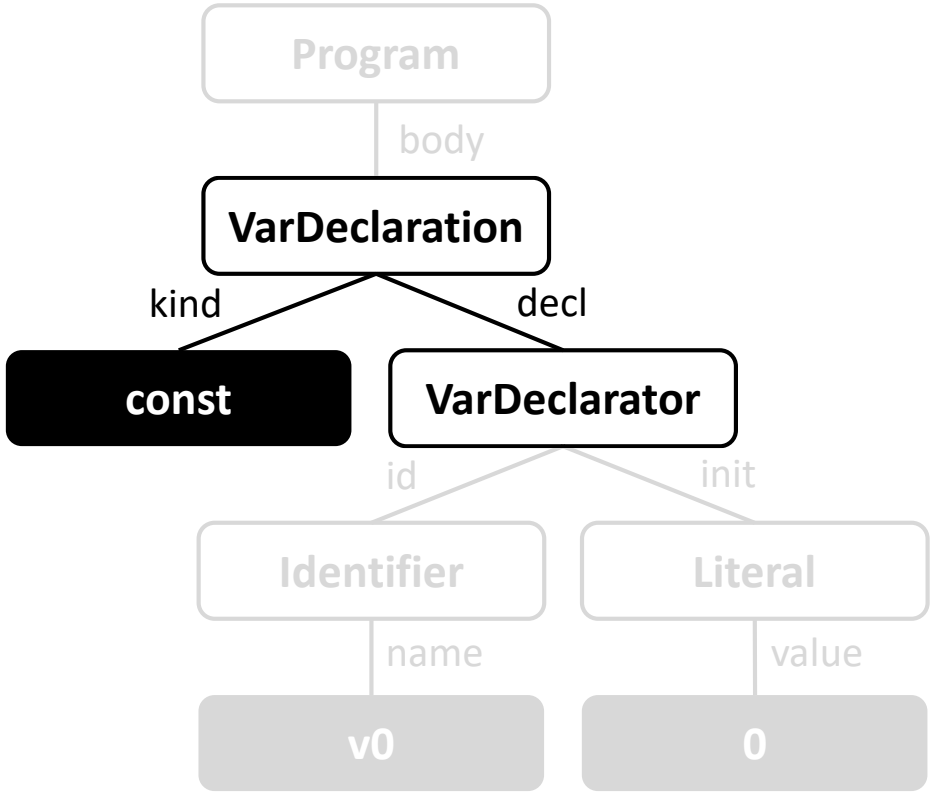


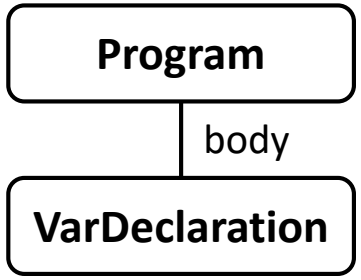
Normalized AST

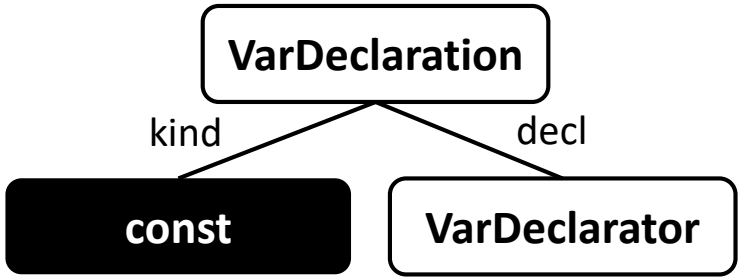
A sequence of fragments

Preprocessing – Fragmentation

- **Fragmentation**





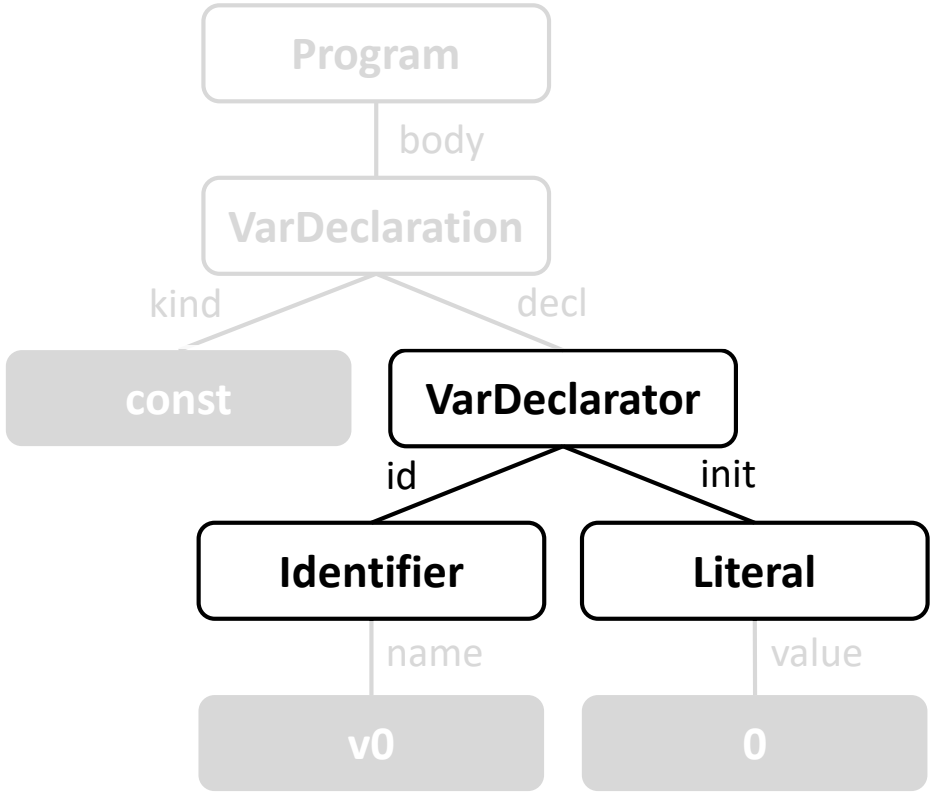


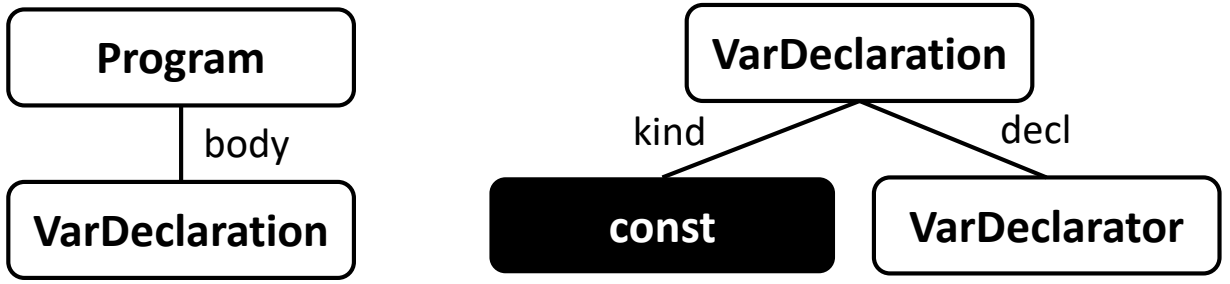
Normalized AST

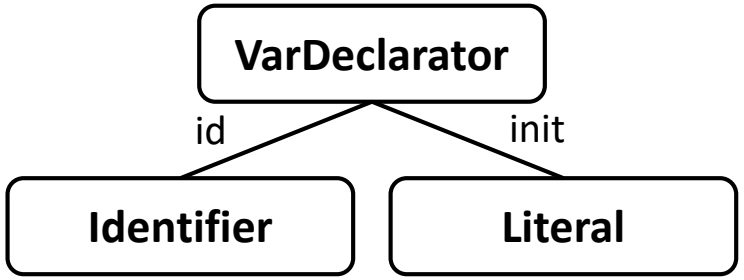
A sequence of fragments

Preprocessing – Fragmentation

- **Fragmentation**





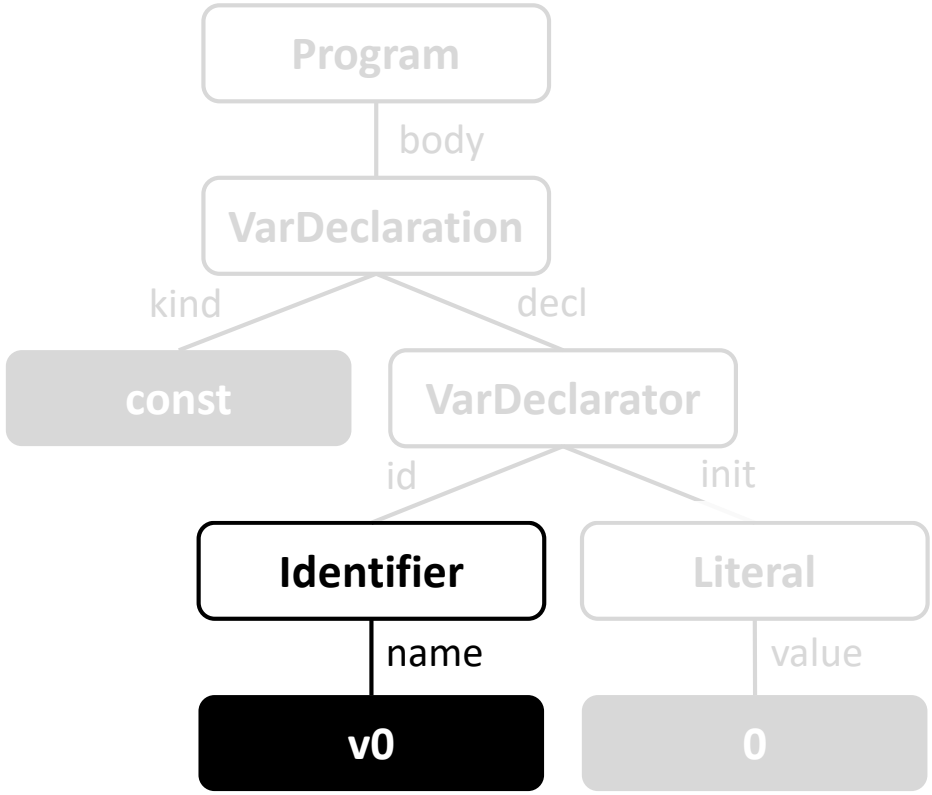


