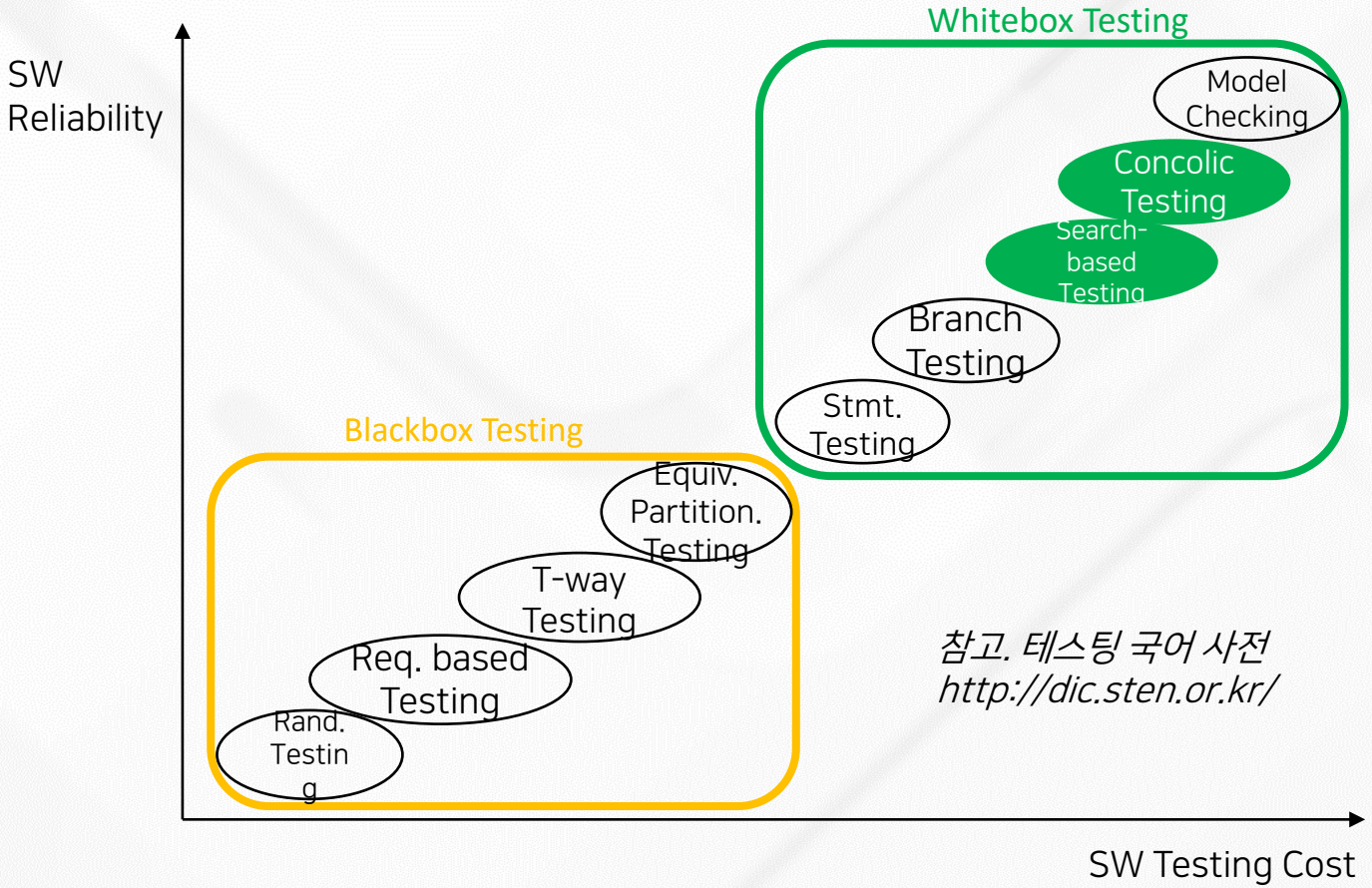
Automated Test Input Generation through Fuzzing

