SWE3002-42: Introduction to Software Engineering

Lecture 11 – Grey-Box Testing

Sooyoung Cha

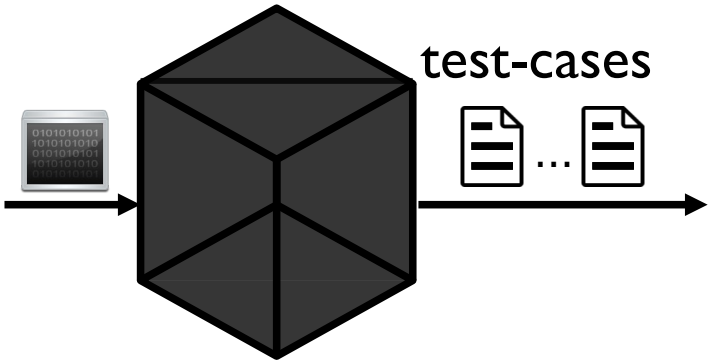
Department of Computer Science and Engineering

Today's Lecture

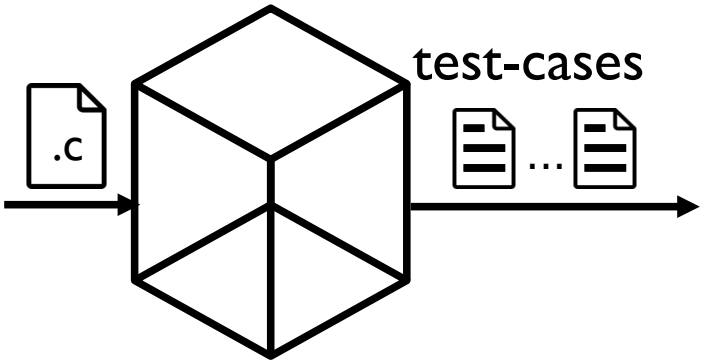
- Grey-Box Testing (Fuzzing)
  - Mutation-based Fuzzing
    - AFL: A representative mutation-based fuzzer
    - Recent trend in mutation-based fuzzing
      - Program-adaptive mutation-based fuzzing
  - Generation-based Fuzzing
    - Grammar generation
    - Test-case generation from grammar

Black-Box vs White-Box Testing

- Difference: with/without using of source code



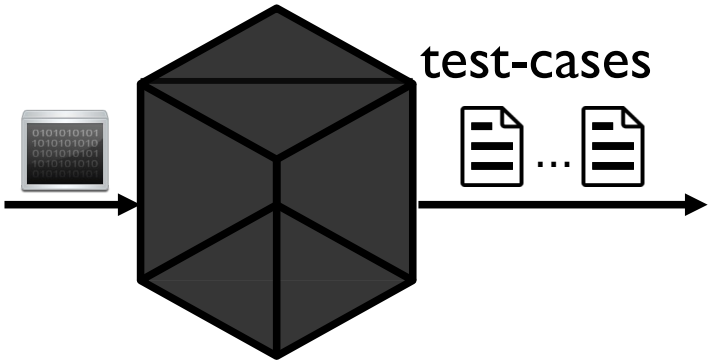
"without source code"
(+) cheap
(-) naive



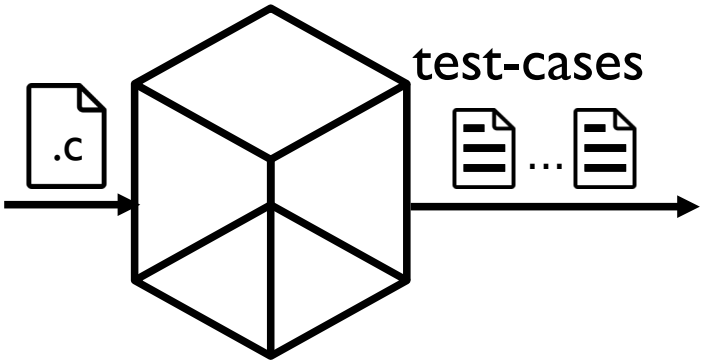
"with source code"
(-) expensive
(+) systematic

Grey-Box Testing (Fuzzing)

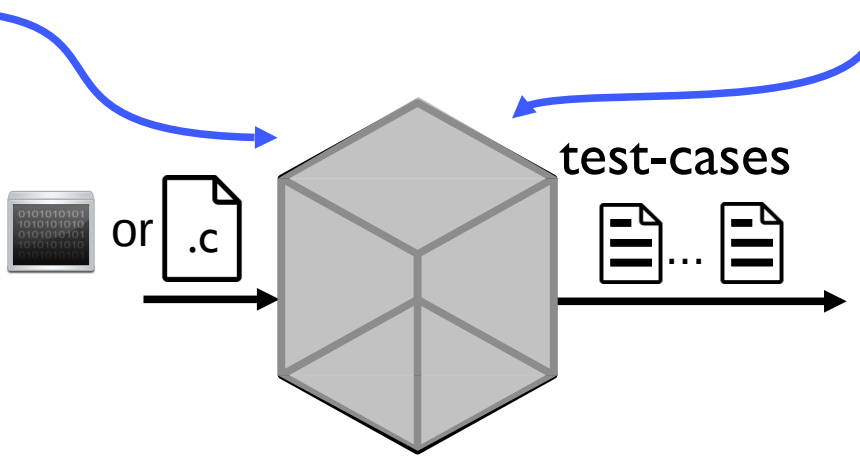
- Include the characteristics of black- and white-box testing.



cheap, but naive



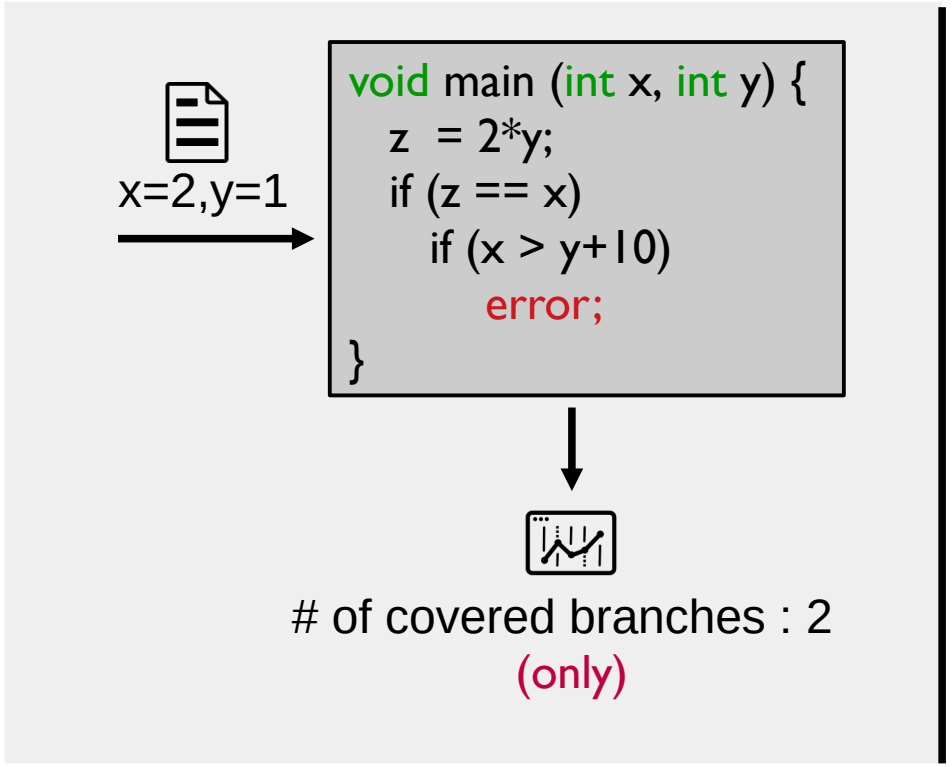
expensive, but systematic

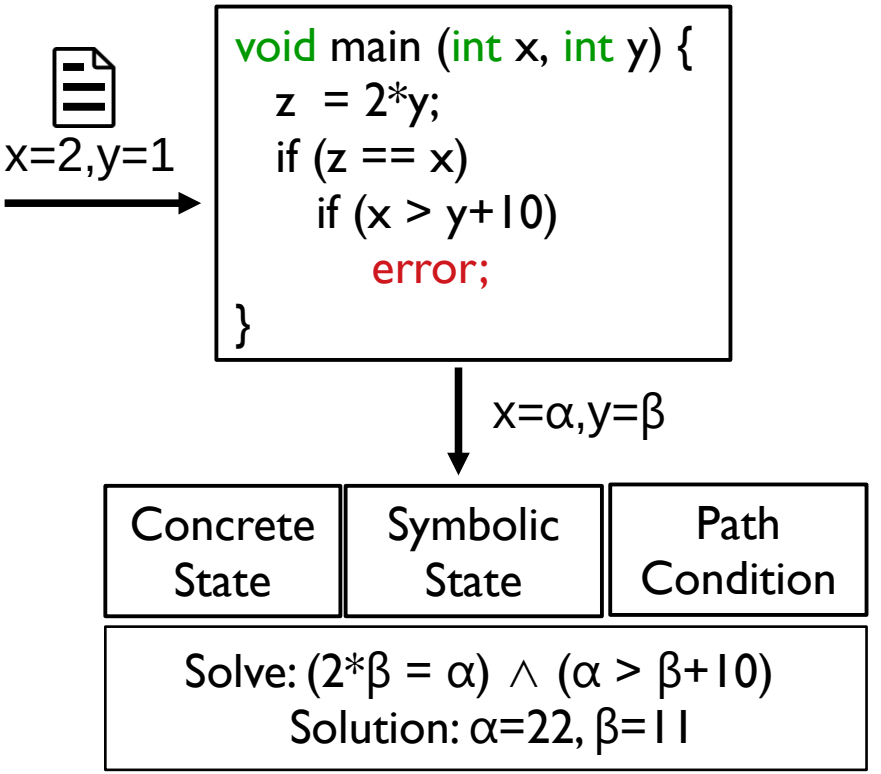


relatively cheap and systematic

Grey-Box vs White-Box Testing

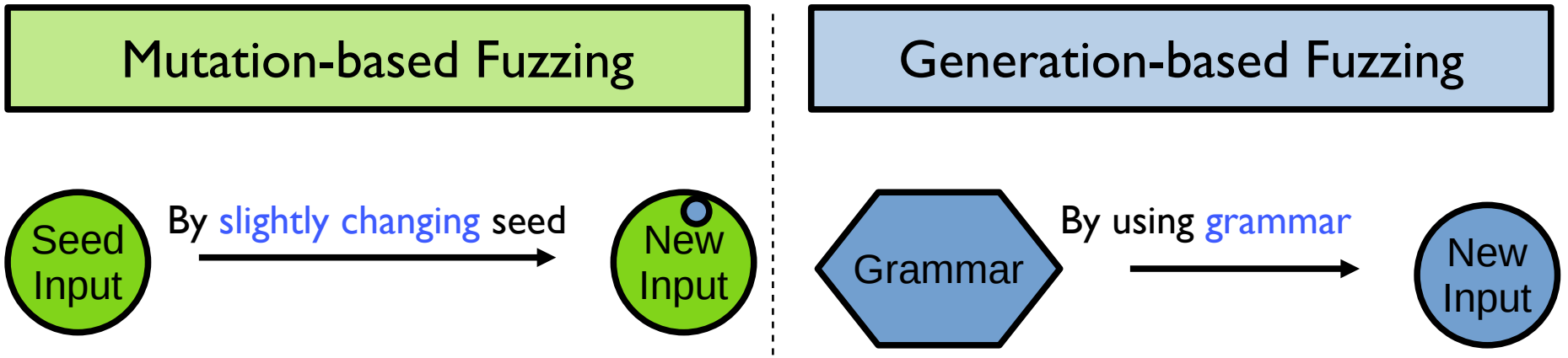
- Key Difference: using source code partially/fully.
- Grey-box Testing
  - Obtaining only partial information from each program execution.





Grey-Box Testing (Fuzzing)

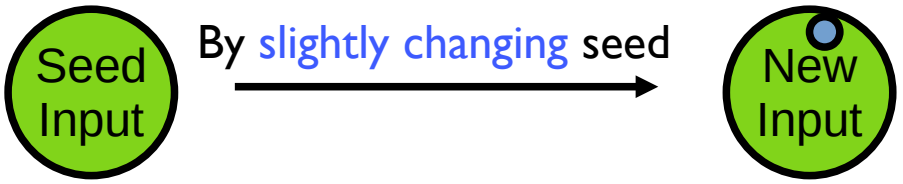
- Categorize into two fuzzing techniques



Grey-Box Testing (Fuzzing)

- Categorize into two fuzzing techniques

Mutation-based Fuzzing



Seed Input = "0 1 0 0 0 0 0 1"

"flip the first bit of seed input"

New Input = "1 1 0 0 0 0 0 1"

Generation-based Fuzzing

Grey-Box Testing (Fuzzing)

- Categorize into two fuzzing techniques

Mutation-based Fuzzing

Seed Input → By slightly changing seed → New Input

Seed Input = "0 1 0 0 0 0 0 1"

"flip the first bit of seed input"

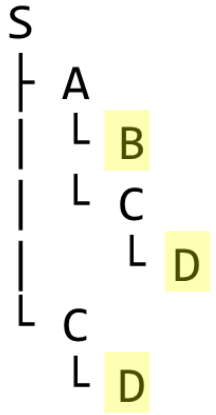
New Input = "1 1 0 0 0 0 0 1"

Generation-based Fuzzing

Grammar → By using grammar → New Input

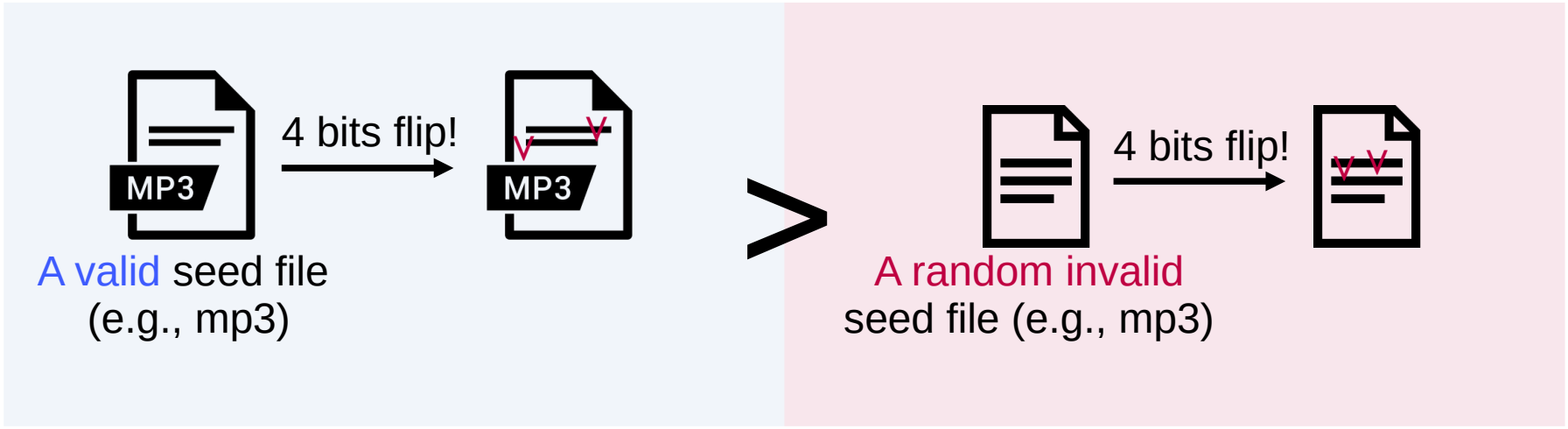
<Grammar>
S → A C
A → B C
C → D | E

New Input = "BDD"



Mutation-based Fuzzing

- Intuition
  - Mutating only a fraction of a "valid" seed may generate good inputs!



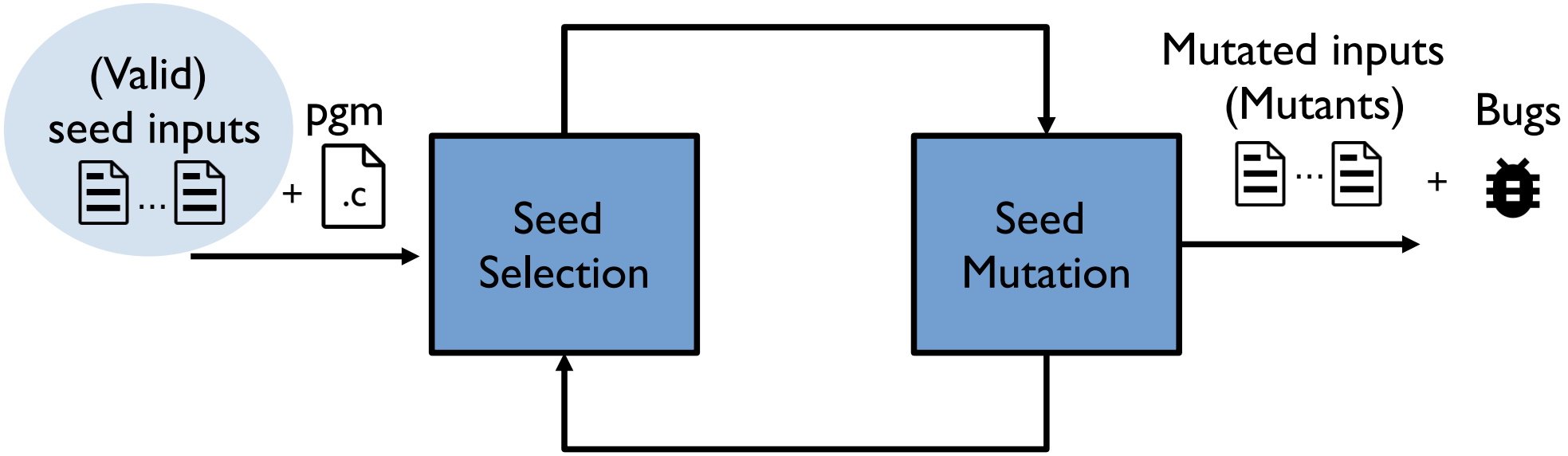
Detail of an MP3 Header

Bits	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Binary	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
Hex	F				F				F				B				A				0				4				0			
Meaning	MP3 Sync Word												Version	Layer		Error Protection	Bit Rate				Frequency		Pad. Bit	Priv. Bit	Mode		Mode Extension (Used With Joint Stereo)		Copy	Original	Emphasis	
Value	Sync Word												1 = MPEG	01 = Layer 3		1 = No	1010 = 160				00 = 44100 Hz		0 = Frame is not padded	Unknown	01 = Joint Stereo		0 = Intensity Stereo Off	0 = MS Stereo Off	0 = Not Copy-righted	0 = Copy Of Original Media	00 = None	

<Mp3 header>

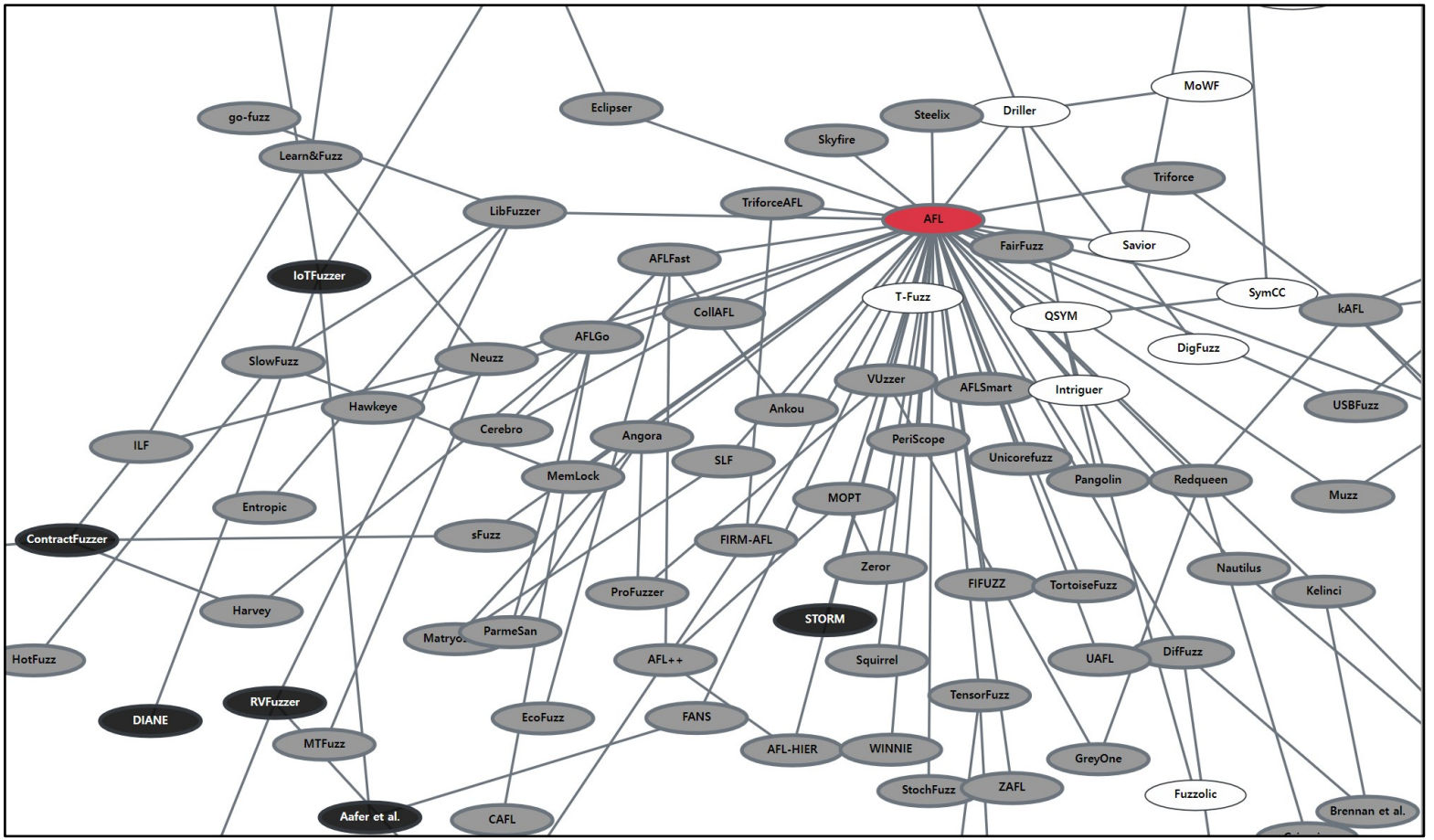
Mutation-based Fuzzing

- Key component
  - Seed Selection & Seed Mutation



AFL (American Fuzzy Lop)

