MARCH 26-29, 2019
MARINA BAY SANDS / SINGAPORE

Efficient Approach to Fuzzing Interpreters

SAMSUNG

Bio

Marcin Dominiak

m.dominiak@samsung.com

Security Engineer

@ Samsung R&D Institute Poland

- Area of research: Fuzzing
- Cryptography, software engineering

Wojciech Rauner

w.rauner@samsung.com

Security Engineer

@ Samsung R&D Institute Poland

- Area of research: IoT/Web/Mobile
- Background: full-stack developer
- Likes to talk about crypto and programming
- Plays CTF in Samsung R&D PL team

Roadmap

1. Attacking interpreters
2. How do interpreters work?
3. Fuzzing interpreters
4. Fluff – our contribution to fuzzing interpreters
5. Evaluation & results
6. Future work
7. Q&A

Attacking interpreters

Attacking interpreters

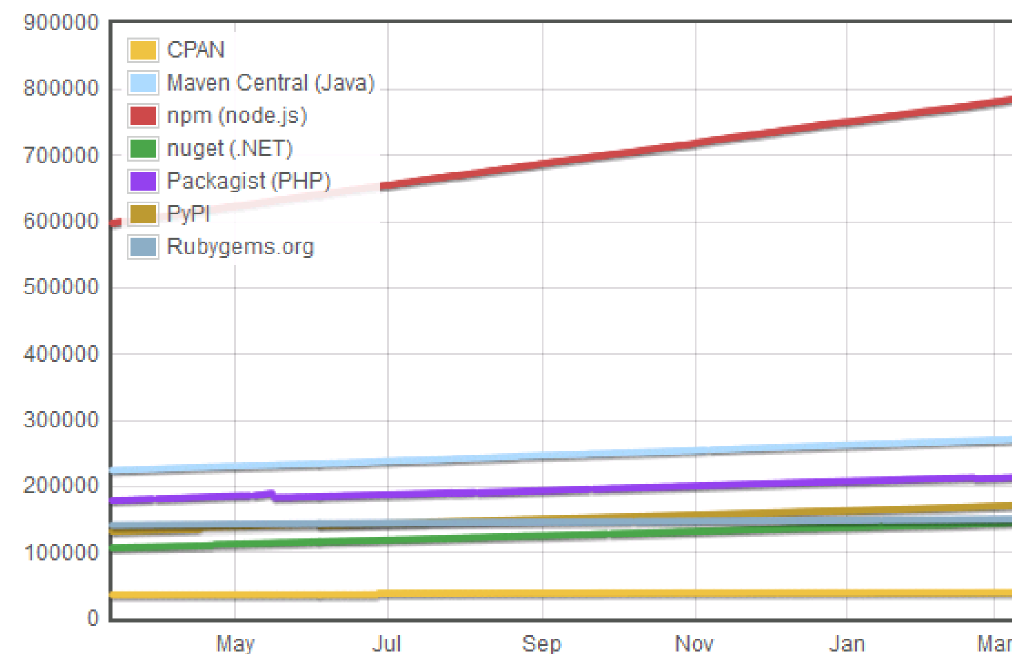
- TIOBE Index
  - JavaScript – 7th place
  - Python – 3rd, PHP – 8th
- StackOverflow Developer Survey
  - Most popular programming language – JS
  - 6th year in a row!
- Focusing on **JavaScript**
- JavaScript engines evolved
  - From interpreters to JIT compiler + VM

Unofficial JavaScript logo by Chris Williams, from GitHub logo.js, under very permissive licensing (WTFPL).

Attacking interpreters – JavaScript

- JavaScript implements ECMAScript standard
  - ActionScript, JScript, QScript
- New features from ECMAScript 6 (+)
  - Classes
  - Proxies
  - Big arrow (=>) functions

  and more
- Growing ecosystem
  - ~784k packages in NPM repository



http://www.modulecounts.com/ (march 2019)

Attacking interpreters – where can you find JS Engines

- Web browsers
  - V8 (Google Chrome)
  - SpiderMonkey (Mozilla Firefox)
  - Chakra (Microsoft Edge)
  - JavaScriptCore (Safari)

Server-side

- Node.JS
- nJS (nginx)

Internet of Things

- Espruino
- Duktape
- mJS
- jerryscript (IoT.js)

Attacking interpreters – vulnerabilities and scenarios

- JavaScript vulnerabilities database
  - https://github.com/tunz/js-vuln-db
- Types of bugs
  - Type confusion
  - All sorts of overflows
  - Use-after-free
  - Race conditions

Attacking interpreters – vulnerabilities and scenarios

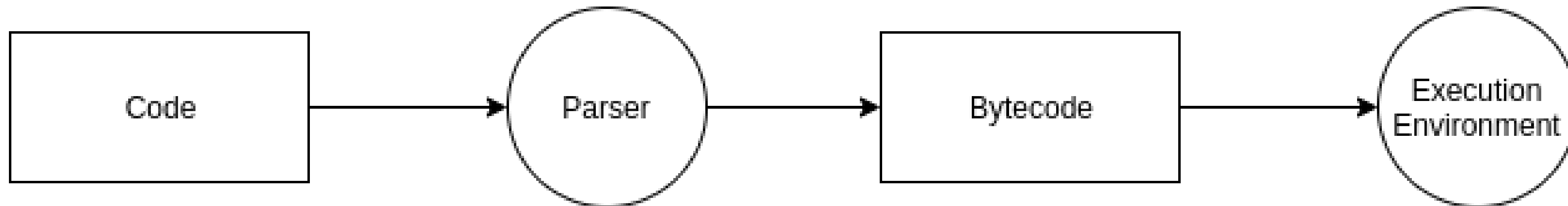
- JavaScript vulnerabilities database
  - https://github.com/tunz/js-vuln-db
- Types of bugs
  - Type confusion
  - All sorts of overflows
  - Use-after-free
  - Race conditions

- Code evaluated by interpreters often comes from untrusted sources
- Example attack scenarios
  - Web browsers – via multiple vectors
  - Online Code Execution Services
  - Continuous Integration Systems
  - Social Engineering
  - Many more

How do interpreters work?

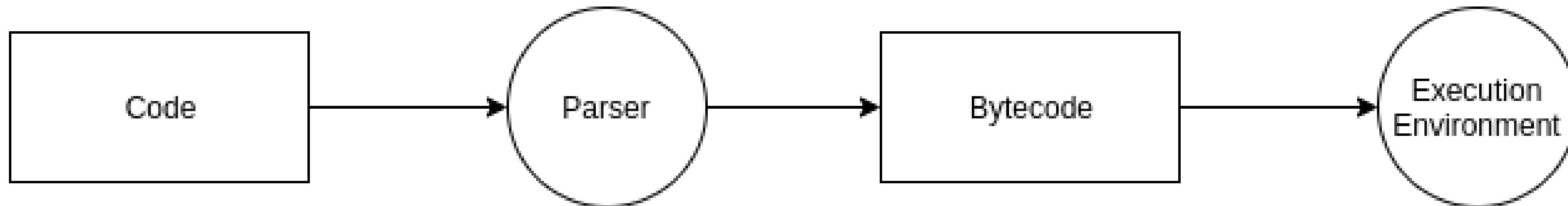
Interpreters 101

- Split input into tokens (lexems)
- Parse tokens into a parsing tree
- Initialize execution environment
- Execute code

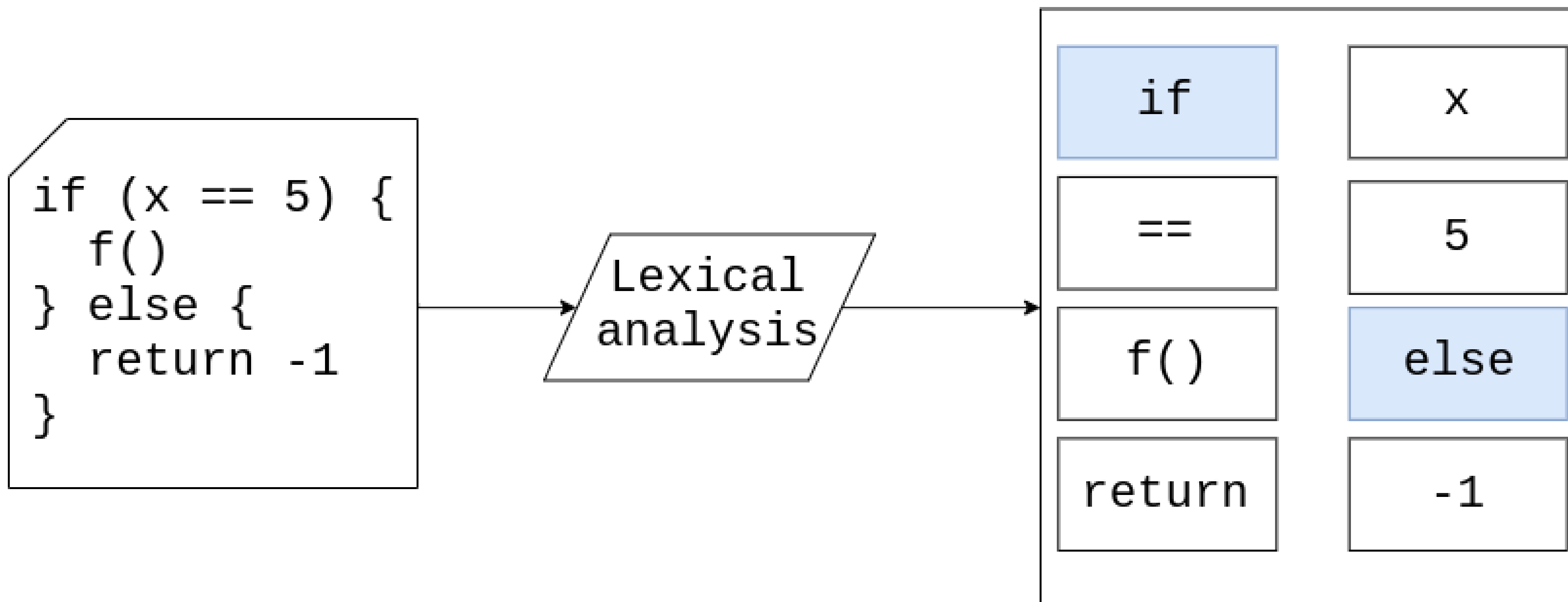


Interpreters 101 – Parsing

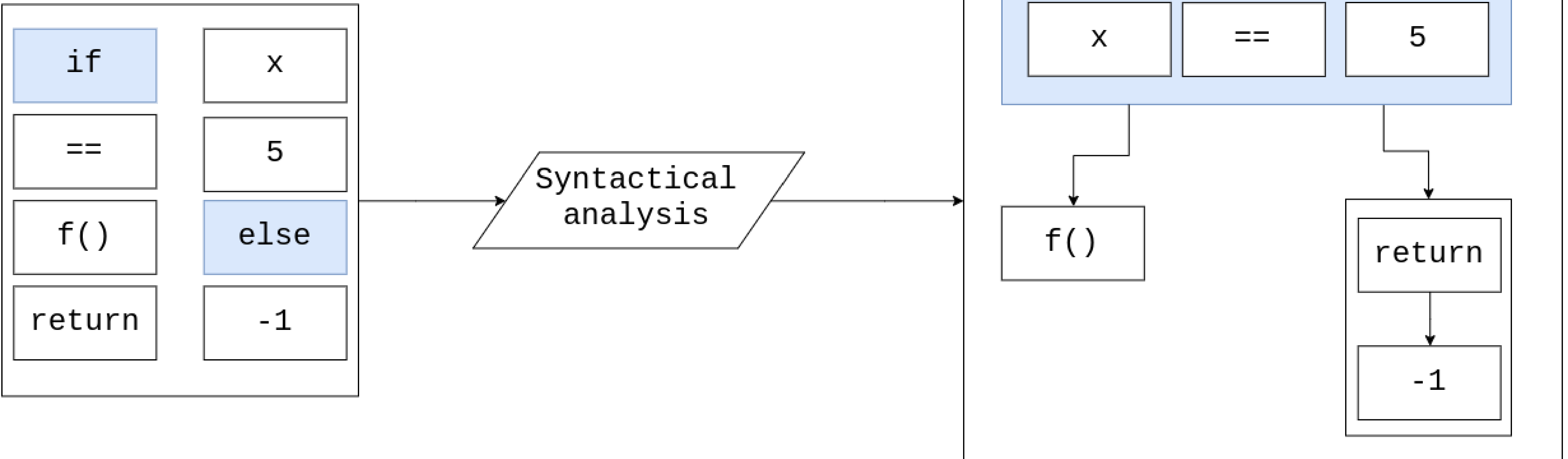
- Analyze input text
- Split input text into tokens
- Construct parse tree
- Structure provided by a formal grammar