Moonzoo Kim

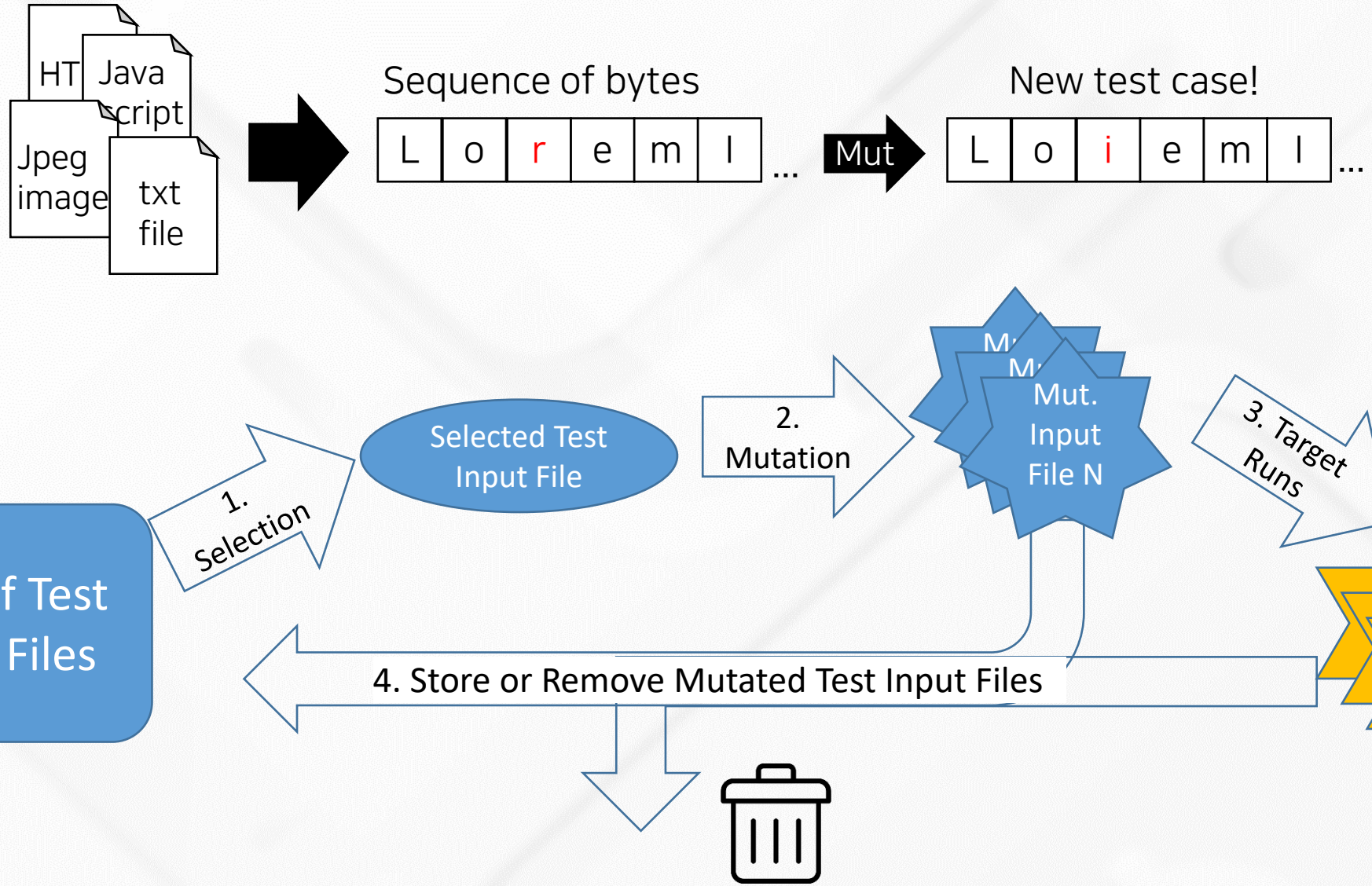
KAIST

Various SW Testing Techniques w/ Different Cost and Effectiveness



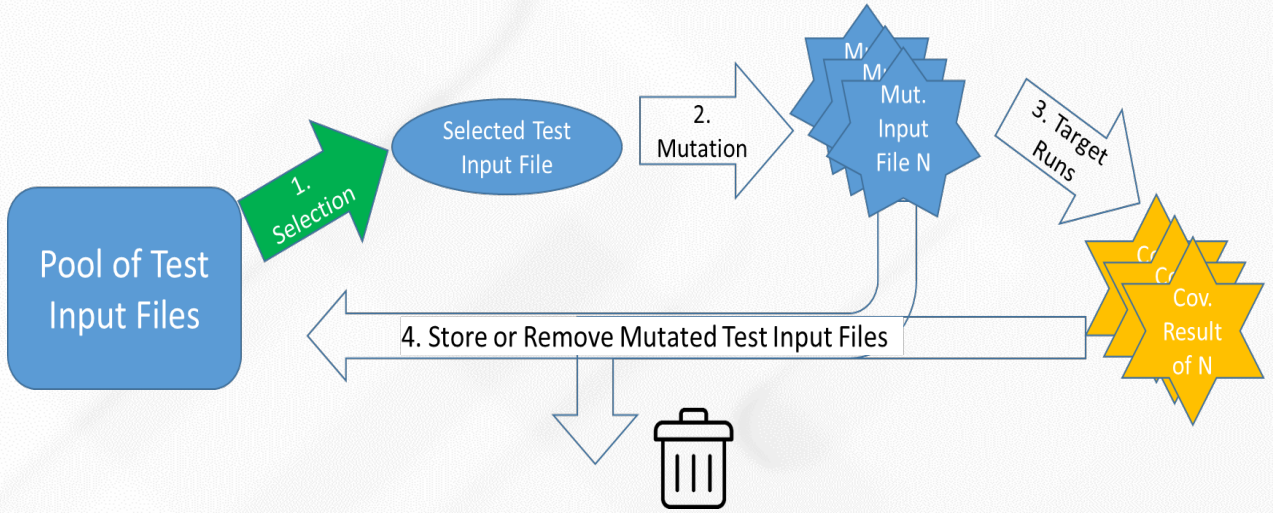
참고. 테스팅 국어 사전
http://dic.sten.or.kr/

Fuzzing - Automated Test Input Generation via Random Mutation



Fuzzing Challenge #1

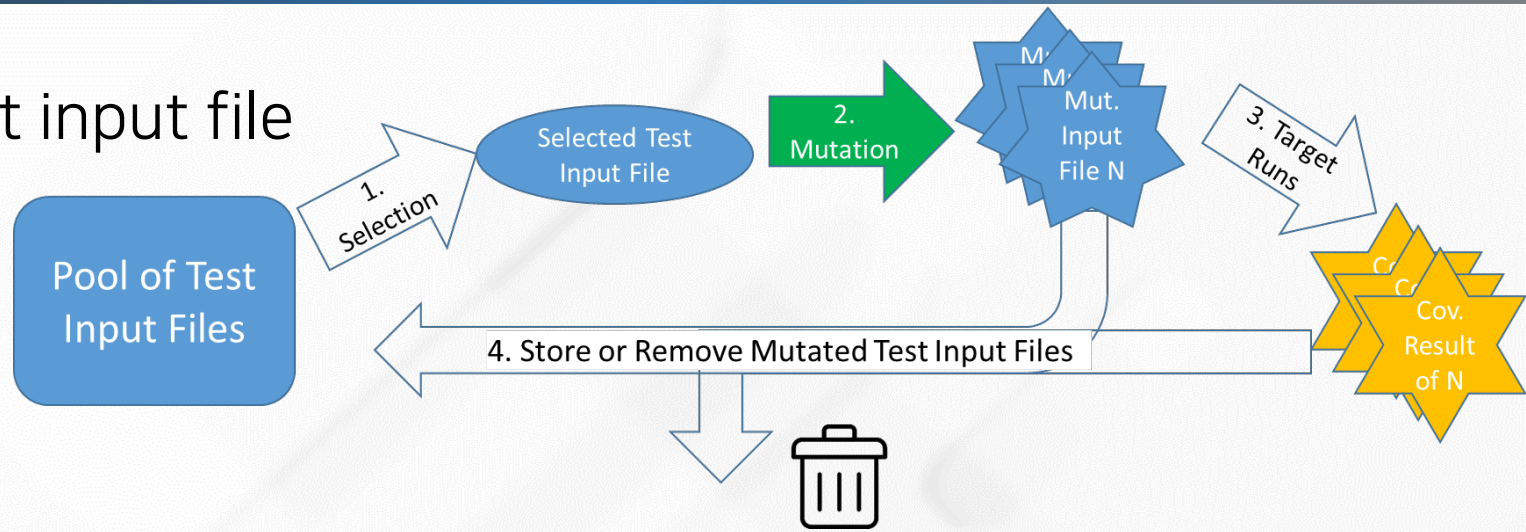
Which test input file to select to mutate?



Fuzzer	Heuristic
AFL	Favor test inputs whose execution time and length are short - semantic characteristics of test inputs are ignored)
FairFuzz (ASE 2018), Vuzzer (NDSS 2017)	Favor test inputs such that those test inputs covers hardly covered branches - semantic characteristics of test inputs are a little bit used
CollAFL (SP 2018), Angora (SP 2018)	Favor test inputs that cover many branches whose branch condition statements are executed but these branches are rarely covered - semantic characteristics of test inputs are a little bit used

Fuzzing Challenge #2

Which bytes in the selected test input file should be mutated?



Fuzzer	Heuristic
AFL	Bytes of random # are randomly selected to mutate - No semantic information on the bytes are utilized
FairFuzz (ASE 2018), Profuzzer(SP 2019), GreyOne (USENIX security '20)	Guess which bytes are important based on the runtime information (e.g., if one byte is mutated and the runtime information does not change at all, a fuzzer does not select that byte afterward)
BuzzFuzz(ICSE 2009), Dowser(USENIX security '13), Vuzzer(NDSS 2017), Angora (SP 2018), Matryoska(CCS '19)	Select the bytes which have dependency with "important" branches through Dynamic Taint Analysis (DTA) - Control dependency is not considered

SBST

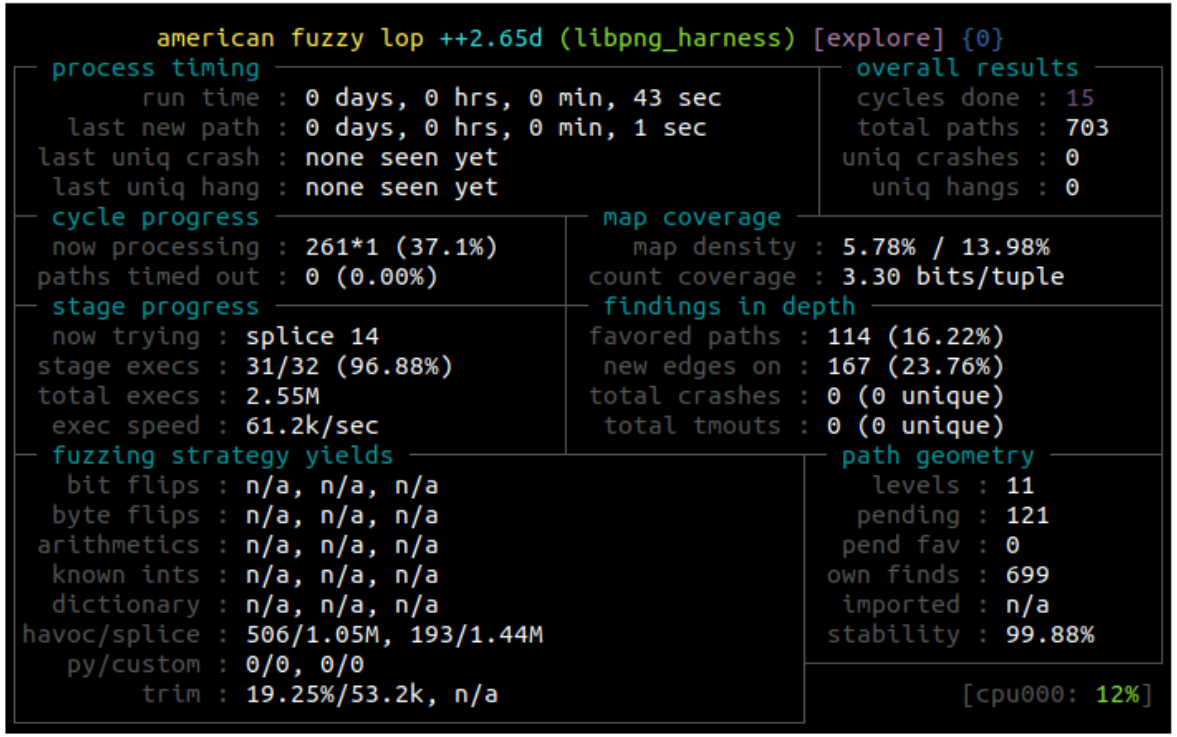
stands for

Search Based Software Testing

Abbreviations.com

AFL++ Overview

AFLplusplus is the daughter of the American Fuzzy Lop fuzzer by Michał "lcamtuf" Zalewski and was created initially to incorporate all the best features developed in the years for the fuzzers in the AFL family and not merged in AFL cause it is not updated since November 2017.



The AFL++ fuzzing framework includes the following:

- A fuzzer with many mutators and configurations: afl-fuzz.
- Different source code instrumentation modules: LLVM mode, afl-as, GCC plugin.
- Different binary code instrumentation modules: QEMU mode, Unicorn mode, QBDI mode.
- Utilities for testcase/corpus minimization: afl-tmin, afl-cmin.
- Helper libraries: libtokencap, libdislocator, libcompcov.

OSS-Fuzz

Fuzz testing is a well-known technique for uncovering programming errors in software. Many of these detectable errors, like buffer overflow, can have serious security implications. Google has found thousands of security vulnerabilities and stability bugs by deploying guided in-process fuzzing of Chrome components, and we now want to share that service with the open source community.

In cooperation with the Core Infrastructure Initiative and the OpenSSF, OSS-Fuzz aims to make common open source software more secure and stable by combining modern fuzzing techniques with scalable, distributed execution. Projects that do not qualify for OSS-Fuzz (e.g. closed source) can run their own instances of ClusterFuzz or ClusterFuzzLite.

We support the libFuzzer, AFL++, and Honggfuzz fuzzing engines in combination with Sanitizers, as well as ClusterFuzz, a distributed fuzzer execution environment and reporting tool.

Currently, OSS-Fuzz supports C/C++, Rust, Go, Python and Java/JVM code. Other languages supported by LLVM may work too. OSS-Fuzz supports fuzzing x86_64 and i386 builds.

Learn more about fuzzing

This documentation describes how to use OSS-Fuzz service for your open source project. To learn more about fuzzing in general, we recommend reading libFuzzer tutorial and the other docs in google/fuzzing repository. These and some other resources are listed on the useful links page.