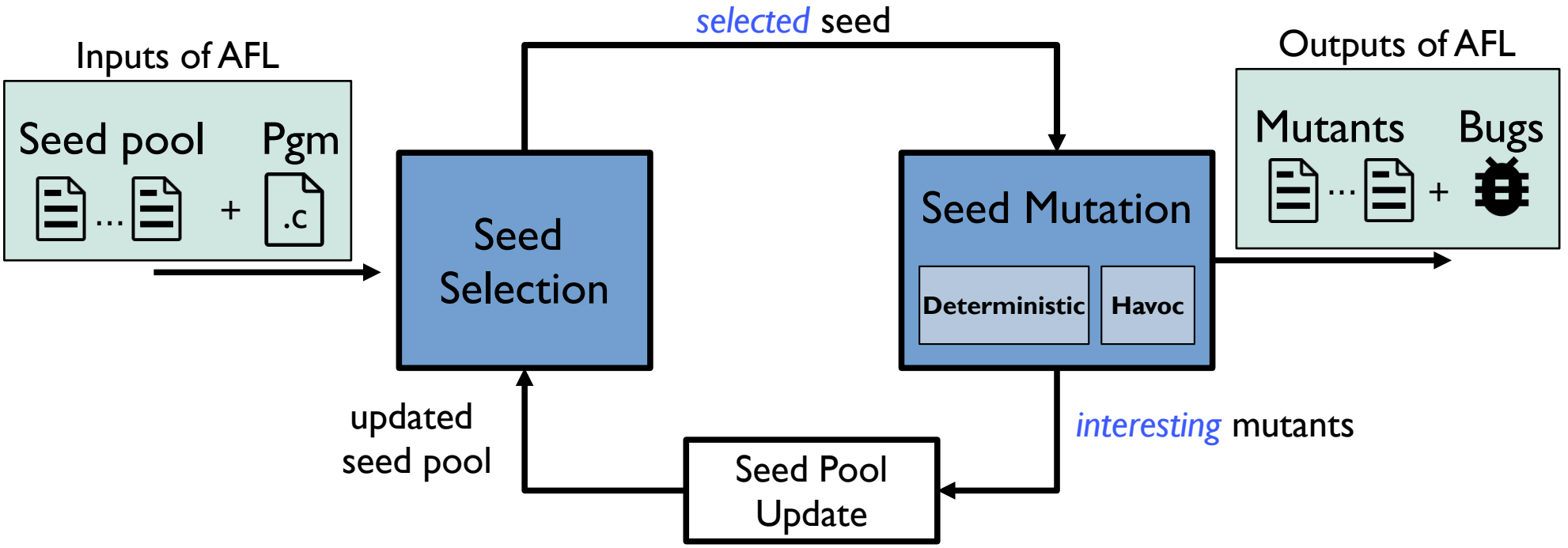
- A representative mutation-based fuzzer
  - Numerous techniques have been implemented on top of AFL[1].



1: https://fuzzing-survey.org/

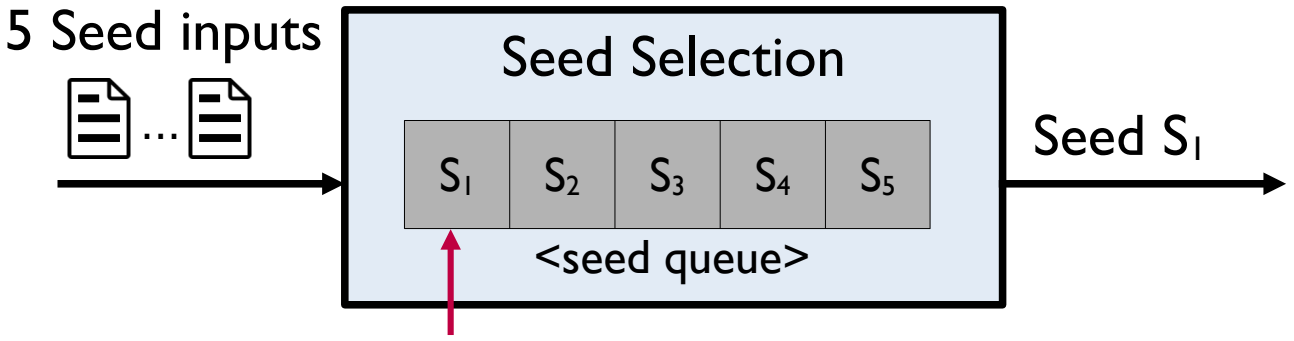
AFL (American Fuzzy Lop)

- Workflow of AFL



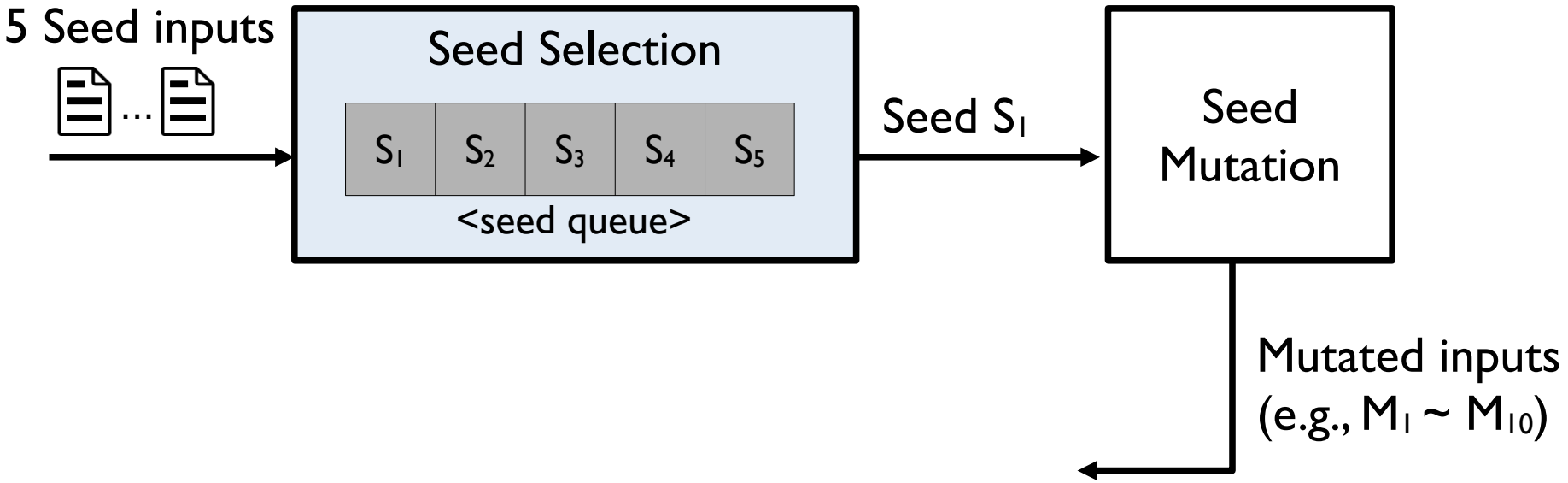
Seed Selection

- Load a set of seed inputs into a queue.
- Sequentially select a seed from the queue.



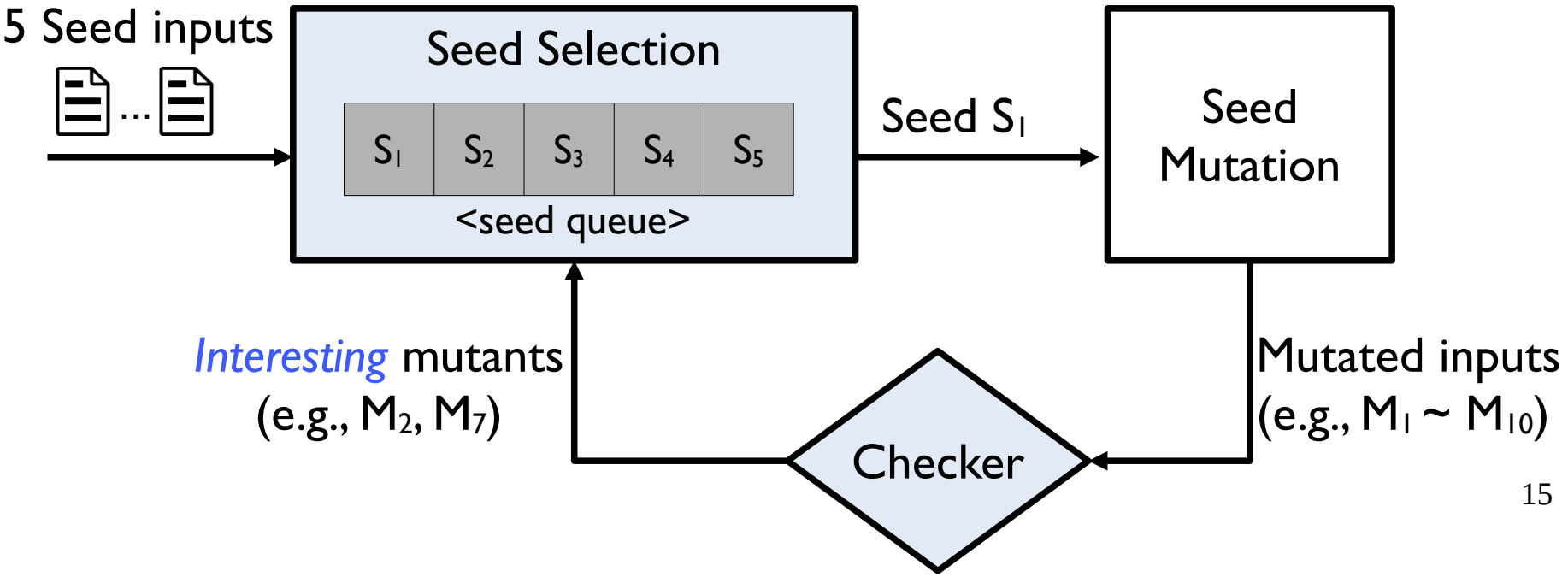
Seed Selection

- Load a set of seed inputs into a queue.
- Sequentially select a seed from the queue.
- Generate the mutated inputs by using the seed.



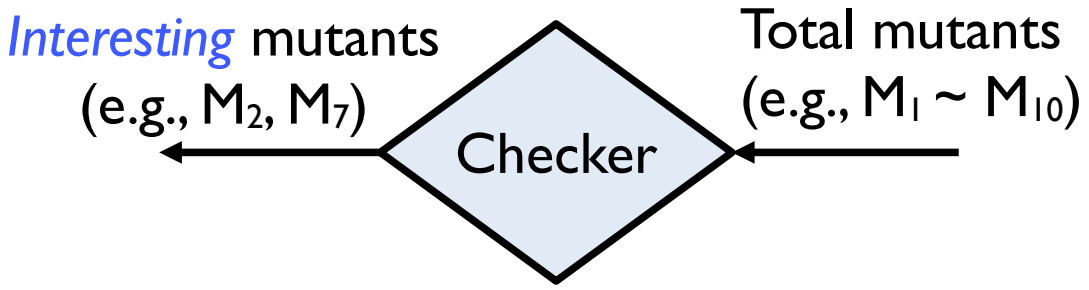
Seed Selection

- Load a set of seed inputs into a queue.
- Sequentially select a seed from the queue.
- Generate the mutated inputs by using the seed.
- Check whether the mutated inputs are interesting



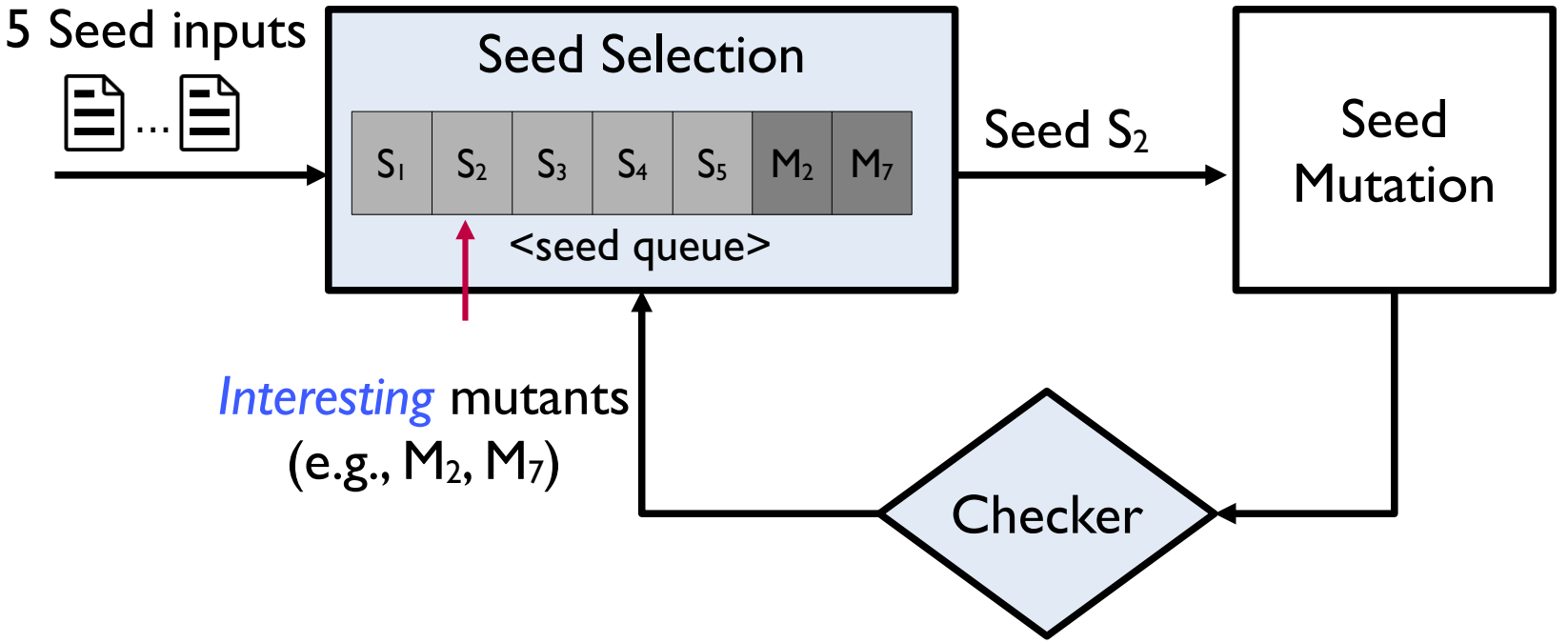
Checker

- Two criteria for interesting inputs.
  - The input causes a crash in the target program.
  - The input covers a new path which has not been covered.



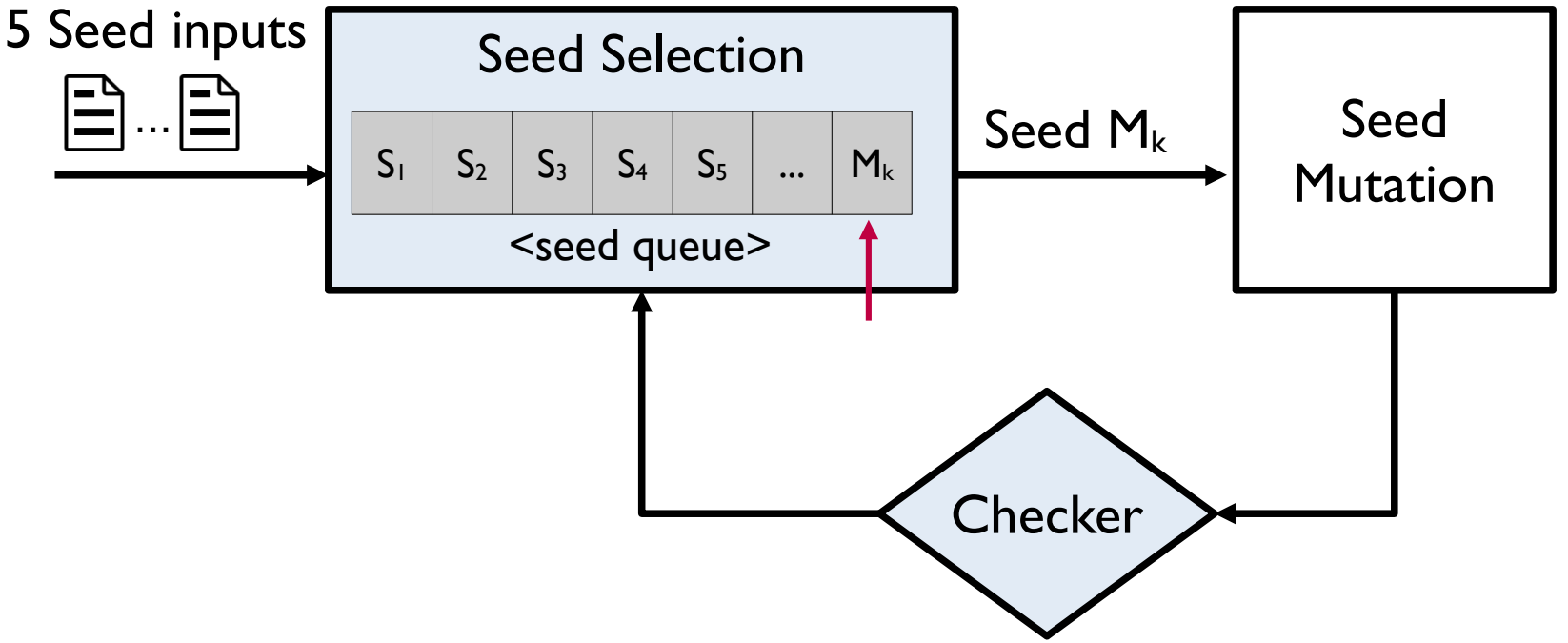
Seed Selection

- Add the interesting inputs into the queue.
- Sequentially select the next seed from the queue.



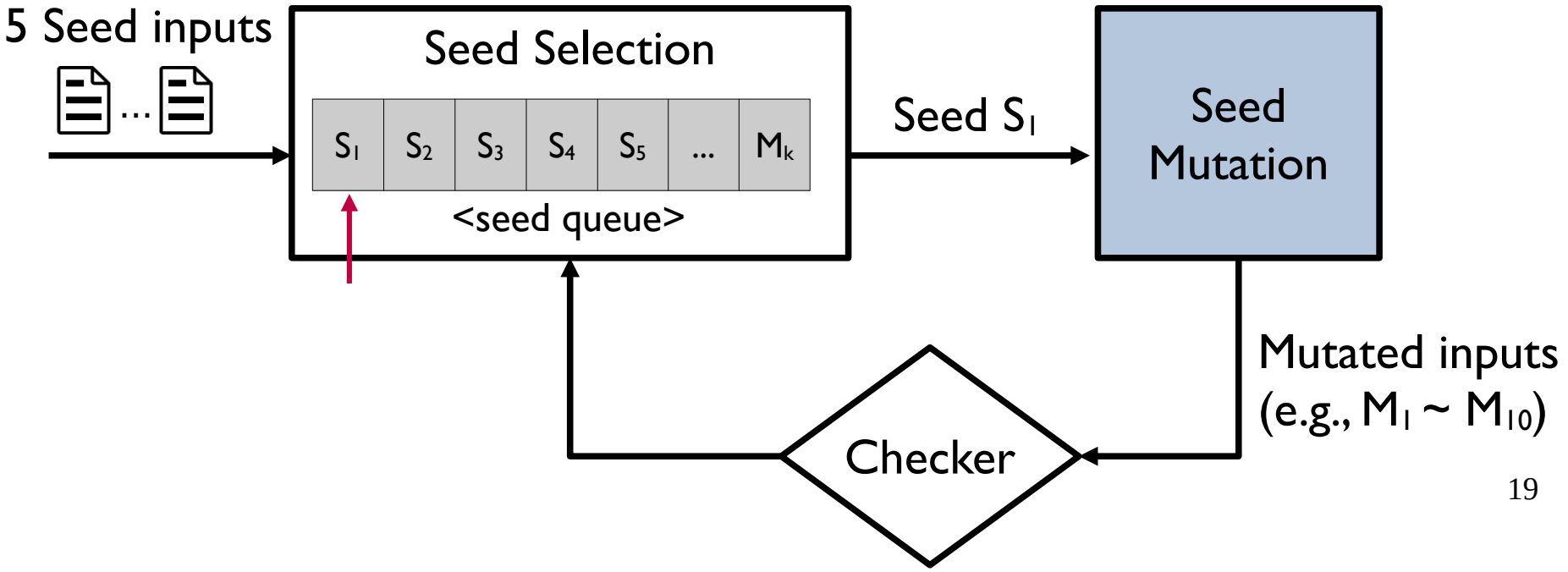
Seed Selection

- Add the interesting inputs into the queue.
- Sequentially select the first seed in the queue again.