Interpreters 101 – Parsing



Interpreters 101 - Parsing



Interpreters 101 – Context Free Grammars

- Formal language
- Invented by Noam Chomsky
- Describes rules for manipulating string productions
- Useful for describing syntax of a language
- Used by parser/lexer generators (bison, yacc, lex)

GNU Bison logo

Interpreters 101 – Context Free Grammars

$$G = (V_N, V_T, v_0, P)$$

- V_N – Set of nonterminals
- V_T – Set of terminals
- $v_0 \in V_N$ – Initial symbol
- P – set of productions, i.e. functions $v \rightarrow x_1 x_2 \dots x_n$, where $x_i \in V_N \cup V_T, v \in V_N$

Interpreters 101 – Context Free Grammars

Notation:

- Big letter – nonterminal
- Small letter – terminal

The following productions are equivalent:

$$A \rightarrow a$$
$$A \rightarrow b$$

$$A \rightarrow a \mid b$$

Interpreters 101 – Context Free Grammars (example)

Grammar generating palindromes over $\{a, b\}$

$$S \rightarrow aSa|bSb|a|b|\epsilon$$

Production for aba

$$S \rightarrow aSa \rightarrow aba$$

Production for $abba$

$$S \rightarrow aSa \rightarrow abSba \rightarrow abba$$

Interpreters 101 – Context Free Grammars (programs)

$$Program \rightarrow Declaration\,;Instruction \mid \epsilon$$

Interpreters 101 – Context Free Grammars (programs)

$$Program \rightarrow Declaration\,; Instruction \mid \epsilon$$

$$Declaration \rightarrow var\ Id \mid Declaration; Declaration \mid \epsilon$$

Interpreters 101 – Context Free Grammars (programs)

$$Program \rightarrow Declaration\,;Instruction \mid \epsilon$$

$$Declaration \rightarrow var\ Id \mid Declaration; Declaration \mid \epsilon$$

$$Instruction \rightarrow Id = Expression \mid print(Expression)$$

$$Instruction \rightarrow Instruction\,;Instruction \mid \epsilon$$

Interpreters 101 – Context Free Grammars (programs)

$$Program \rightarrow Declaration\,; Instruction \mid \epsilon$$

$$Declaration \rightarrow var\ Id \mid Declaration; Declaration \mid \epsilon$$

$$Instruction \rightarrow Id = Expression \mid print(Expression)$$

$$Instruction \rightarrow Instruction\,; Instruction \mid \epsilon$$

$$Expression \rightarrow Integer \mid Id \mid Expression + Expression$$

$$Expression \rightarrow Expression * Expression$$

Interpreters 101 – Context Free Grammars (programs)

$$Program \rightarrow Declaration\ ; Instruction \mid \epsilon$$

$$Declaration \rightarrow var\ Id \mid Declaration; Declaration \mid \epsilon$$

$$Instruction \rightarrow Id = Expression \mid print(Expression)$$

$$Instruction \rightarrow Instruction\ ; Instruction \mid \epsilon$$

$$Expression \rightarrow Integer \mid Id \mid Expression + Expression$$

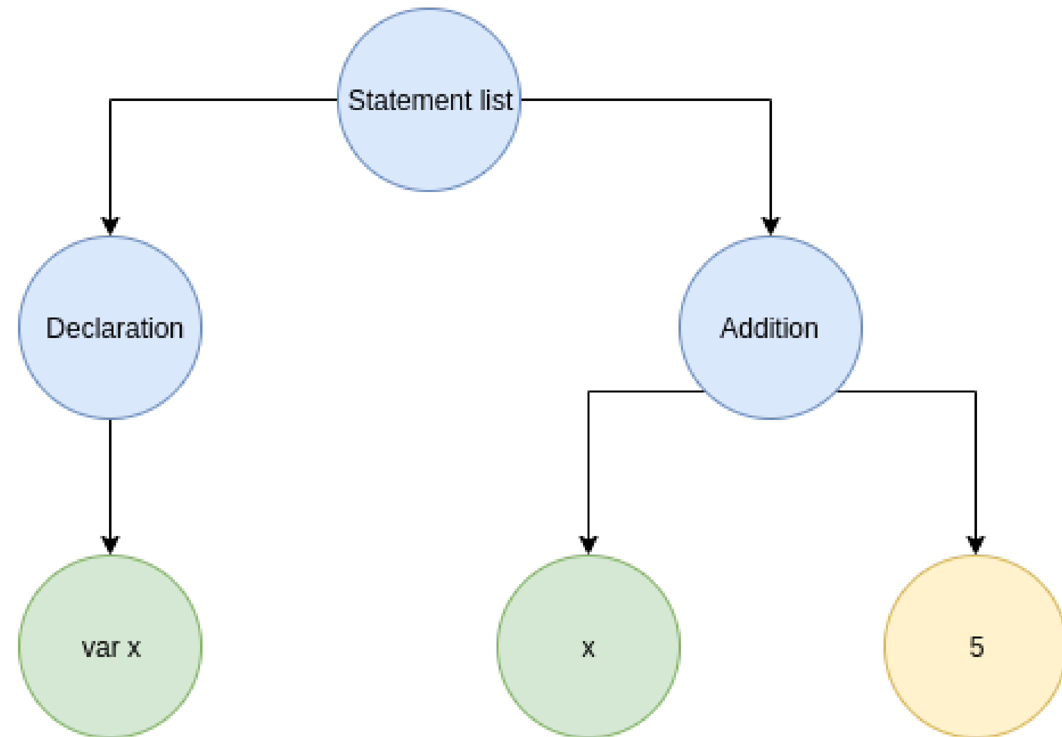
$$Expression \rightarrow Expression\ * Expression$$

Assume that Id and $Integer$ are defined.

Interpreters 101 – Abstract Syntax Tree

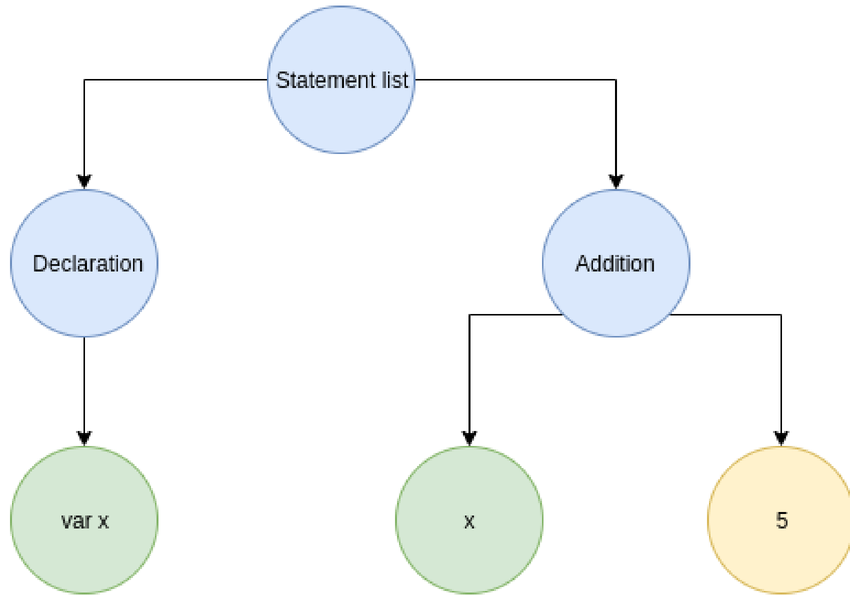
- Tree structure
- Represents source code
- Structure controlled by CFG
- Convenient processing

$$var\ x;$$
$$x + 5;$$



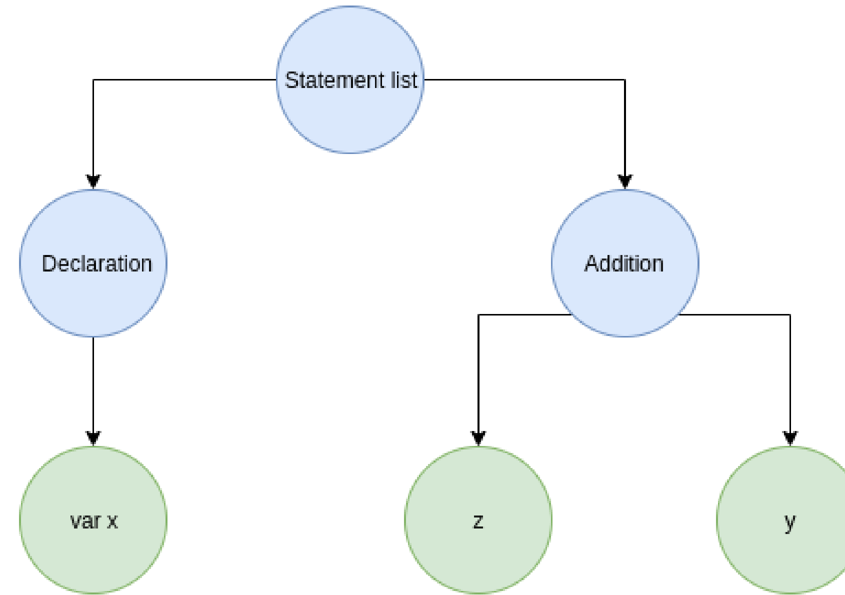
Interpreters 101 – Legal decompositions

- Valid program



$var\ x;$
$x + 5;$

- Invalid program



$var\ x;$
$z + y;$

Interpreters 101

Further actions

- Analysis
- Type checking
- Optimizations
- JIT compilation
- Execution

Fuzzing interpreters

Mutation based fuzzing

- Maintain corpus of test cases
- Apply mutations
- Observe target execution
- Further mutate interesting test cases

Mutation based fuzzing

- Maintain corpus of test cases
- Apply mutations
- Observe target execution
- Further mutate interesting test cases

Pros

- Fuzzer learns language features
- Well tested parser
- Easy setup

Cons

- Semantically incorrect test cases
- Syntactically incorrect test cases
- Weak coverage of the backend

Mutation based fuzzing

- Maintain corpus of test cases
- Apply mutations
- Observe target execution
- Further mutate interesting test cases

Sample tools
- American Fuzzy Lop
- hongfuzz
- libFuzzer
- zuff

Pros
- Fuzzer learns language features
- Well tested parser
- Easy setup

Cons
- Semantically incorrect test cases
- Syntactically incorrect test cases
- Weak coverage of the backend

Mutation based fuzzing (with keywords)

- Maintain corpus of test cases
- Apply mutations
- Observe target execution
- Further mutate interesting test cases
- Incorporate predefined keywords

Sample tools

- American Fuzzy Lop
- hongfuzz
- libFuzzer
- zuff

Pros

- Fuzzer learns language features
- Well tested parser
- Easy setup
- Faster detection of the language