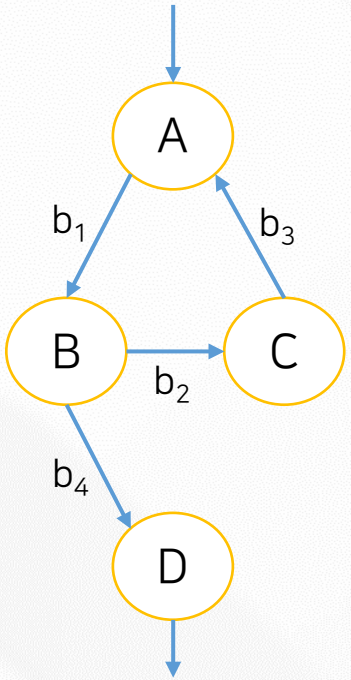
Trophies

As of June 2021, OSS-Fuzz has found over 30,000 bugs in 500 open source projects.

Fuzzing Algorithm of AFL/AFL++

1. How execution paths of test input files are defined and analyzed
2. How to select test input files
3. How to mutate selected test input files

Example of a New Path



- A path is considered new, if it covers a new branch, or a branch's hit count is in a new range of hit counts.
  - The list of ranges are as follows:
    - [1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+]
    - These ranges are called buckets.
- For 2 test input files that have the same branch coverage
  - tc1: A-B-C-A-B-D
  - tc2: A-B-C-A-B-C-A-B-D
- Branch hit count of tc1:

Branch ID	b_1	b_2	b_3	b_4
Hit count	2	1	1	1

- Branch hit count of tc2:

Branch ID	b_1	b_2	b_3	b_4
Hit count	3	2	2	1

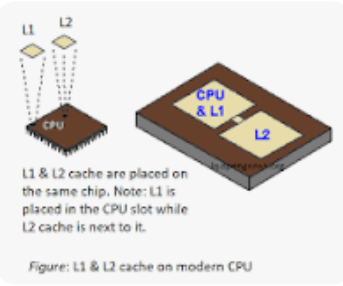
- tc2 covers a new path compared to tc1
  - the hit count for b_1, b_2, and b_3 of tc1 and tc2 cover different buckets.

How Path Information is Stored by AFL

- Branch hit count is used to find a new path
- AFL keeps the path coverage data in a 64kB mem. Each byte represent hit count bucket for each branch as follows:

L1 Cache Size

Usually, the size of L1 cache range from 16KB to 64KB. Higher the L1 cache size, Higher is the System Performance in general. Note: In few systems, the size of Instruction Cache is more than the size of Data Cache while the common



1 byte (8 bits)

1 hit bucket: 1	1 hit bucket: 0
2 hits bucket: 1	2 hits bucket: 1
3 hits bucket: 1	3 hits bucket: 0
4-7 hits bucket: 0	4-7 hits bucket: 0
8-15 hits bucket: 0	8-15 hits bucket: 0
16-31 hits bucket: 0	16-31 hits bucket: 1
32-127 hits bucket: 0	32-127 hits bucket: 0
128+ hits bucket: 0	128+ hits bucket: 0

...

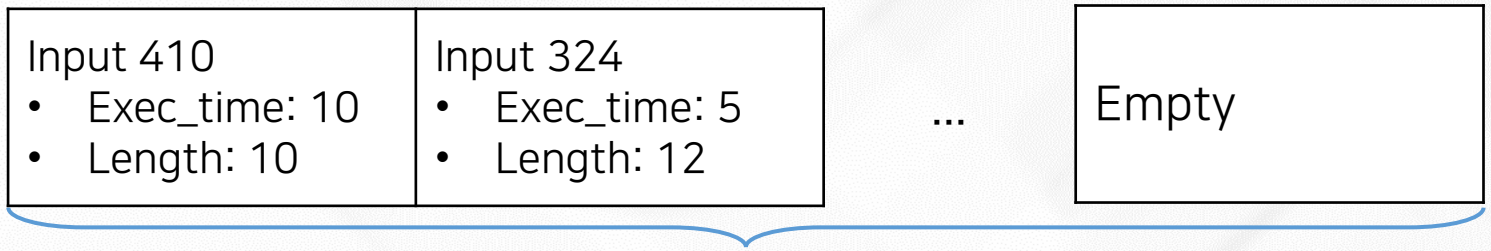
1 hit bucket: 0
2 hits bucket: 0
3 hits bucket: 0
4-7 hits bucket: 1
8-15 hits bucket: 1
16-31 hits bucket: 0
32-127 hits bucket: 0
128+ hits bucket: 0

64K branches

- Note: multiple branches may use the same byte
  - the location of a branch's byte is determined by hashing (i.e., hash collision)

Finding the Set of Favored Inputs

- Def) a favored input for a branch *b*:
  - input that has the lowest performance score for *b*:
  - Performance score: exec_time(μs) x length(byte)
    - exec_time: time it takes for the target program to execute the input in nanoseconds
    - length: size of the input in bytes

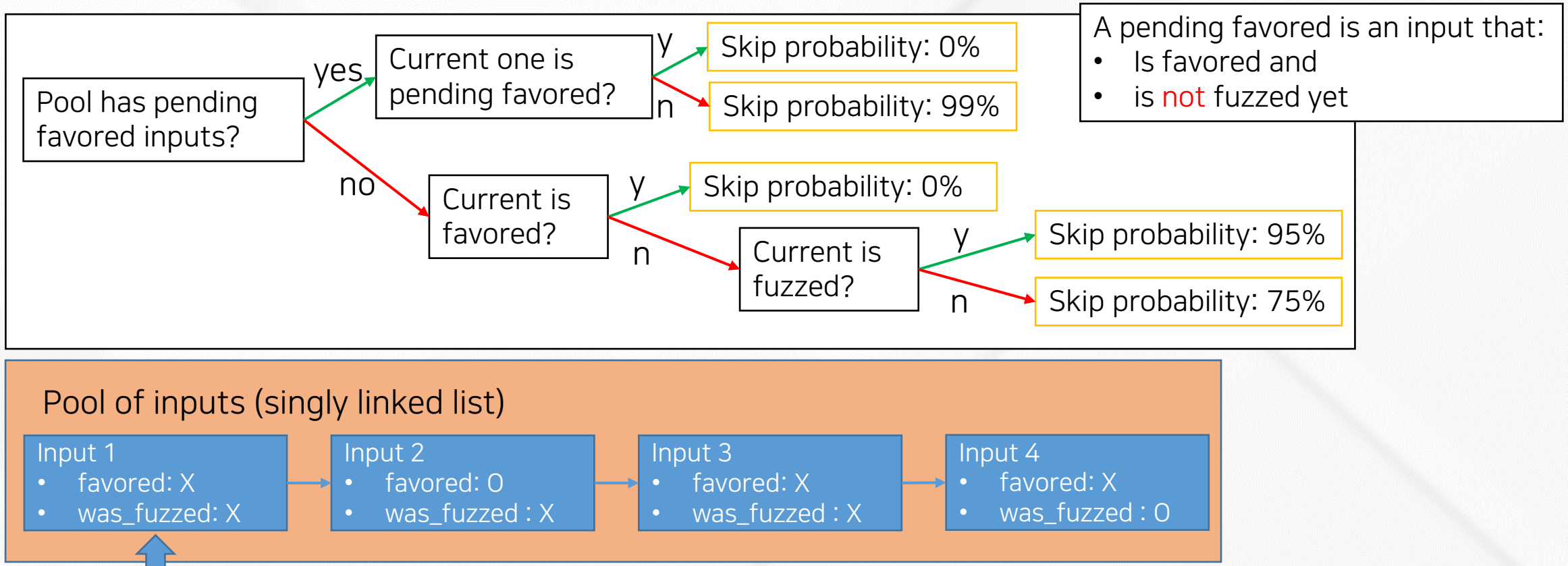


64K favored inputs corresponding to the 64K branches

- AFL selects favored inputs at:
  - the initialization phase after calculating performance score based on the initial seed inputs, and
  - the beginning of each fuzzing cycle

Selecting Input from Pool (1/4)