Seed Selection

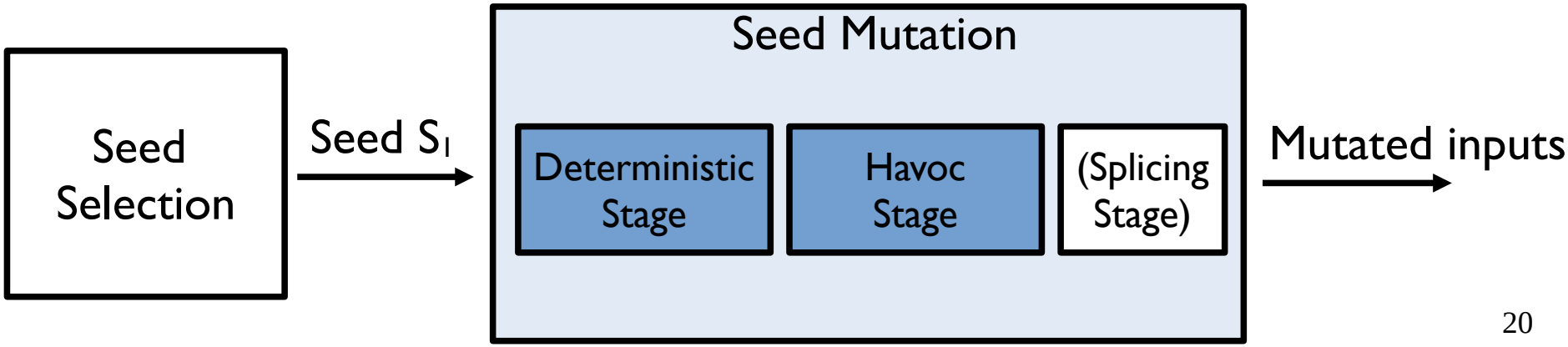
- Add the interesting inputs into the queue.
- Sequentially select the first seed in the queue again.



Seed Mutation

- Deterministic stage (with no randomness)

- Havoc stage (with randomness)

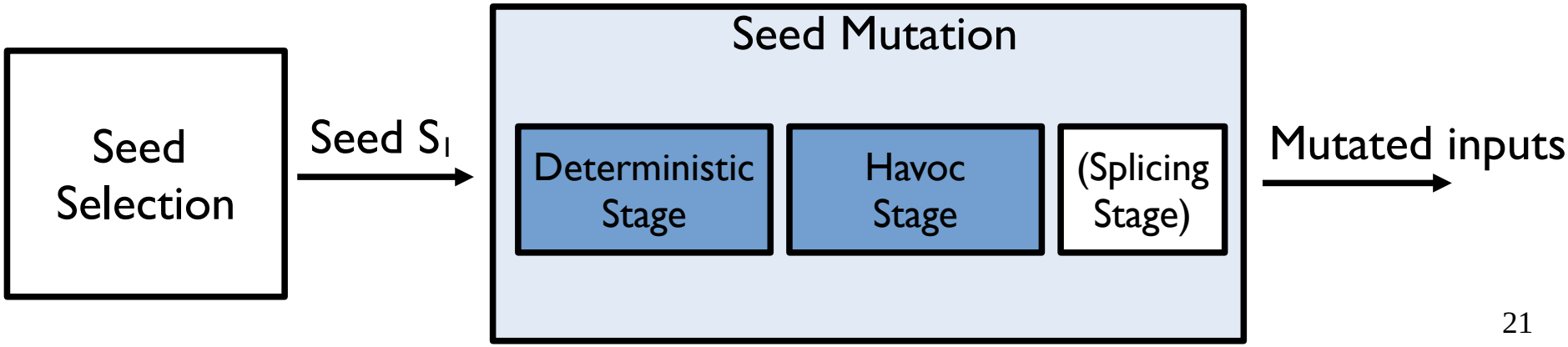


Seed Mutation

- Deterministic stage (with no randomness)
  - Generating mutants by using a mutation operator

    (Small change to seed input)
- Havoc stage (with randomness)
  - Generating mutants by using multiple operators

    (Big change to seed input)



Seed Mutation

- Deterministic stage
  - Apply each mutation operator to the input from the start to the end
  - Operators: bitflip(1,2,4 bit), byteflip(1,2,4 byte), arithmetic inc/dec, ...

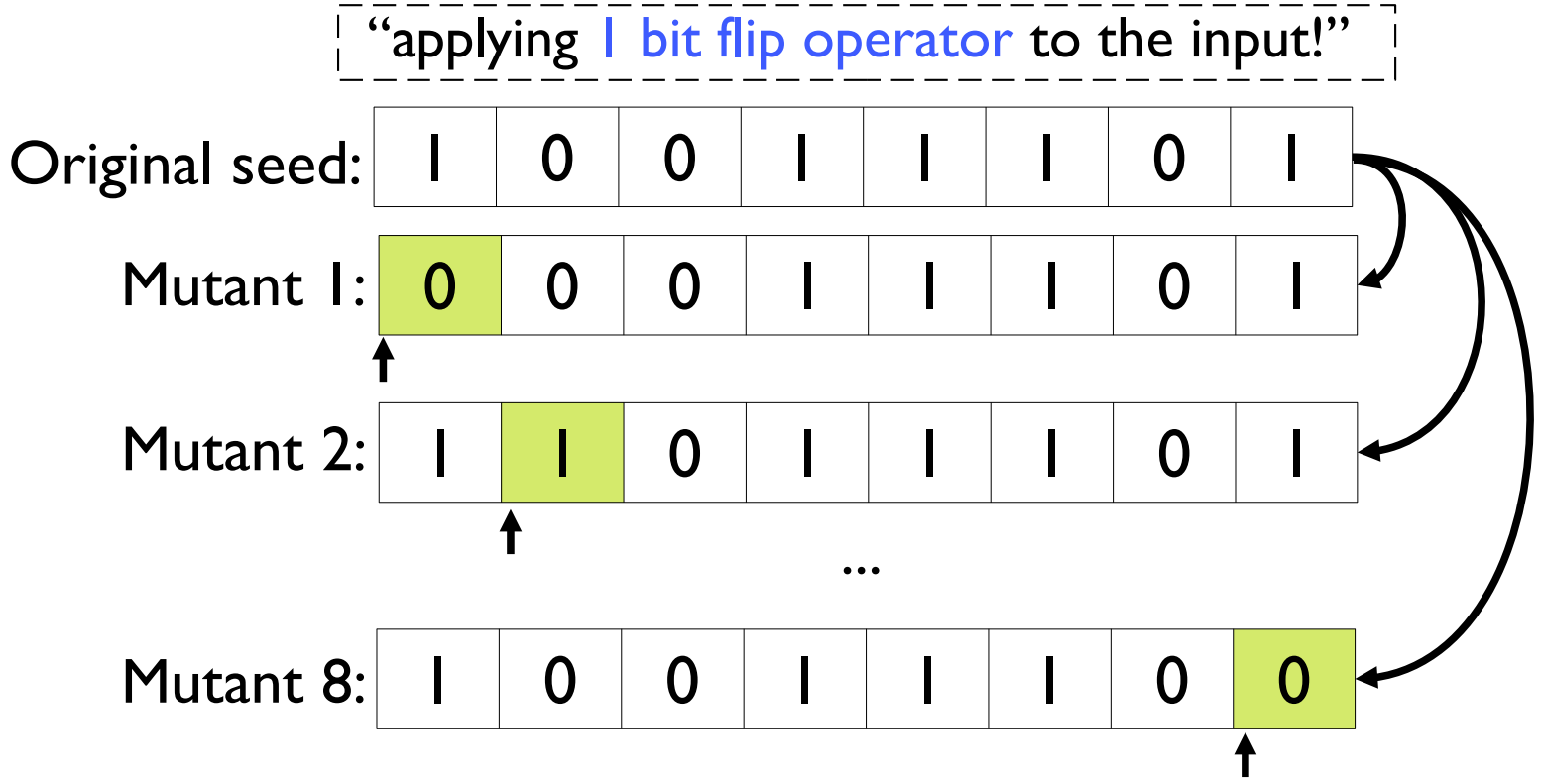
"applying 1 bit flip operator to the input!"

Original seed:

1	0	0	1	1	1	0	1

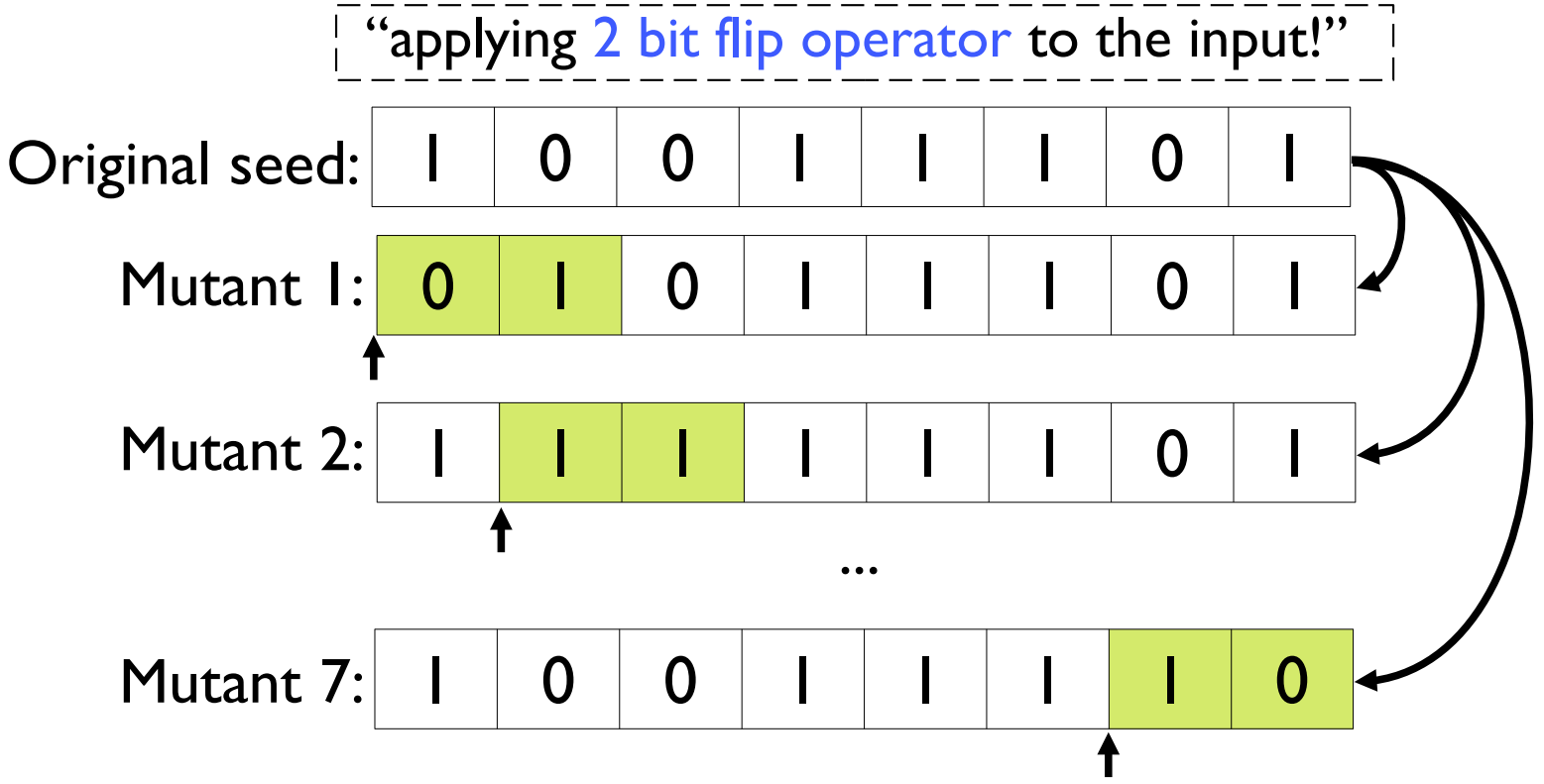
Seed Mutation

- Deterministic stage
  - Apply each mutation operator to the input from the start to the end
  - Operators: bitflip(1,2,4 bit), byteflip(1,2,4 byte), arithmetic inc/dec, ...



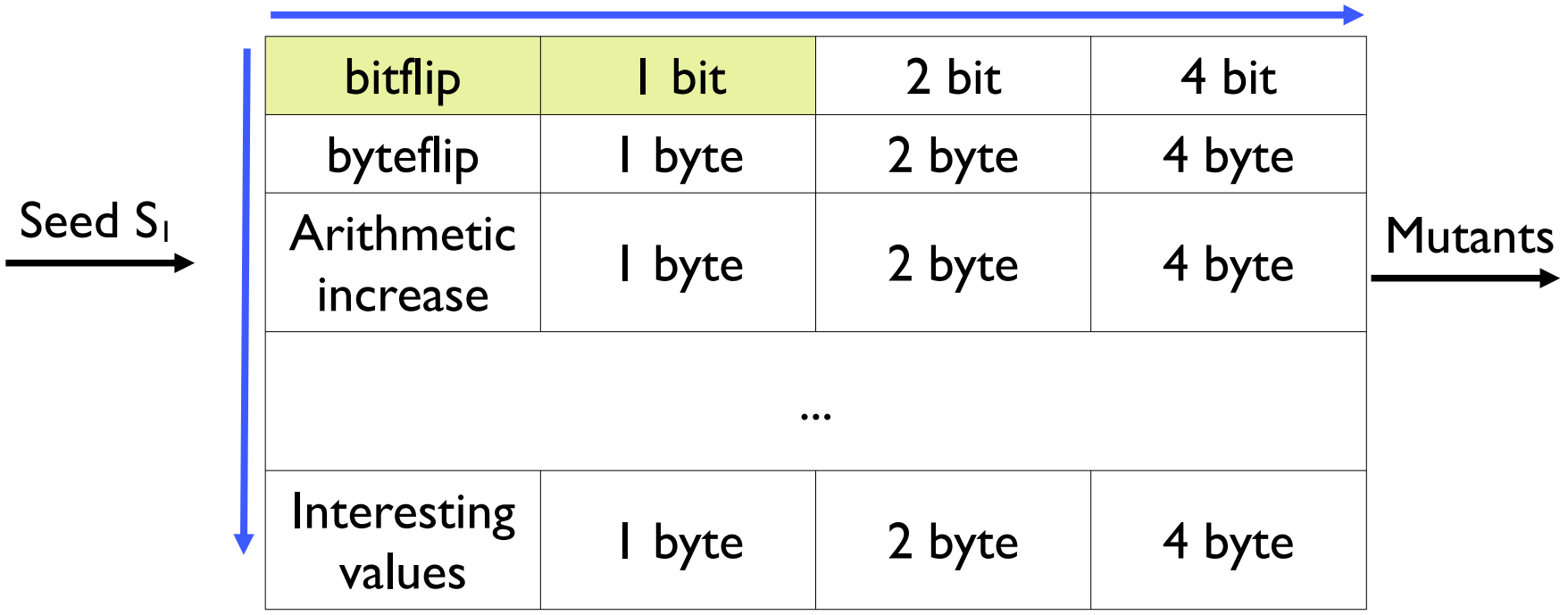
Seed Mutation

- Deterministic stage
  - Apply each mutation operator to the input from the start to the end
  - Operators: bitflip(1,2,4 bit), byteflip(1,2,4 byte), arithmetic inc/dec, ...



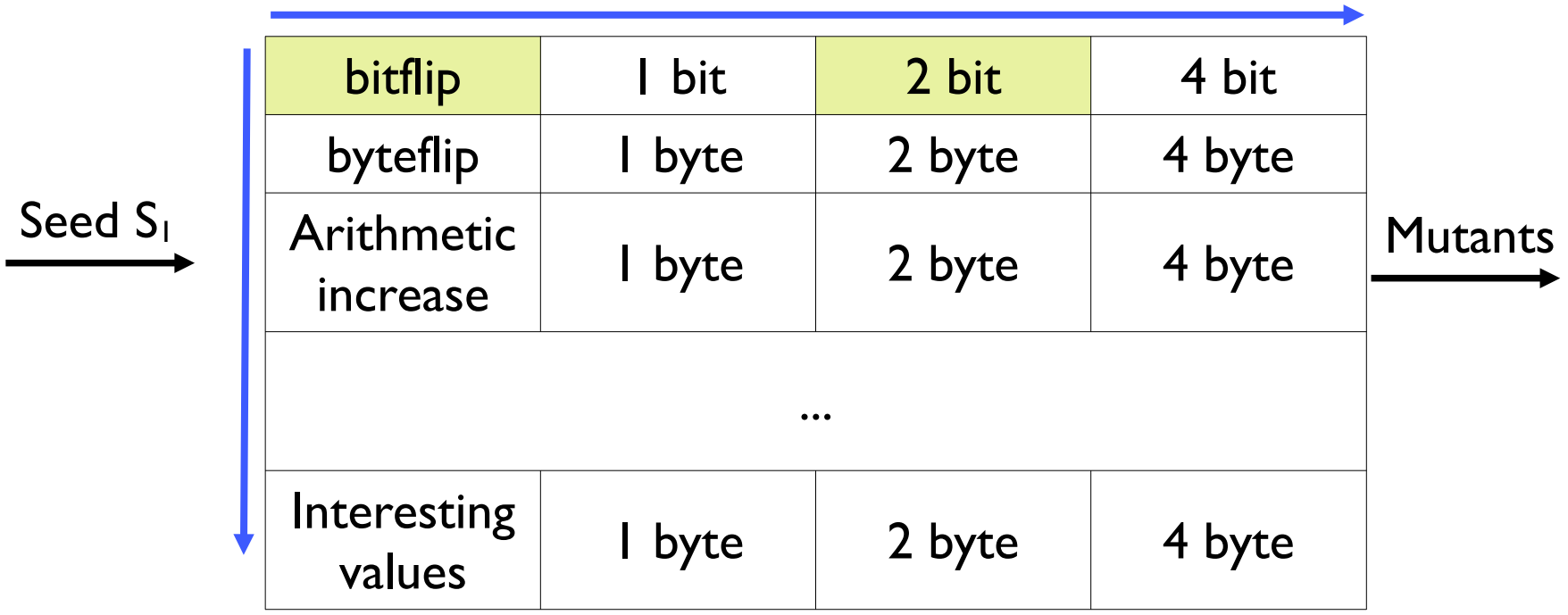
Seed Mutation

- Deterministic stage
  - Apply each mutation operator to the input from the start to the end
  - Perform all mutation operators in order



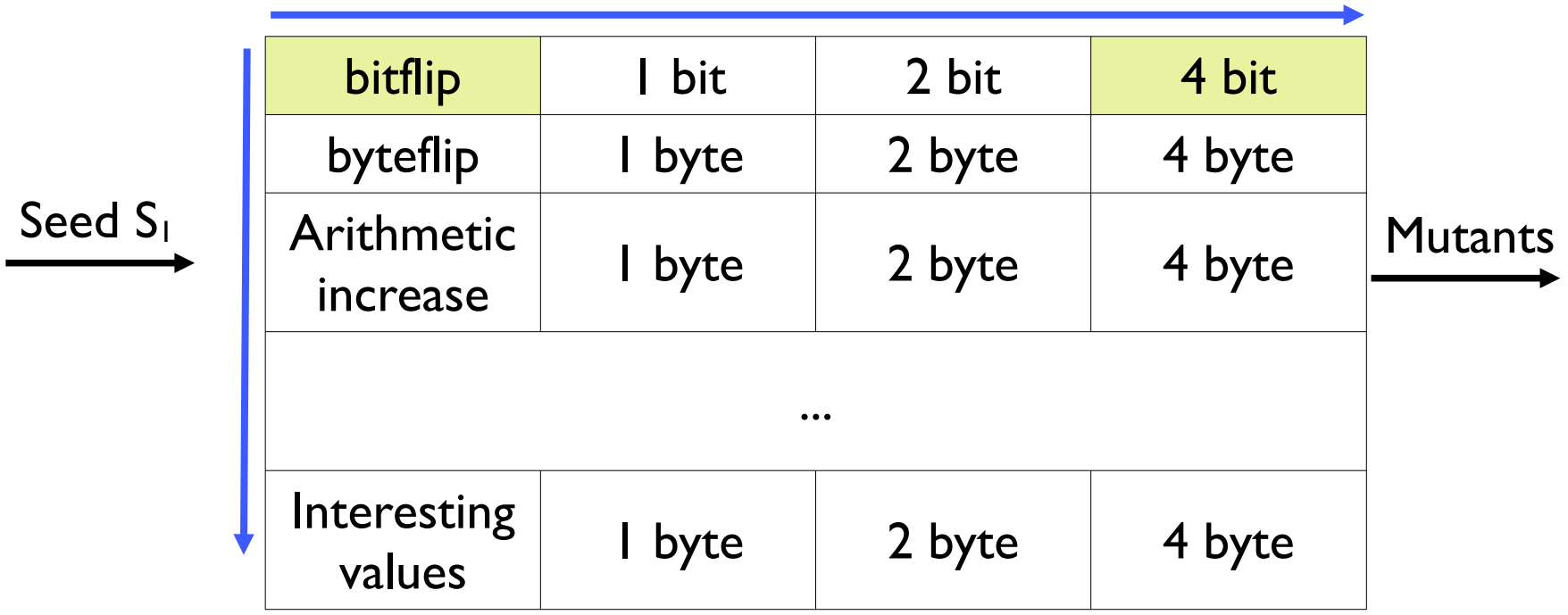
Seed Mutation

- Deterministic stage
  - Apply each mutation operator to the input from the start to the end
  - Perform all mutation operators in order