Cons

- Semantically incorrect test cases
- Syntactically incorrect test cases
- Weak coverage of the backend
- Preparation of keywords

Grammar based fuzzing

- Construct grammar of target language
- Generate conforming test cases
- Execute

Grammar based fuzzing

- Construct grammar of target language
- Generate conforming test cases
- Execute

Pros

- Inclusion of language features
- Syntactically correct test cases

Cons

- Preparation of grammar
- Semantically incorrect test cases
- Weak coverage of parser
- Lack of test harness

Grammar based fuzzing

- Construct grammar of target language
- Generate conforming test cases
- Execute

Sample tools

- grammarinator
- langfuzz

Pros

- Inclusion of language features
- Syntactically correct test cases

Cons

- Preparation of grammar
- Semantically incorrect test cases
- Weak coverage of parser
- Lack of test harness

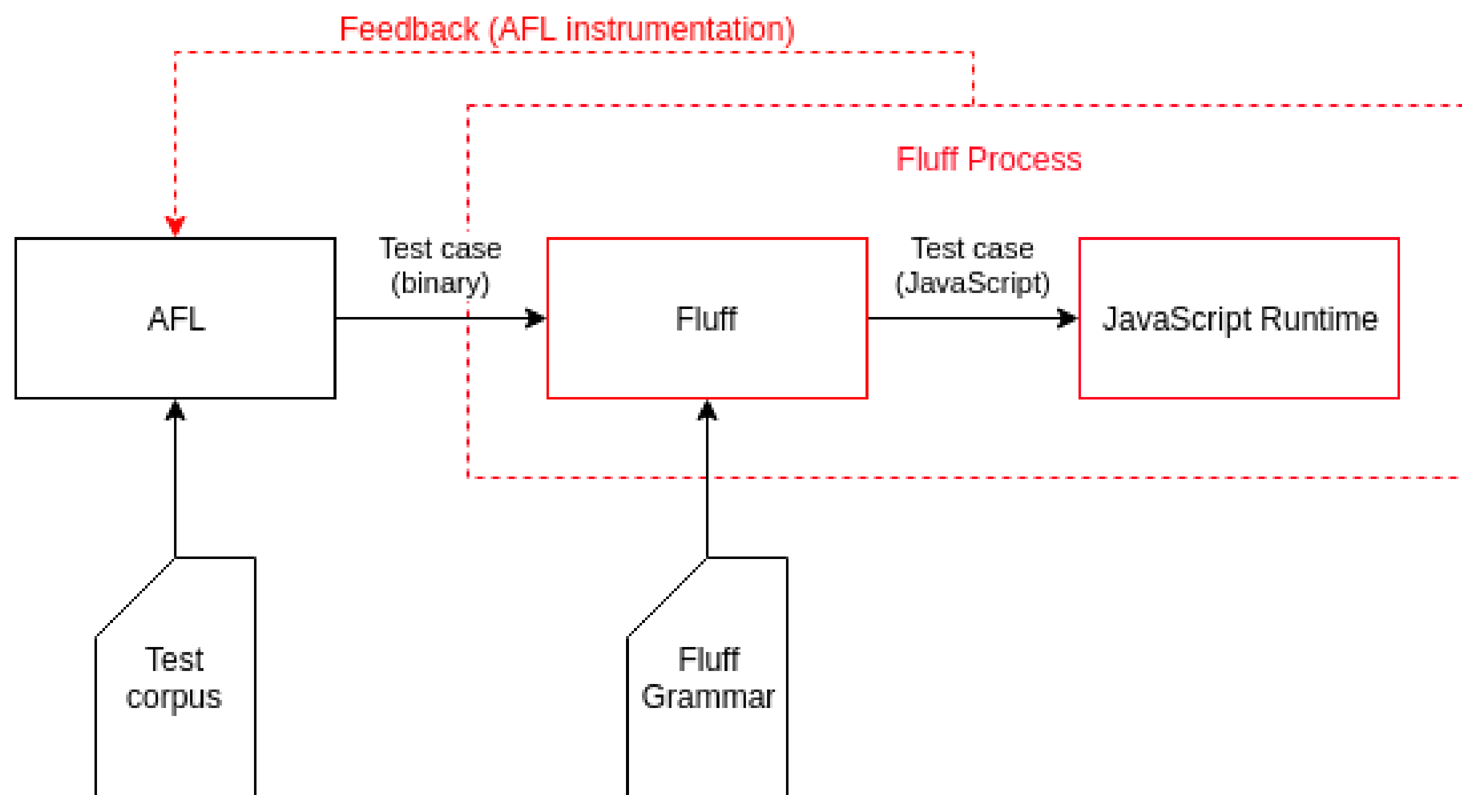
Fluff

our contribution to fuzzing interpreters

Fluff – design goals

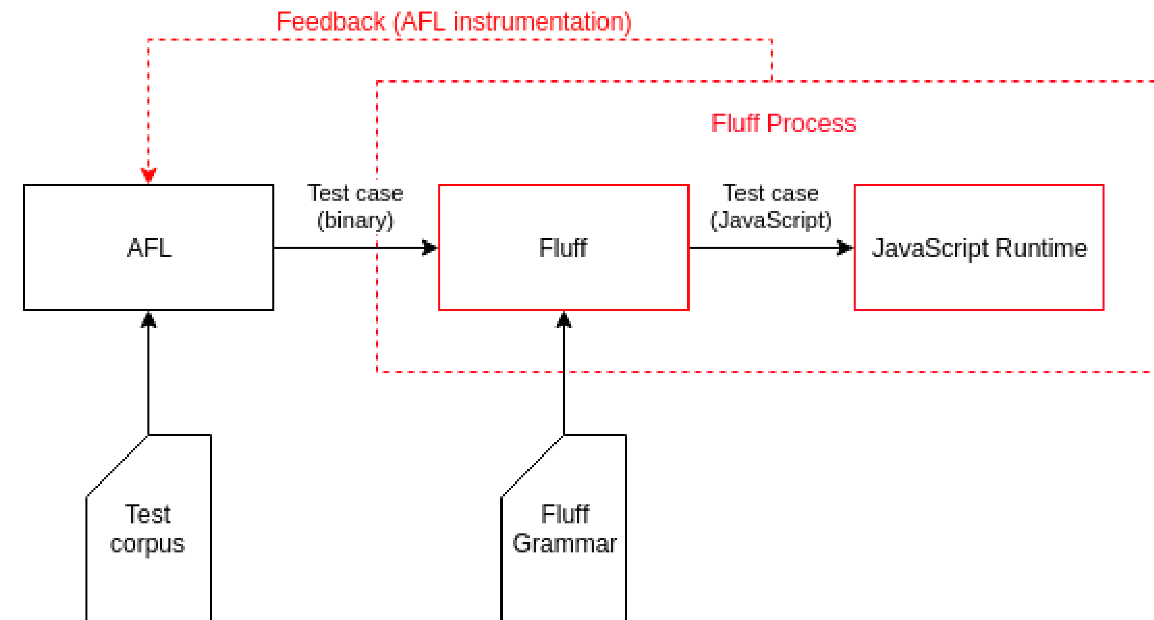
- Construct semantically correct test cases
- Provide test harness
  - Integration with AFL
  - Use path detection provided by AFL
- Allow testing multiple implementations
- Effectiveness
  - Speed
  - Support for ECMAScript 6

Fluff – design



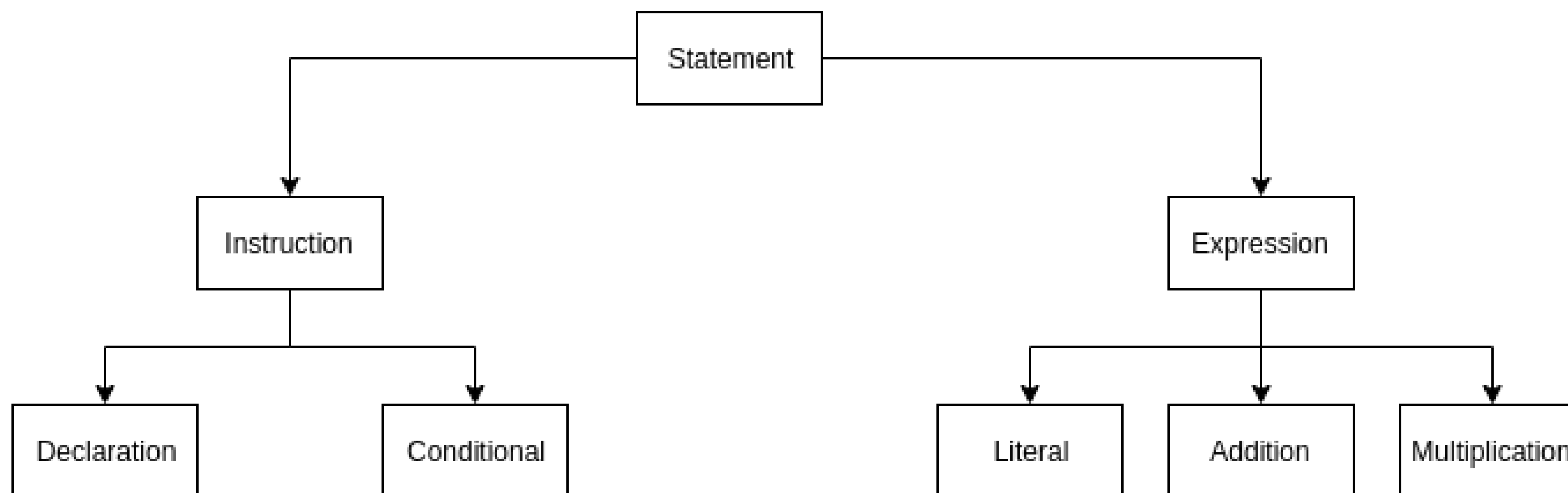
Fluff – design

- Input – encoded program
- Decode input
  - Permissive parsing
- Construct (and modify) AST
- Pass generated code for execution



