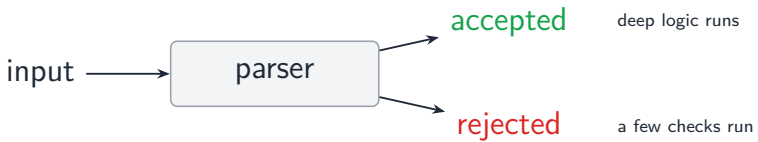
L16: Grammar-Based Fuzzing

Software Analysis

IIT Guwahati · September 2026

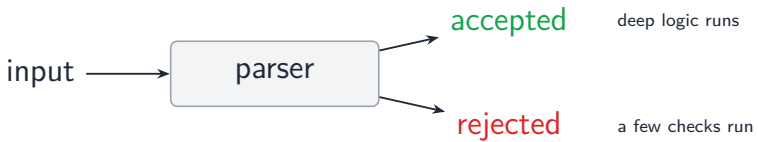
The Problem with Random Fuzzing




```
123 + 45 * 7          <- accepted: evaluation
x$#@91!!              <- rejected at the first character
```

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

The Problem with Random Fuzzing

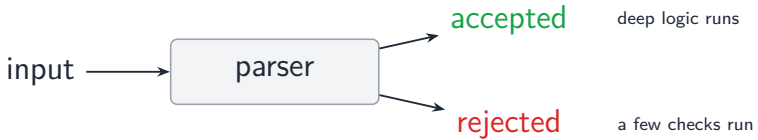



```
123 + 45 * 7          <- accepted: evaluation
x$#@91!!              <- rejected at the first character
```

Question: what fraction of uniformly random bytes will reach the interesting parser logic?

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

The Problem with Random Fuzzing




```
123 + 45 * 7          <- accepted: evaluation
x$#@91!!              <- rejected at the first character
```

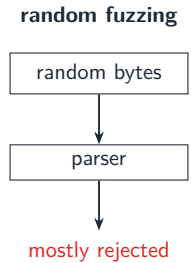
Question: what fraction of uniformly random bytes will reach the interesting parser logic?

Practically none. Random fuzzers are efficient but ineffective, because "the large majority of inputs generated is syntactically invalid".

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

What Do We Want?

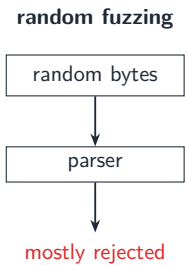
Generate **inputs** that obey the target's structure, while still exploring a huge input space.

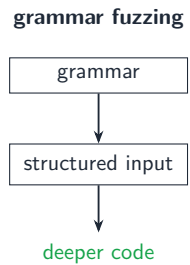


Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

What Do We Want?

Generate **inputs** that obey the target's structure, while still exploring a huge input space.





Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

Background: Languages and Grammars

alphabet	the symbols an input is made of — bytes, characters, tokens
string	a finite sequence of those symbols
language	a *set* of strings: the ones the program accepts

Two ways to describe such a set:

regular expression

```
[0-9]+ ( "." [0-9]+ )?
```

a number: 7, 42, 3.14

grammar

```
EXPR -> NUMBER
| EXPR + NUMBER
| ( EXPR )
NUMBER -> DIGIT | DIGIT NUMBER
DIGIT -> 0 | 1 | 2 | ...  | 9
```

an expression, which may contain another expression

Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

Background: Languages and Grammars

alphabet	the symbols an input is made of — bytes, characters, tokens
string	a finite sequence of those symbols
language	a *set* of strings: the ones the program accepts

Two ways to describe such a set:

regular expression

```
[0-9]+ ( "." [0-9]+ )?
```

a number: 7, 42, 3.14

grammar

```
EXPR -> NUMBER
| EXPR + NUMBER
| ( EXPR )
NUMBER -> DIGIT | DIGIT NUMBER
DIGIT -> 0 | 1 | 2 | ...  | 9
```

an expression, which may contain another expression

A regular expression describes **flat** structure — what a token looks like. A grammar describes **nested** structure — how parts contain other parts.

Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

Background: Grammar

A grammar is a set of rules for constructing valid strings.

```
EXPR -> NUMBER
EXPR -> EXPR + NUMBER
```

Strings this grammar can build:

```
1          12          1 + 2          42 + 17          1 + 2 + 3
```

Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

Background: Grammar

Concept	Meaning
terminal	a symbol that appears in the input itself
nonterminal	a symbol standing for a piece of structure
production	a rule for replacing a nonterminal
start symbol	the nonterminal generation begins with

```
EXPR -> EXPR + NUMBER
```

Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

Background: Grammar

Concept	Meaning
terminal	a symbol that appears in the input itself
nonterminal	a symbol standing for a piece of structure
production	a rule for replacing a nonterminal
start symbol	the nonterminal generation begins with

```
EXPR -> EXPR + NUMBER
```

`EXPR` and `NUMBER` are **nonterminals**; `+` is a **terminal**; the line itself is a **production**; `EXPR` is the **start symbol**.

Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

Derivation: Generating Strings from a Grammar

```
EXPR
  |
EXPR + NUMBER              EXPR -> EXPR + NUMBER
  |
NUMBER + NUMBER            EXPR -> NUMBER
  |
12 + 45                    NUMBERs expanded to 12 and 45
```

Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

Derivation: Generating Strings from a Grammar

```
EXPR
  |
EXPR + NUMBER              EXPR -> EXPR + NUMBER
  |
NUMBER + NUMBER            EXPR -> NUMBER
  |
12 + 45                    NUMBERs expanded to 12 and 45
```

Start from an abstract description and repeatedly expand until only terminals remain. The sequence of steps is a **derivation**.

Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

Recursion

```
EXPR -> NUMBER
EXPR -> EXPR + NUMBER
EXPR -> ( EXPR )
```

```
EXPR  ->  ( EXPR )  ->  ( EXPR + NUMBER )  ->  ( 12 + 45 )
```

Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

Recursion

```
EXPR -> NUMBER
EXPR -> EXPR + NUMBER
EXPR -> ( EXPR )
```

```
EXPR  ->  ( EXPR )  ->  ( EXPR + NUMBER )  ->  ( 12 + 45 )
```

Question: why can this grammar generate arbitrarily deep structures?

Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

Recursion

```
EXPR -> NUMBER
EXPR -> EXPR + NUMBER
EXPR -> ( EXPR )
```

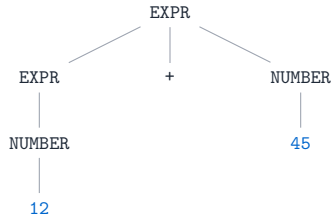
```
EXPR  ->  ( EXPR )  ->  ( EXPR + NUMBER )  ->  ( 12 + 45 )
```

Question: why can this grammar generate arbitrarily deep structures?

Because `EXPR` can expand to something that contains `EXPR` again — **recursion**.

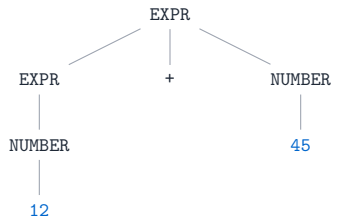
Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

Parse Tree: Structure



Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

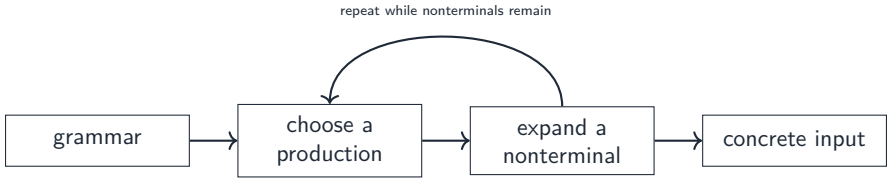
Parse Tree: Structure



A grammar does not only describe strings; it describes their **structure**. The leaves, read left to right, give 12 + 45.