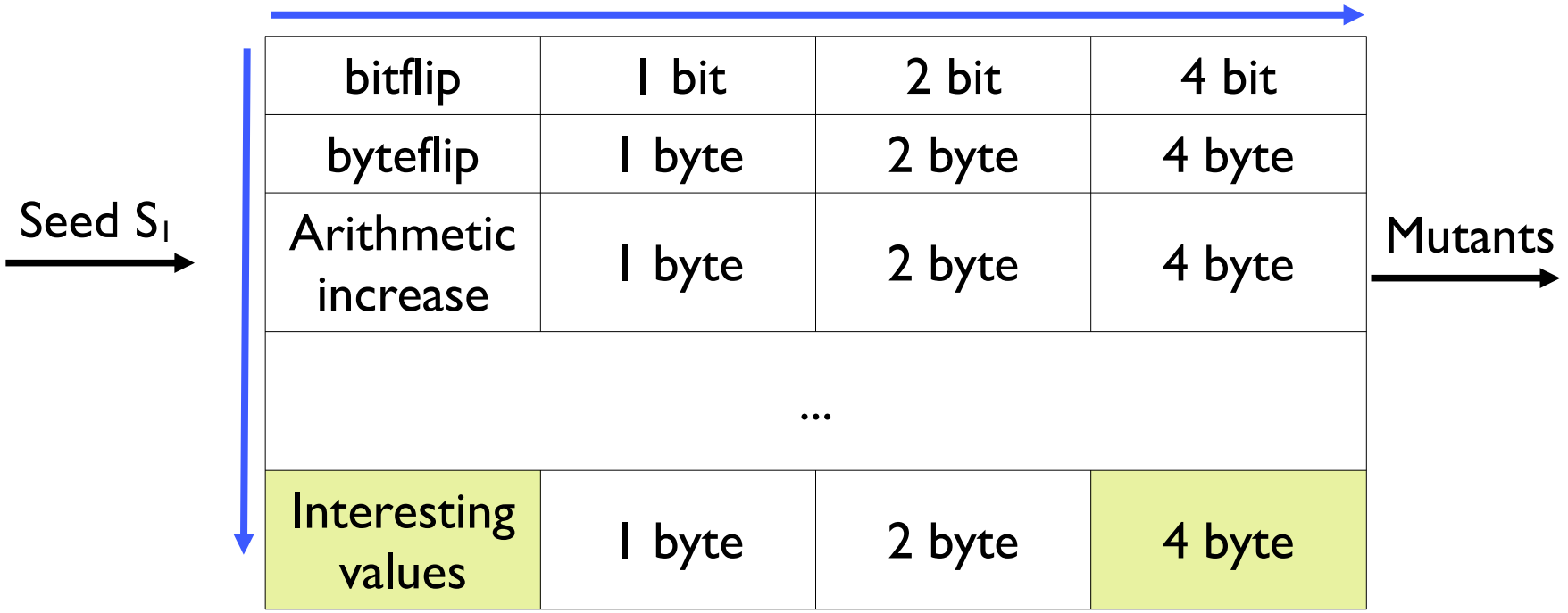
Seed Mutation

- Deterministic stage
  - Apply each mutation operator to the input from the start to the end
  - Perform all mutation operators in order



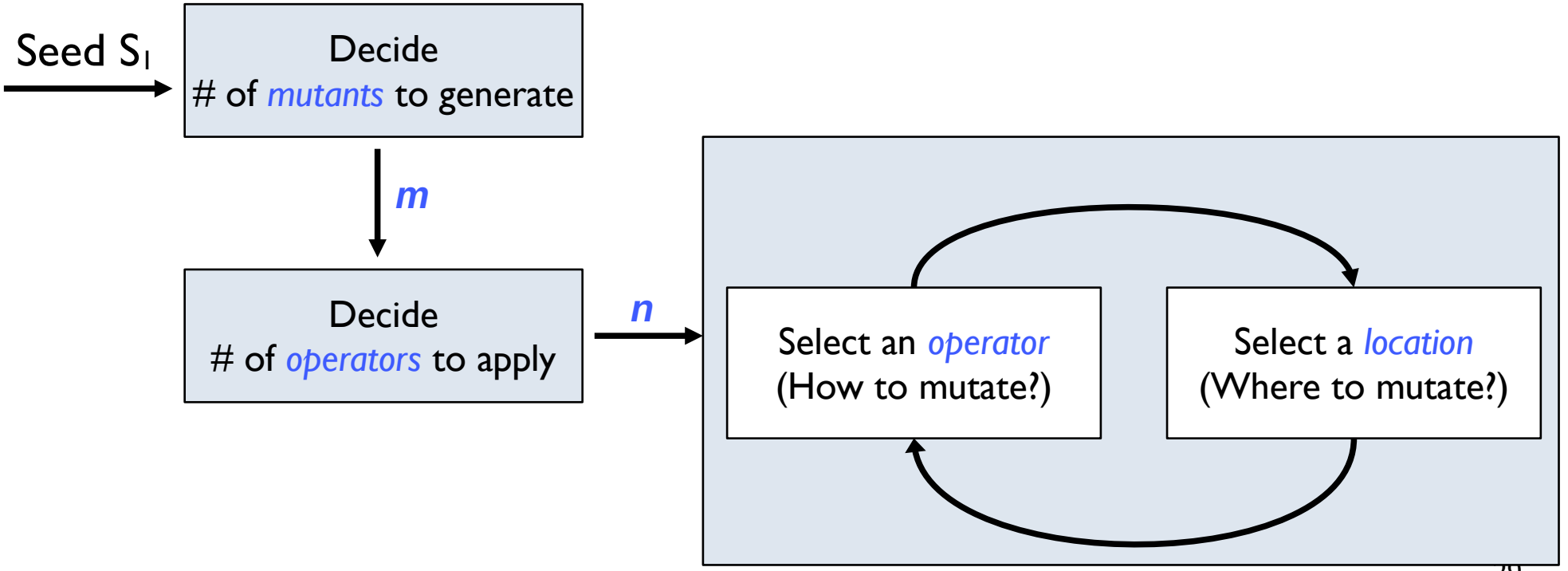
Seed Mutation

- Deterministic stage
  - Apply each mutation operator to the input from the start to the end
  - Perform all mutation operators in order



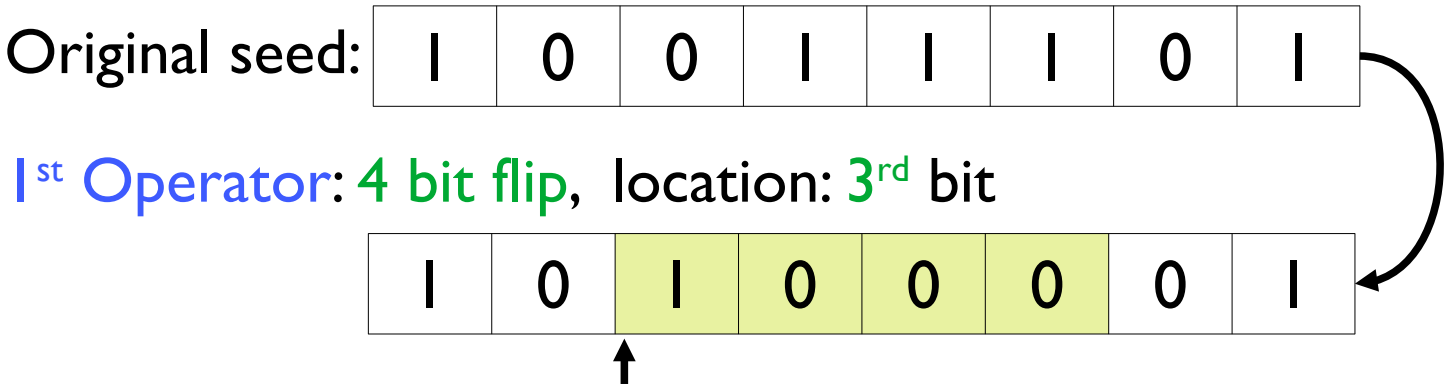
Seed Mutation

- Havoc stage
  - Apply ***n*** mutation operators to mutate the seed input.
  - Repeat the above to produce ***m*** mutants.



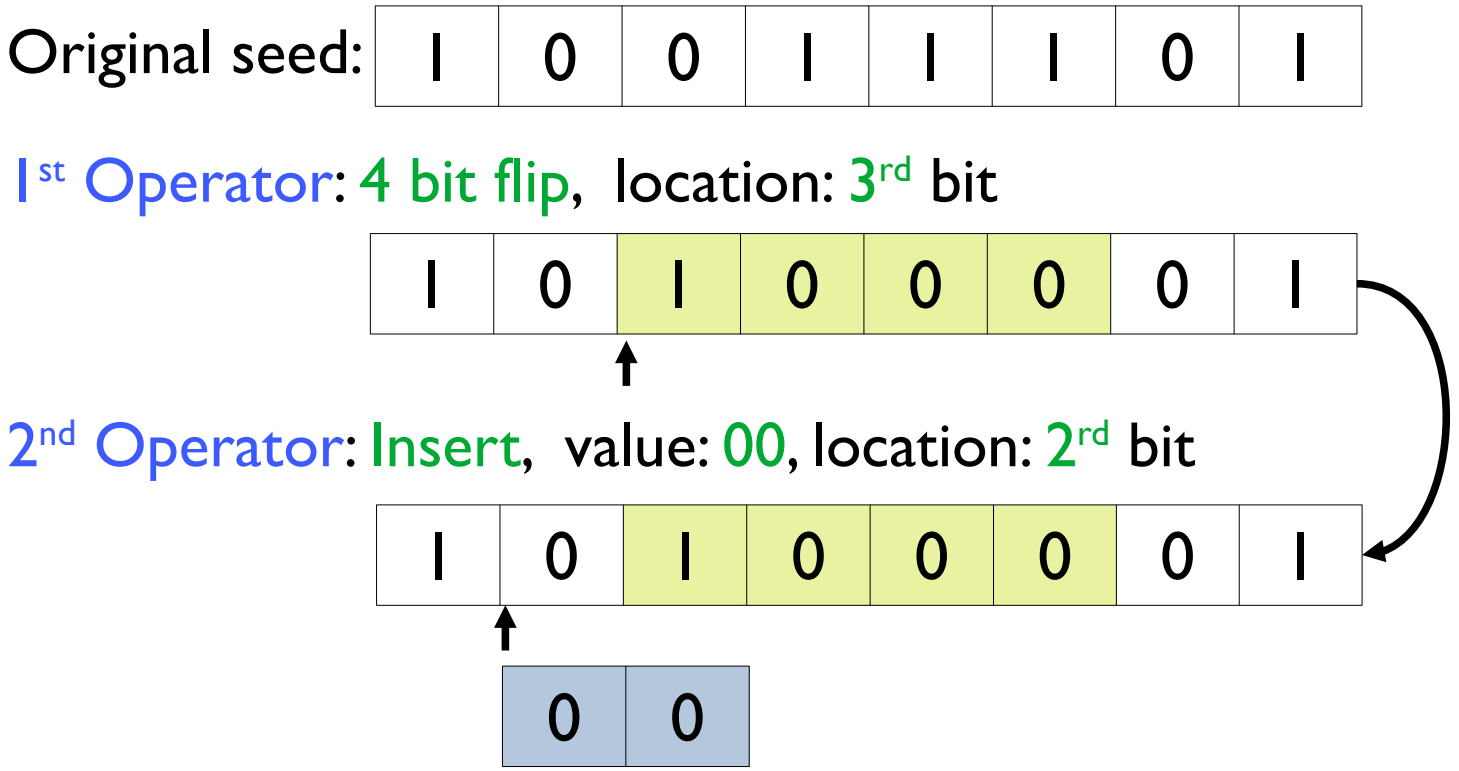
Seed Mutation

- Havoc stage
  - Apply ***3*** mutation operators to mutate the seed input.



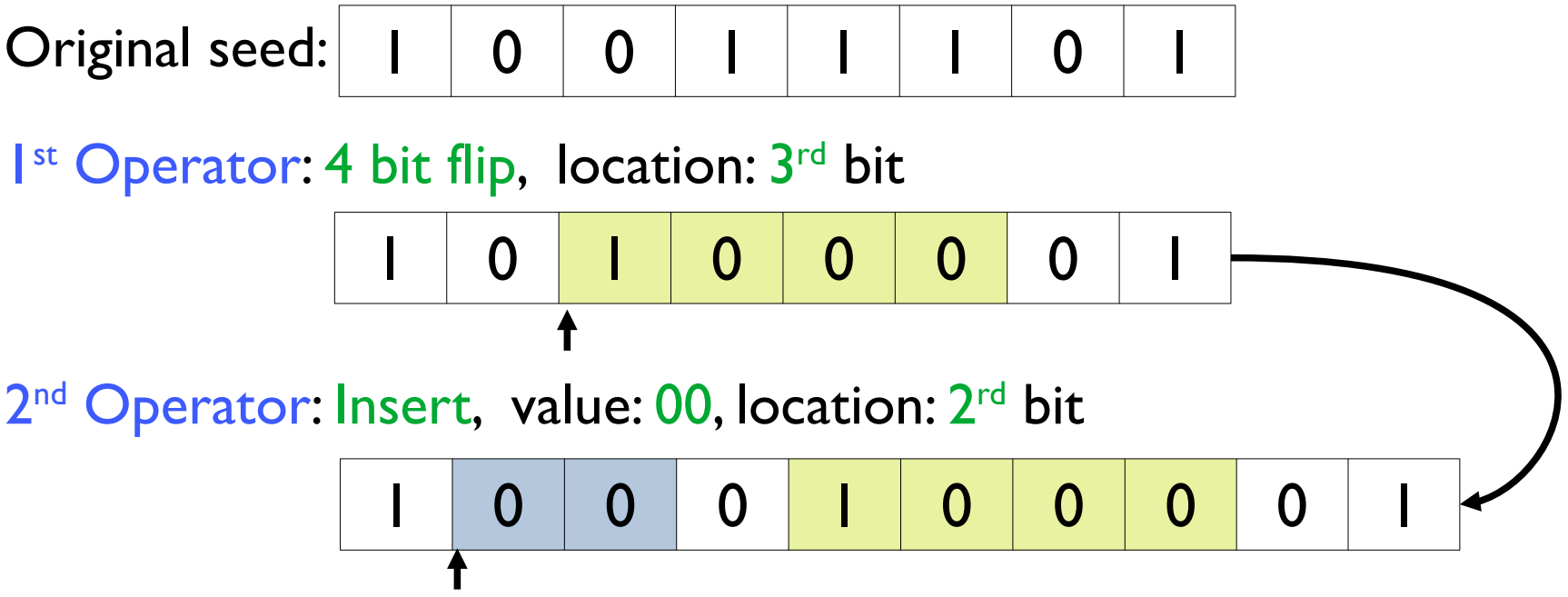
Seed Mutation

- Havoc stage
  - Apply ***3*** mutation operators to mutate the seed input.



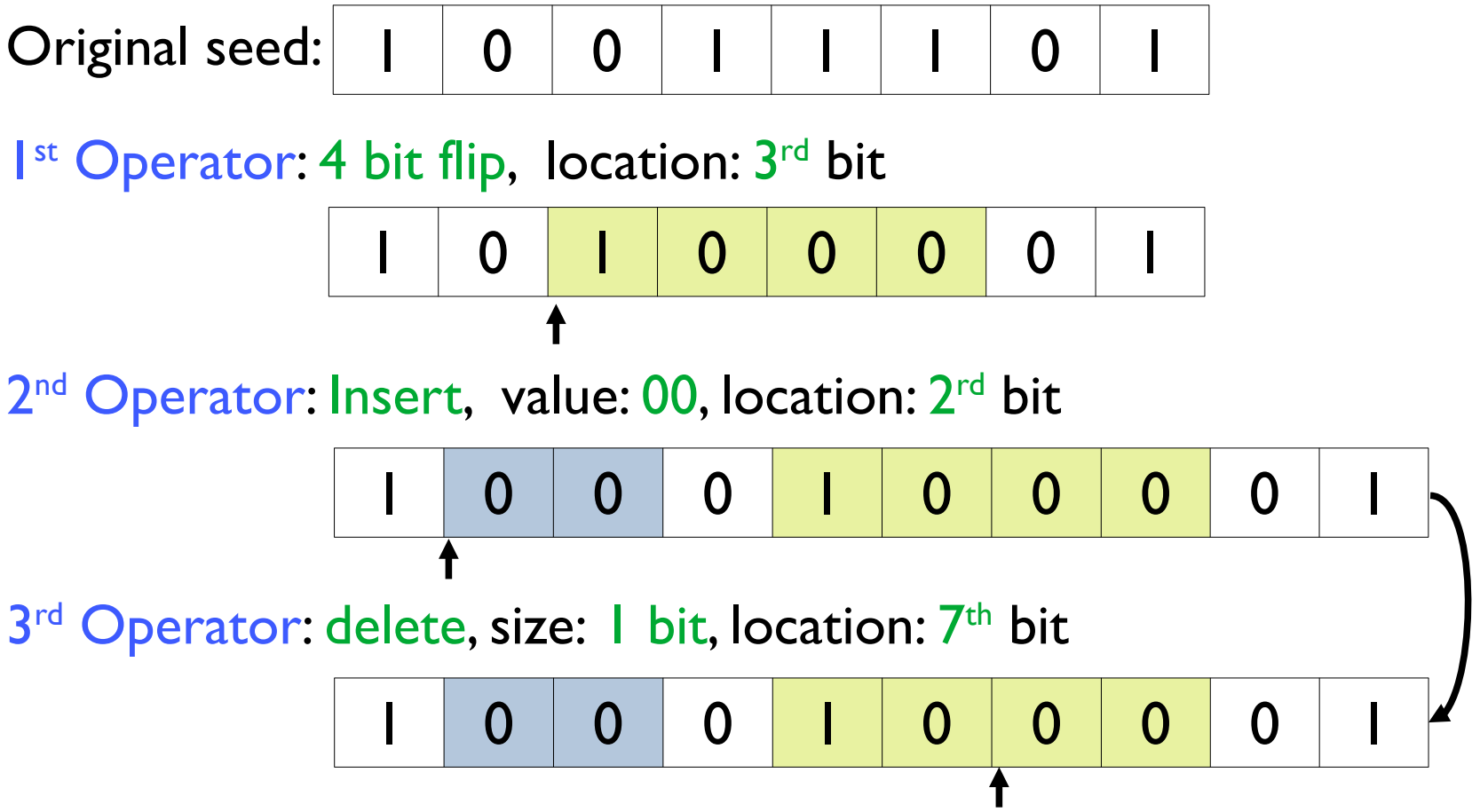
Seed Mutation

- Havoc stage
  - Apply ***3*** mutation operators to mutate the seed input.



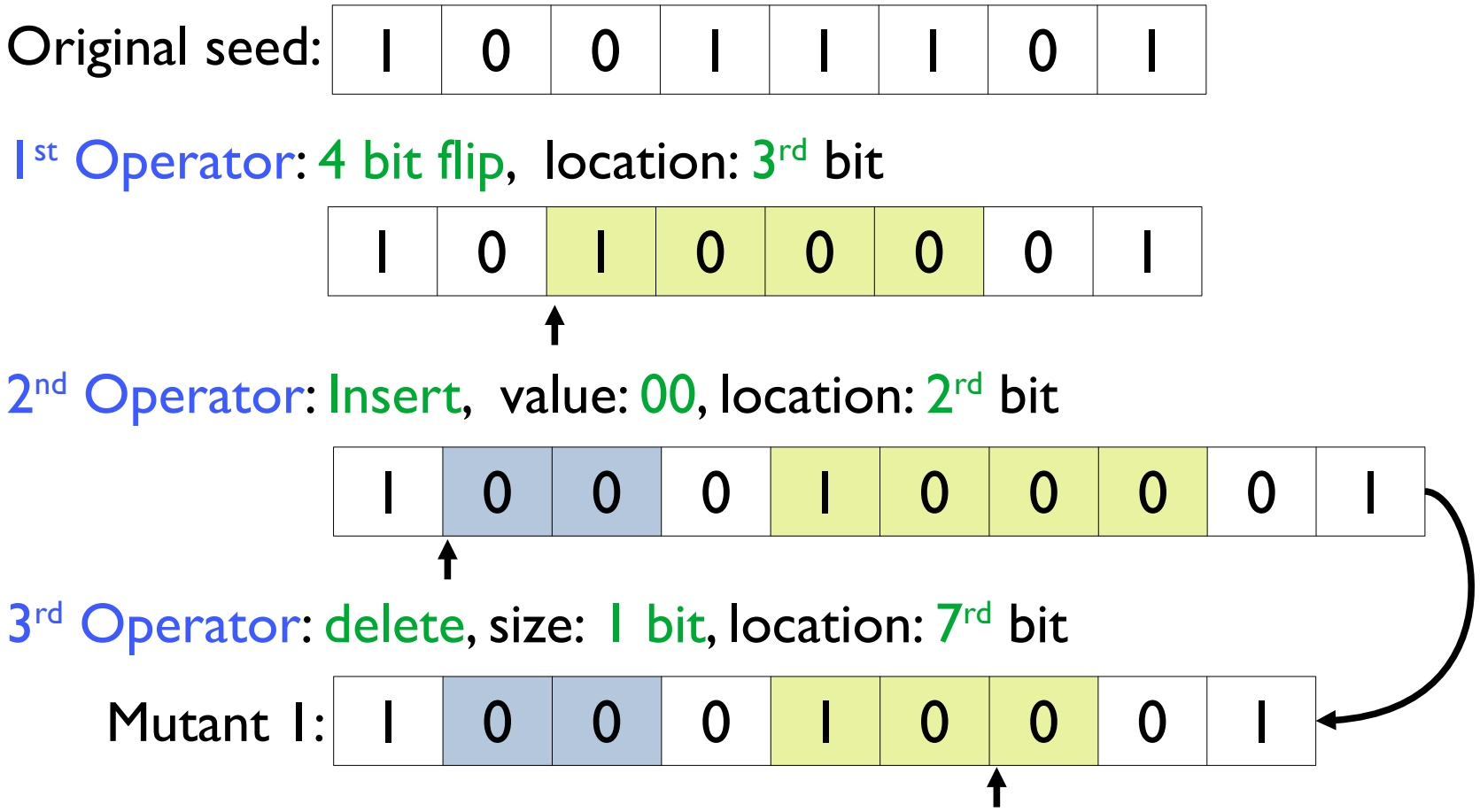
Seed Mutation

- Havoc stage
  - Apply ***3*** mutation operators to mutate the seed input.