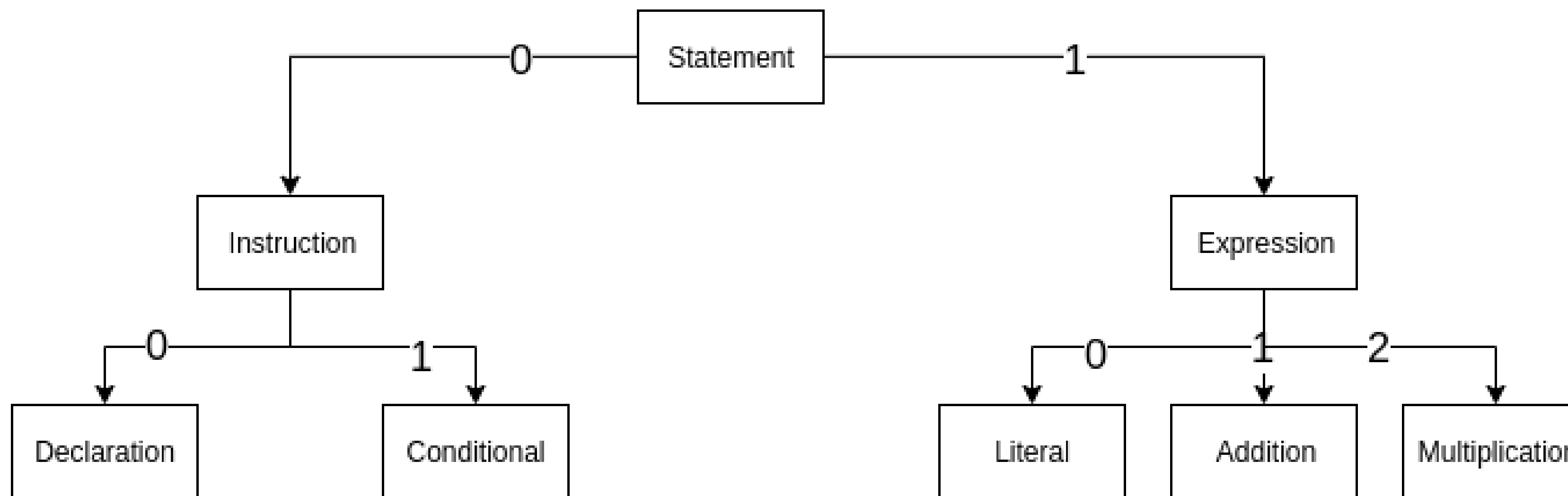
Fluff – technical example

- Construct simplified grammar



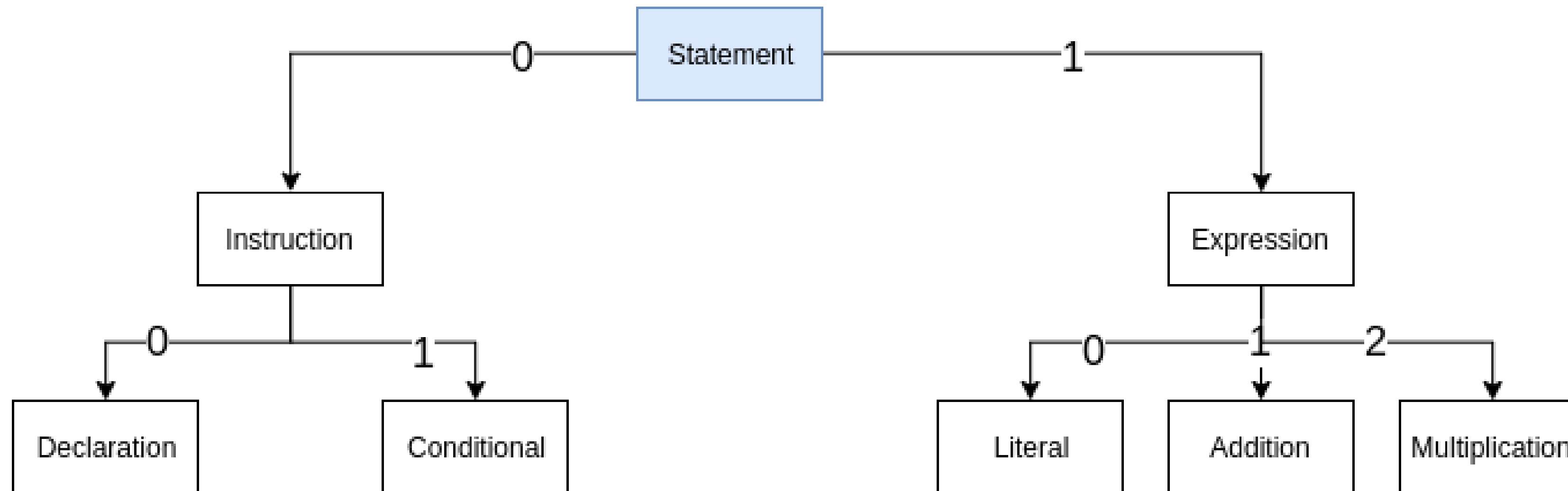
Fluff – technical example

- Label each edge



Fluff – technical example

- Begin parsing in root

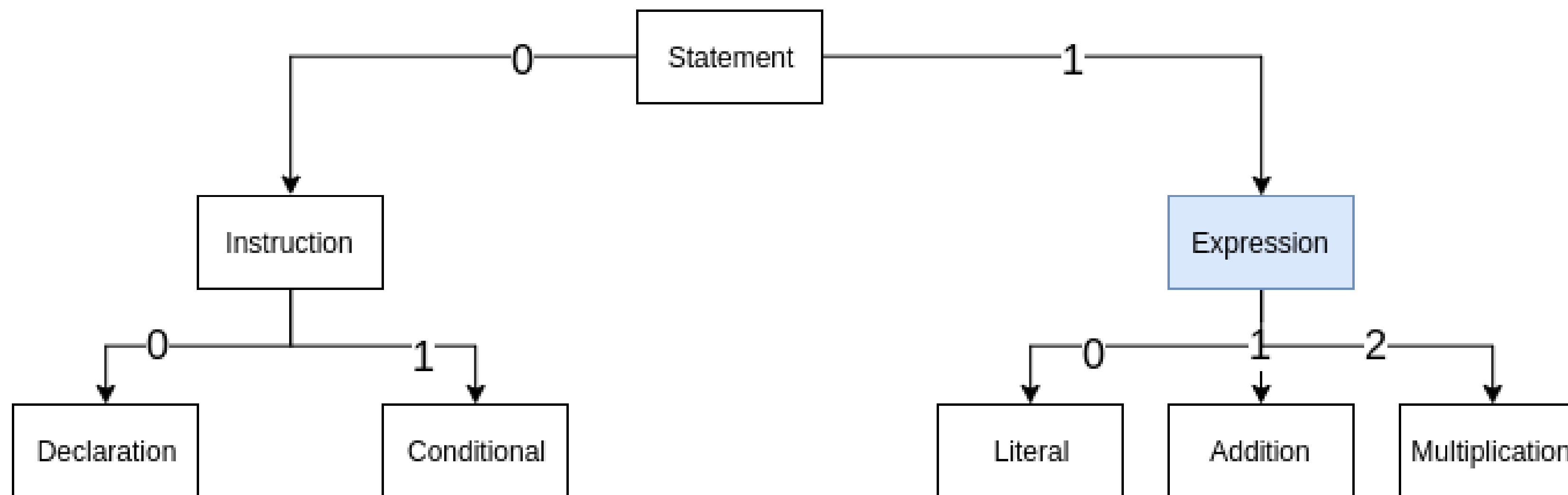


Input: $0x01, 0x04, 0x00, 0x17, 0x06, 0x13$

Result:

Fluff – technical example

- Use input to iterate over tree

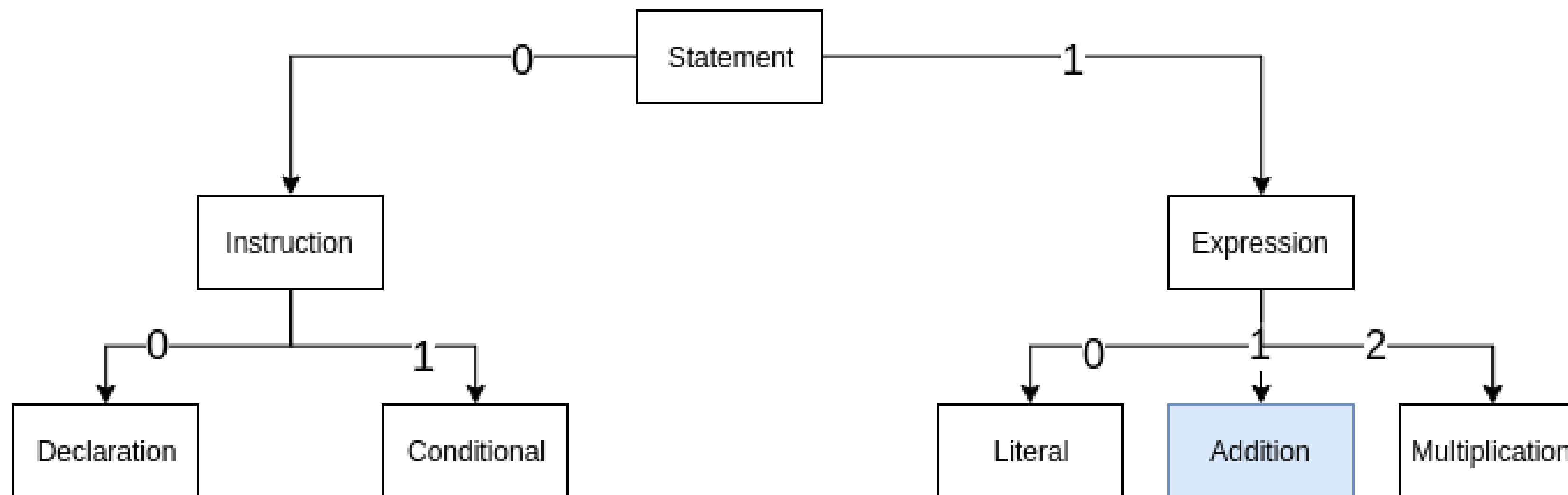


Input: $0x01$, $0x04$, $0x00$, $0x17$, $0x06$, $0x13$

Result:

Fluff – technical example

- Use input to iterate over tree

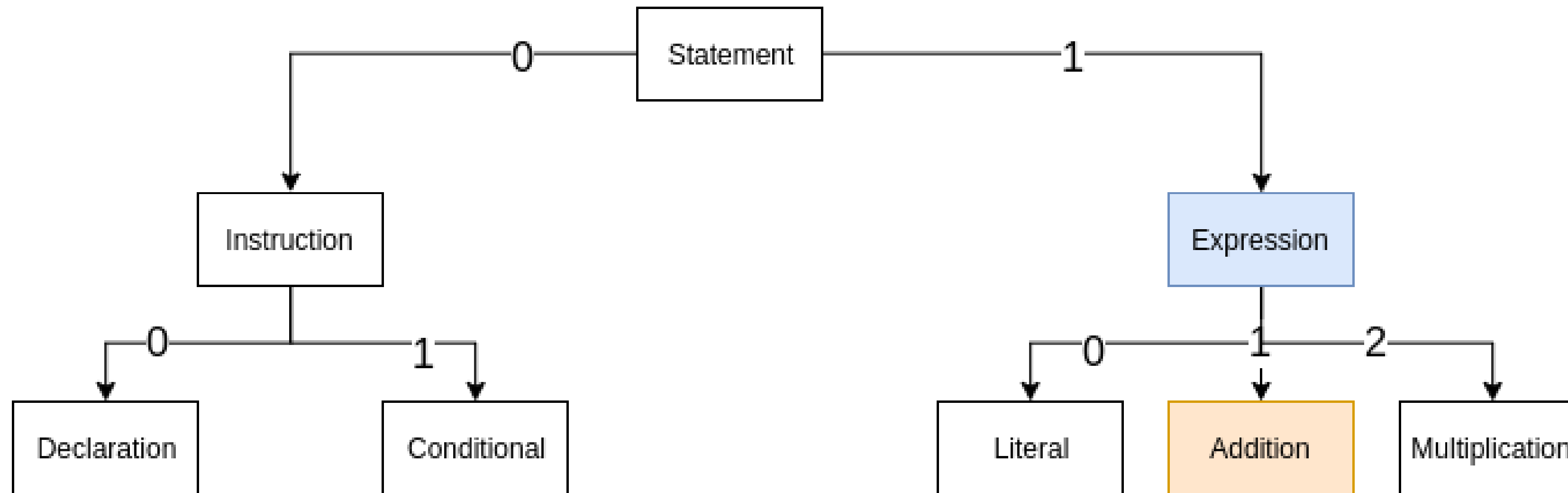


Input: $0x01$, $0x04$, $0x00$, $0x17$, $0x06$, $0x13$

Result: _ + _

Fluff – technical example

- Use input to iterate over tree

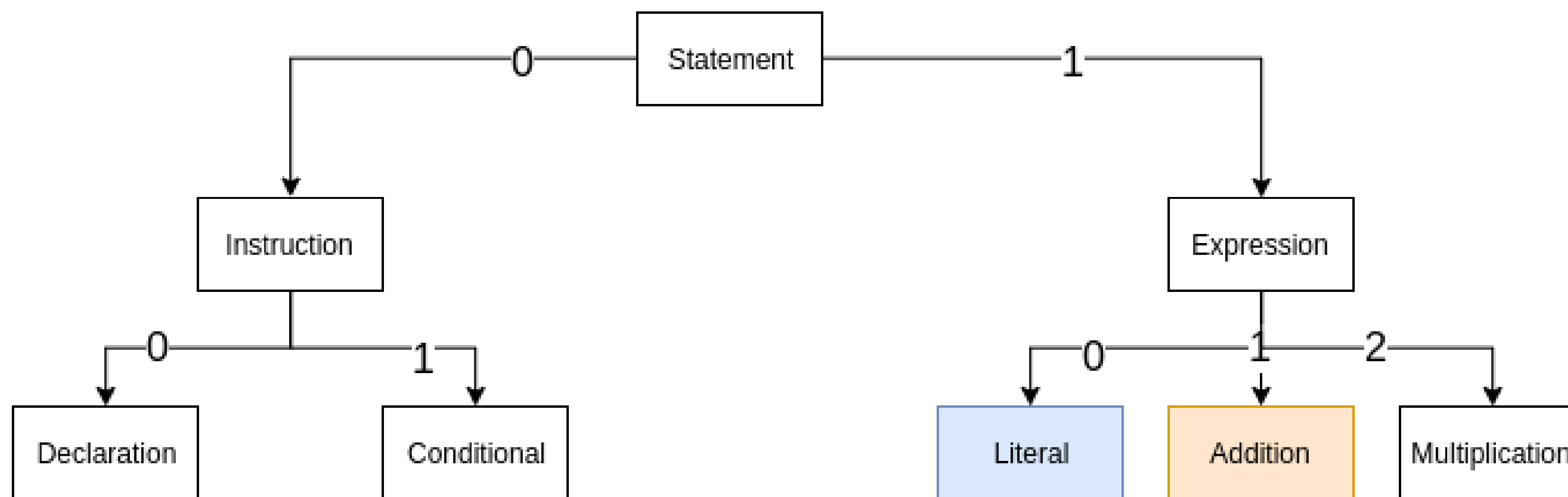


Input: $0x01$, $0x04$, $0x00$, $0x17$, $0x06$, $0x13$

Result: _ + _

Fluff – technical example

- Use input to iterate over tree

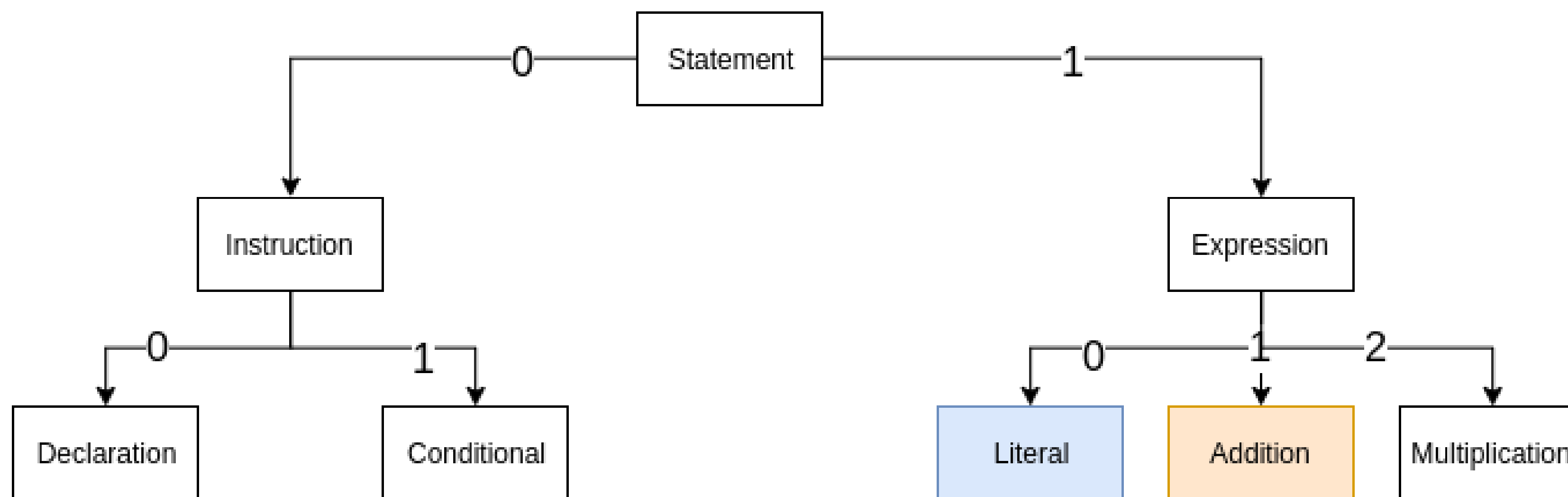


Input: $0x01$, $0x04$, $0x00$, $0x17$, $0x06$, $0x13$

Result: _ + _

Fluff – technical example

- Use input to iterate over tree

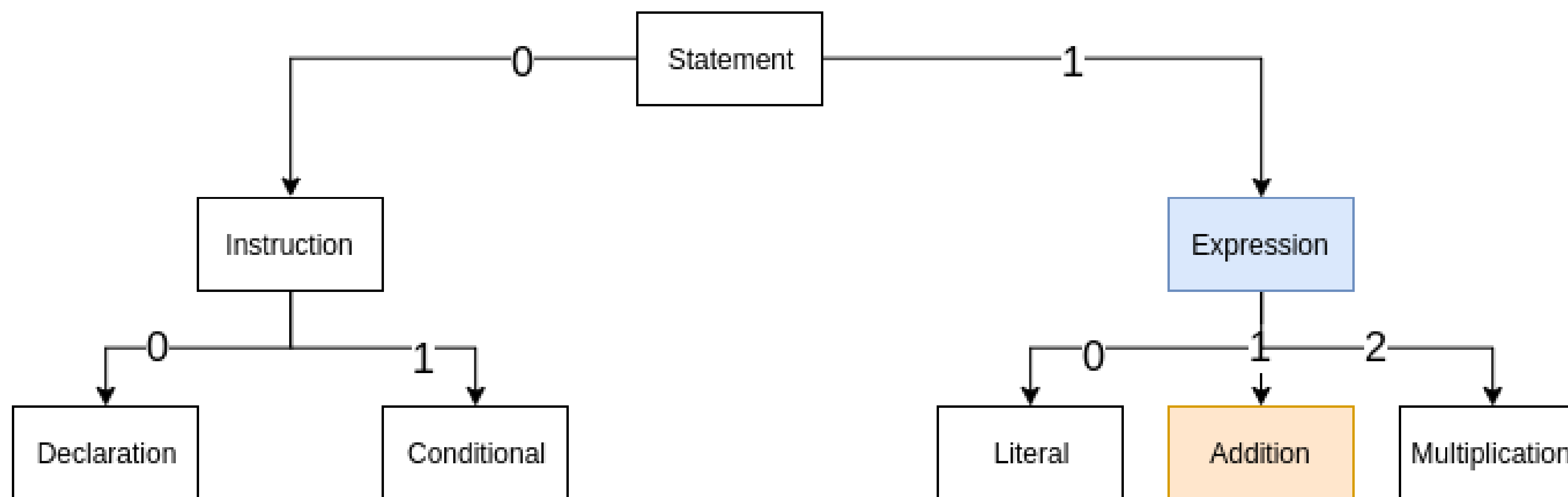


Input: $0x01, 0x04, 0x00, 0x17, 0x06, 0x13$

Result: $23 + _$

Fluff – technical example

- Use input to iterate over tree

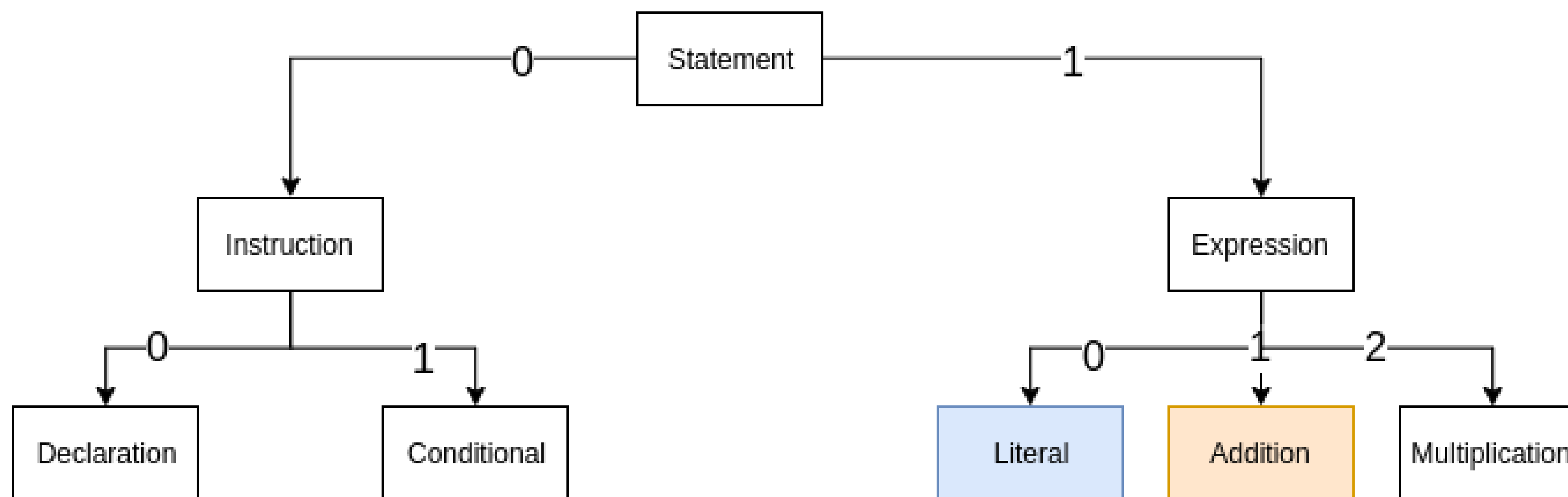


Input: $0x01$, $0x04$, $0x00$, $0x17$, $0x06$, $0x13$

Result: $23\ +\ _$

Fluff – technical example

- Use input to iterate over tree

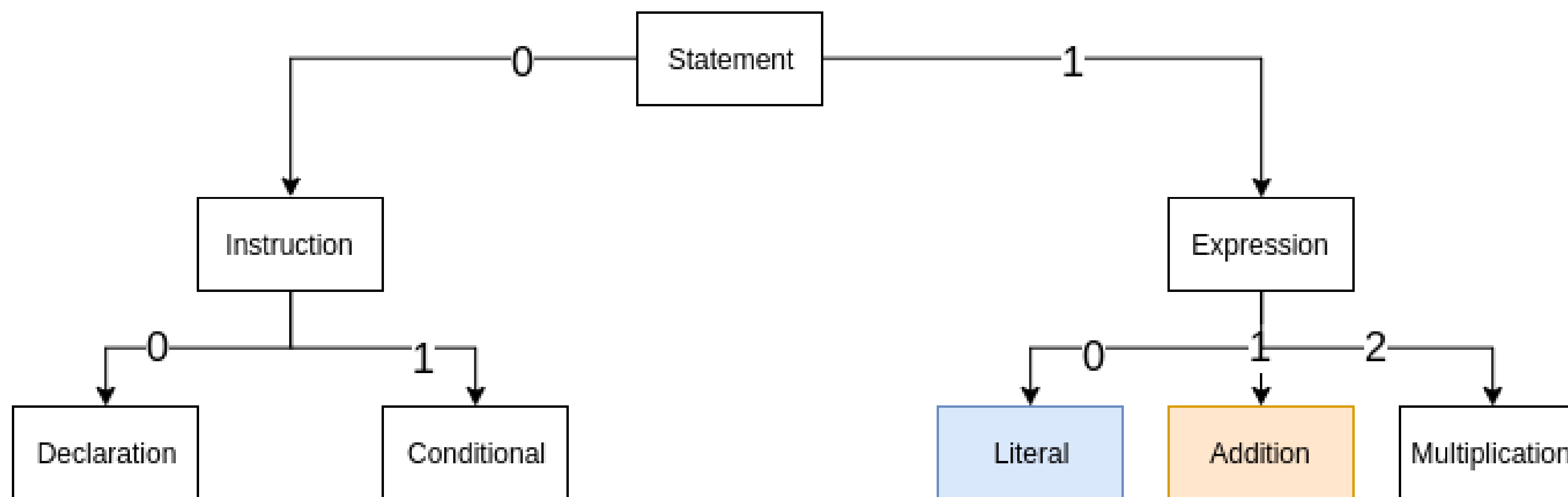


Input: $0x01$, $0x04$, $0x00$, $0x17$, $0x06$, $0x13$

Result: $23\ +\ _$

Fluff – technical example

- Use input to iterate over tree



Input: $0x01$, $0x04$, $0x00$, $0x17$, $0x06$, $0x13$

Result: $23 + 19$

Fluff – other language features

- Functions
  - Maintain a registry of identifiers
  - Read the number of instructions from input