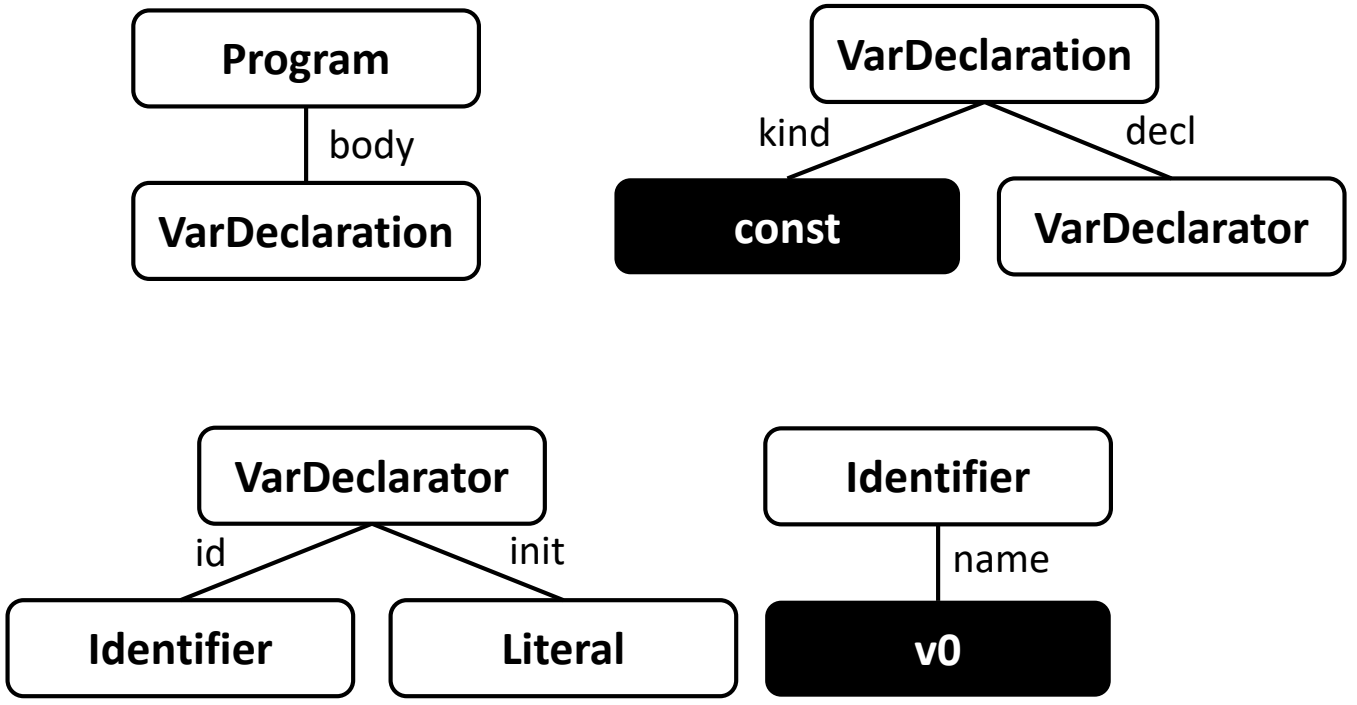
Normalized AST

A sequence of fragments

Preprocessing – Fragmentation

- **Fragmentation**



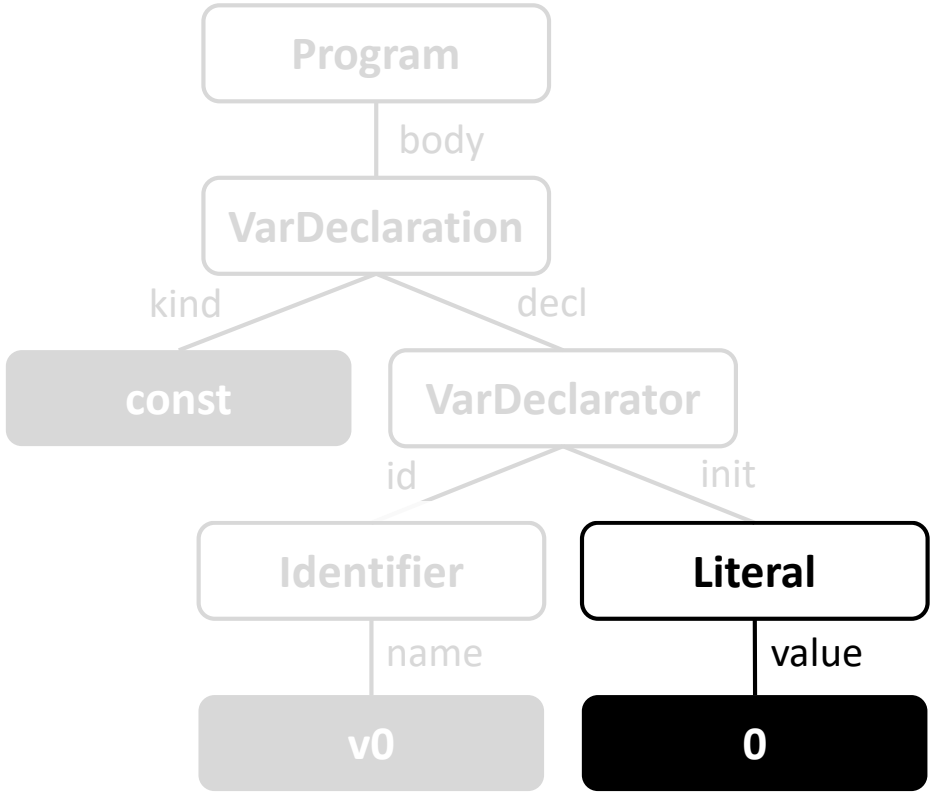


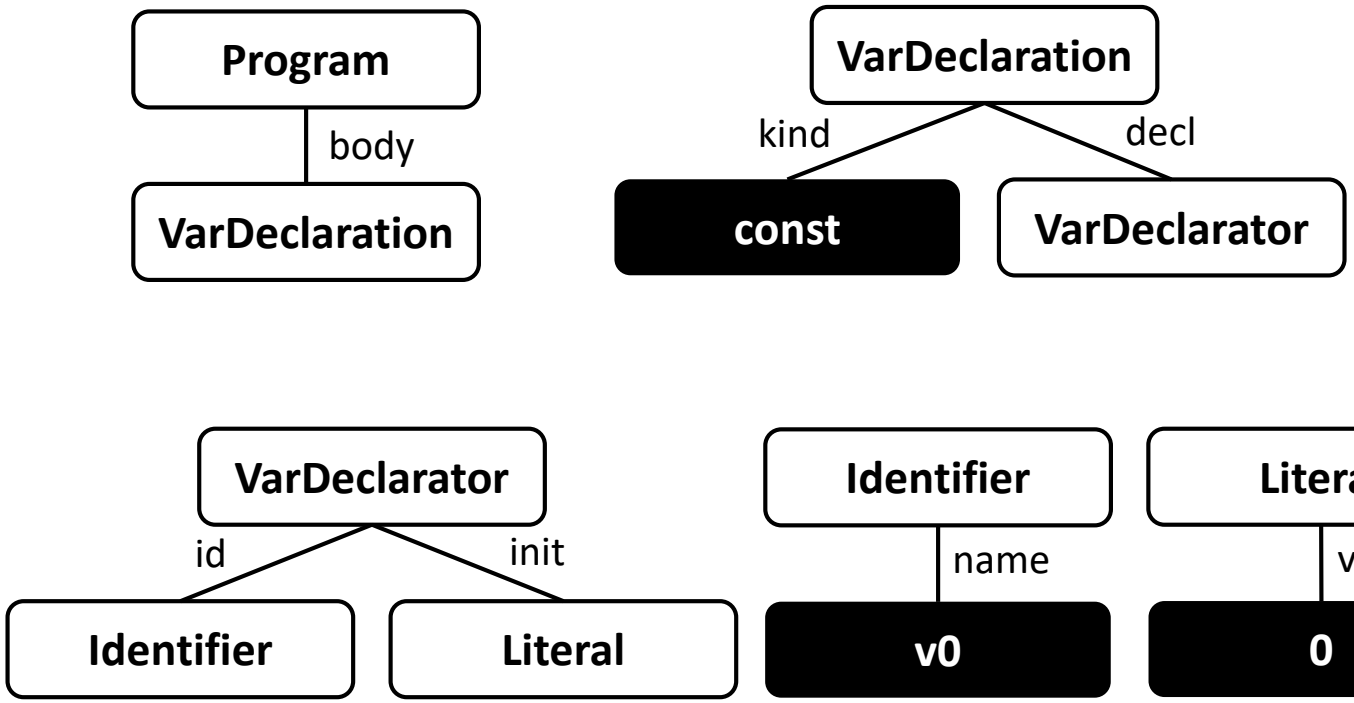
Normalized AST

A sequence of fragments

Preprocessing – Fragmentation

- **Fragmentation**



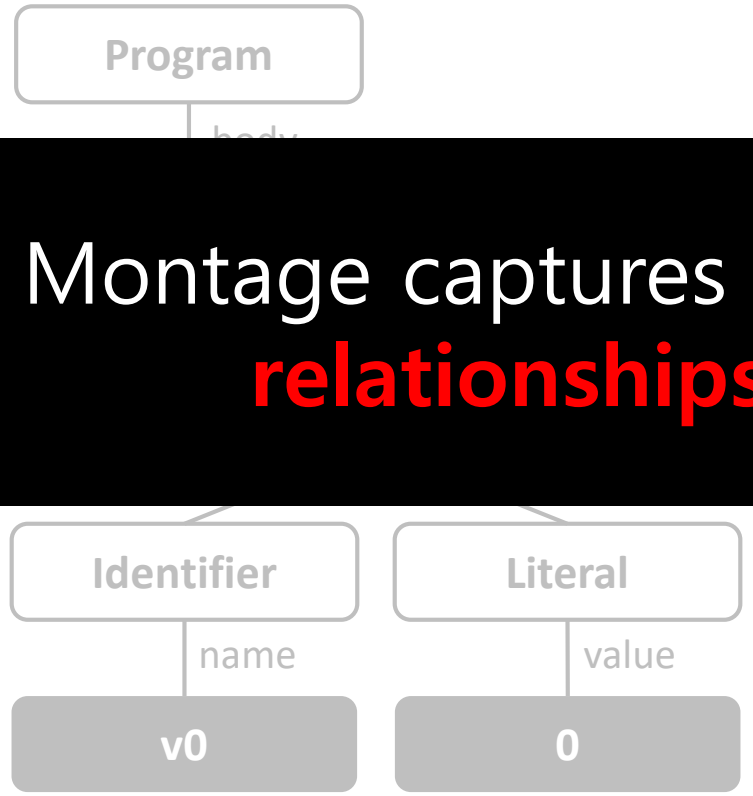


Normalized AST

A sequence of fragments

Preprocessing – Fragmentation

- Fragmentation



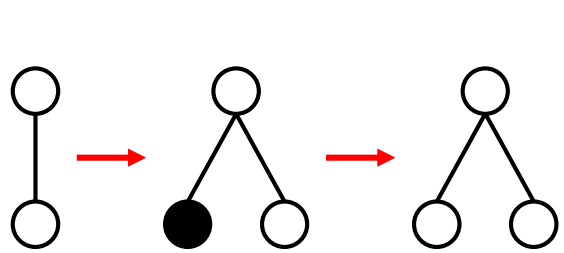
Normalized AST