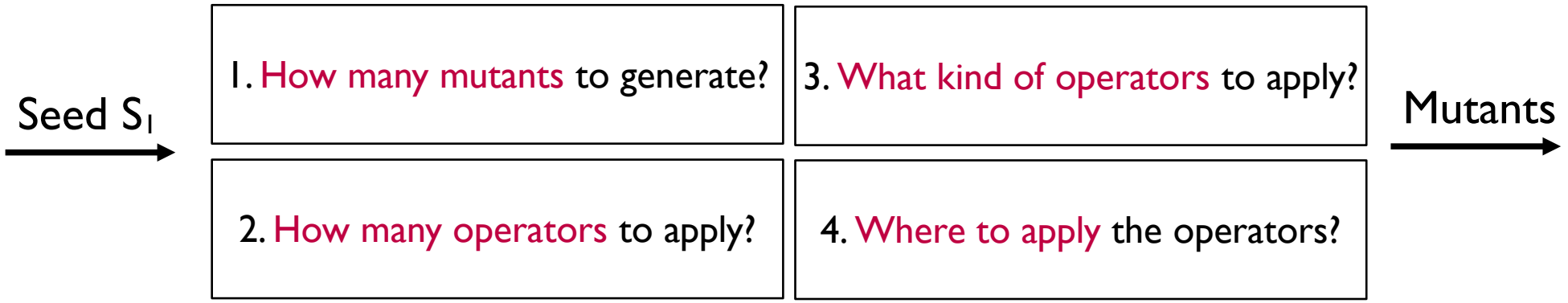
Seed Mutation

- Havoc stage
  - Apply ***3*** mutation operators to mutate the seed input.



Seed Mutation

- Havoc stage
  - Totally rely on randomness to generate mutants.

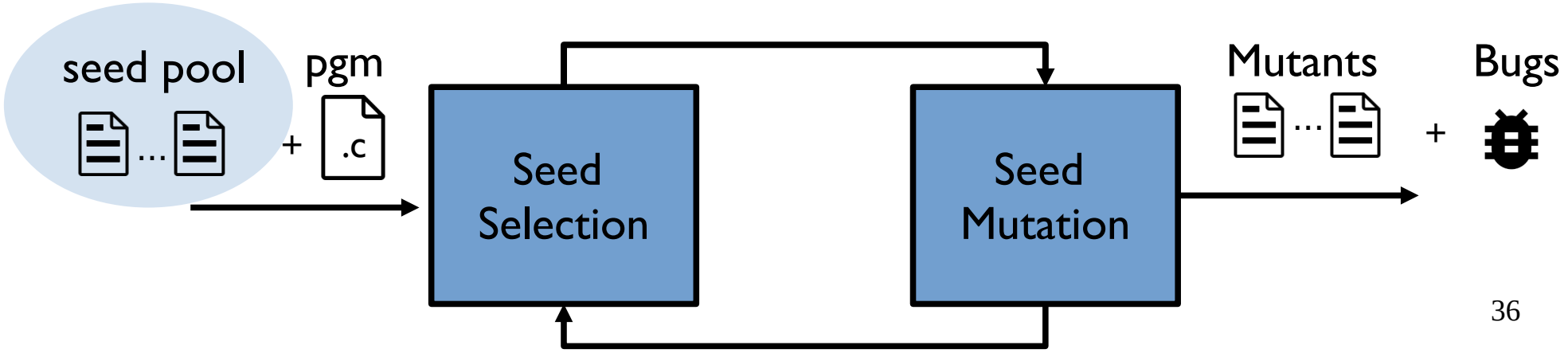


- Which one is more effective at finding bugs?

Deterministic Stage vs Havoc Stage

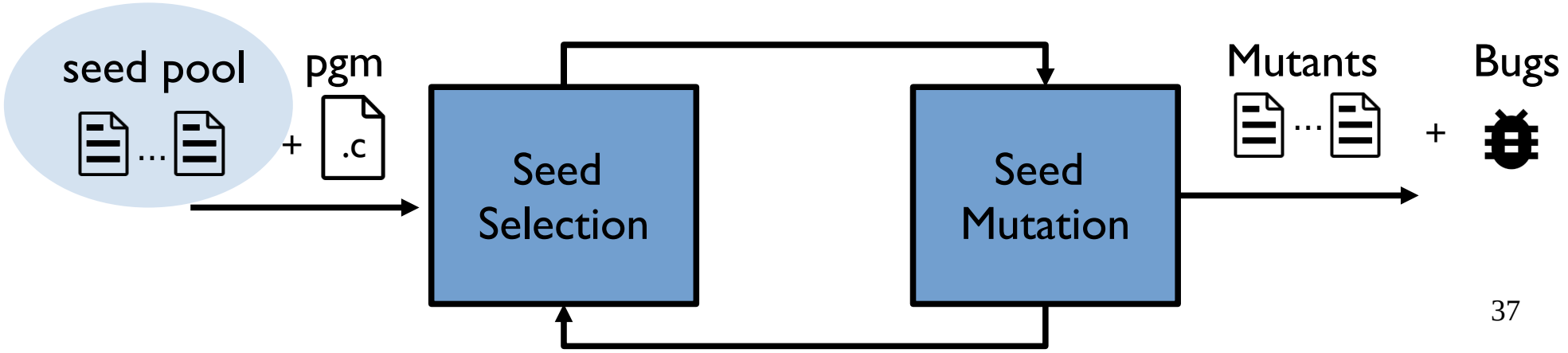
Topics in Mutation-based Fuzzing

- Seed Pool Generation
  - How to effectively build an initial seed pool?



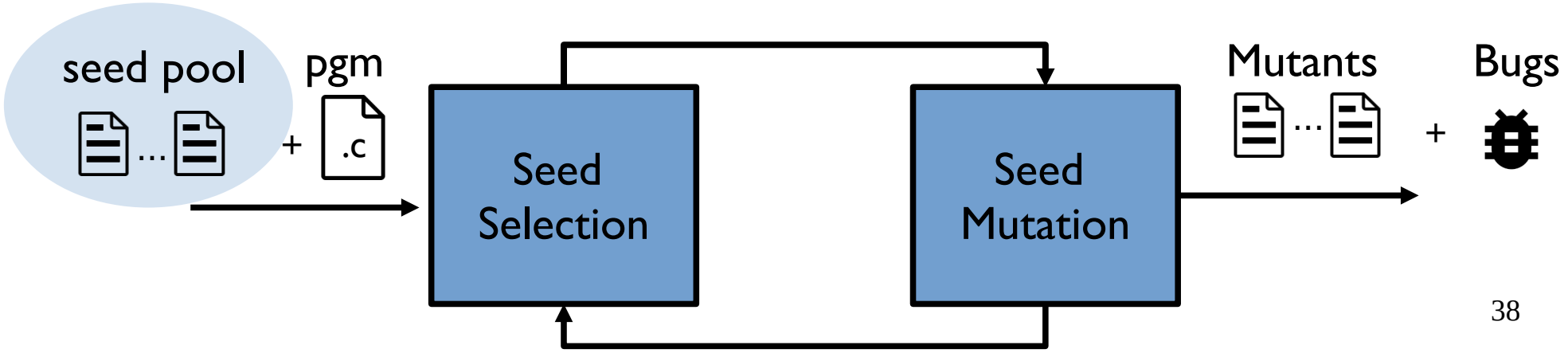
Topics in Mutation-based Fuzzing

- Seed Pool Generation
  - How to effectively build an initial seed pool?
- Seed Selection
  - Which seed from the pool to select first?



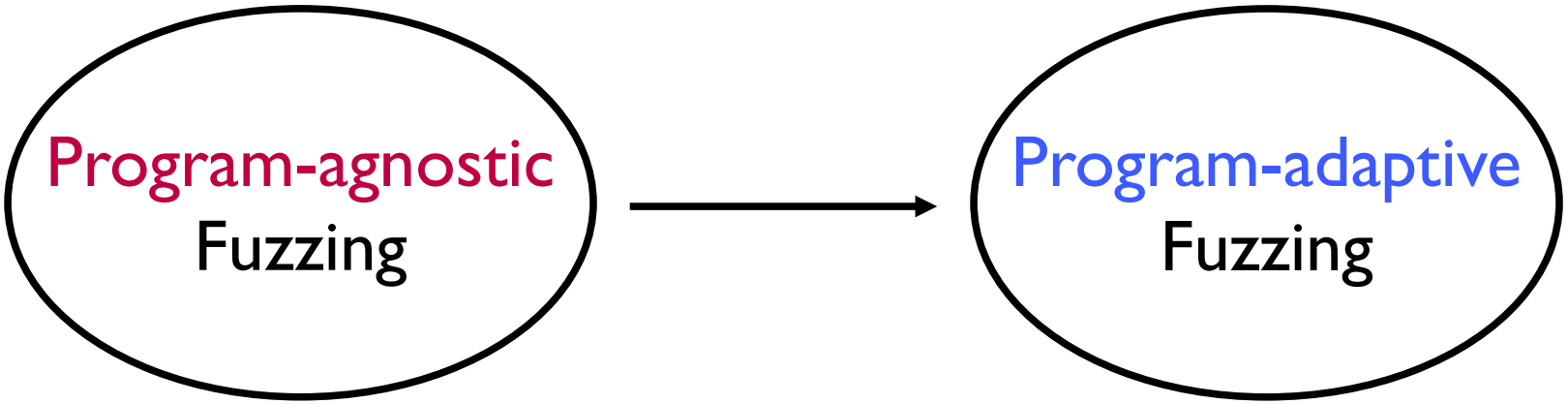
Topics in Mutation-based Fuzzing

- Seed Pool Generation
  - How to effectively build an initial seed pool?
- Seed Selection
  - Which seed from the pool to select first?
- Seed Mutation
  - How to mutate the selected seed?



Recent Trend

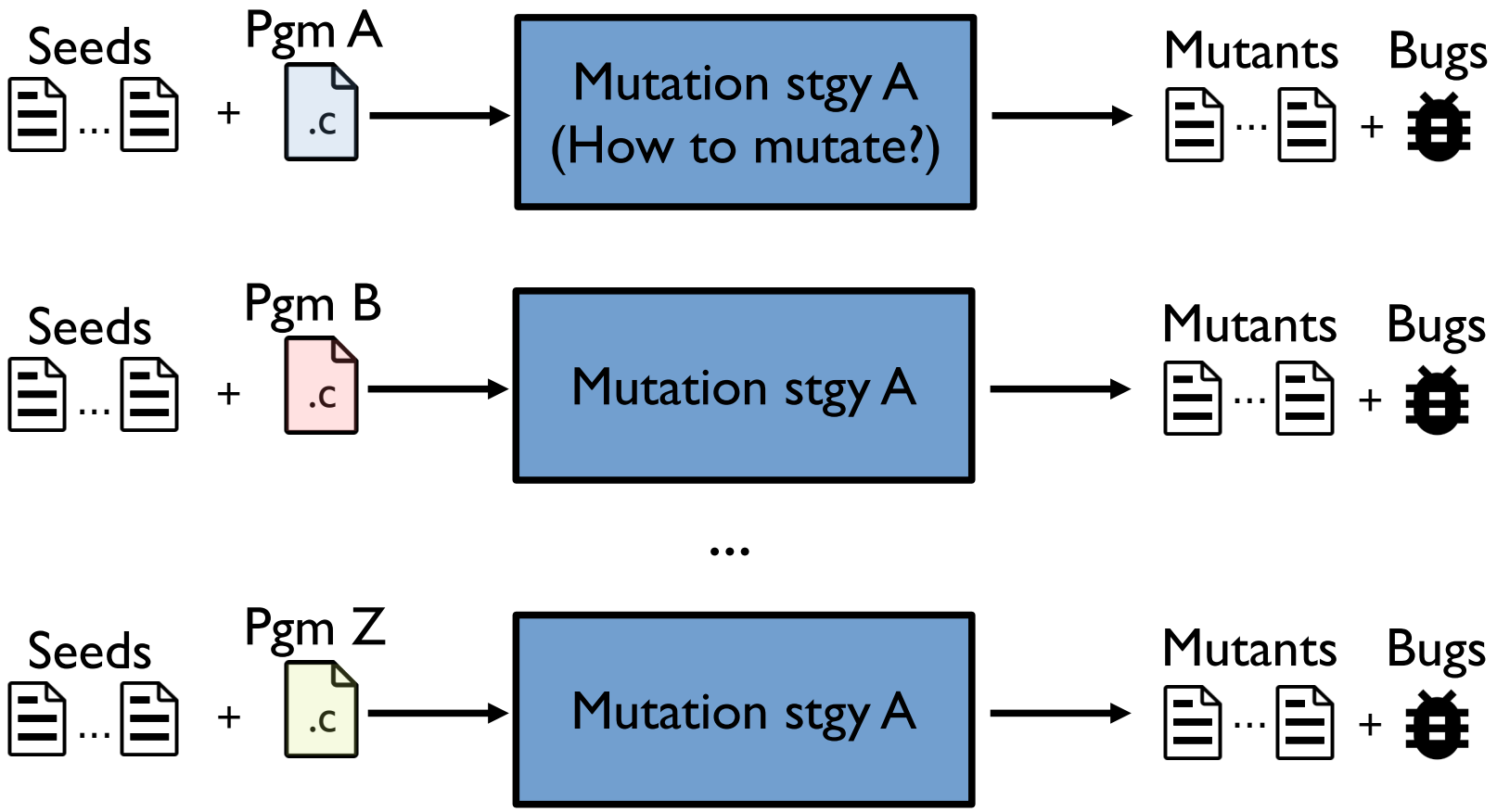
- Recent trend in mutation-based fuzzing



  - Program-agnostic fuzzing
    - Using a fixed seed mutation (or selection) strategy regardless of the target program. (e.g., AFL)
  - Program-adaptive fuzzing
    - Using an adaptive seed mutation (or seed selection) strategy depending on the target program. (e.g., MOPT)

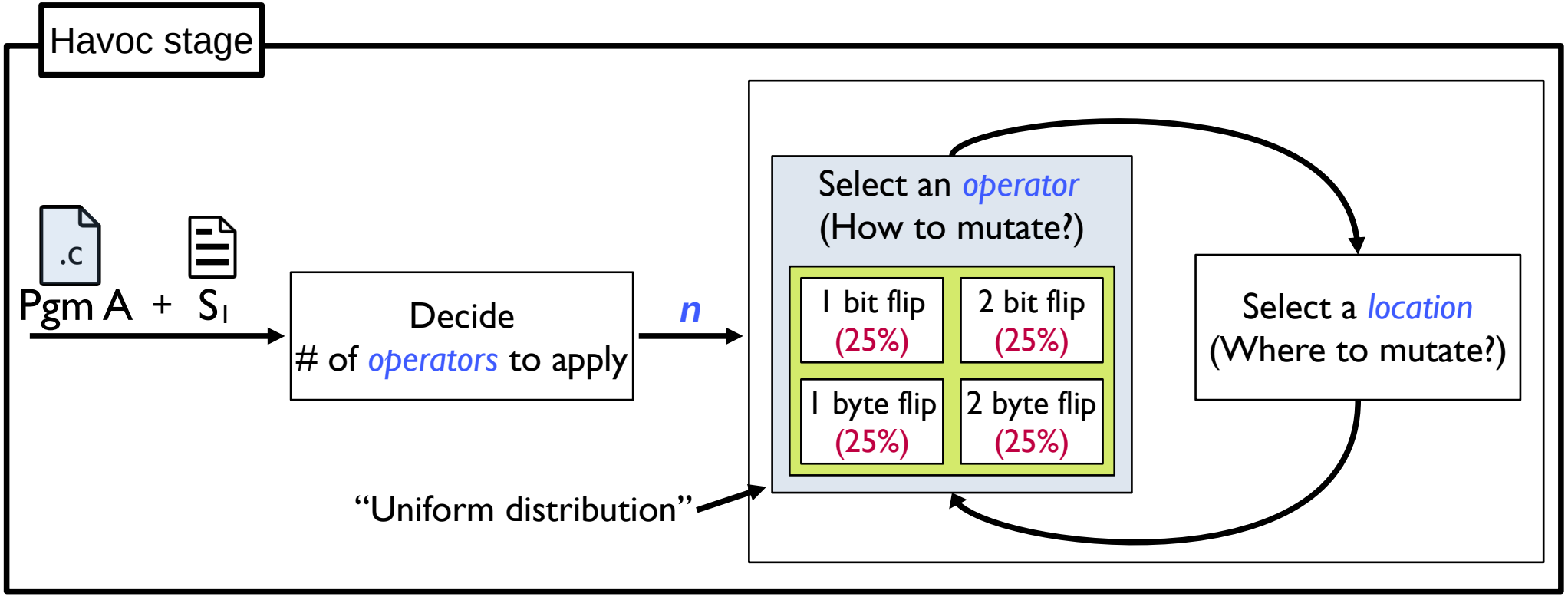
Recent Trend

- Program-agnostic seed mutation strategy
  - Unchanging the seed mutation strategy.



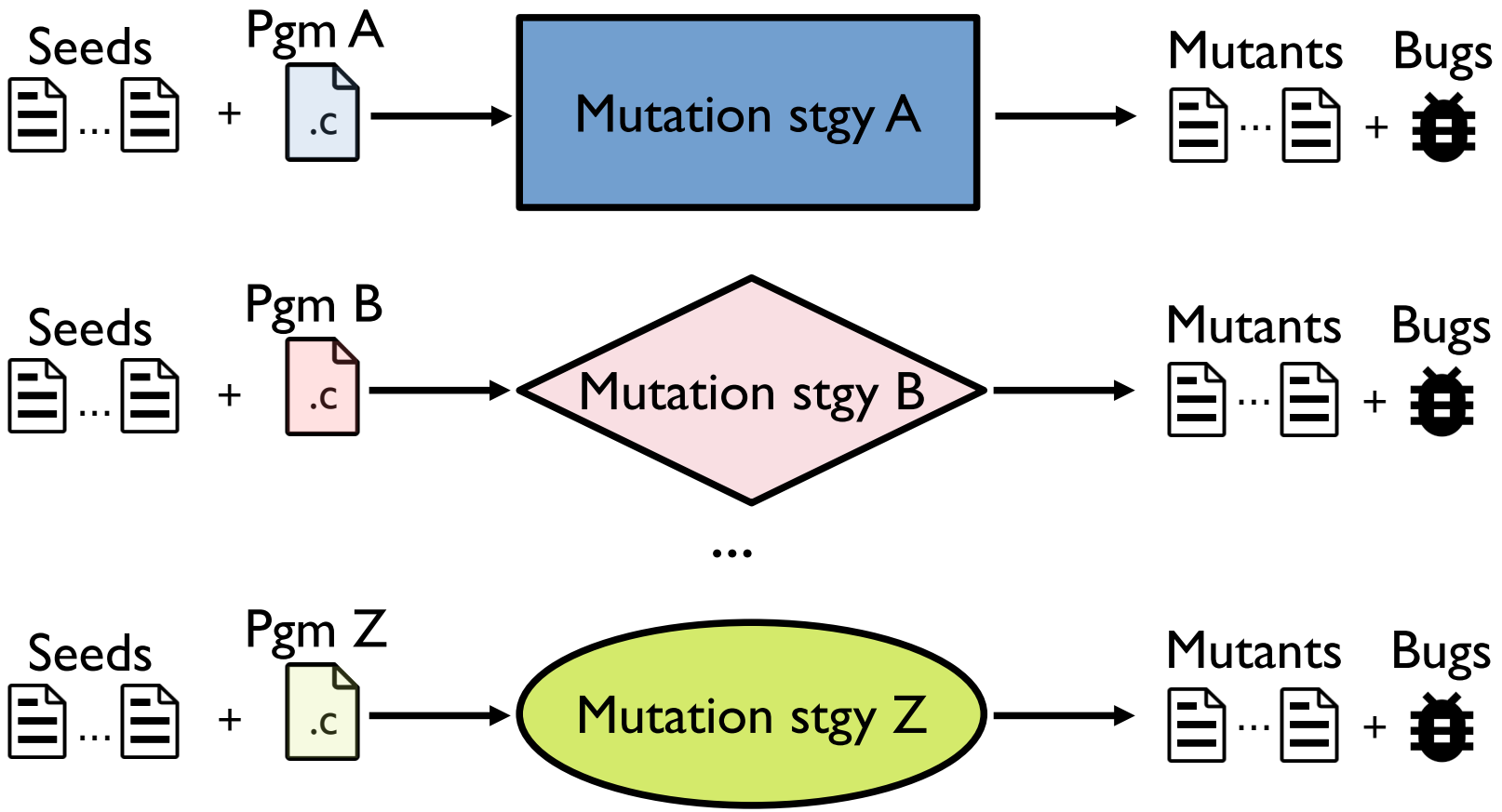
Program-Agnostic Mutation Strategy

- Seed mutation strategy (in AFL)
  - Randomly select the predefined mutation operators.



Recent Trend

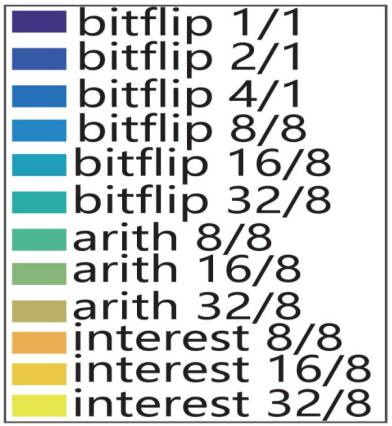
- Program-adaptive seed mutation strategy
  - Changing the seed mutation strategy depending on the program.



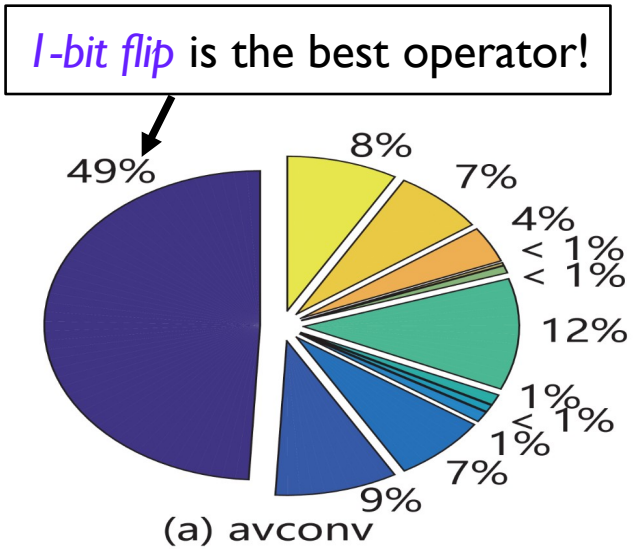
Security'19

MOPT

- Motivation (Key observation)
  - Effective mutation operators are different from the target program.[1]



Mutation operator types.