Fluff – other language features

- Functions
  - Maintain a registry of identifiers
  - Read the number of instructions from input
- Classes
  - Syntactic sugar for functions

Fluff – other language features

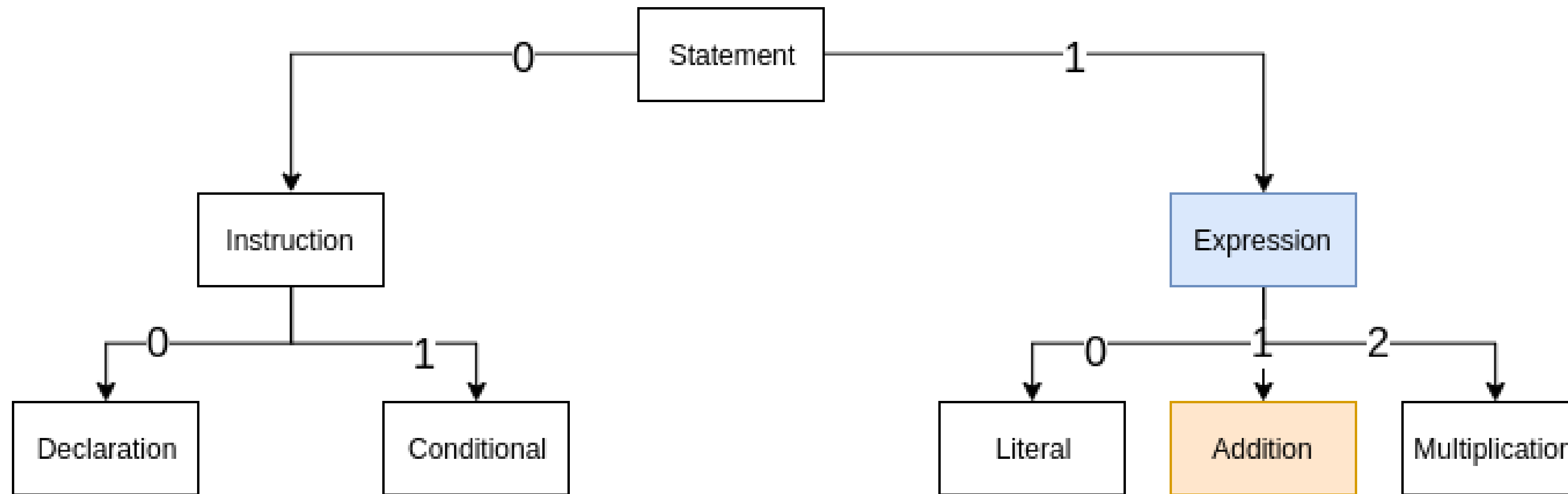
- Functions
  - Maintain a registry of identifiers
  - Read the number of instructions from input
- Classes
  - Syntactic sugar for functions
- Built-ins
  - Preload identifier registers

Fluff – unfinished statements

- Not enough data to finish a statement
- Provide default closures

Fluff – unfinished statements

- Not enough data to finish a statement
- Provide default closures

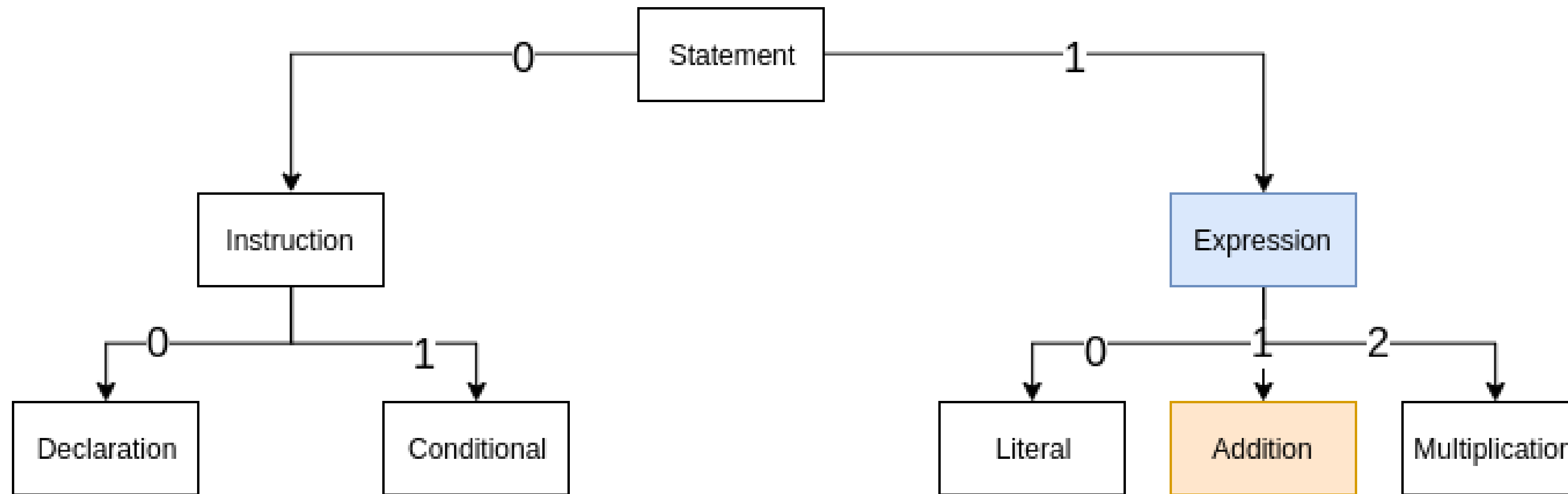


Input: $0x01$, $0x04$, $0x00$, $0x17$

Result: 23 + _

Fluff – unfinished statements

- Not enough data to finish a statement
- Provide default closures

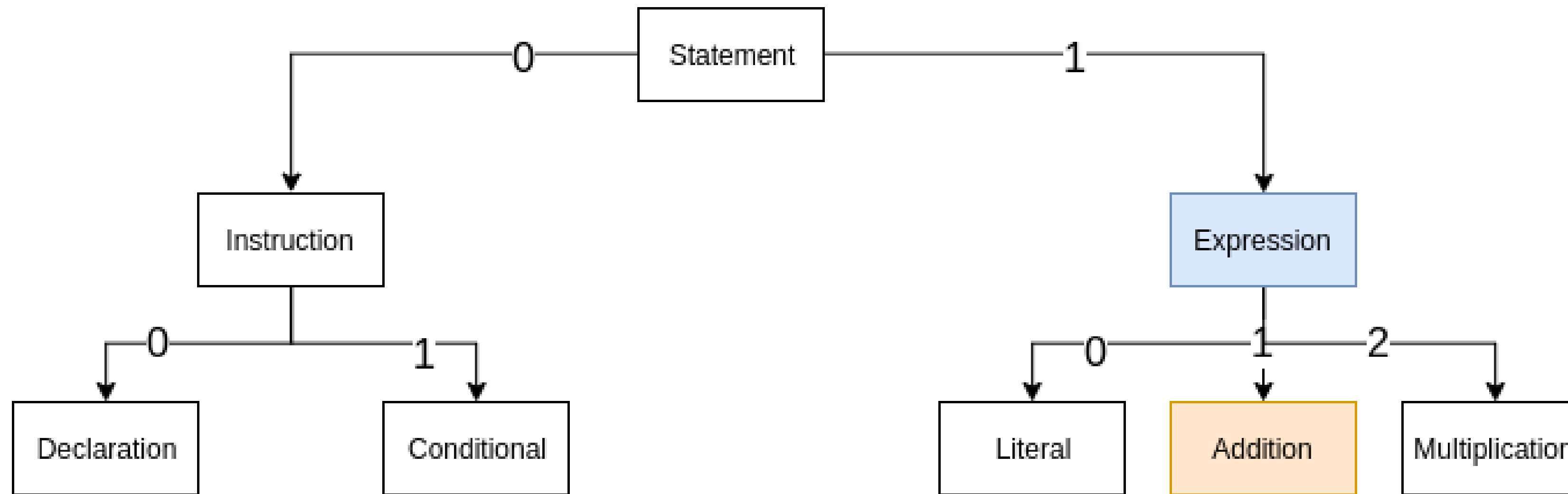


Input: $0x01$, $0x04$, $0x00$, $0x17$

Result: $23 + ?$

Fluff – unfinished statements

- Not enough data to finish a statement
- Provide default closures

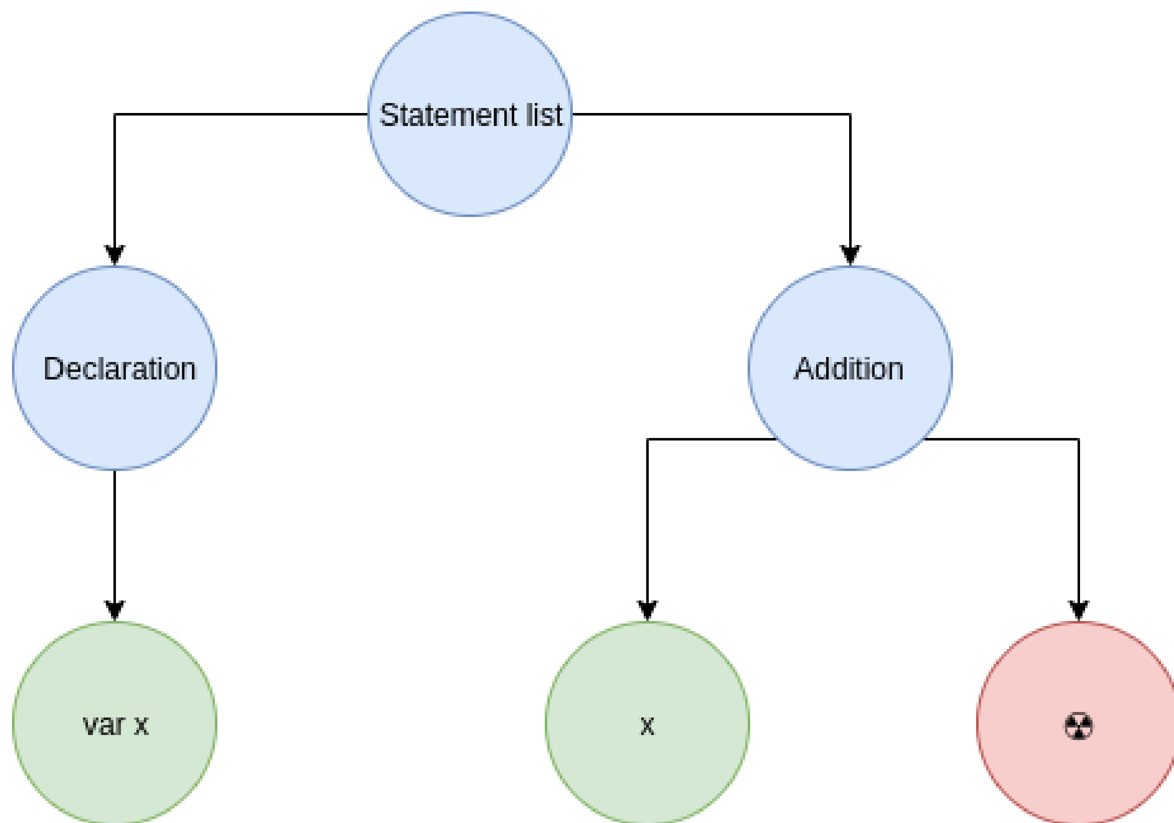


Input: $0x01$, $0x04$, $0x00$, $0x17$

Result: $23 + 0$

Fluff – corruptions

- Modifications of a constructed test case
- Introduce unexpected constructions
- Introduce syntax errors
- Inject runtime errors