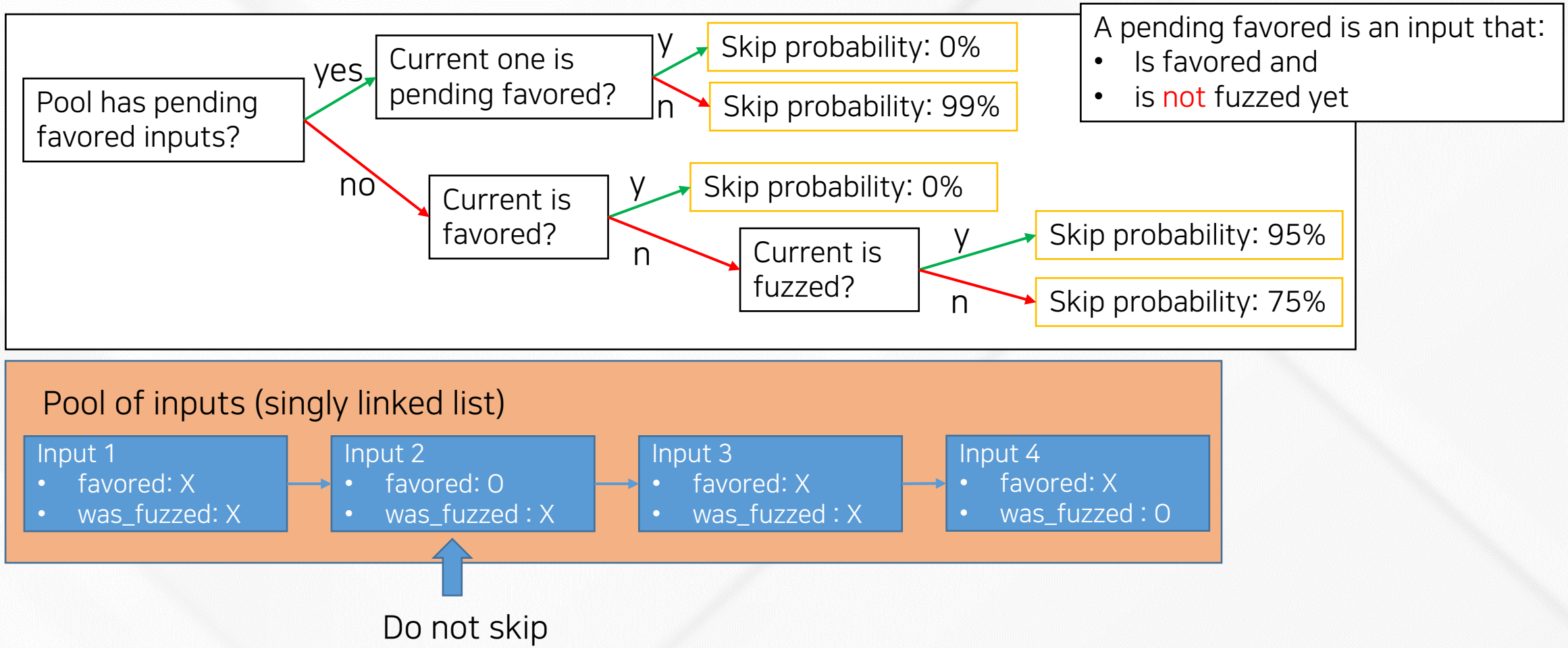
AFL can skip inputs in the pool with probability decided by the following algorithm:



Skip with 99% chance

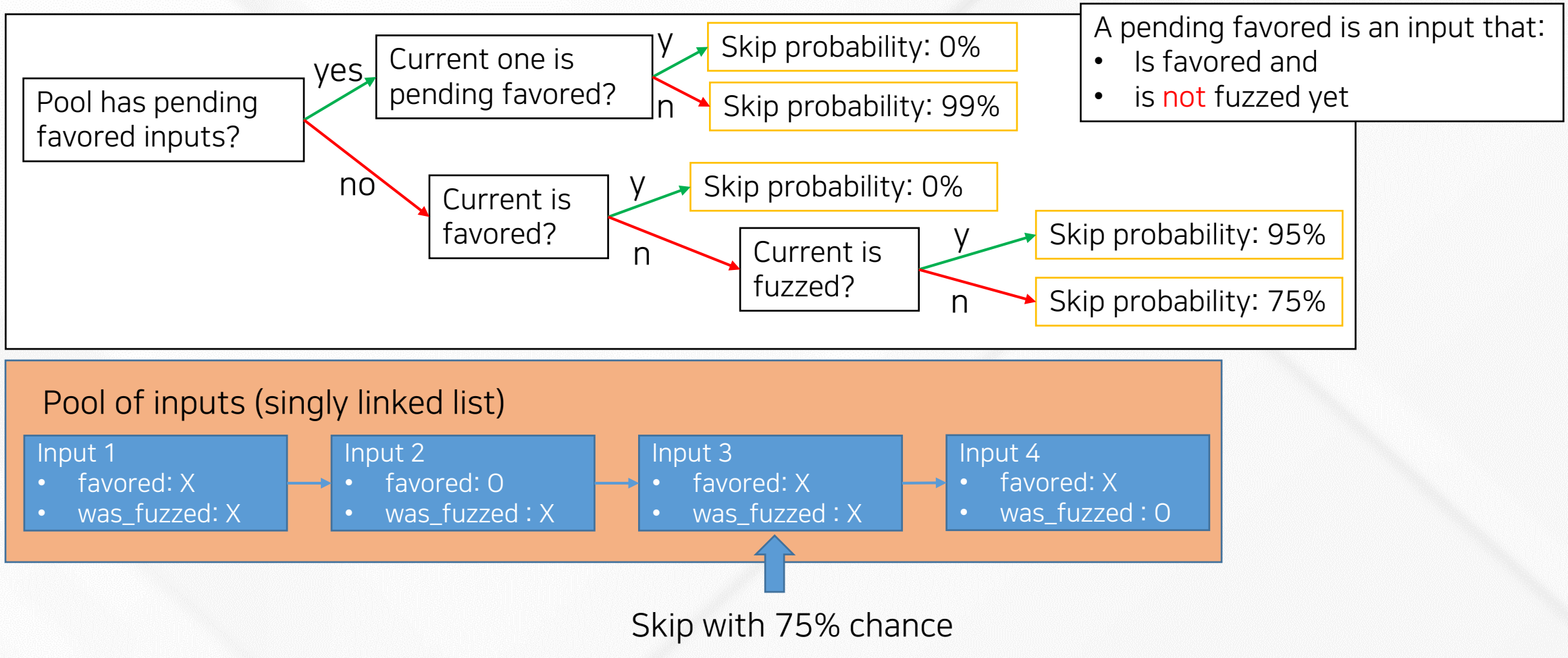
Selecting Input from Pool (2/4)

AFL can skip inputs in the pool with probability decided by the following algorithm:



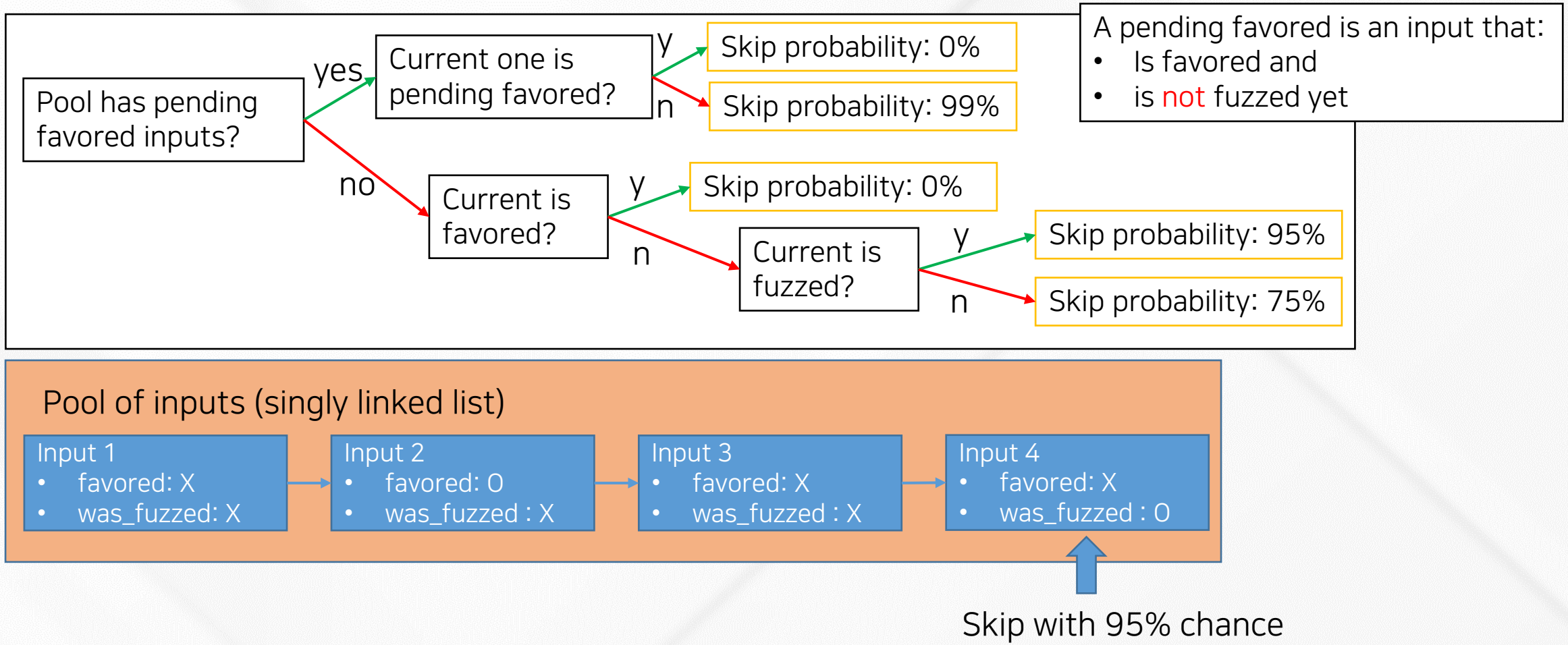
Selecting Input from Pool (3/4)

AFL can skip inputs in the pool with probability decided by the following algorithm:



Selecting Input from Pool (3/4)

AFL can skip inputs in the pool with probability decided by the following algorithm:



Fuzzing/Mutating Input Bytes

The selected input file is fuzzed using the following fuzzing methods in order:

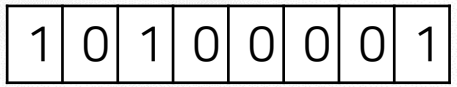
1. **Bitflip** – flip 1 or 2 or 4 bits of the input
2. **Byteflip** – flip 1 or 2 or 4 bytes of the input
3. **Arithmetic** – add or subtract an integer up to 35 to 8-bit or 16-bit or 32-bit values of the input
4. **Interest** – similar to arithmetic, but overwrite interesting values instead of add or subtract
5. **Extras** – overwrite or insert to the input using user-given or auto-generated terms
6. **Havoc** – makes a random number of modifications to the input using the above 5 methods
7. **Splice** – splice the input with another in the pool and apply havoc