Montage captures **the global compositional relationships** between fragments

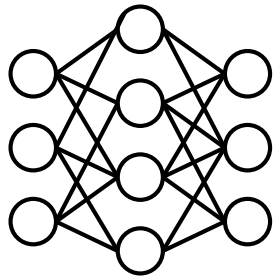
A sequence of fragments

Training Objectives

1. **Given a sequence, predict the distribution of next fragments**



A sequence of preceding fragments

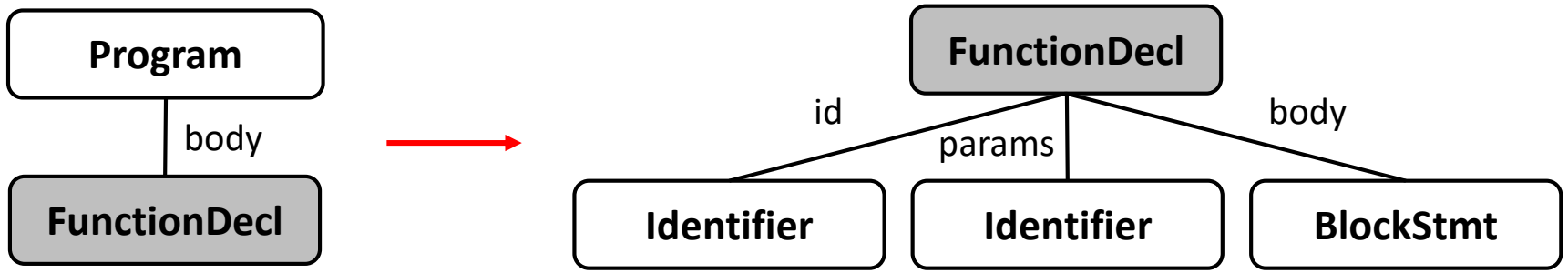


LSTM model

Probability distribution of next fragment

Training Objectives

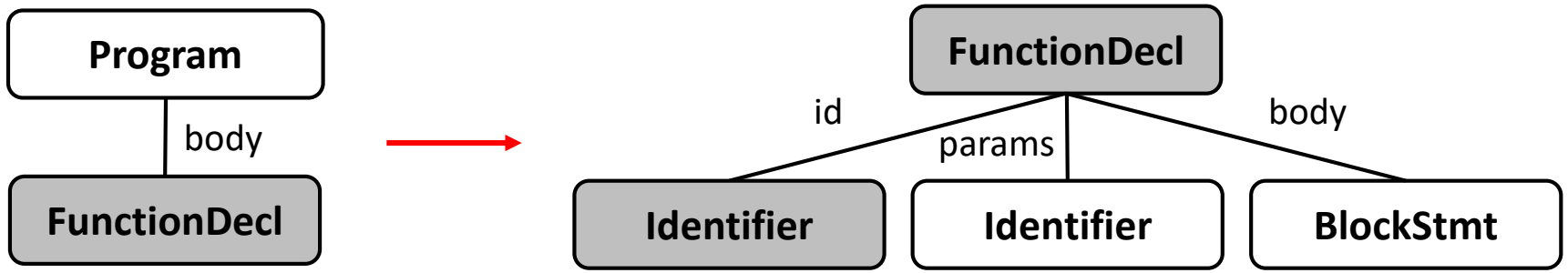
2. Prioritize the fragments that have a correct type