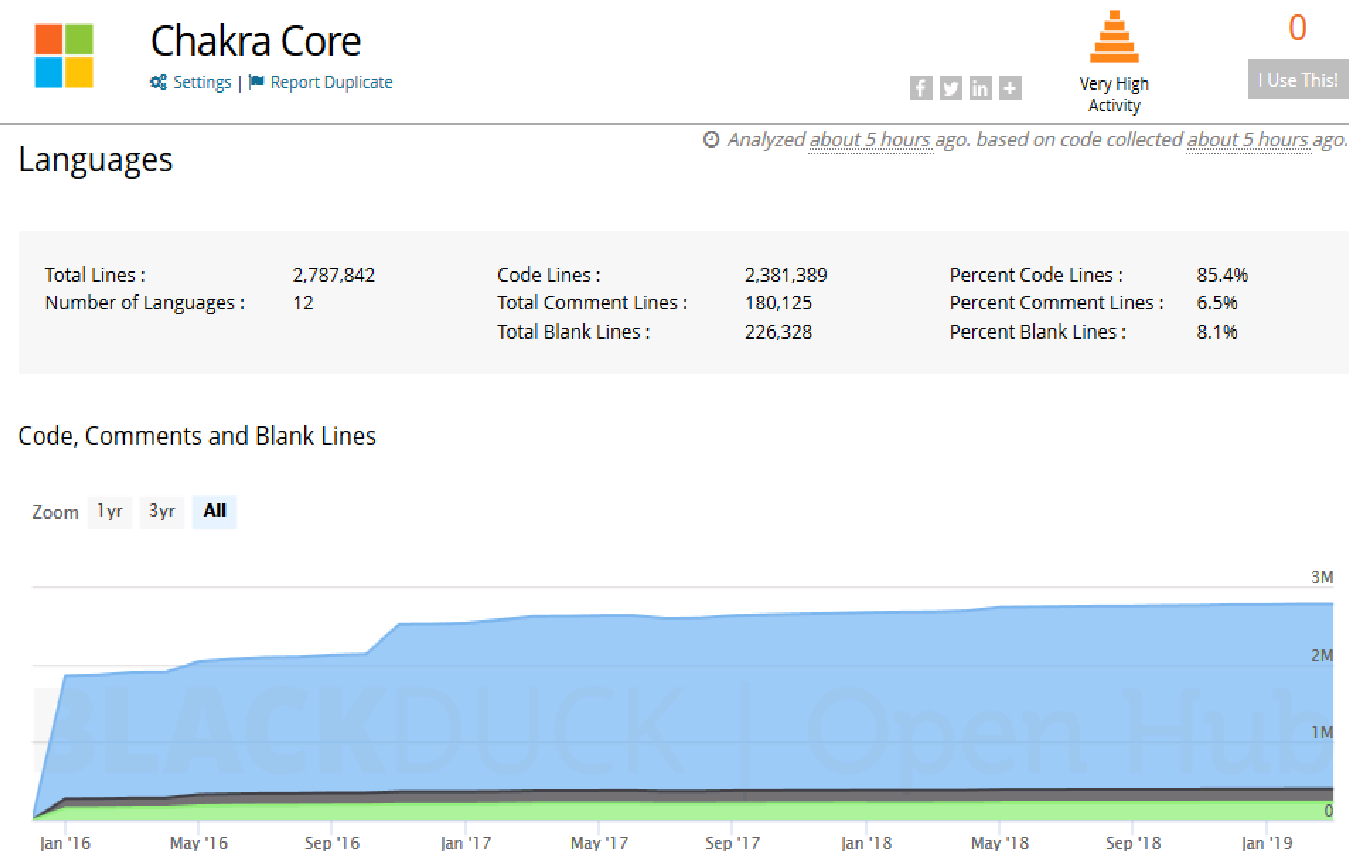
Results & Evaluation

Evaluation

- Based on ”Evaluating fuzz testing” by Klees et al.
- Performance metric
  - Ideal – crashes found/time
  - Realistic – code coverage
  - Running for 25 hours
- Dealing with randomness of fuzz testing
  - 24 independent instances of each fuzzer
  - Mann-Whitney U test

Evaluation – fuzzers for comparison and target

- Fuzzers for comparison
  - American Fuzzy Lop v2.52b
  - AFL with JS dictionary
  - grammarinator
- Target selection: ChakraCore
  - Modern, relevant, ES2015
  - Large codebase
  - Used in Microsoft Edge
  - Easily compiled and embeddable



Black Duck Open Hub – ChakraCore (march 2019)

Evaluation methodology

- Two phases of testing
  1) Fuzz target with AFL coverage
  2) Calculate coverage based on interesting test cases found by AFL

Evaluation methodology

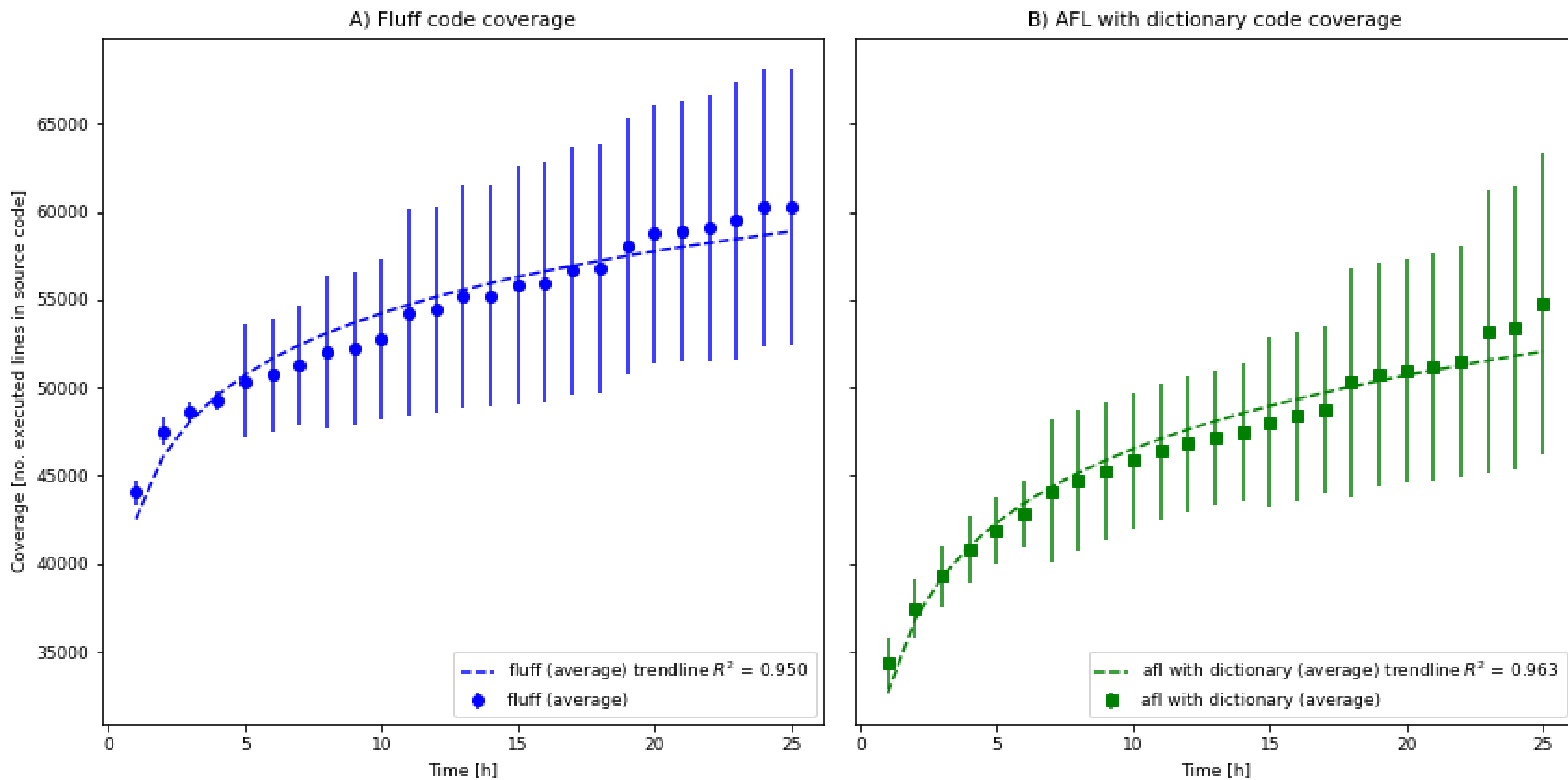
- Two phases of testing
  1) Fuzz target with AFL coverage
  2) Calculate coverage based on interesting test cases found by AFL
- Issue with grammarinator – no AFL integration
  - Approx. number of test cases executed by other fuzzers: 3.5m
  - Our approach
    - Loop
      - Generate test cases and minimize them in **afl-cmin**
      - Merge the result with new test cases generated by grammarinator
    - Calculate coverage from all collected test cases