Fuzzing Inputs – bitflip, byteflip

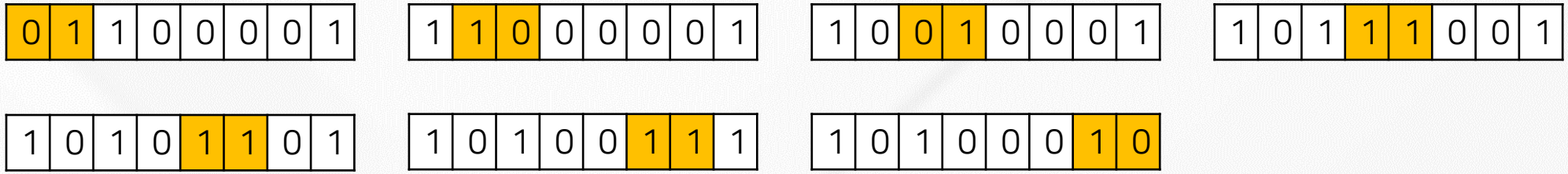
AFL flips L bits at a time, stepping over the input file by S-bit increments. The possible L/S variants are:

- 1/1, 2/1 and 4/1 for bitflip
- 8/8, 16/8, 32/8 for byteflip

Example. The following input (of size 1 byte) is represented in bits as follows:



If we apply bitflip 2/1 to the input, it will produce the following fuzzed inputs:

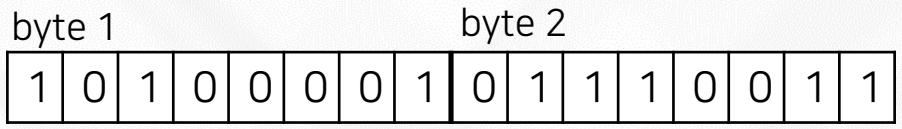


Fuzzing Inputs – arithmetic

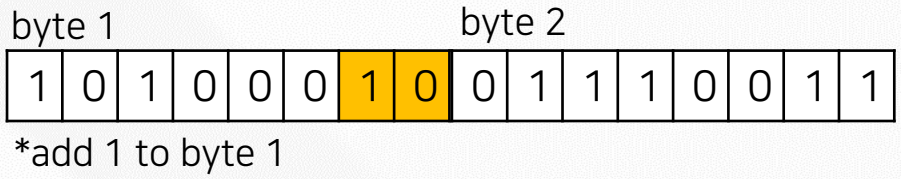
AFL adds or subtracts integers ranging from 1 to 35 to 8-bit, 16-bit and 32-bit values of the input while stepping over by 8 bits. The possible variants are:

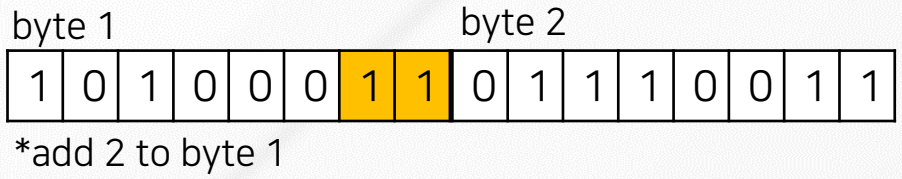
- arith 8/8, arith 16/8, and arith 32/8

Example. The following input (of size 2 bytes) is represented in bits as follows:

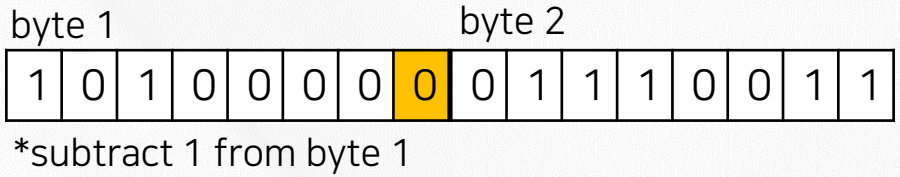


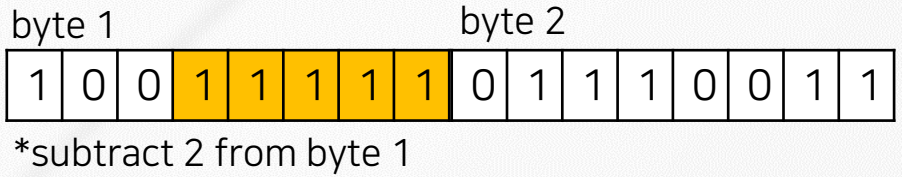
If we apply arith 8/8 to the input, it will produce the following fuzzed inputs:





... (up to 35)





... (up to 35)

Fuzzing Inputs – interest (1/2)

AFL overwrites interesting values to 8-bit, 16-bit and 32-bit values of the input while stepping over by 8 bits. The possible variants are:

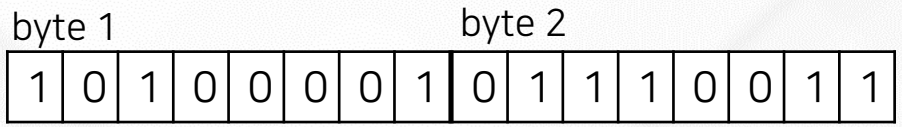
- interest 8/8, interest 16/8, and interest 32/8

The list of interesting values are as follows:

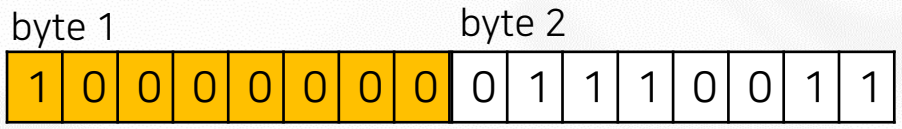
- interesting 8-bit values:
  - -128, -1, 0, 1, 16, 32, 64, 100, 127
- interesting 16-bit values:
  - -32768, -129, 128, 255, 256, 512, 1000, 1024, 4096, 32767
- interesting 32-bit values:
  - -2147483648, -100663046, -32769, 32768, 65535, 65536, 100663045, 2147483647

Fuzzing Inputs – interest (2/2)

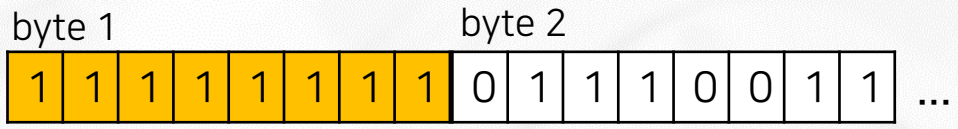
Example. The following input (of size 2 bytes) is represented in bits as follows:



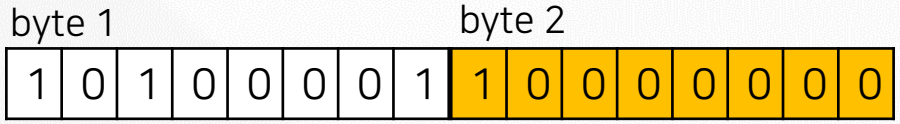
If we apply interest 8/8 to the input, it will produce the following fuzzed inputs:



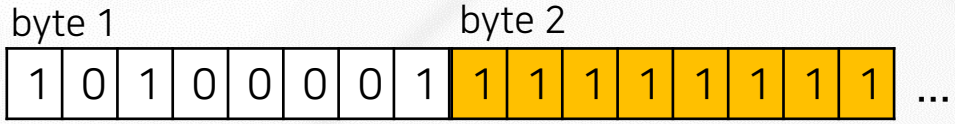
*overwrite -128 to byte 1



*overwrite -1 to byte 1



*overwrite -128 to byte 2



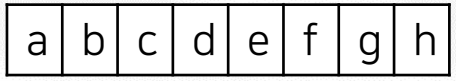
*overwrite -1 to byte 2

Fuzzing Inputs – extras

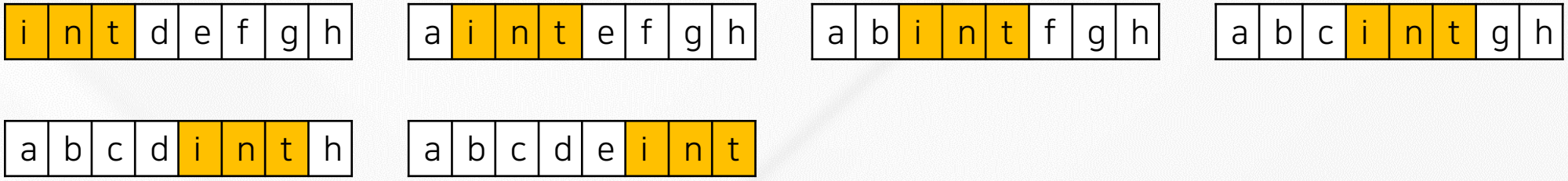
AFL overwrites or inserts dictionary terms to the input. The dictionary terms can be given by the user or automatically generated by the fuzzer. The possible variants are:

- user extras (over), user extras(insert) - overwrite or insert user given terms
- auto extras (over), auto extras(insert) - overwrite or insert auto generated terms

Example. The following input (of size 8 bytes) is represented in characters:



If we overwrite an arbitrary dictionary term “int”, it will produce the following fuzzed inputs:



Fuzzing Inputs – havoc

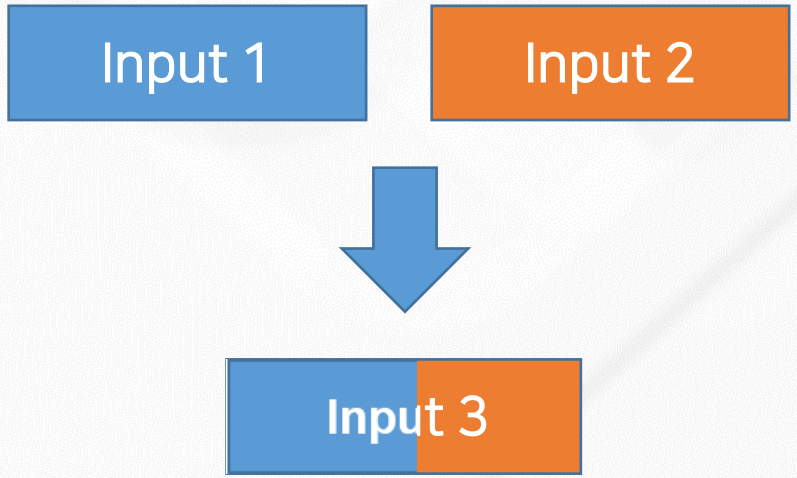
AFL makes a random number (max 128) of random edits to the input. The number of fuzzed inputs produced is proportional to the performance score of the input.

The list of possible edits are as follows:

- bitflip 1/1
- interest 8/8, 16/8, 32/8
- arith 8/8, 16/8, 32/8
- User extra (over,insert), auto extra (over,insert)
- Set a random byte to a random value
- Remove random number of bytes from random location
- Copy random number of bytes to random location

Fuzzing Inputs – splice

AFL splices together two random inputs from the queue at some arbitrarily selected midpoint and apply havoc
- Applied only after the first full queue cycle ends with no new paths.



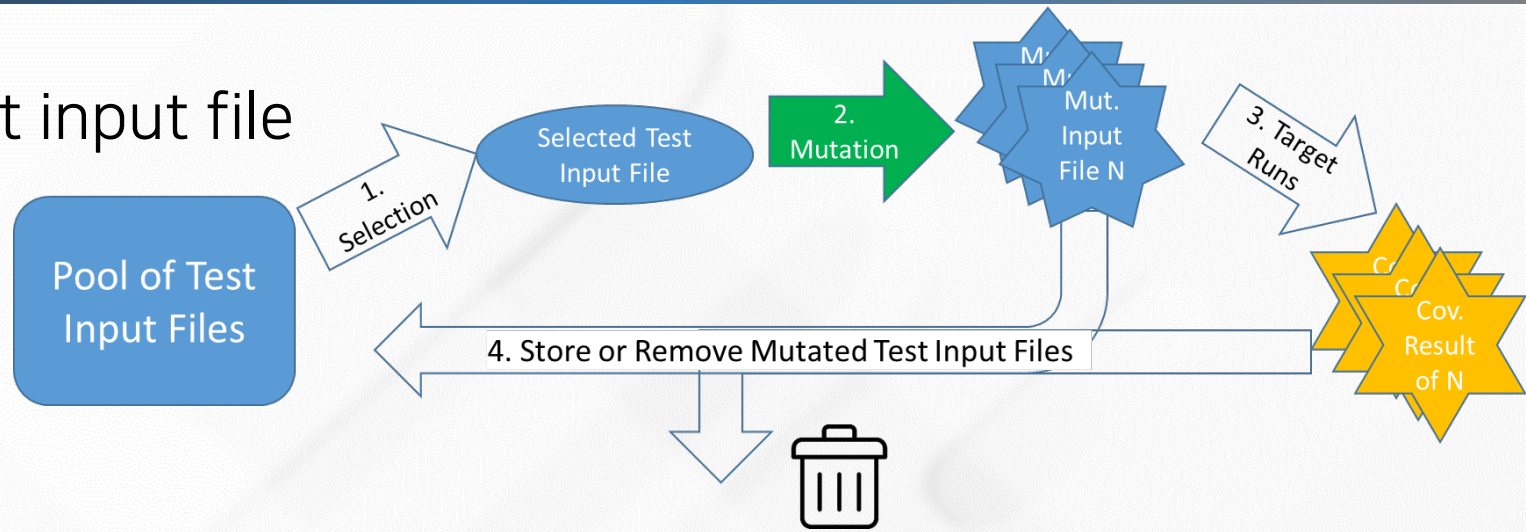
Fuzzing Challenge #1-alternatives

Fuzzing Techniques of Different Characteristics

- PerfFuzz (ISSTA 2018)
  - Aims to generate test inputs whose execution time is very long
    - Testing target program performance
  - Favors and mutates test input files that execute a large # of instructions
- MemFuzz (ICST 2019)
  - Aims to generate test inputs that contain many memory read/write operations
    - conjecture: many SW bugs are caused by memory access operations
  - Favors and mutates test input files that contains a large # of instructions for memory access operations

Fuzzing Challenge #2

Which bytes in the selected test input file should be mutated?



Note that a long test input may not cover a target branch.

```
FILE * f1 = open("tmp.txt", "rb")
char buf[4000];
fread(buf, 1, 4000, f1);
if (buf[2030] == 'h') {   // 2031 th byte value should be 'h'
…
```

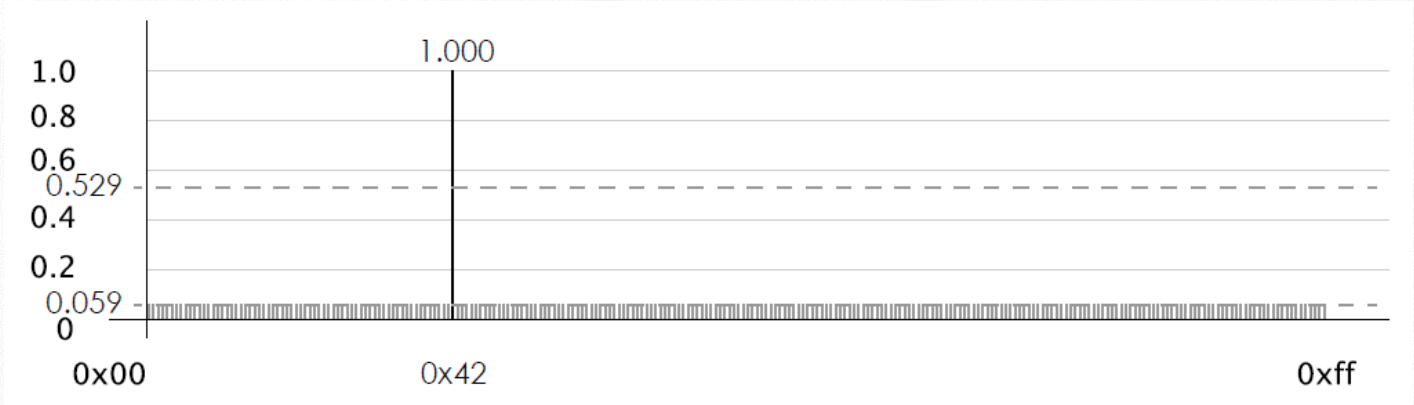
Thus, fuzzers utilize characteristics of input bytes to generate small test inputs

입력 바이트 타입 분류

Profuzzer (SP 2019)의 경우 입력 바이트를
5가지 타입으로 분류하여 퍼징을 진행.

1. Assertion type
   - 특정한 값이 들어와야만 특정 분기를 달성할 수 있는 경우
   - 해당 바이트에 0~255 값을 넣었을 때 특정 하나의 값(0x42)에서만 다른 실행을 보이는 것을 감지가능

```
1  int main (){
2    //input = 4 bytes input;
3    char a = input[1];
4    if (a != 0x34) exit_error();
5    ...
```



입력 바이트 타입 분류

2. Enumeration type
   - switch의 case같은 경우 (특정 값들 (0x08, 0x10, …) 이 유효하게 작용)
   - Assertion과 비슷하게 탐지가 가능

```
1  int main (){
2    char a = input[1];
3    switch (a) {
4    case 0x08 : foo(); break;
5    case 0x10 : foo2(); break;
6    case 0x18 : foo3(); break;
7    case 0x20 : foo4(); break;
8    default : exit_error(); }
9    ...
```

입력 바이트 타입 분류

3. Loop count
   - Loop가 몇번 실행될 것인지 정하는 입력 바이트
   - 바이트의 값을 바꿔보았을 때, 비슷한 실행을 보이지만, 얼마나 loop를 많이 실행하는지가 다른 경우가 발생하는지 탐지

```
1  int main (){
2    //input = 4 bytes input;
3    char a = input[1];
4    for (int i = 0; i < a ; i ++){
5     ...
```

4. Raw Data
   - 해당 바이트가 입력의 실행에 거의 영향을 안주는 경우.
   - 이경우에는 변이하는 것이 의미가 없는 경우가 많음.

입력 바이트 타입 분류

5. Offset

- 이후 입력을 얼마나 사용할 것인가를 결정하는 바이트

```
1  int main (){
2    //input = 40 bytes input file;
3    char offset = input[0];
4    char buffer[400];
5    if (fread(buffer, 1, offset , input + 1) == NULL) { …
```

- offset의 값에 따라서, buffe에 얼마나 많은 바이트를 복사할 건지를 결정.
- 특정 값 이상이 들어오는 경우, 보통 프로그램이 오류가 발생하고 종료되기 때문에, 이를 쉽게 감지 가능.