A given sequence of preceding fragments

Training Objectives

2. Prioritize the fragments that have a correct type

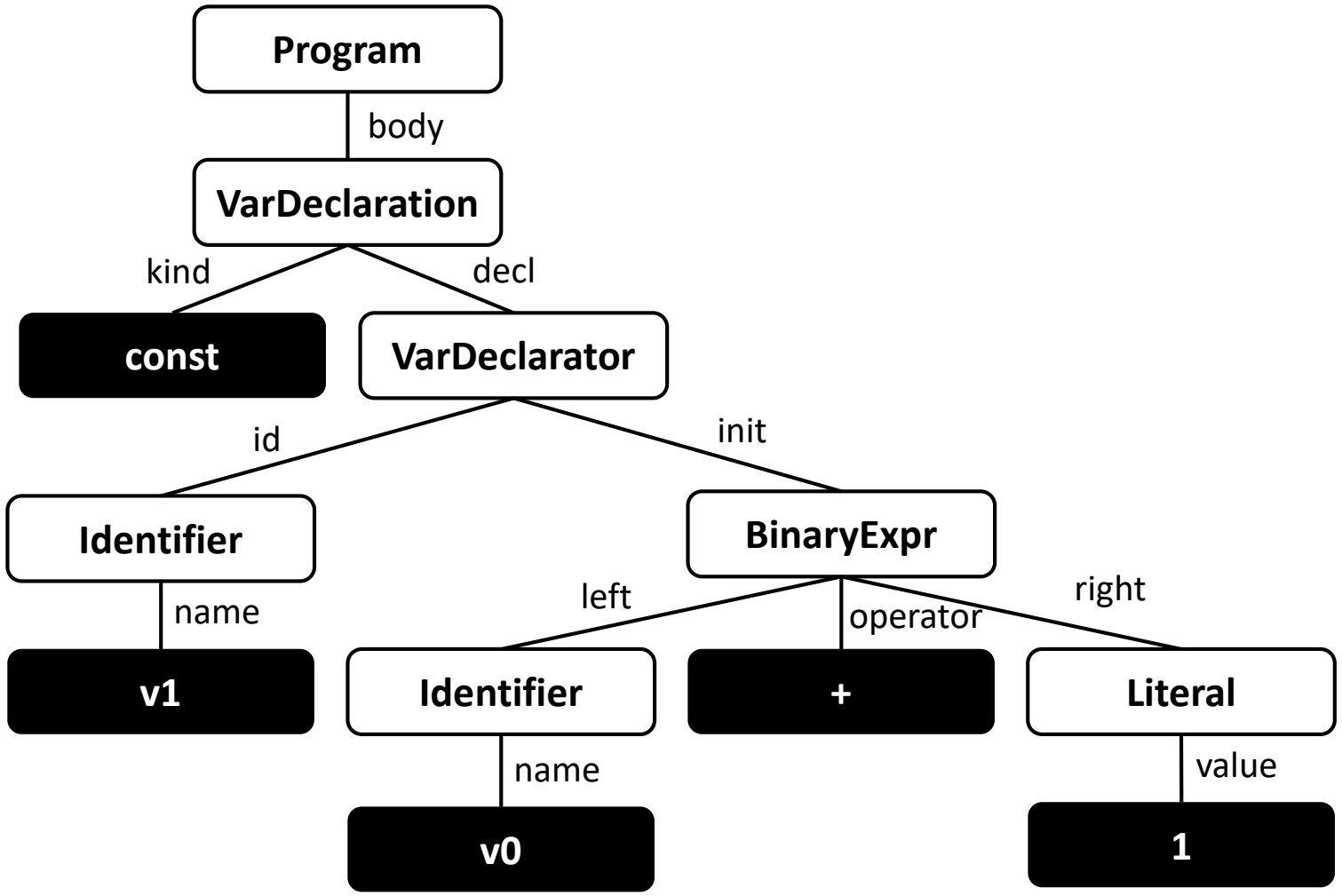


A given sequence of preceding fragments

To be syntactically correct, the root of the next fragment should be **Identifier!**

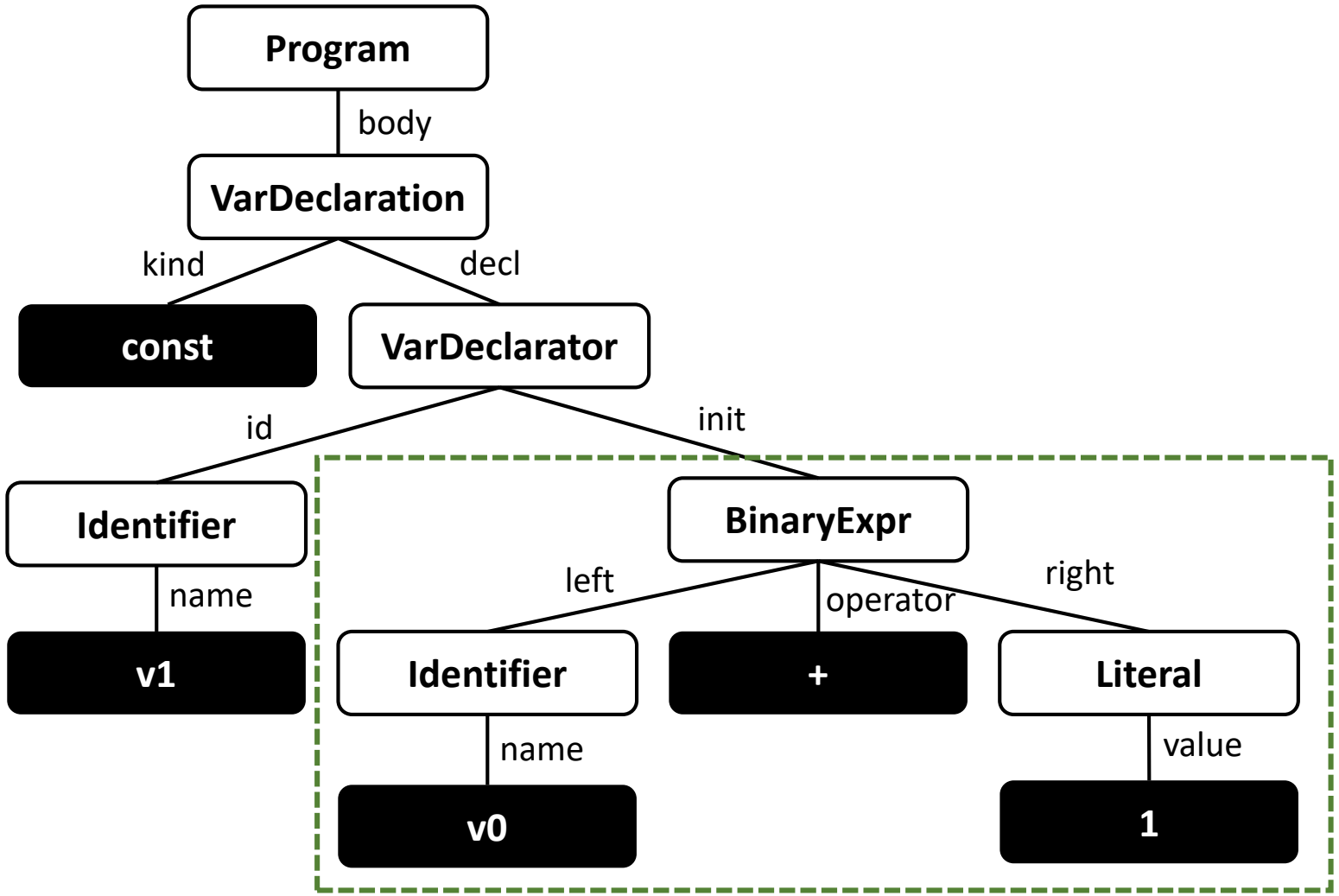
AST Mutation

```
const v1 = v0 + 1;
```