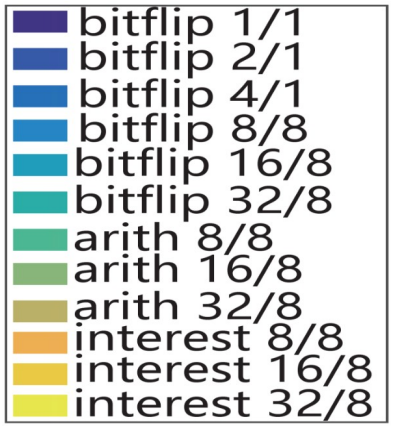
Percentages of *interesting* mutants generated by each mutation operator

1. MOPT: Optimize Mutation Scheduling for Fuzzers

Security'19

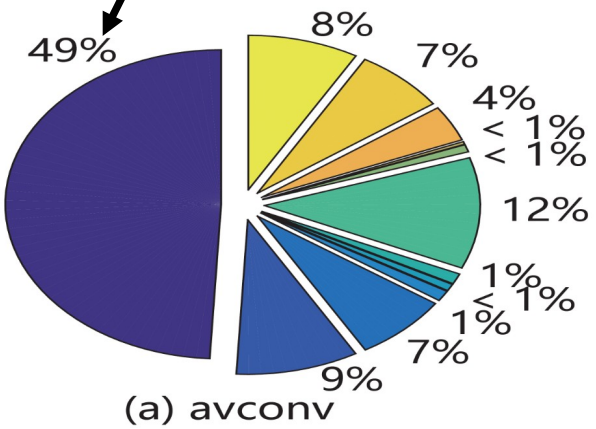
MOPT

- Motivation (Key observation)
  - Effective mutation operators are different from the target program.[1]



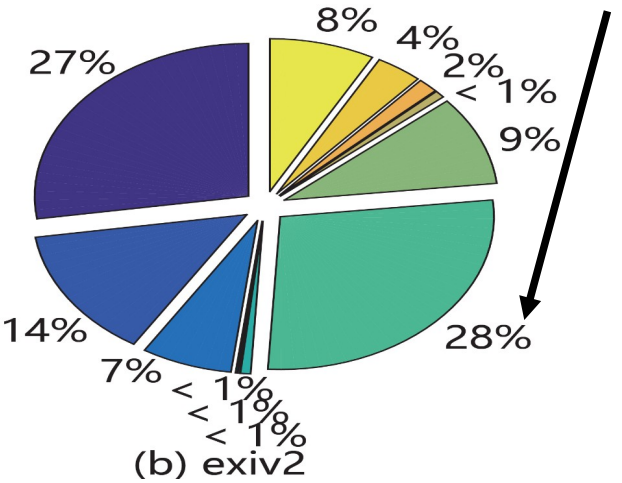
Mutation operator types.

1-bit flip is the best operator!



(a) avconv

arithmetic increase is the best!



(b) exiv2

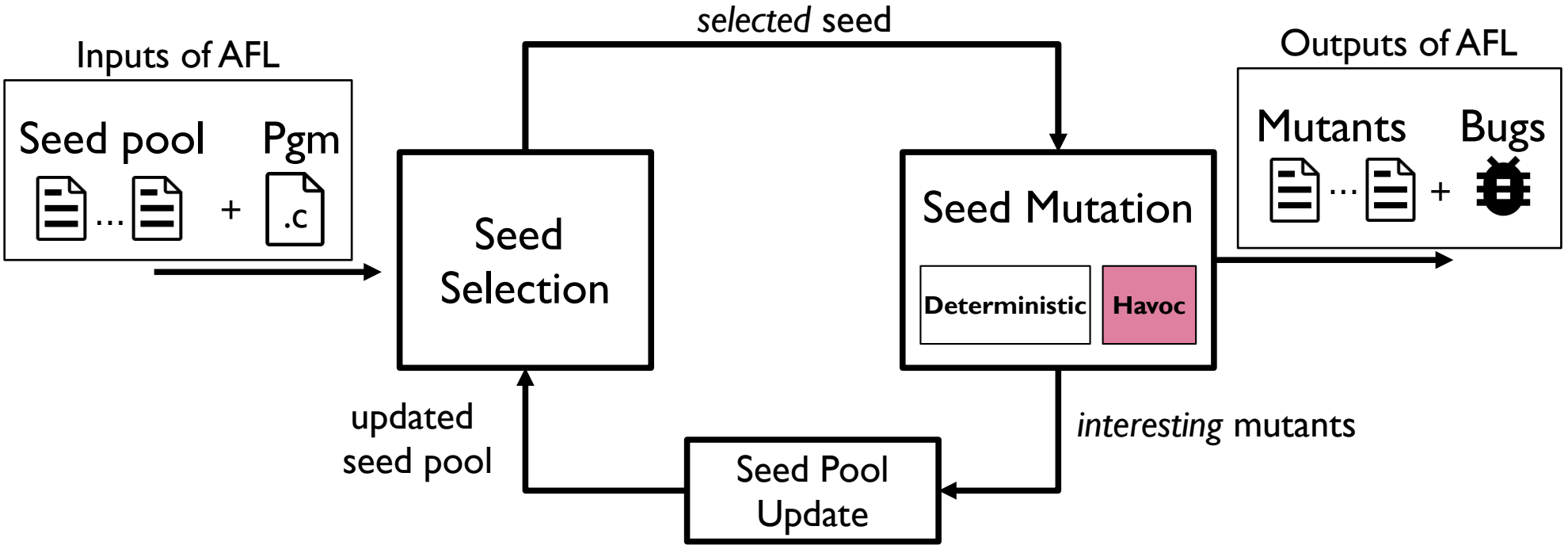
Percentages of *interesting* mutants generated by each mutation operator

1. MOPT: Optimize Mutation Scheduling for Fuzzers

Security'19

MOPT

- Goal
  - Finding an optimal mutation strategy in havoc stage for the program.

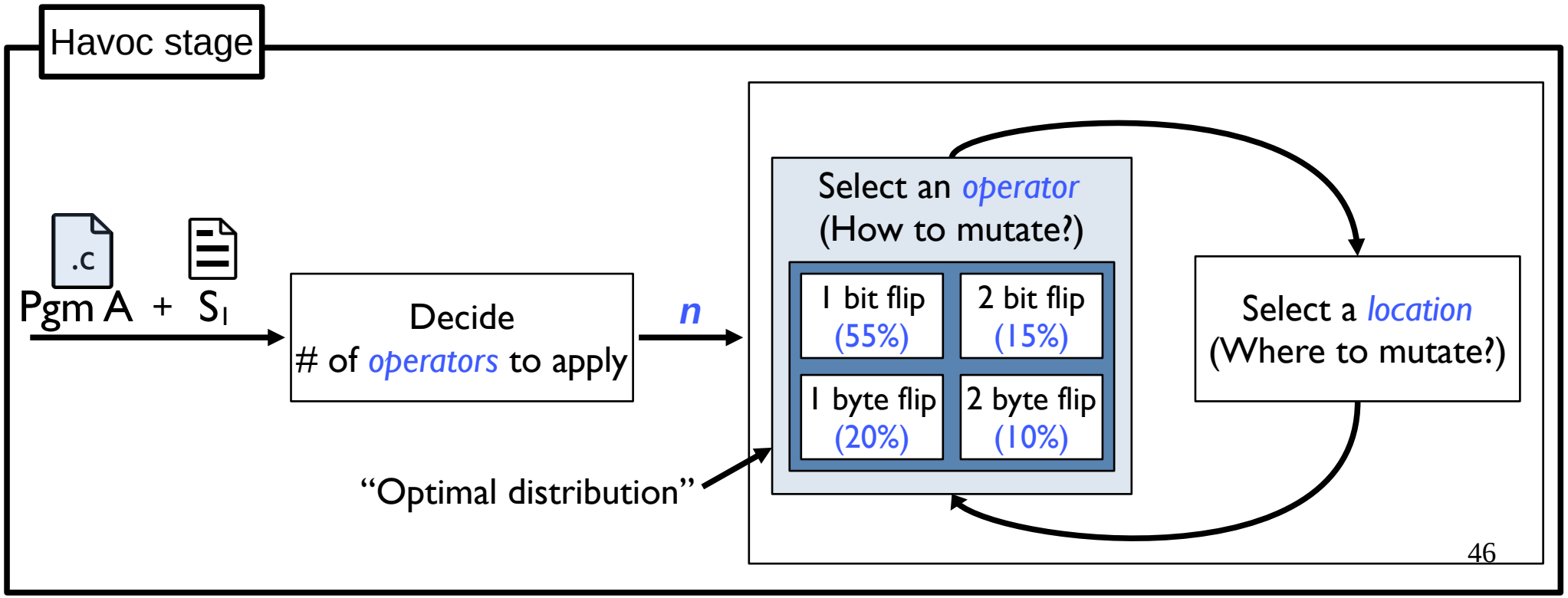


Security'19

MOPT

- Goal
  - Finding an optimal mutation strategy in havoc stage for the program.

    = Finding an optimal probabilistic distribution of selecting operators.

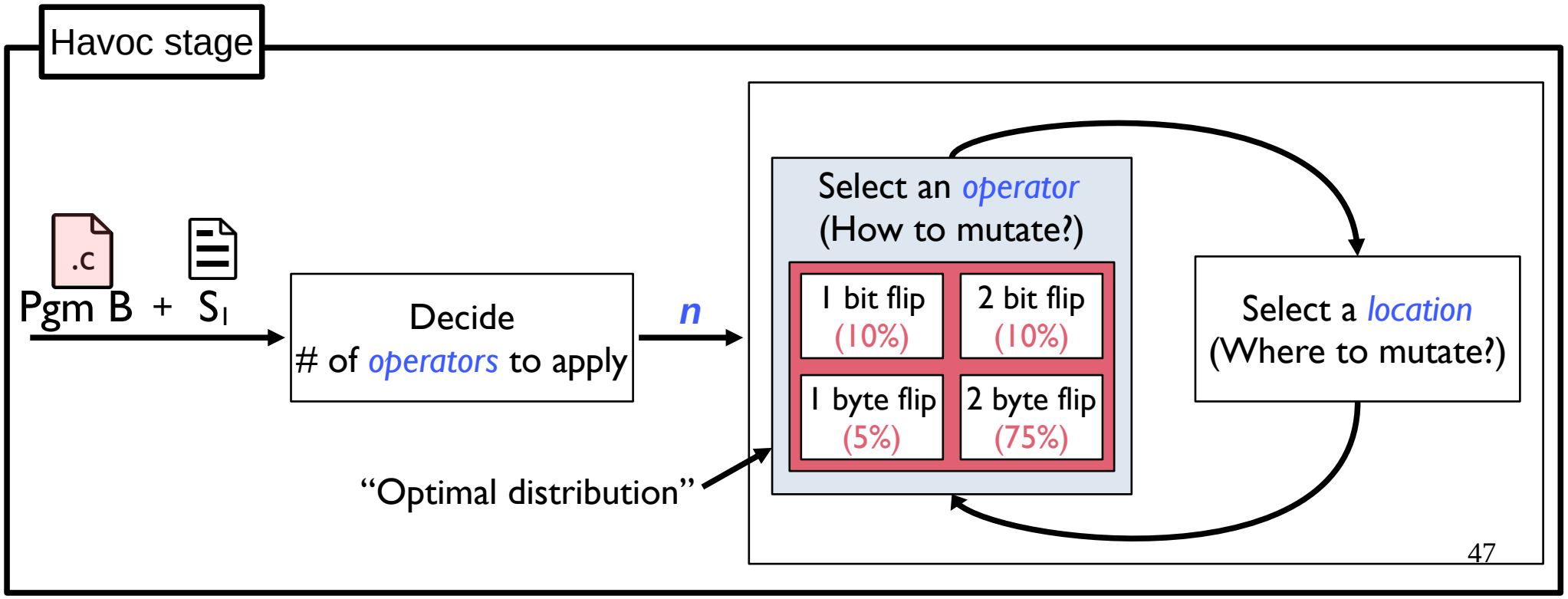


Security'19

MOPT

- Goal
  - Finding an optimal mutation strategy in havoc stage for the program.

    = Finding an optimal probabilistic distribution of selecting operators.



Security'19

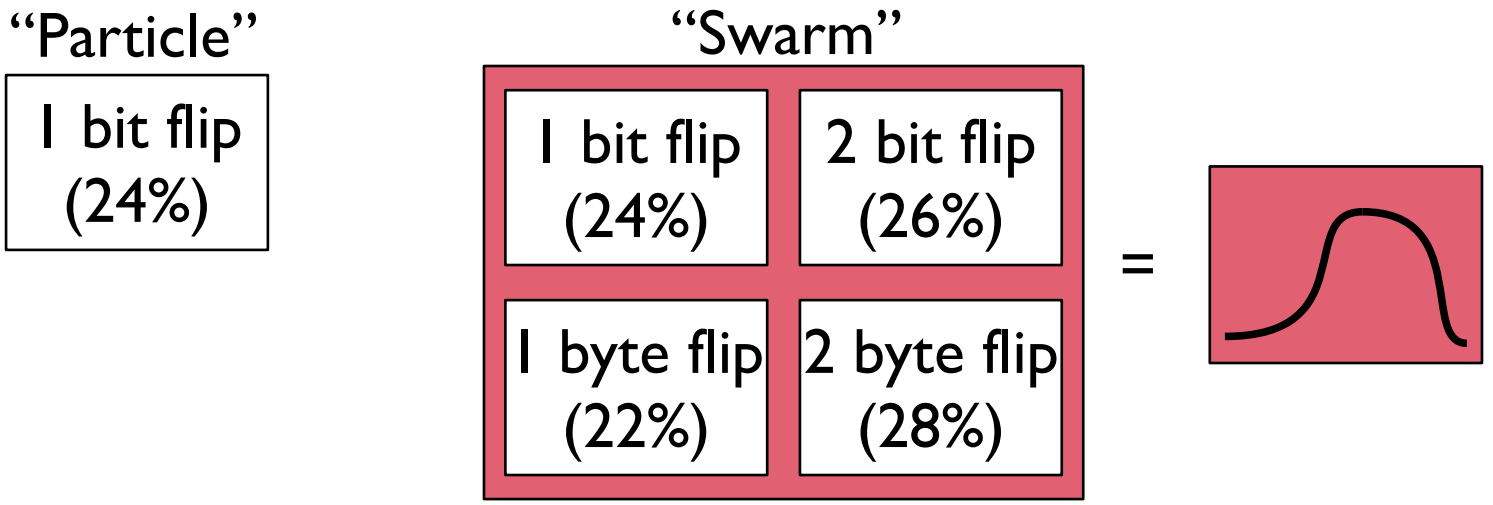
MOPT

- Key Idea: online learning algorithm
  - Present a customized particle swarm optimization algorithm.

Security'19

MOPT

- Key Idea: online learning algorithm
  - Present a customized particle swarm optimization algorithm.
    - A particle = A probability of selecting a single operator.
    - A swarm = A probabilistic distribution of selecting operators.



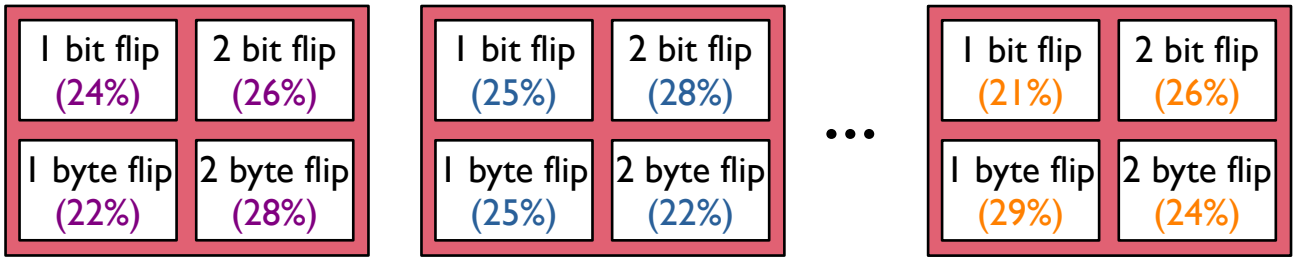
Security'19

MOPT

- Key Idea

  (1) Initialize *k* swarms. *(k=5)*

  - An initial swarm = a random distribution of mutation operators



Security'19

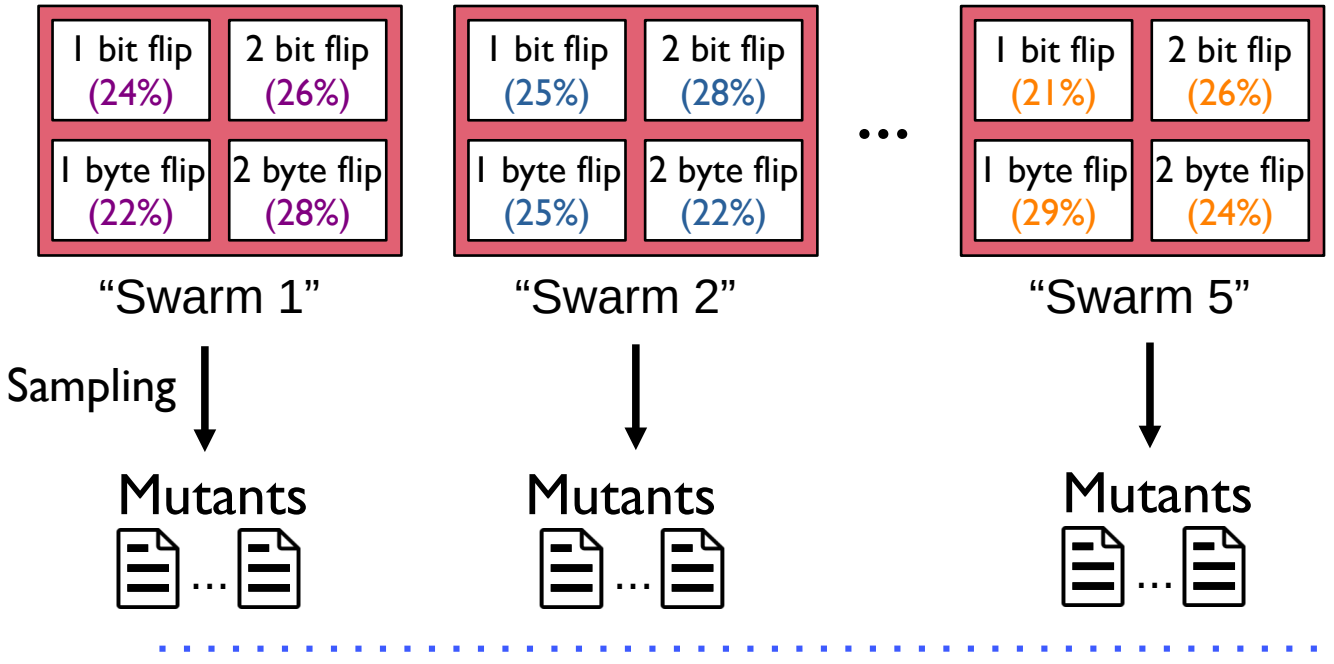
MOPT

- Key Idea

  (1) Initialize *k* swarms. *(k=5)*

  - An initial swarm = a random distribution of mutation operators

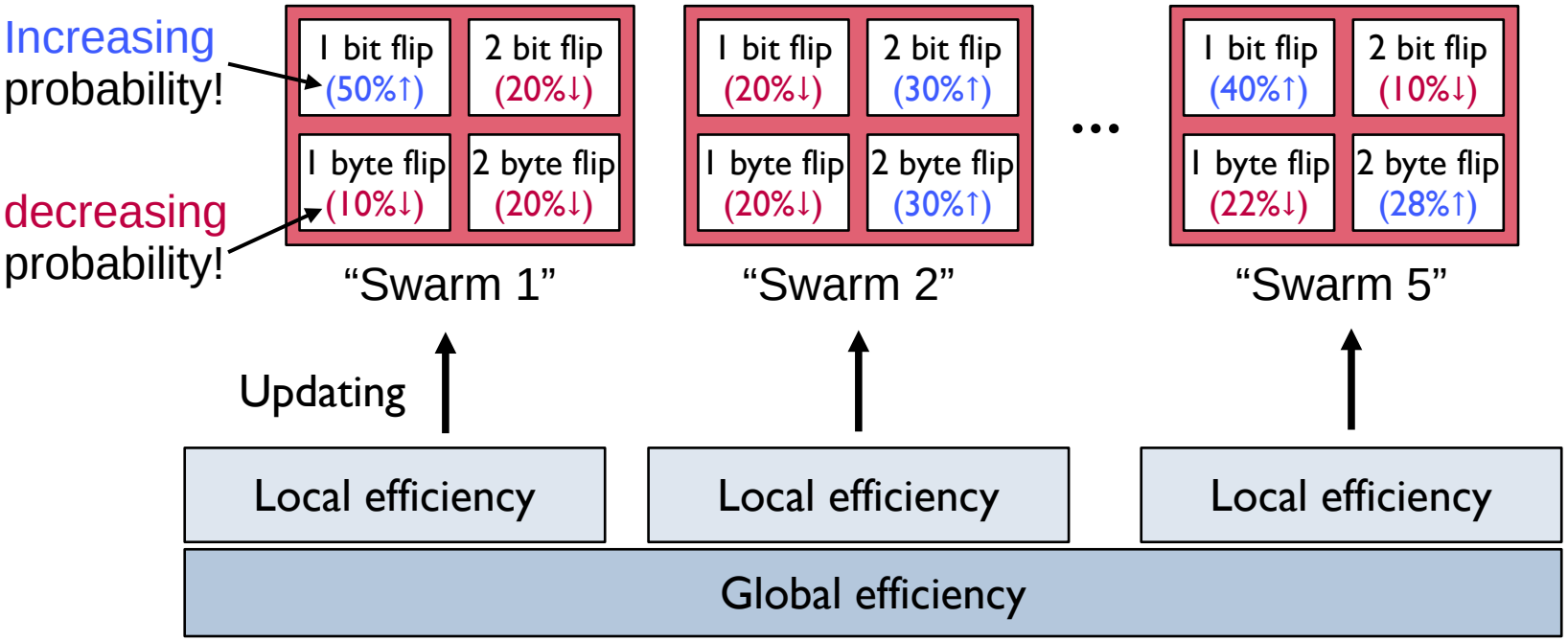
  (2) Generate mutants by sampling from *k* swarms, respectively.