Dynamic taint analysis (동적 분석)

동적 데이터 의존성 (data dependency)분석으로 얻은
바이트에 관한 정보도 사용. (Angora, SP 2018)

```
1   int main (){
2     char input[9];
3     int a = input[4];
4     int b = input[5];
5     if (input[1] == 30) {
6       if (a + b > 20) {
7         return 1;
8       }
9     }
10    return 0;
11  }
```

초기 입력값: 9-byte 길이의 0

각 바이트를 특정 색깔 (id)로 표지를 하고 실행

Dynamic taint analysis (동적 분석)

동적 데이터 의존성 (data dependency)분석으로 얻은 바이트에 관한 정보도 사용. (Angora, SP 2018)

```
1   int main (){
2     char input[9];
3     int a = input[4];
4     int b = input[5];
5     if (input[1] == 30) {
6       if (a + b > 20) {
7         …
8       }
9     }
10    …
11  }
```

1. 변수 a가 하늘색 5번째 바이트 값을 사용하므로, 하늘색으로 표지
2. 변수 b가 남색 6번째 바이트 값을 사용하므로, 남색으로 표지
3. 5번째 줄의 if문이 주황색 2번째 바이트 값을 사용하므로, 주황색으로 표지
4. 6번째 줄의 if문이 하늘색 a와 남색 b의 값을 사용하므로, 두가지 색으로 표지

… 이를 계속해서 반복

특정 분기가 어떤 바이트를 참조하는지 알아내어, 특정 분기를 빠르게 달성할 수 있음. (6번째 줄의 if문을 달성하기 위해, 5번째, 6번째 바이트를 집중적으로 변이)

관련 연구

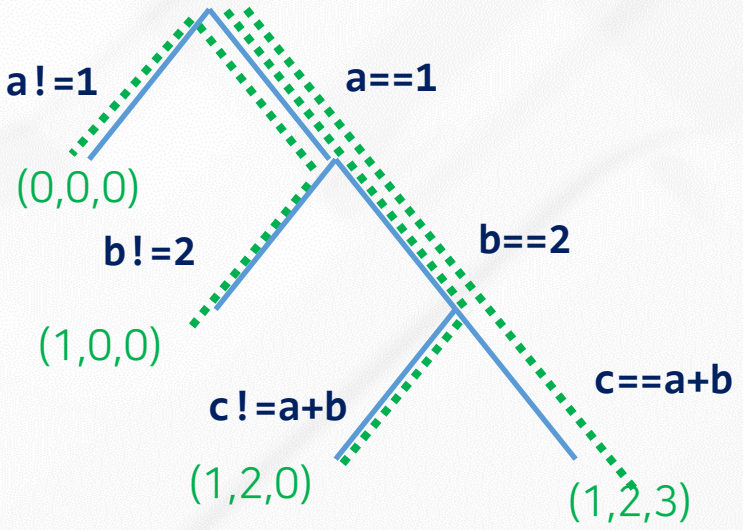
- ✓ Concolic 테스팅 최신 연구
  - ✓ 광대한 심볼릭 경로 공간(symbolic path space) 안에서 중복되는 탐색을 최소화하며
    - ✓ "조립식 동적 심볼릭 테스팅"
  - ✓ 모든 탐색 공간(search space)을 살펴보기보다 특정한 코드 영역을 집중하여 탐색하고
    - ✓ "목표지향적 탐색 알고리즘",
  - ✓ 또한 동적 심볼릭 테스팅 기술의 한계를 탐색 기반 테스팅 기술과 결합하여 극복하는 기술
    - ✓ "Concolic 테스팅과 탐색 기반 테스팅 기술을 혼합 적용"
  - ✓ 분산 시스템을 활용한 동적 심볼릭 테스팅의 속도를 향상하기 위한 기술
    - ✓ "분산 Concolic 테스팅"

- ✓ Fuzzing 최신 연구
  - ✓ GREYONE: Data Flow Sensitive Fuzzing (USENIX Security2020)
  - ✓ Fuzzing Symbolic Expressions (ICSE 2021)
  - ✓ IntelliGen: Automatic Driver Synthesis for Fuzz Testing (ICSE 2021 SEIP)
  - ✓ The Use of Likely Invariants as Feedback for Fuzzers (USENIX Security2021)
  - ✓ Constraint-guided Directed Greybox Fuzzing (USENIX Security2021)
  - ✓ POWER: Program Option-Aware Fuzzer for High Bug Detection Ability (ICST'2022)

Concolic 테스팅의 약점 예

- 첫번째 TC 생성이 매우 중요
  - 첫번째 TC에 따라, 탐색하는 search space가 크게 변동됨
  - 심지어, 첫번째 TC가 crash 인 경우, 해당 유닛 후속 테스팅 진행이 불가능

```
// Symbolic input: a, b, c
void f(int a, int b, int c) {
  if (a == 1)
    if (b == 2)
      if (c == a + b)
        Error();
}
```



Fuzzing의 약점 예

- 복잡한 조건식을 만족하는 TC 생성이 거의 불가능
  - 랜덤하게 변이한 TC 입력값이 특정 조건을 만족하기 어려움

Applied AFL++ on unit drivers generated with CROWN2
cflow, include_symbol (30 lines)

2 hours run

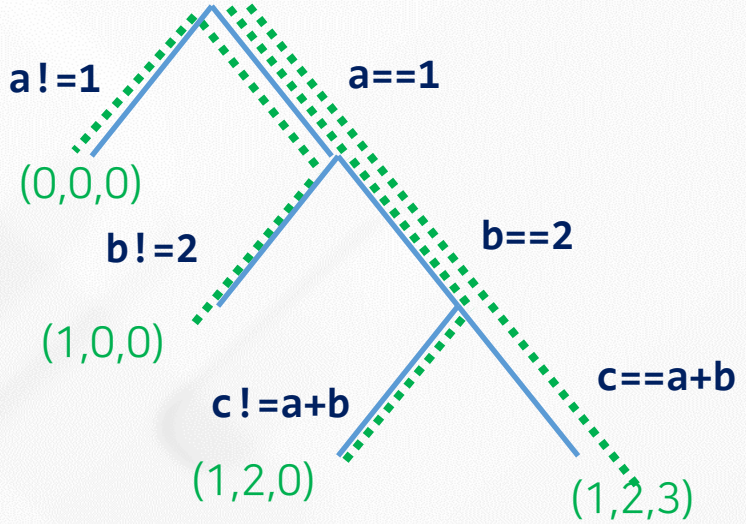
```
560 : include_symbol(Symbol *sym)
    : {
560 :       int type = 0;
    :
560 :       if (!sym)
205 :           return 0;
    :
355 :       if (sym->visible != output_visible)
 25 :           return 0;
    :
330 :       if (sym->type == SymIdentifier) {
  0 :           if (sym->name[0] == '_' && !(symbol_map & SM_UNDERSCORE))
  0 :               return 0;
    :
  0 :           if (sym->storage == StaticStorage)
  0 :               type |= SM_STATIC;
  0 :           if (sym->arity == -1 && sym->storage != AutoStorage)
  0 :               type |= SM_DATA;
  0 :           else if (sym->arity >= 0)
  0 :               type |= SM_FUNCTIONS;
    :
  0 :           if (!sym->source)
  0 :               type |= SM_UNDEFINED;
    :
330 :       } else if (sym->type == SymToken) {
  5 :           if (sym->token_type == TYPE && sym->source)
  0 :               type |= SM_TYPEDEF;
    :           else
  5 :               return 0;
    :       }
325 :       return (symbol_map & type) == type;
```

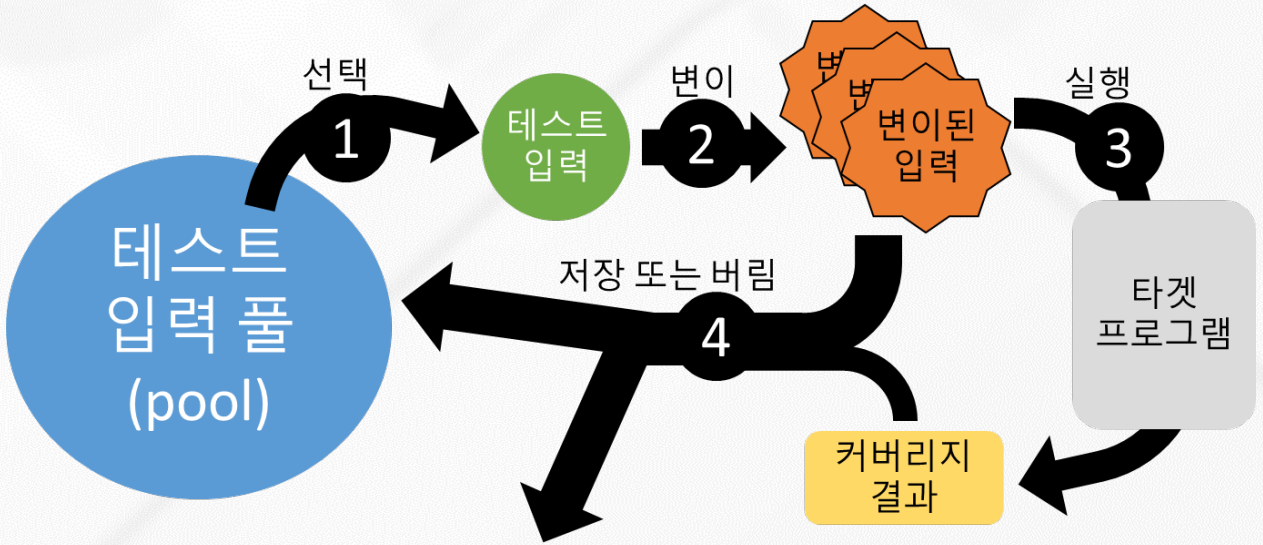
Concolic vs. Fuzzing

- 퍼징의 장점인 **빠르고 다양한 테스트 입력값 생성**과 Concolic 테스팅의 장점인 **정확한 SW 동적 분석 능력**을 결합하여, 보다 효과적인 테스팅 효과 달성