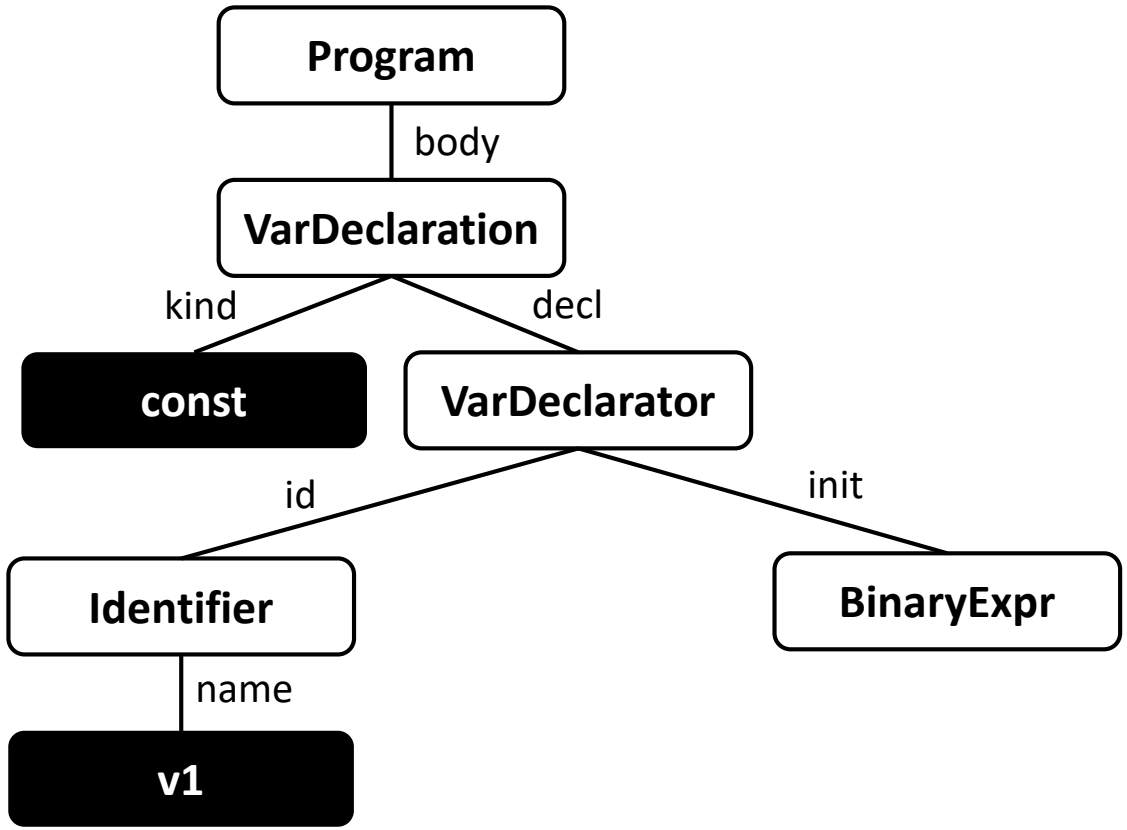
Seed AST

AST Mutation

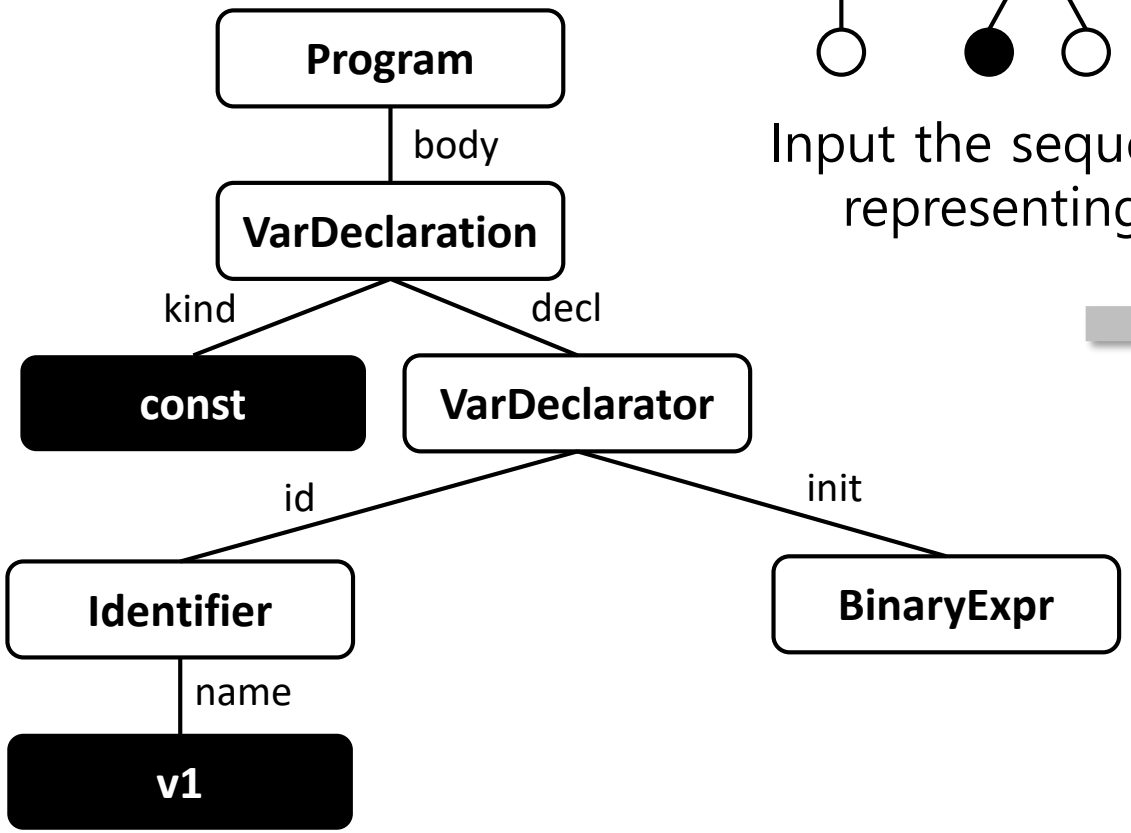


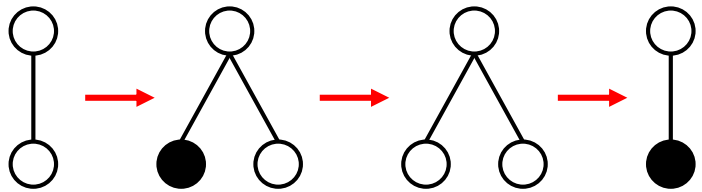
Remove a subtree

AST Mutation

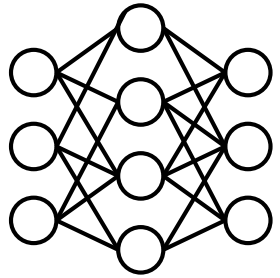


AST Mutation



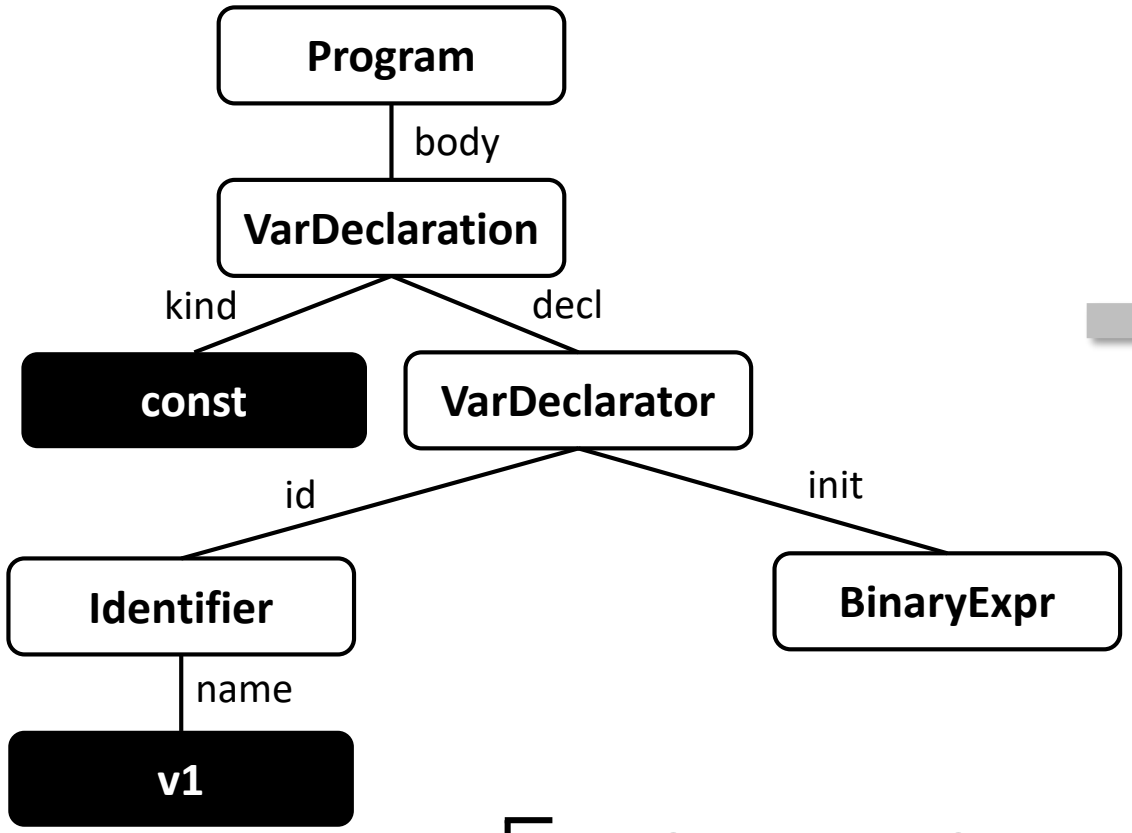


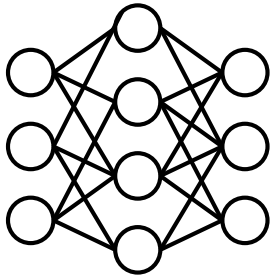
Input the sequence of 4 fragments representing the current AST