Security'19

MOPT

- Key Idea
  - (3). Update *k* swarms based on local and global efficiency.
  - Intuition for the efficiency
    - How effective is each particle in generating interesting mutants?

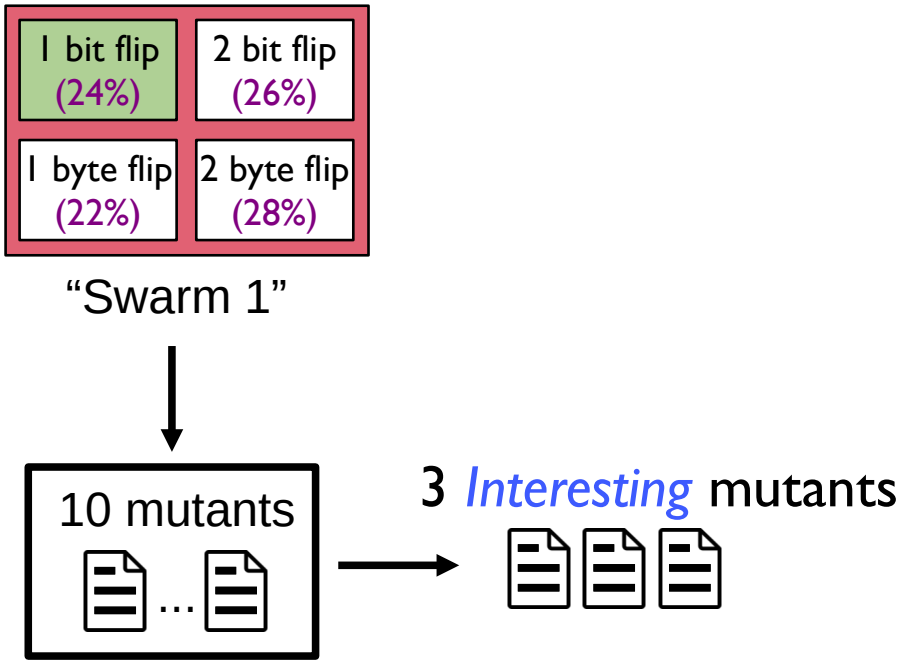


Security'19

MOPT

- Key Idea
  - (2). Updating *k* swarms based on local and global efficiency.
    - Local efficiency of a particle in a swarm

      (ex) Local efficiency of the particle ($op_{1\,bit\text{-}flip}$) for swarm 1

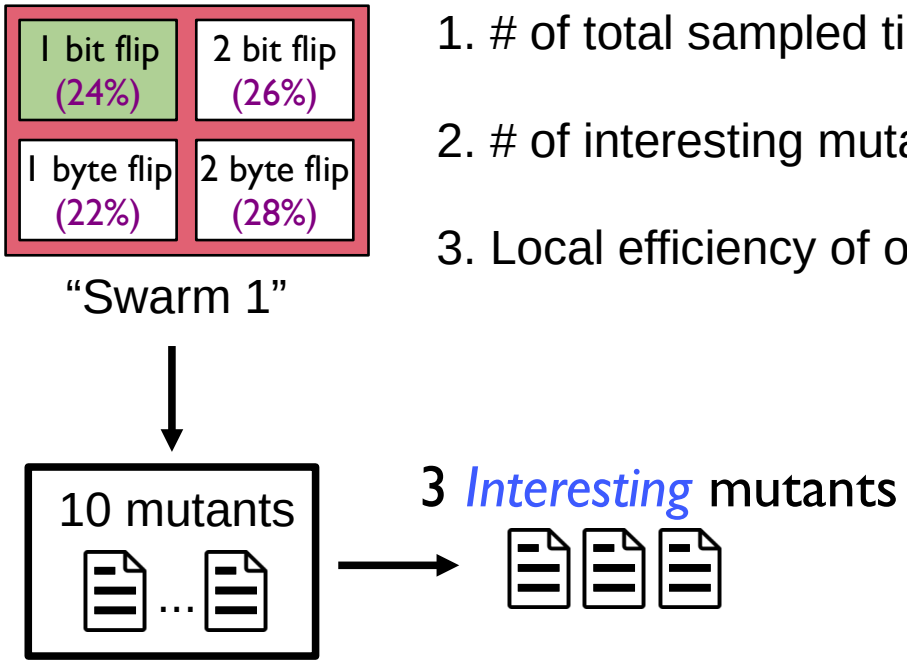


Security'19

MOPT

- Key Idea
  - (2). Updating *k* swarms based on local and global efficiency.
    - Local efficiency of a particle in a swarm

      (ex) Local efficiency of the particle ($op_{1bit\text{-}flip}$) for swarm 1



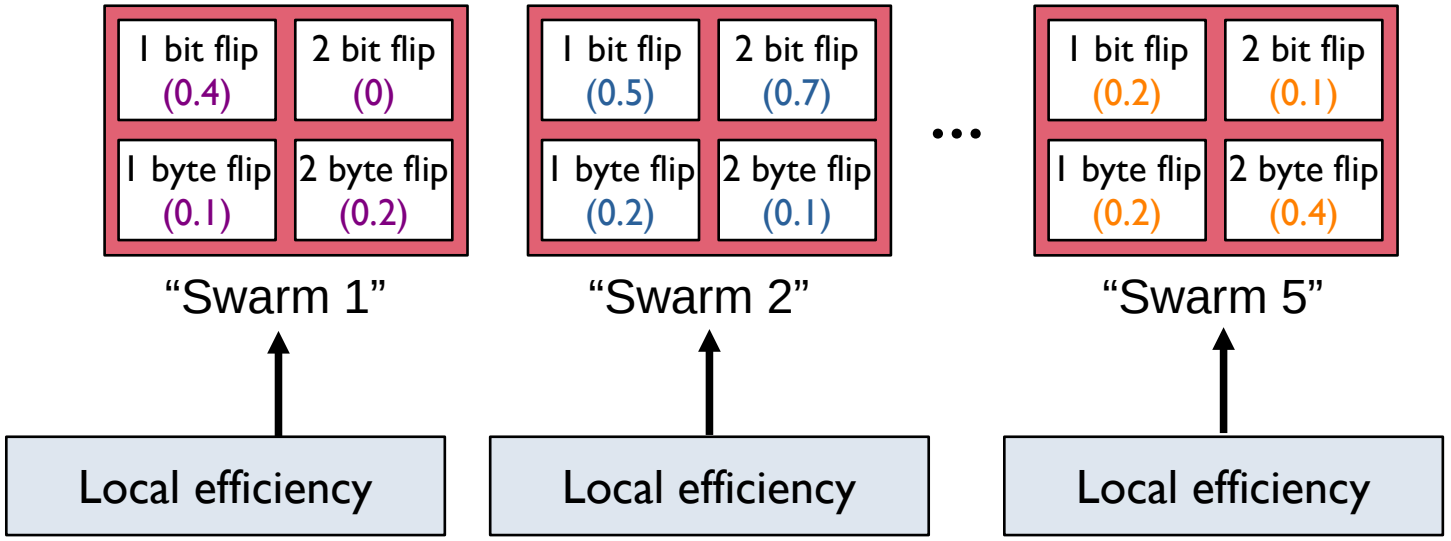
1. # of total sampled times for $op_{1bit\text{-}flip}$: 5
2. # of interesting mutants contributed by $op_{1bit\text{-}flip}$: 2
3. Local efficiency of $op_{1bit\text{-}flip}$: 0.4 (2/5)

Security'19

MOPT

- Key Idea
  - (2). Updating *k* swarms based on local and global efficiency.
    - Local efficiency of a particle in a swarm

      (ex) Local efficiency of the particle ($op_{1\,bit\text{-}flip}$) for swarm 1

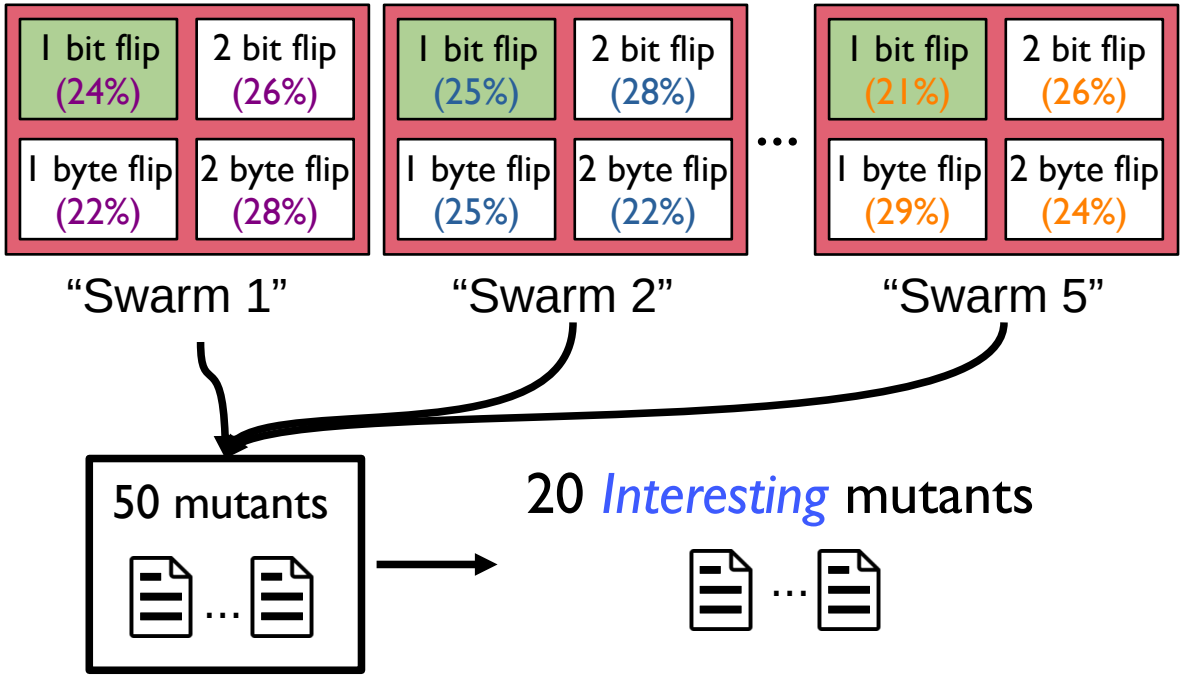


Security'19

MOPT

- Key Idea
  - (2). Updating *k* swarms based on local and global efficiency.
    - Global efficiency of a particle in 5 swarms

      (ex) Global efficiency of the particle ($op_{1bit\text{-}flip}$) for 5 swarms

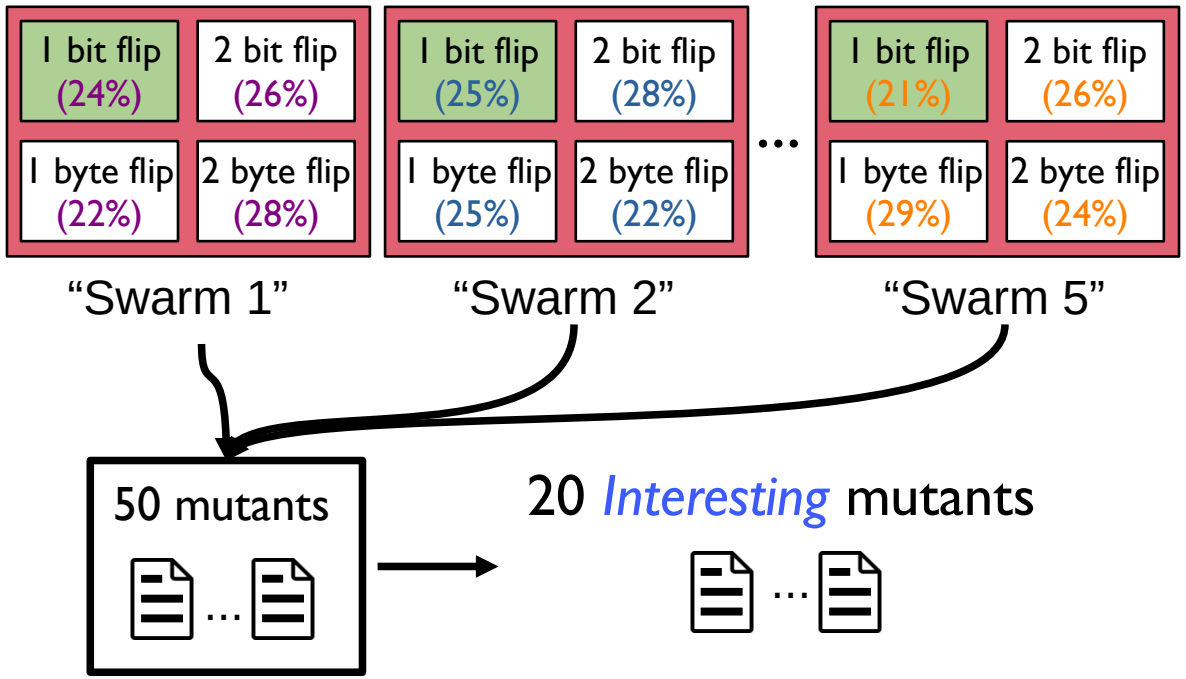


Security'19

MOPT

- Key Idea
  - (2). Updating *k* swarms based on local and global efficiency.
    - Global efficiency of a particle in 5 swarms

      (ex) Global efficiency of the particle ($op_{1bit\text{-}flip}$) for 5 swarms

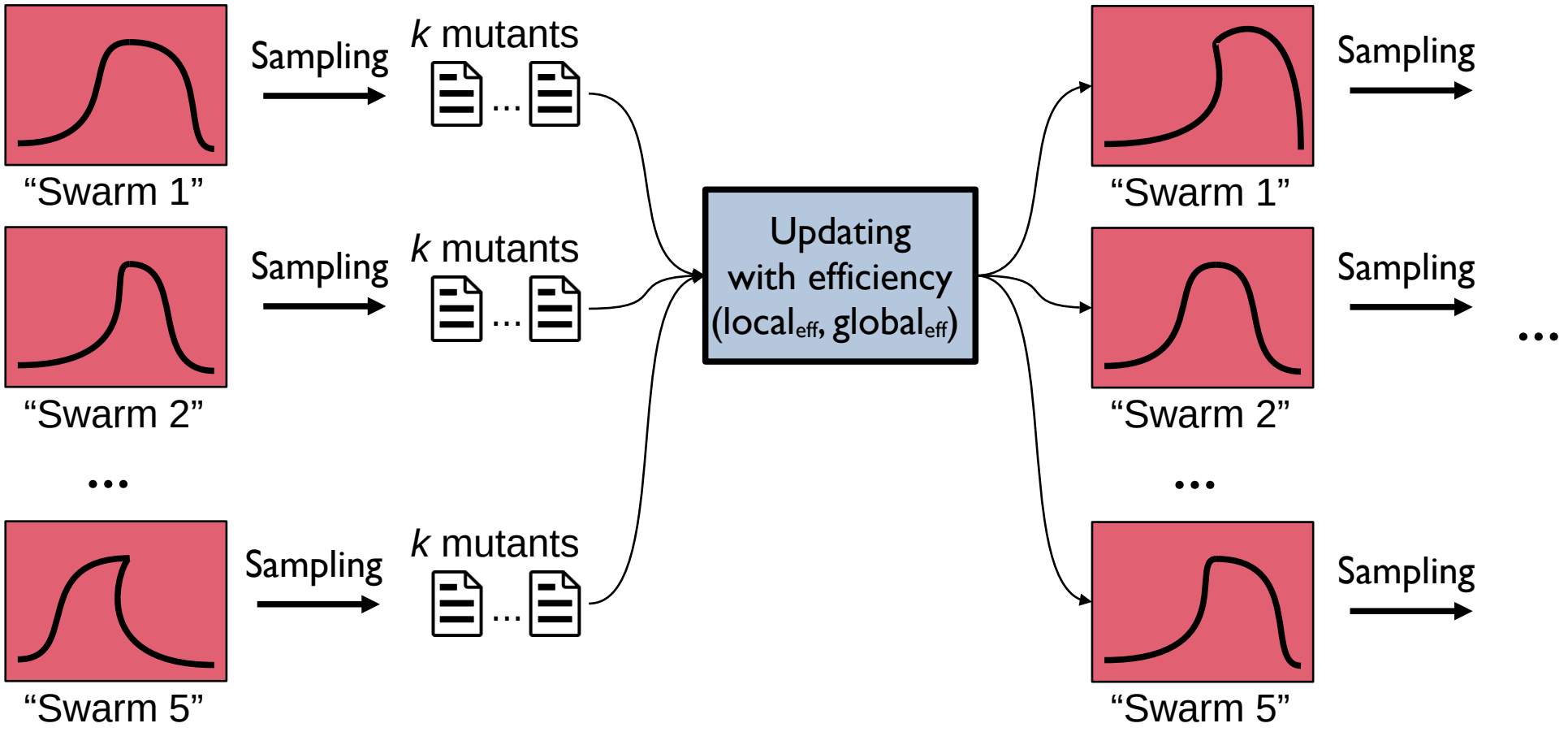


1. # of total times for $op_{1bit\text{-}flip}$: 20
2. # of interesting mutants by $op_{1bit\text{-}flip}$: 10
3. Global efficiency of $op_{1bit\text{-}flip}$: 0.5 (10/20)

Security'19

MOPT

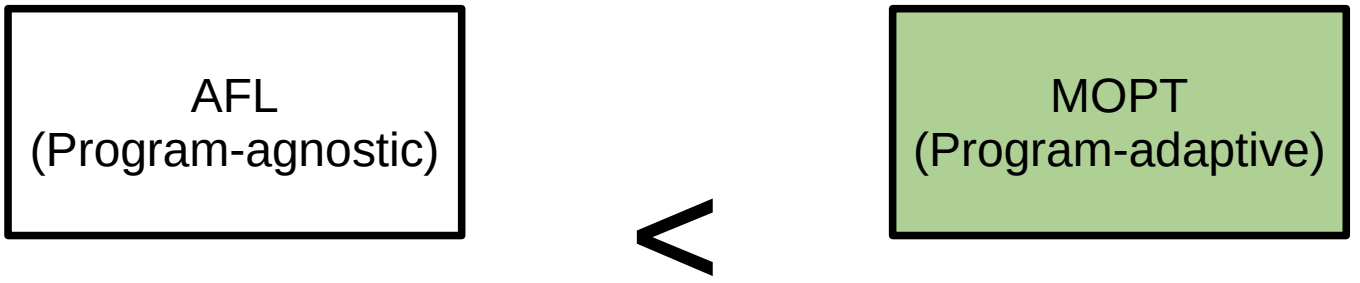
- Repeat the sampling and updating process.



Security'19

MOPT

- Effectiveness
  - Total benchmarks : 13 open-source linux programs.
  - Testing budget: 10 days.