Aho, Lam, Sethi & Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed., ch. 4

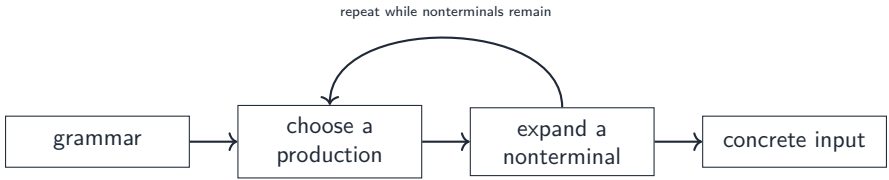
A Grammar Can Drive a Generator




```
EXPR  ->  EXPR + NUMBER  ->  NUMBER + NUMBER  ->  12 + 45
```

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

A Grammar Can Drive a Generator




```
EXPR  ->  EXPR + NUMBER  ->  NUMBER + NUMBER  ->  12 + 45
```

A grammar can be used directly to **generate valid inputs**: start from the grammar and repeatedly expand its nonterminals.

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

Random Production Selection

```
EXPR -> NUMBER
      | EXPR + NUMBER
      | EXPR - NUMBER
      | ( EXPR )
```

At each step the fuzzer picks one alternative at random:

```
EXPR
  ->  ( EXPR )
  ->  ( EXPR + NUMBER )
  ->  ( ( EXPR ) + NUMBER )
  ->  ( ( 12 ) + 45 )
```

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

Random Production Selection

```
EXPR -> NUMBER
      | EXPR + NUMBER
      | EXPR - NUMBER
      | ( EXPR )
```

At each step the fuzzer picks one alternative at random:

```
EXPR
  ->  ( EXPR )
  ->  ( EXPR + NUMBER )
  ->  ( ( EXPR ) + NUMBER )
  ->  ( ( 12 ) + 45 )
```

Fuzzing with a grammar is **random exploration of the derivation space**: the grammar fixes what is legal, the random choices decide which legal input appears.

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

Byte-Level versus Grammar-Level Mutation

Start from a valid input:

```
1 + 2 * 3
```

Byte mutation changes one byte and usually destroys the input:

```
1 + X * 3          rejected by the lexer
```

Byte-Level versus Grammar-Level Mutation

Start from a valid input:

```
1 + 2 * 3
```

Byte mutation changes one byte and usually destroys the input:

```
1 + X * 3          rejected by the lexer
```

Structure-aware mutation changes a subtree and keeps it valid:

```
1 + ( 2 * 3 )      still an expression
( 1 + 2 ) * 3      still an expression, different structure
```

Byte-Level versus Grammar-Level Mutation

Start from a valid input:

```
1 + 2 * 3
```

Byte mutation changes one byte and usually destroys the input:

```
1 + X * 3          rejected by the lexer
```

Structure-aware mutation changes a subtree and keeps it valid:

```
1 + ( 2 * 3 )      still an expression
( 1 + 2 ) * 3      still an expression, different structure
```

The second kind of mutation knows that `2 * 3` is an expression, and that an expression may be replaced by another expression.

Valid Is Not the Same as Interesting

A grammar fuzzer will happily produce

```
123 + 456
```

a thousand times over. Validity alone tests very little. What we want as well:

very deep nesting	`(((((  1  )))))`
very long tokens	a number with 500 digits
empty constructs	an empty argument list
boundary values	0, −1, 2147483647
unusual combinations	`+-+-+1`
near-valid structures	one parenthesis missing

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

Valid Is Not the Same as Interesting

A grammar fuzzer will happily produce

```
123 + 456
```

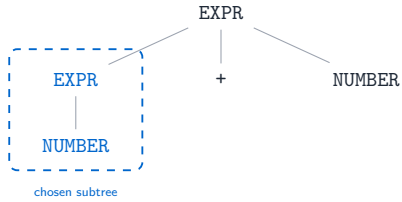
a thousand times over. Validity alone tests very little. What we want as well:

very deep nesting	`(((((  1  )))))`
very long tokens	a number with 500 digits
empty constructs	an empty argument list
boundary values	0, −1, 2147483647
unusual combinations	`+-+-+1`
near-valid structures	one parenthesis missing

valid structure + **interesting variation**

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

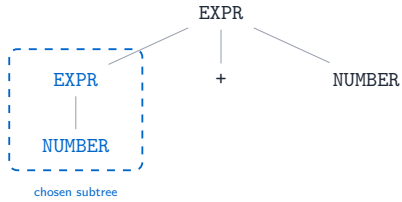
Grammar-Based Mutation



Replace it by *any* derivation of the same nonterminal:

```
NUMBER  ->  ( EXPR )        or        EXPR  ->  EXPR * NUMBER
```