- Deep path를 탐색하는 능력이 제한적인 퍼징의 약점 및 Concolic Testing의 핵심이지만 속도가 느린 symbolic execution 엔진 상호 보완

```
// Symbolic input: a, b, c
void f(int a, int b, int c) {
  if (a == 1)
    if (b == 2)
      if (c == a + b)
        Error();
}
```





연구 내용

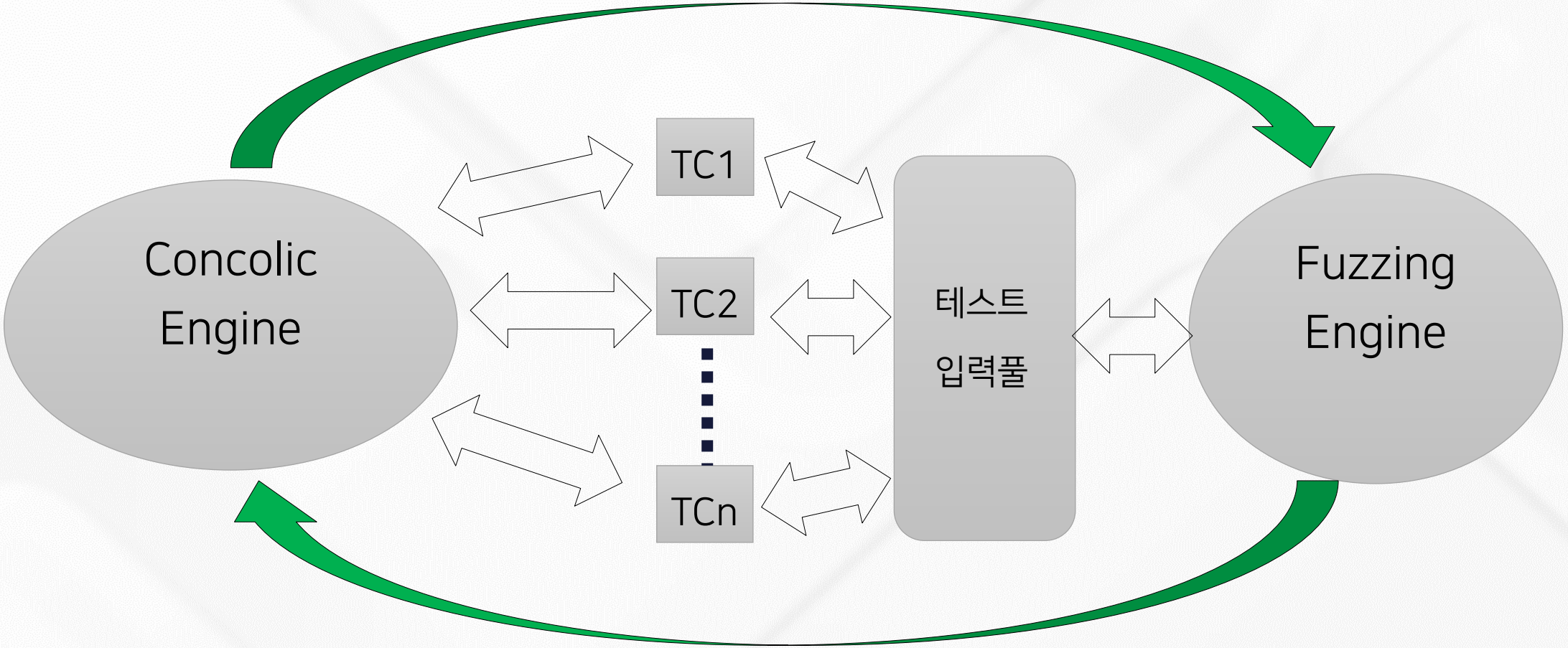
- 퍼징과 Concolic testing을 상호보완적으로 결합한 CONUZZ Framework 연구

1) 퍼징이 필요로 하는 test input pool에, concolic testing이 생성한 test input들을 test input들의 특성에 기반하여 선택적으로 추가

2) 제한된 테스팅 시간 안에 Concolic 테스팅이 커버하기 힘든 복잡한 자료구조를 활용하는 로직을 Fuzzing을 통해 검사

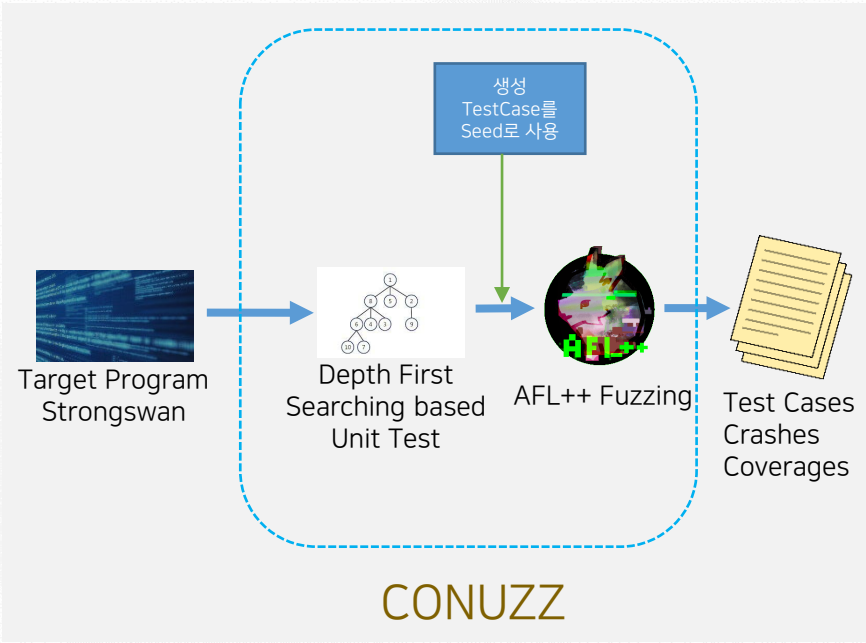
CONUZZ (CONcolic + fUZZing): KCSE '22 발표

테스트 입력 파일 기반 융합 프레임워크

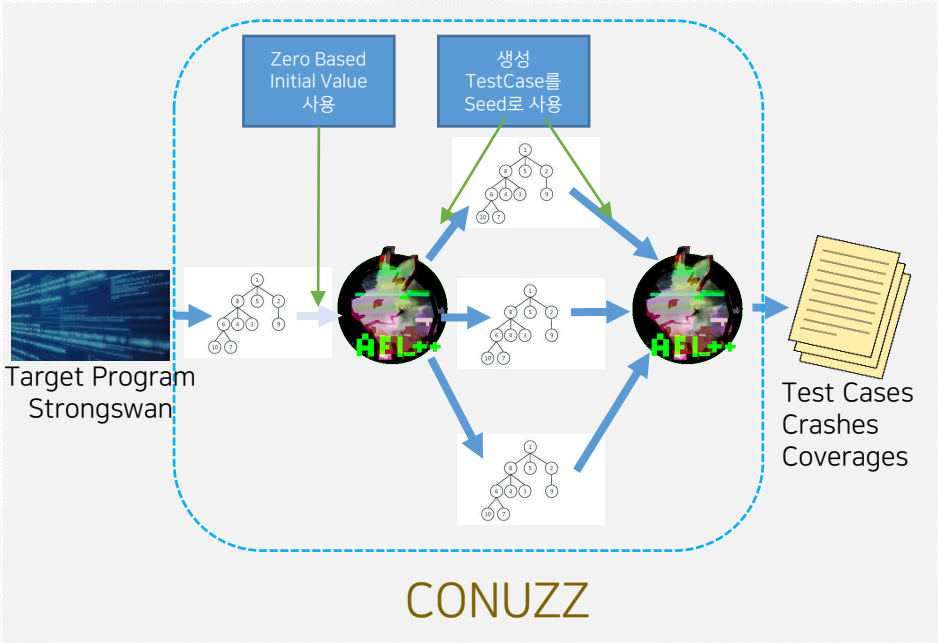


Concolic Testing과 Fuzzing의 교차 이용

DFS 사용시 Init TestCase에서 Crash가 발생하지 않는 경우



DFS 사용시 Init TestCase에서 Crash가 발생하는 경우



결론

- 퍼징의 장점인 **빠르고 다양한 테스트 입력값 생성**과 Concolic 테스팅의 장점인 **정확한 SW 동적 분석 능력**을 결합하여, 보다 효과적인 테스팅 효과 달성

- Deep path를 탐색하는 능력이 제한적인 퍼징의 약점 및 Concolic Testing의 핵심이지만 속도가 느린 symbolic execution 엔진 상호 보완

```
// Symbolic input: a, b, c
void f(int a, int b, int c) {
  if (a == 1)
    if (b == 2)
      if (c == a + b)
        Error();
}
```