610 unique crashes

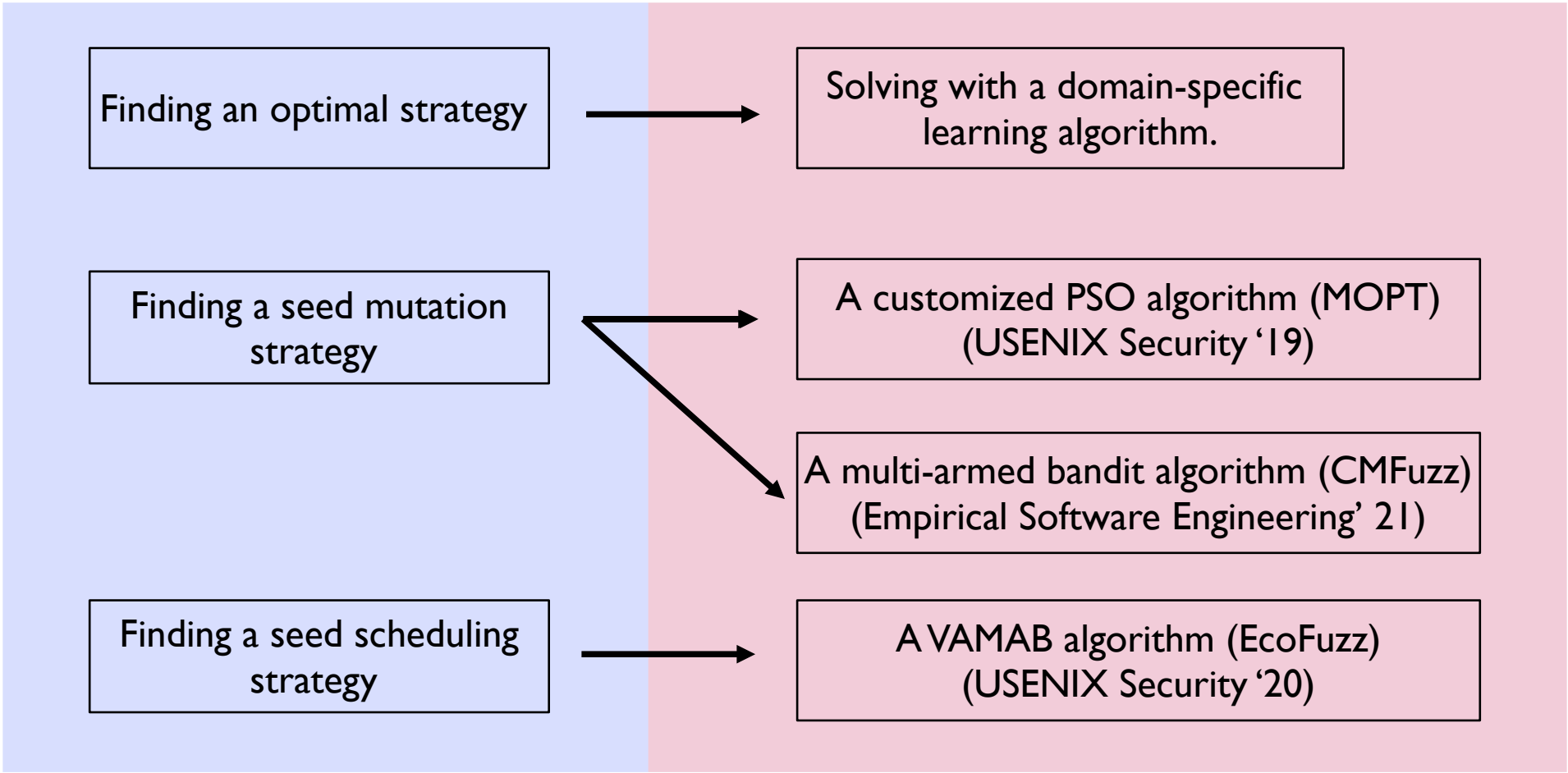
47,618 unique paths

2,944 unique crashes (x4.8)

104,133 unique paths (x2.2)

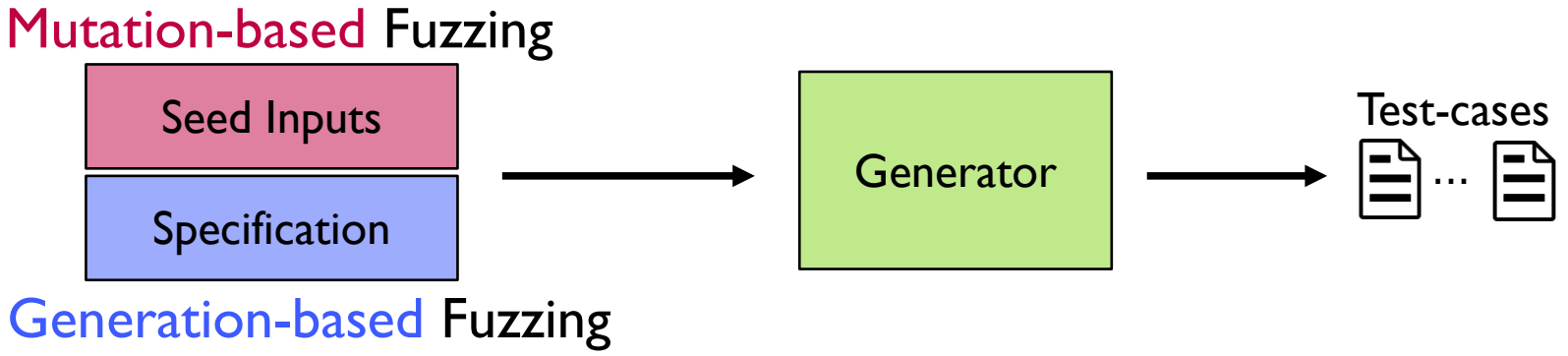
Recent Mutation-based Fuzzing

- Mutation-based Fuzzing + Learning



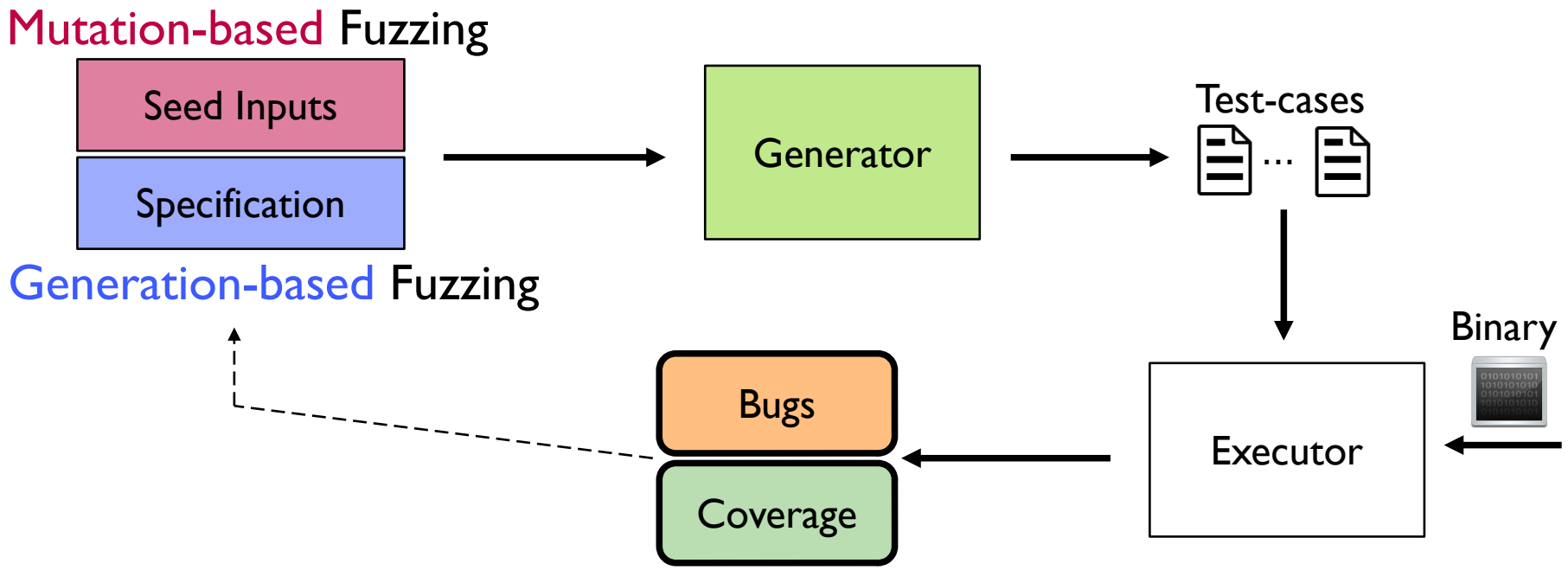
Generation-based Fuzzing

- Generation-based fuzzing vs Mutation-based fuzzing



Generation-based Fuzzing

- Generation-based fuzzing vs Mutation-based fuzzing



Generation-based Fuzzing

- Generate test-cases by using a specification.[1]
  - ex) a specification: grammars of valid inputs to a target program.

- A Grammar
  - Express the syntactical structure of an input. (e.g., javascript)
  - Consists of a start symbol and a set of production rules.
    - A start symbol: <start>
    - A production rule: <A> ::= <B>

1. https://www.fuzzingbook.org/

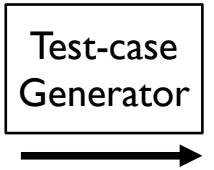
Generation-based Fuzzing

- Generate test-cases by using a specification.[1]
  - ex) a specification: grammars of valid inputs to a target program.

- A Grammar
  - Express the syntactical structure of an input. (e.g., javascript)
  - Consists of a start symbol and a set of production rules.
    - A start symbol: <start>
    - A production rule: <A> ::= <B>

<start> ::= <digit><digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Example Grammar>



{00, 01, …, 99}

1. https://www.fuzzingbook.org/

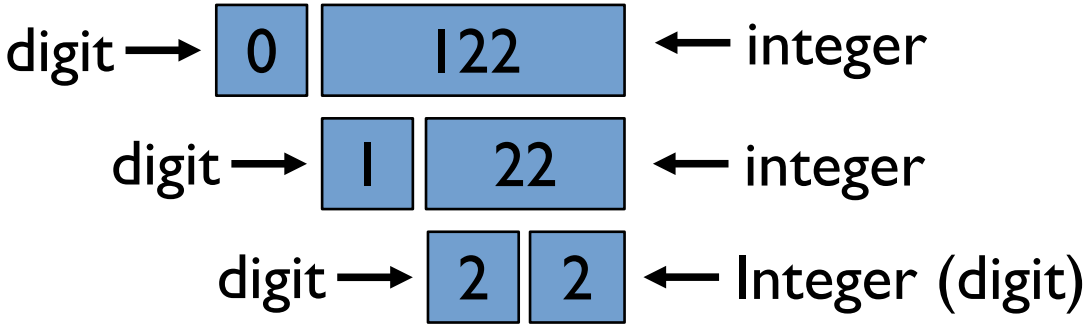
Grammars

- Grammars can be recursive.[1]

```
<start>   ::= <integer>
<integer> ::= <digit> | <digit><integer>
<digit>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<Example Grammar>

ex) A test-case: 0122



1. https://www.fuzzingbook.org/

Grammars

- More complex grammars for arithmetic expressions
  - Cover full arithmetic expressions

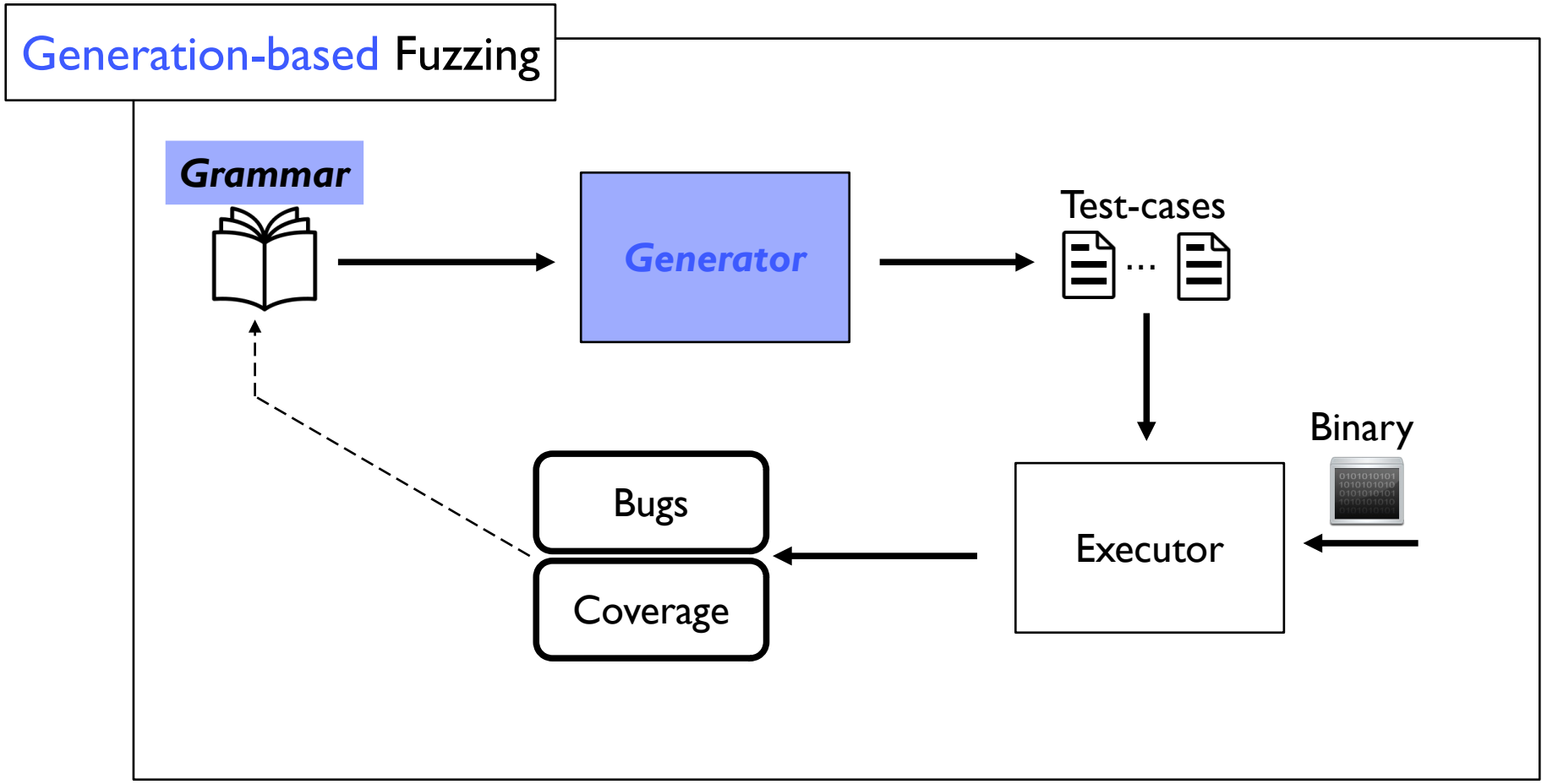
```
<start>   ::= <expr>
<expr>    ::= <term> + <expr> | <term> - <expr> | <term>
<term>    ::= <term> * <factor> | <term> / <factor> | <factor>
<factor>  ::= +<factor> | -<factor> | (<expr>) | <integer> | <integer>.<integer>
<integer> ::= <digit><integer> | <digit>
<digit>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

ex) A test-case: (1 + 2) * (3.4 / 5.6)

<start> → <expr> → <term> → <term> * <factor> → <factor> * <factor> →
(<expr>)*<factor> → (<term> + <expr>) * <factor> → (<factor> + <expr>) * <factor> →
(<integer> + <expr>) * <factor> → (<digit> + <expr>) * <factor> → (1 + <expr>) * <factor> →
(1 + <term>) * <factor> → (1 + <factor>) * <factor> → (1 + <integer>) * <factor> →
(1 + <digit>) * <factor> → (1 + 2) * <factor> → ... → (1 + 2) * (3.4 /5.6)

Topics in Generation-based Fuzzing

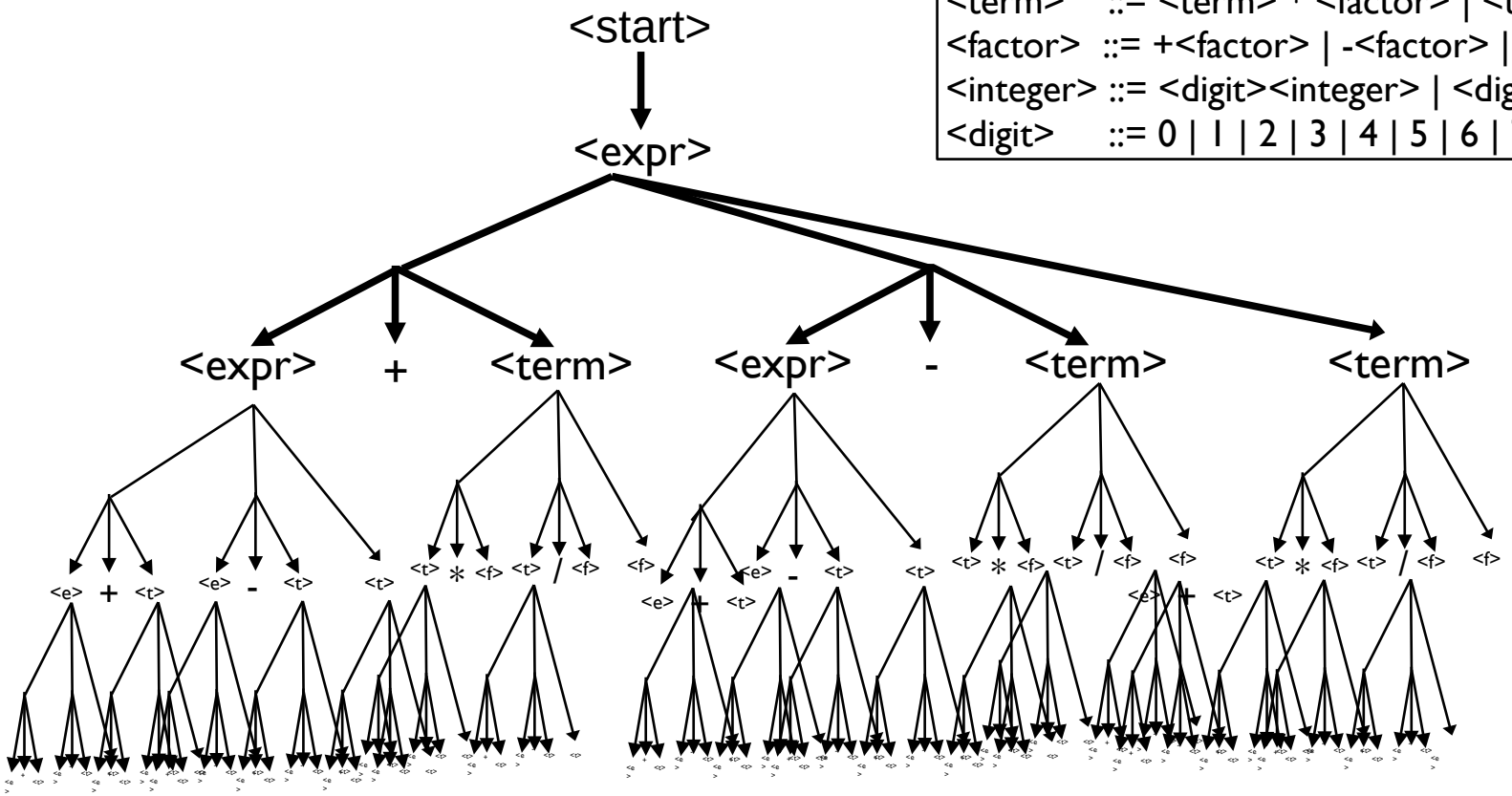
- How to build a good grammar?
- How to expand from a given grammar?



Generator

- How to expand from a given grammar?

```
<start>   ::= <expr>
<expr>    ::= <expr> + <term> | <expr> - <term> | <term>
<term>    ::= <term> * <factor> | <term> / <factor> | <factor>
<factor>  ::= +<factor> | -<factor> | (<expr>) | <integer>
<integer> ::= <digit><integer> | <digit>
<digit>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```



...

Generator

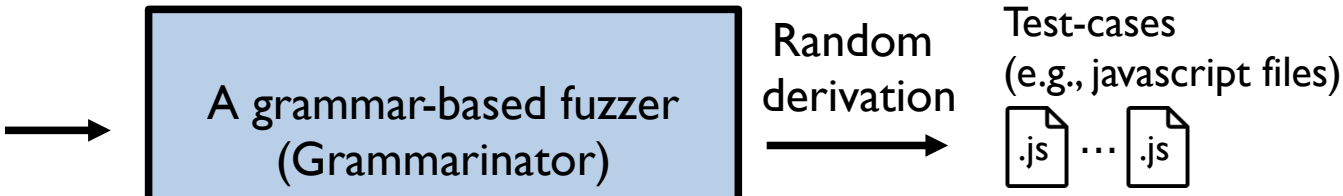
- Smart generation from the grammar is very important to the performance of generation-based fuzzing.

```
program
    : HashBangLine? sourceElements? EOF
    ;

sourceElement
    : statement
    ;

statement
    : block
    | variableStatement
    | importStatement
    | exportStatement
    | emptyStatement
    | classDeclaration
    | expressionStatement
    | ifStatement
    | iterationStatement
    | continueStatement
    | breakStatement
    | returnStatement
        ...
```

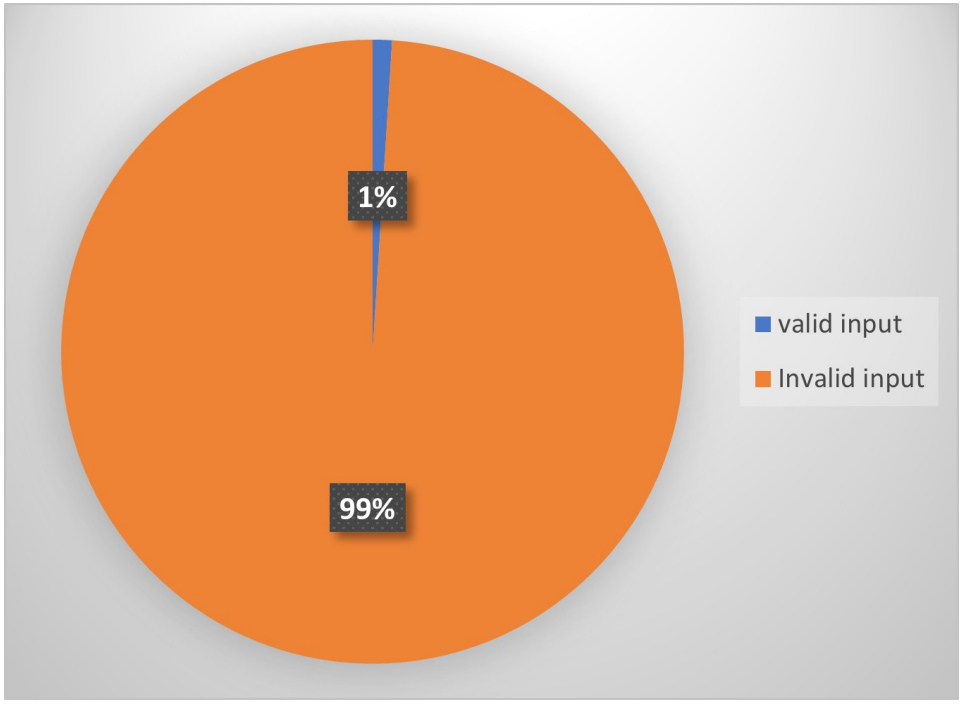
grammar



1. Grammarinator: A Grammar-Based Open Source Fuzzer

Generator

- Random generation from the grammar does not generate valid JavaScript files.



Examples for 1% valid Inputs

(ex1) class async{ ; }
(ex2) function async( ) { }
(ex3) try{ } catch{ }
(ex4) try{ } finally{ }

Grammar

- Building a good grammar is also essential.

```
program
    : HashBangLine? sourceElements? EOF
    ;

sourceElement
    : statement
    ;

statement
    : block
    | variableStatement
    | importStatement
    | exportStatement
    | emptyStatement
    | classDeclaration
    | expressionStatement
    | ifStatement
    | iterationStatement
    | continueStatement
    | breakStatement
    | returnStatement

        ...
```

Naive grammar

An example for 99% invalid Inputs

(ex1). return var I = 4 + 3
(Exception: Syntax Error: Return statements are only valid inside functions)

Summary

- Grey-box testing is classified into two methods.
  - Mutation-based fuzzing / Generation-based fuzzing
- Mutation-based fuzzing has three key components.
  - Seed pool generation / Seed selection / Seed mutation
- Recent trend in mutation-based fuzzing is …
  - Program-agnostic Fuzzing → Program-adaptive Fuzzing
- Generation-based Fuzzing
  - How to build a good grammar?
  - How to expand from a given grammar?

Thank You