Grammar-Based Mutation

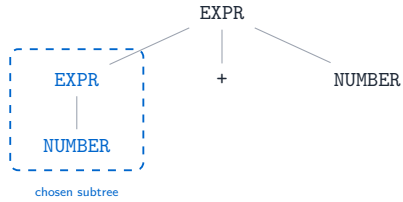


Replace it by *any* derivation of the same nonterminal:

```
NUMBER  ->  ( EXPR )        or        EXPR  ->  EXPR * NUMBER
```

then serialise the tree back to bytes. The result is a different input that is still **valid by construction**.

Grammar-Based Mutation



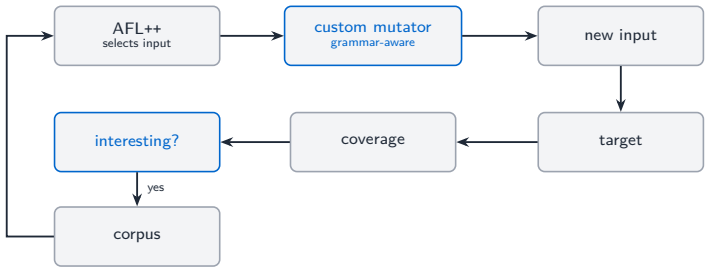
Replace it by *any* derivation of the same nonterminal:

```
NUMBER  ->  ( EXPR )        or        EXPR  ->  EXPR * NUMBER
```

then serialise the tree back to bytes. The result is a different input that is still **valid by construction**.

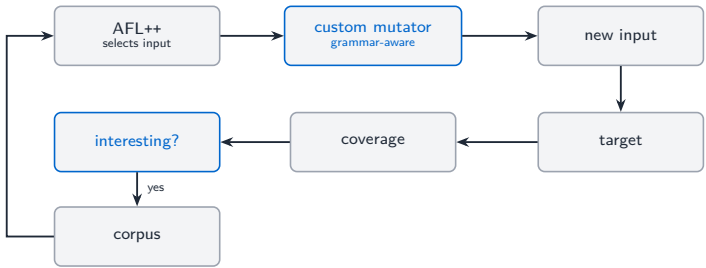
Byte mutation asks “what if this character were different?”; structural mutation asks “what if this *expression* were a different expression?”.

AFL++ Custom Mutator



AFL++, *Custom Mutators* documentation

AFL++ Custom Mutator



the **grammar**	how to construct a structured input
the **coverage**	which inputs are worth keeping

AFL++, *Custom Mutators* documentation

The Custom-Mutator Hook

AFL++ calls your code instead of its own mutator:

```
queue entry  ->  afl_custom_fuzz()  ->  new input  ->  target  ->  coverage
```

AFL++, *Custom Mutators* documentation

The Custom-Mutator Hook

AFL++ calls your code instead of its own mutator:

```
queue entry  ->  afl_custom_fuzz()  ->  new input  ->  target  ->  coverage
```

- written in **C/C++** (a shared object) or in **Python**; Rust support is experimental;

AFL++, *Custom Mutators* documentation

The Custom-Mutator Hook

AFL++ calls your code instead of its own mutator:

```
queue entry  ->  afl_custom_fuzz()  ->  new input  ->  target  ->  coverage
```

- written in **C/C++** (a shared object) or in **Python**; Rust support is experimental;
- loaded through `AFL_CUSTOM_MUTATOR_LIBRARY` or `AFL_PYTHON_MODULE`; several may be chained.

AFL++, *Custom Mutators* documentation

The Custom-Mutator Hook

AFL++ calls your code instead of its own mutator:

```
queue entry  ->  afl_custom_fuzz()  ->  new input  ->  target  ->  coverage
```

- written in **C/C++** (a shared object) or in **Python**; Rust support is experimental;
- loaded through `AFL_CUSTOM_MUTATOR_LIBRARY` or `AFL_PYTHON_MODULE`; several may be chained.
- `AFL_CUSTOM_MUTATOR_ONLY` switches off AFL++'s own mutations entirely.