Evaluation results

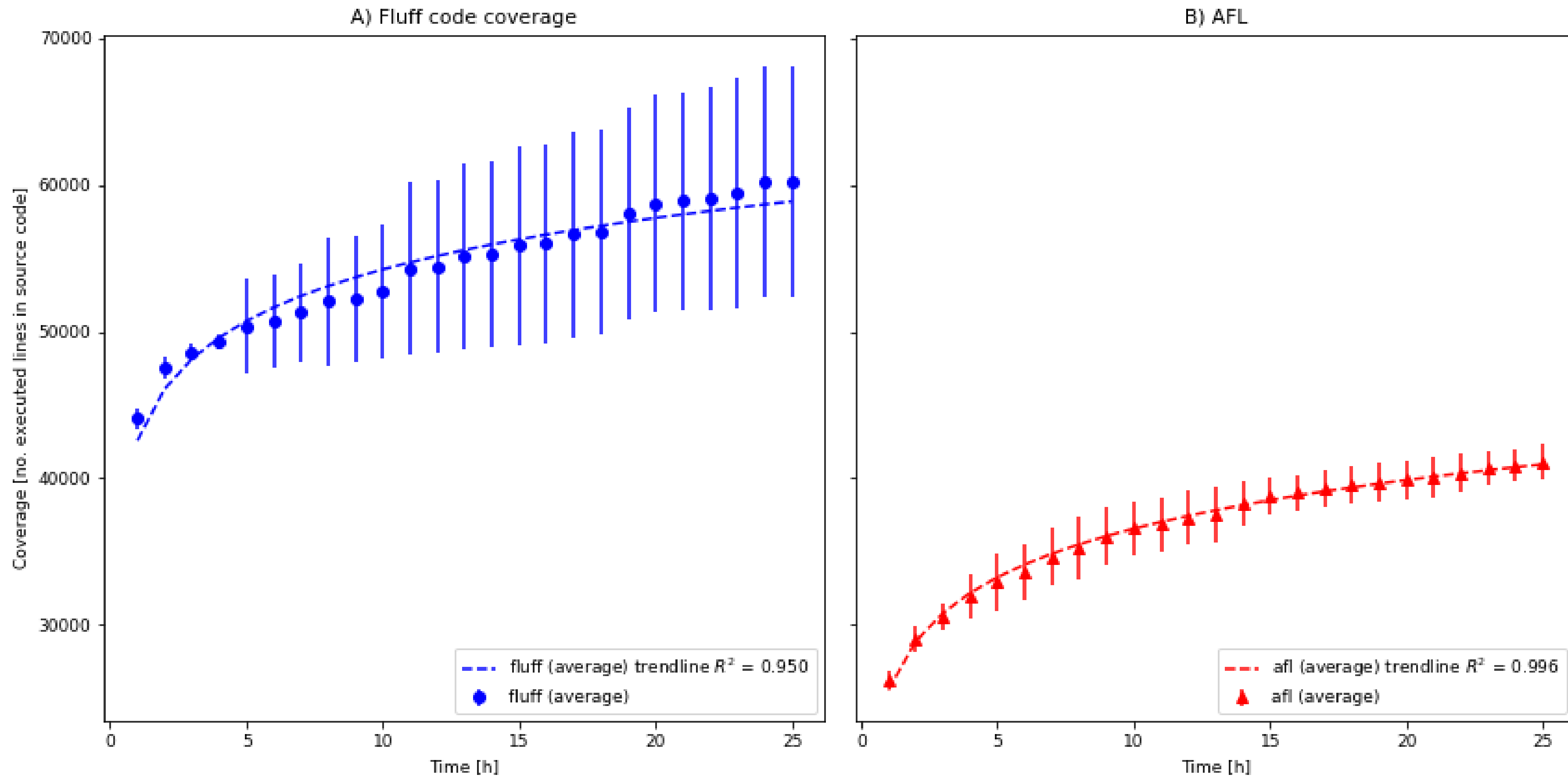
- After 25 hours of fuzzing
    - Fluff 1.0 was ahead with coverage
    - AFL 2.52b with dictionary slightly behind
    - No crashes found by any fuzzer
    - p-value: 0.001

	Mean code coverage (lines)	Mean code coverage (%)	Std. deviation
Fluff 1.0	60282	27	7874
AFL 2.52b with dictionary	54744	24	8563
grammarinator 18.10	42741	18	463
AFL 2.52b	41094	18	1174

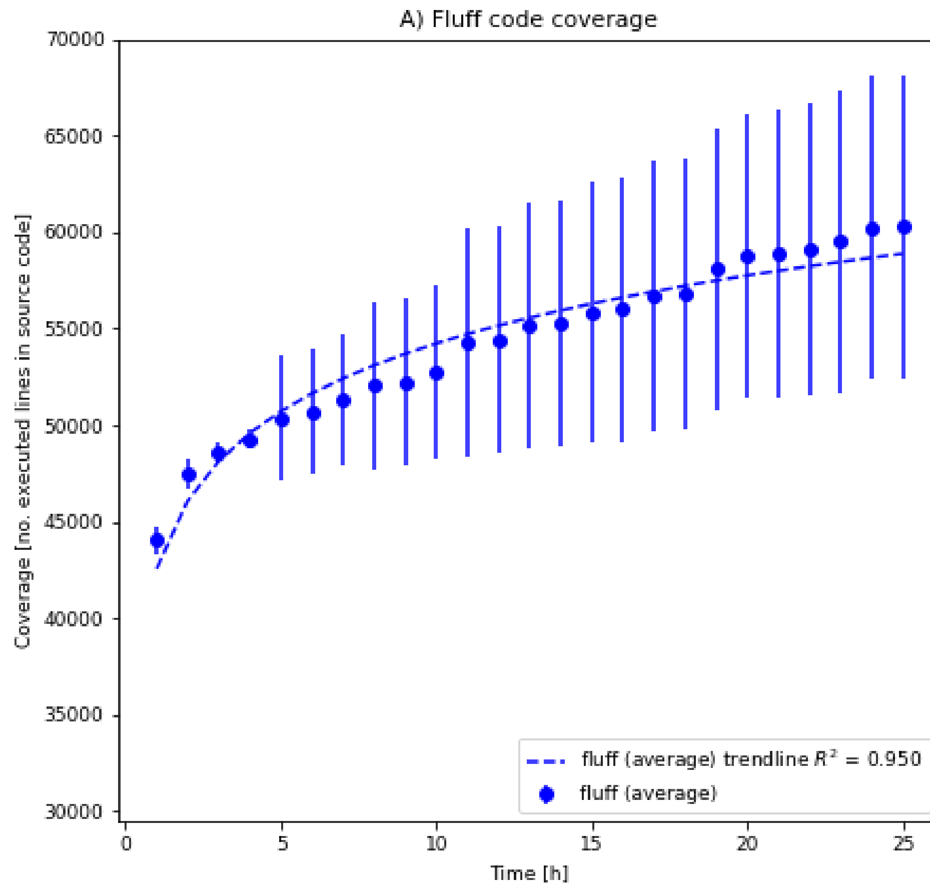
Results

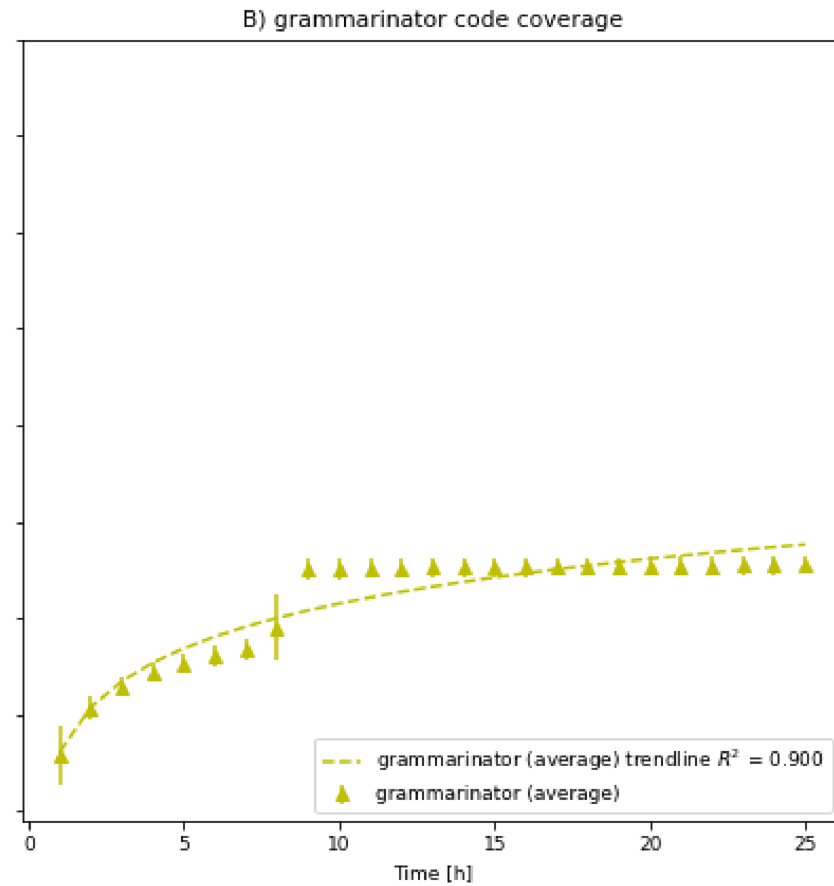


Results



Results





Conclusions

- High coverage helps – not the only factor
- Every fuzzer has advantages
  - AFL tested parser/lexer
  - grammarinator utilized existing JS ANTLR v4 grammar
  - Fluff fuzzed backend
- Fuzzing large codebases takes time

Fuzzing strategies should utilize many different approaches working together!!!

Fluff – crashes found

- Running Fluff in parallel manner on various JS Engines
- Results:
  - 24 bugs found in various JS Engines: jerryscript, Espruino, jsish, ChakraCore
  - CVE-2018-1000636
  - CVE-2018-1000655
  - CVE-2018-1000661
  - CVE-2018-1000663
  - CVE-2018-1000668

Fluff – crashes found

- Running Fluff in parallel manner on various JS Engines
- Results:
  - 24 bugs found in various JS Engines: jerryscript, Espruino, jsish, ChakraCore
  - CVE-2018-1000636
  - CVE-2018-1000655
  - CVE-2018-1000661
  - CVE-2018-1000663
  - CVE-2018-1000668

- Types of bugs found:
  - Uncontrolled recursion
  - Uncontrolled resource consumption
  - **Null pointer dereference**
  - **Heap buffer overflow**
  - **Buffer overread**
  - **Buffer overwrite**
  - Memory leak
  - Assertion reachable

Fluff – sample findings

A bug found in the jerryscript:

```
for (v0=new Uint16Array(function v1() {});
(v0);
v0.filter((parseInt), ++((parseInt)), ++((parseInt))(parseInt), (String))) {};
```

afl-tmin

```
(new Int8Array(0)).filter(parseInt)
```

```
ASAN:SIGSEGV
=================================================================
==7815==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000004
(pc 0x000000402bc1 bp 0x000000000193 sp 0x7fff3cb1ba80 T0)
    #0 0x402bc0 in jmem_heap_free_block jerry-core/jmem/jmem-heap.c:463
    [...]
```

Fluff – sample findings

It’s not always this pretty – Espruino crash

```
while (new Float64Array((((Serial4)|new ArrayBuffer(new HASH(E()))-(new
ArrayBuffer(((InternalError)=E.bind(((InternalError)-new
ArrayBuffer(setWatch())), (D24), (encodeURIComponent))))-
(((E.apply((((E.replaceWith((InternalError))*new
ArrayBuffer(setWatch()))/(I2C))==(SPI)), (true+((quit)+=false)), setWatch())-
(D24))+(edit))+((InternalError)=E.apply((E.replaceWith((InternalError))*new
ArrayBuffer(setWatch())), (I2C), (HIGH)))))))) { } ;
if (((quit)+=false)) { }
else { } ;
```

Fluff – sample findings

CVE-2014-1705 (by geohot) written in ”Fluff bytecode”

```
f000 041b 0301 0400 0800 0111 0003 0204
030a 6279 7465 4c65 6e67 7468 1800 0100
0510 0102 0403 0a30 7846 4646 4646 4646
4304 0010 0000 0004 1b02 0105 030a
```

Fluff

```
var v0=new ArrayBuffer(8)
v0.__defineGetter__(”byteLength”, funtion v1() {
  return parseInt(”0xFFFFFFFC”, 16);
})
var v2=new Uint32Array((v0))
```

Future Work

- Fuzzing more JS interpreters
  - V8
  - SpiderMonkey
  - mJS
- Implementations for other languages
- Language agnostic implementation
  - Interpreters
  - Compilers
- Continuous Fuzzing
- Further integration with AFL

Black Hat Sound Bytes

Key takeaways

1. Generate semantically correct test cases in your fuzzing strategy
2. Fluff – new approach to fuzzing interpreters
3. Use novel methods to fuzz targets which were traditionally difficult to test

Efficient Approach to Fuzzing Interpreters

Q&A

Efficient Approach to Fuzzing Interpreters

Thank you