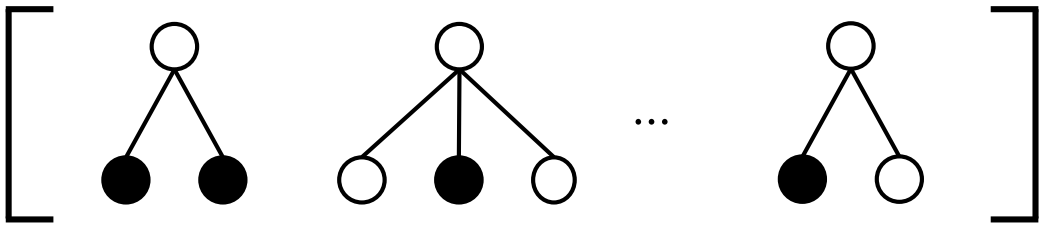
Trained LSTM model

AST Mutation





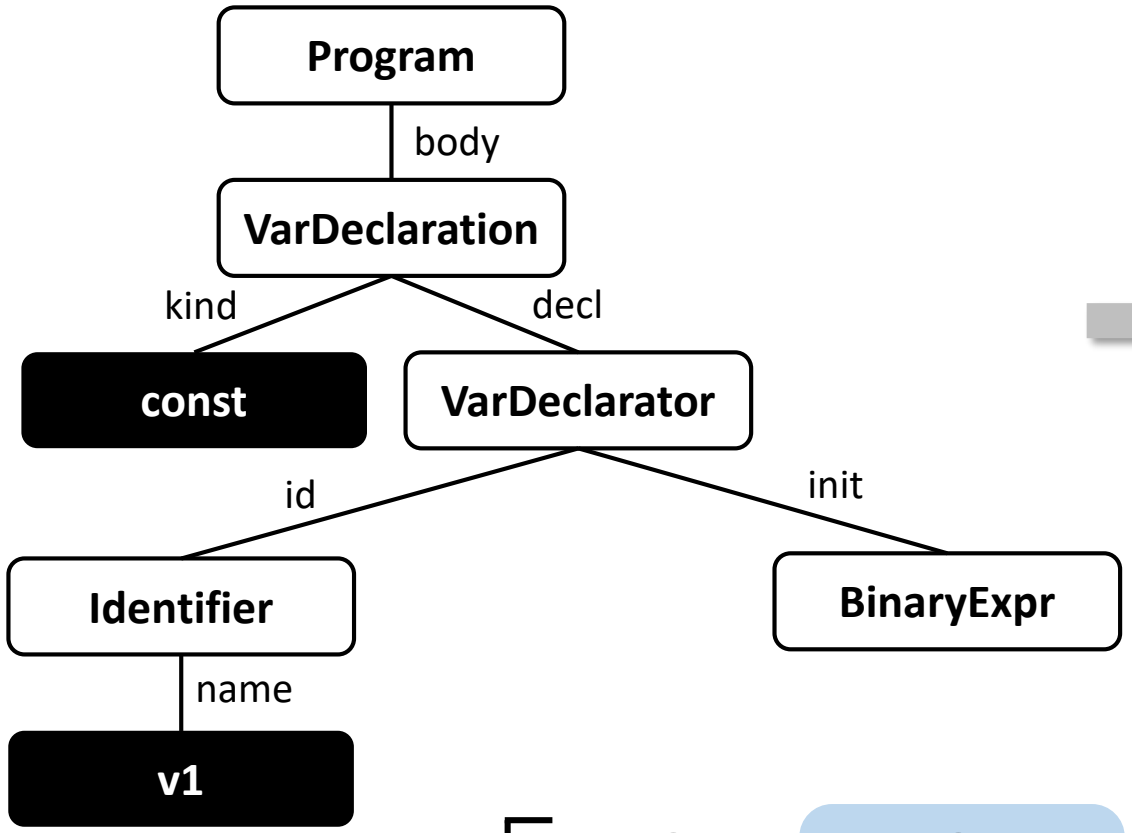
Trained LSTM model

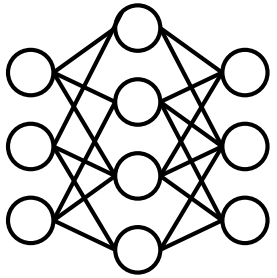


The probability distribution of the next fragment

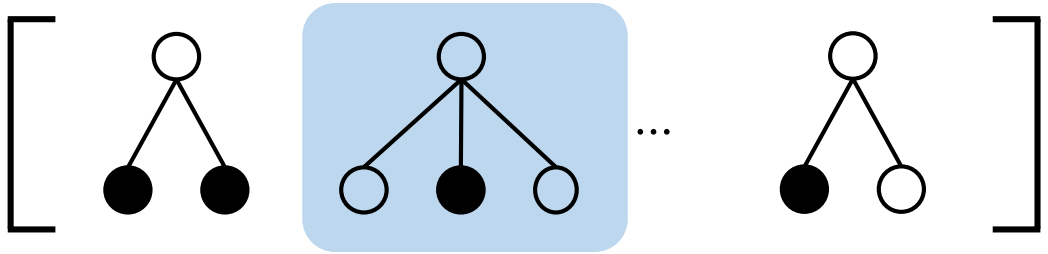
KAIST

AST Mutation





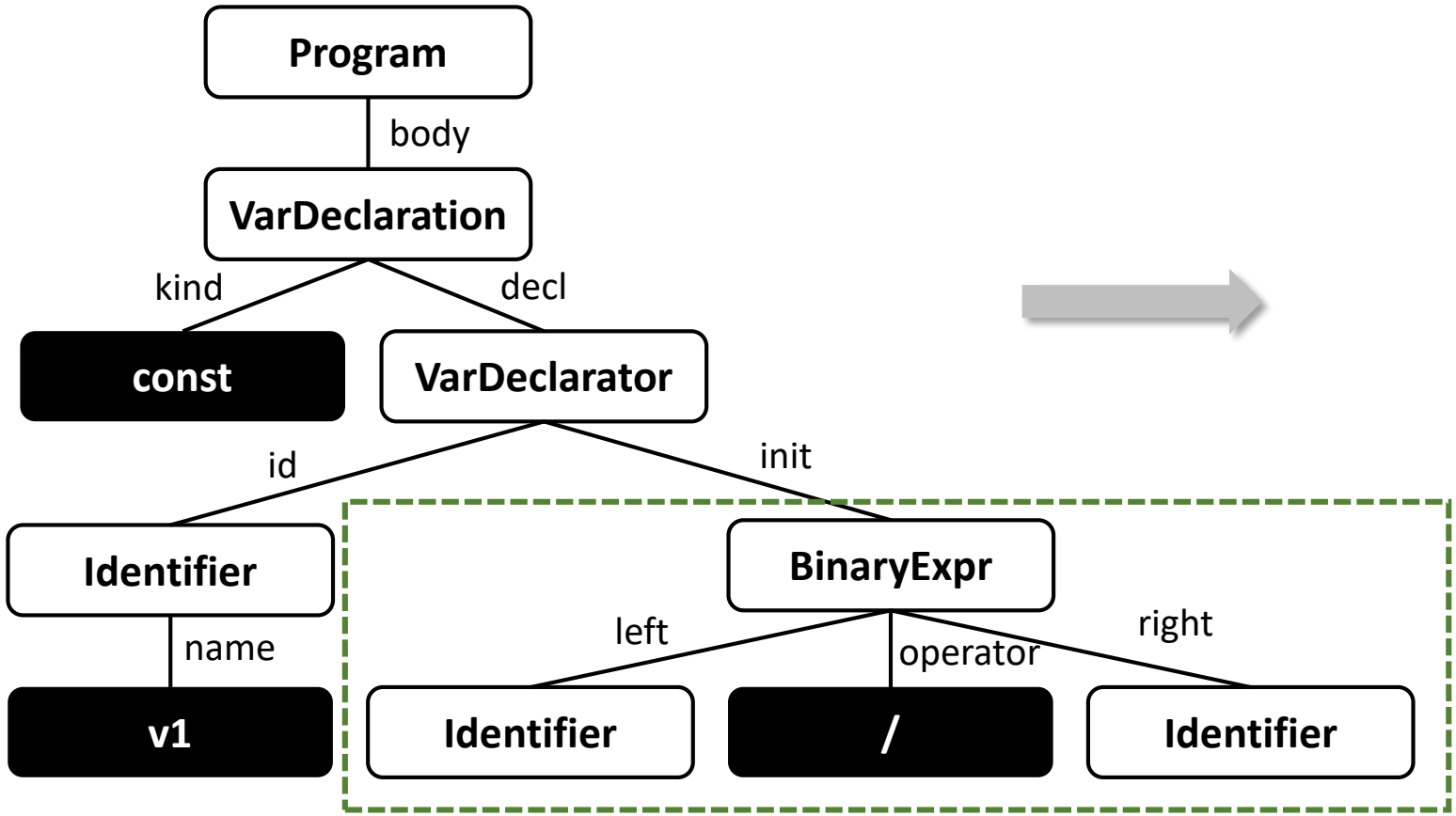
Trained LSTM model

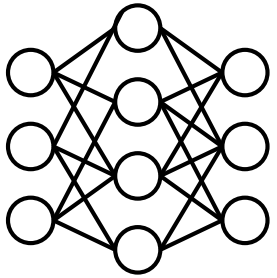


Randomly select one from Top K fragments

KAIST

AST Mutation

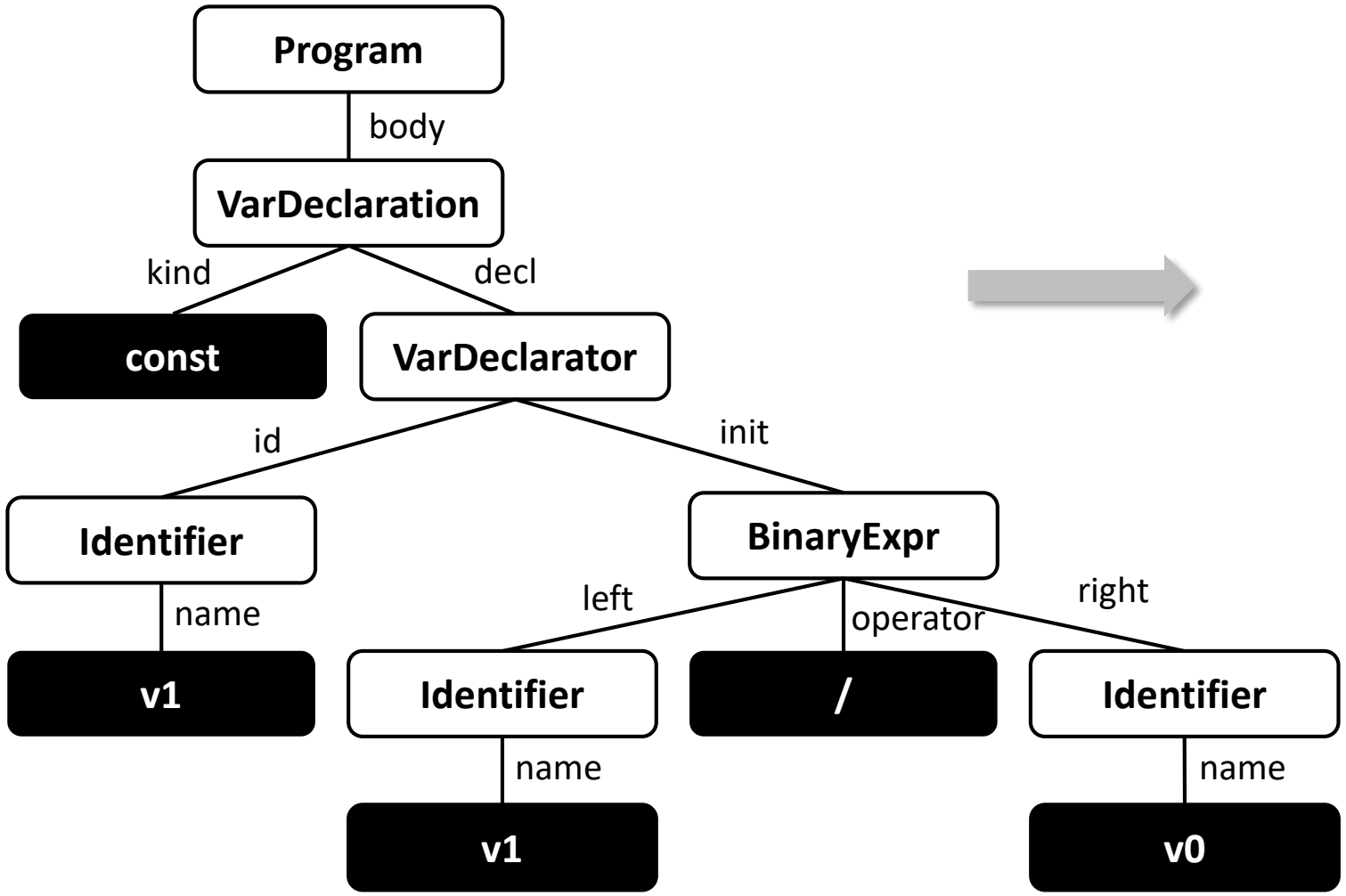


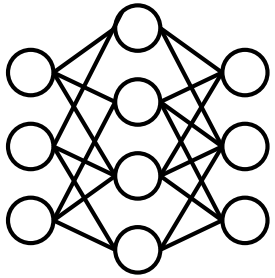


Trained LSTM model

AST Mutation

```
const v1 = v1 / v0;
```





Trained LSTM model

Repeat until all leaf nodes become terminal symbols

Resolving Reference Errors

```
var str = ‘Hello World’;

foo();

function foo () {
    var obj = Object();
    num = 10;
    b.toUpperCase(); // reference error
}
```

Code generated from the previous step

Resolving Reference Errors

```
var str = ‘Hello World’;

foo();

function foo () {
    var obj = Object();
    num = 10;
    b.toUpperCase();
}
```

```
global scope:
        str => string
        foo => function
        num => number
foo:
        obj => object
```

Identifier map

Resolving Reference Errors

```
var str = 'Hello World';

foo();

function foo () {
    var obj = Object();
    num = 10;
    b.toUpperCase();
}
```

```
global scope:
        str => string
        foo => function
        num => number
foo:
        obj => object
```

Identifier map

If possible, statically infer the type of undeclared identifiers!

Resolving Reference Errors

```
var str = 'Hello World';

foo();

function foo () {
    var obj = Object();
    num = 10;
    b.toUpperCase();
}
```

b is a string

```
global scope:
        str => string
        foo => function
        num => number
foo:
        obj => object
```

Identifier map

If possible, statically infer the type of undeclared identifiers!