AFL++, *Custom Mutators* documentation

Example: a Tiny Grammar Fuzzer

```
EXPR   -> NUMBER
        | EXPR + NUMBER
        | EXPR * NUMBER
        | ( EXPR )

NUMBER -> DIGIT | DIGIT NUMBER

DIGIT  -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Task: generate random expressions from this grammar. Expected shape of the output:

```
7          42          1 + 93          (7 + 3)          ((4 * 8) + 12)
```

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

Then Add Limits

Question: what goes wrong if the recursive alternatives are chosen with the same probability as the others?

```
((((((((((((((((((((((((((((((((((((((( ...          may never terminate
1 + 2 + 7 + 4 + 0 + 9 + ... (3000 characters)       useless test case
```

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

Then Add Limits

Question: what goes wrong if the recursive alternatives are chosen with the same probability as the others?

```
((((((((((((((((((((((((((((((((((((((( ...          may never terminate
1 + 2 + 7 + 4 + 0 + 9 + ... (3000 characters)       useless test case
```

Three standard remedies:

- a **maximum depth** — at the limit, choose only productions that terminate.

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

Then Add Limits

Question: what goes wrong if the recursive alternatives are chosen with the same probability as the others?

```
((((((((((((((((((((((((((((((((((((((( ...          may never terminate
1 + 2 + 7 + 4 + 0 + 9 + ... (3000 characters)       useless test case
```

Three standard remedies:

- a **maximum depth** — at the limit, choose only productions that terminate.
- a **maximum length** — abandon and restart when the string grows past it.

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

Then Add Limits

Question: what goes wrong if the recursive alternatives are chosen with the same probability as the others?

```
((((((((((((((((((((((((((((((((((((((( ...          may never terminate
1 + 2 + 7 + 4 + 0 + 9 + ... (3000 characters)       useless test case
```

Three standard remedies:

- a **maximum depth** — at the limit, choose only productions that terminate.
- a **maximum length** — abandon and restart when the string grows past it.
- **production probabilities** — make the recursive alternatives less likely than the terminating ones.

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

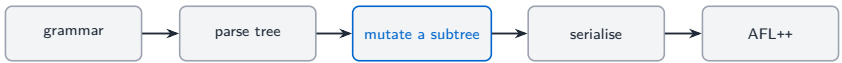
Then Add Mutation

Start from an input that is already in the corpus:

```
(12 + 7) * 3
```

Parse it, choose a subtree, replace it, serialise:

```
subtree chosen:      12 + 7
replaced by:         999 + 1
result:              (999 + 1) * 3
```



Then Add Mutation

Start from an input that is already in the corpus:

```
(12 + 7) * 3
```

Parse it, choose a subtree, replace it, serialise:

```
subtree chosen:      12 + 7
replaced by:         999 + 1
result:              (999 + 1) * 3
```

In AFL++ terms: parse and mutate inside `afl_custom_fuzz`, serialise there or in `afl_custom_post_process`.

AFL++, *Custom Mutators* documentation

An Experiment You Can Run

Same target, same time budget, same machine:

	A: byte-level	**B: grammar mutator**
executions per second		
edge coverage		
unique paths		
crashes		
time to first crash		
corpus size		

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

An Experiment You Can Run

Same target, same time budget, same machine:

	A: byte-level	**B: grammar mutator**
executions per second		
edge coverage		
unique paths		
crashes		
time to first crash		
corpus size		

Run each configuration several times with different seeds: a single run of a randomised tool tells you very little.

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019

An Experiment You Can Run

Same target, same time budget, same machine:

	A: byte-level	**B: grammar mutator**
executions per second		
edge coverage		
unique paths		
crashes		
time to first crash		
corpus size		

Run each configuration several times with different seeds: a single run of a randomised tool tells you very little.

B usually executes *fewer* inputs per second and reaches *more* of the program. Which wins depends on how much of the target sits behind the parser.

Gopinath & Zeller, *Building Fast Fuzzers*, arXiv:1911.07707, 2019