Resolving Reference Errors

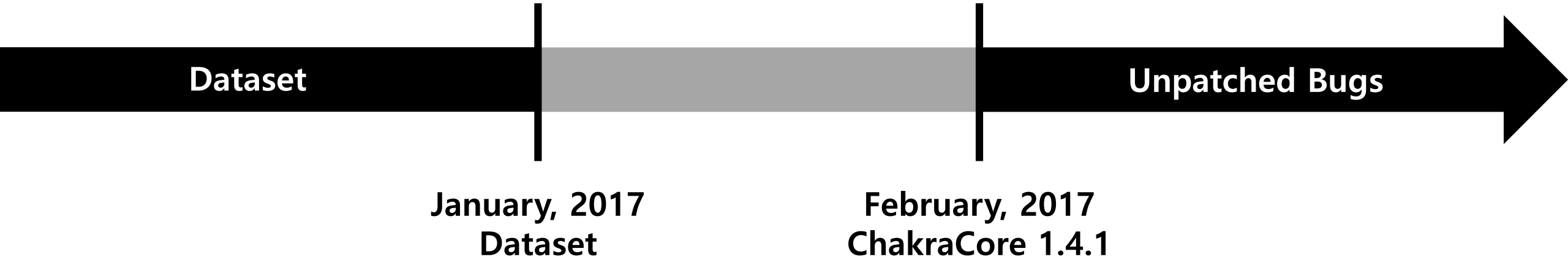
```
var str = 'Hello World';

foo();

function foo () {
    var obj = Object();
    num = 10;
    str.toUpperCase();
}
```

```
global scope:
        str => string
        foo => function
        num => number
foo:
        obj => object
```

Identifier map

Replace **b** with a declared identifier **str**

Experiment Setup



- Collected **33.5K** unique JS files
  - Regression tests from repository of four major JS engines and Test262
  - PoCs of known CVEs
- Ran fuzzers against **ChakraCore 1.4.1**
- JS code testing unpatched bugs are not in our dataset!

KAIST

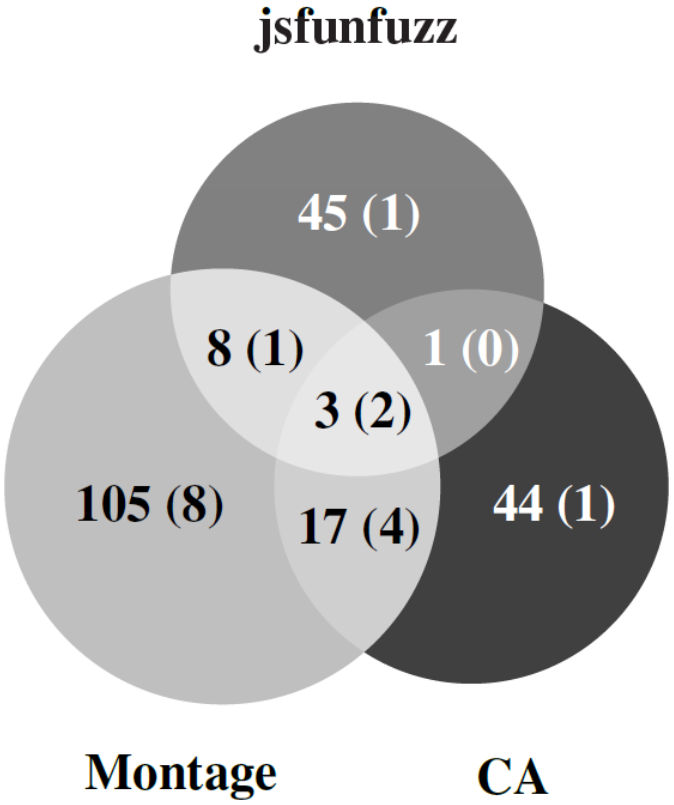
Comparison to State-of-the-art Fuzzers

- **For each fuzzer, ran 5 trials of a 72 hours-long fuzzing campaign**
  - CodeAlchemist: A **state-of-the-art** semantics-aware JS fuzzer, *NDSS'19*
  - IFuzzer: An evolutionary JS fuzzer, *ESORICS'16*
  - jsfunfuzz: A JS fuzzer developed by Mozilla

Metric	Build	# of Unique Crashes (Known CVEs)			
		Montage	CodeAlchemist	jsfunfuzz	IFuzzer
Median	Release	23 (7)	15 (4)	27 (3)	4 (1)
	Debug	49 (12)	26 (6)	27 (4)	6 (1)

Comparison to State-of-the-art Fuzzers

- **The # of found unique crashes (known CVEs)**



Effect of Language Models

- **For each approach, ran 5 trials of a 72 hours-long fuzzing campaign**

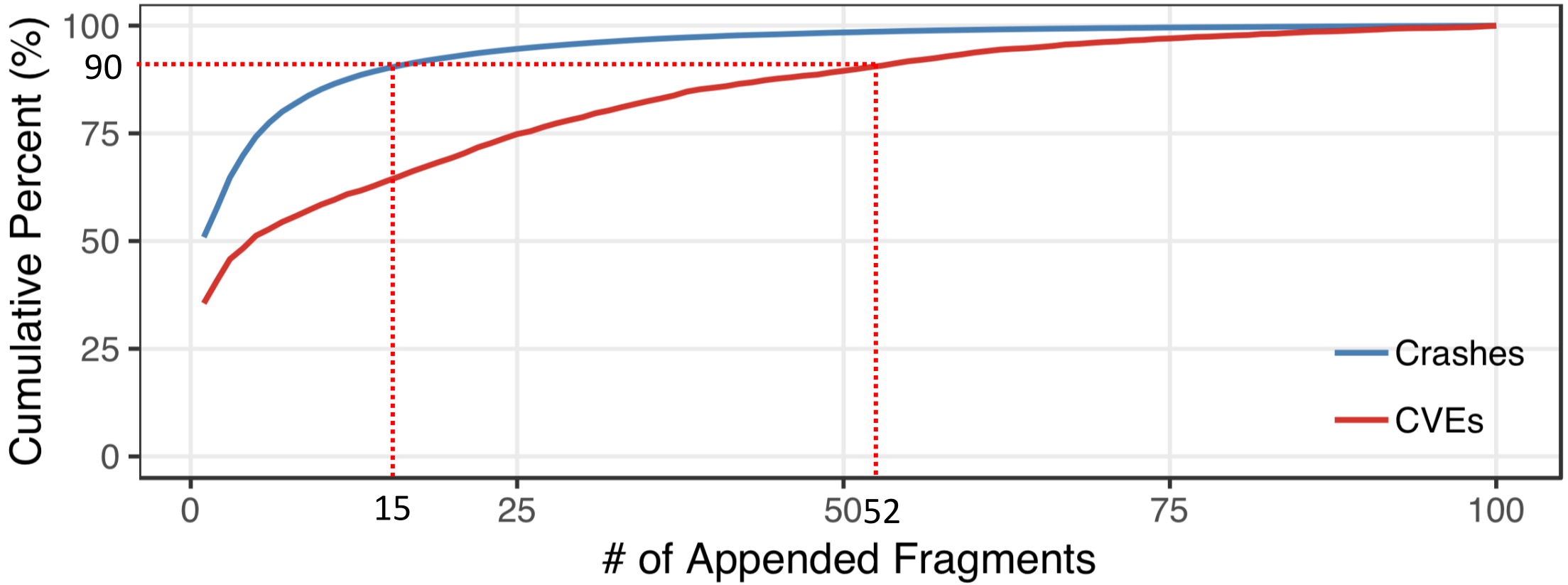
1. A random fragment selection w/o model: The **baseline** of Montage
2. A char/token-level RNN: A prevalent neural language model
3. A Markov model: A **simple** language model

Metric	Build	# of Unique Crashes (Known CVEs)			
		Montage	**Random**	**ch/token RNN**	**Markov**
Median	Release	23 (7)	12 (3)	1 (0)	19 (6)
	Debug	49 (12)	31 (7)	3 (0)	44 (11)

KAIST

Effect of the LSTM model

- **The # of appended fragments to compose a new subtree**



Effect of Resolving Reference Errors

- **For each approach, ran 5 trials of a 72 hours-long fuzzing campaign**

Metric	Build	# of Unique Crashes (Known CVEs)	
		Montage	**Montage w/o resolving step**
Median	Release	23 (7)	12 (4)
	Debug	49 (12)	41 (9)

The resolving step helps to find more bugs!

Montage still finds many bugs without the resolving step!

KAIST

Finding Real-World Bugs

- **We ran Montage on the four major JS engines for 1.5 months**
  - Found **37 previous bugs** in total.
    - 34 bugs including **two CVEs** from ChakraCore 1.11.7
    - One bug from V8 7.4.0 (beta)
    - Two bugs including **one CVE** from JSC 2.23.3

Conclusion

- Conducted systematic **studies on JS engine vulnerabilities**
- Proposed the first NNLM-guided JS engine fuzzing tool
- Found **37 real-world bugs** from the latest JS engines

Question?