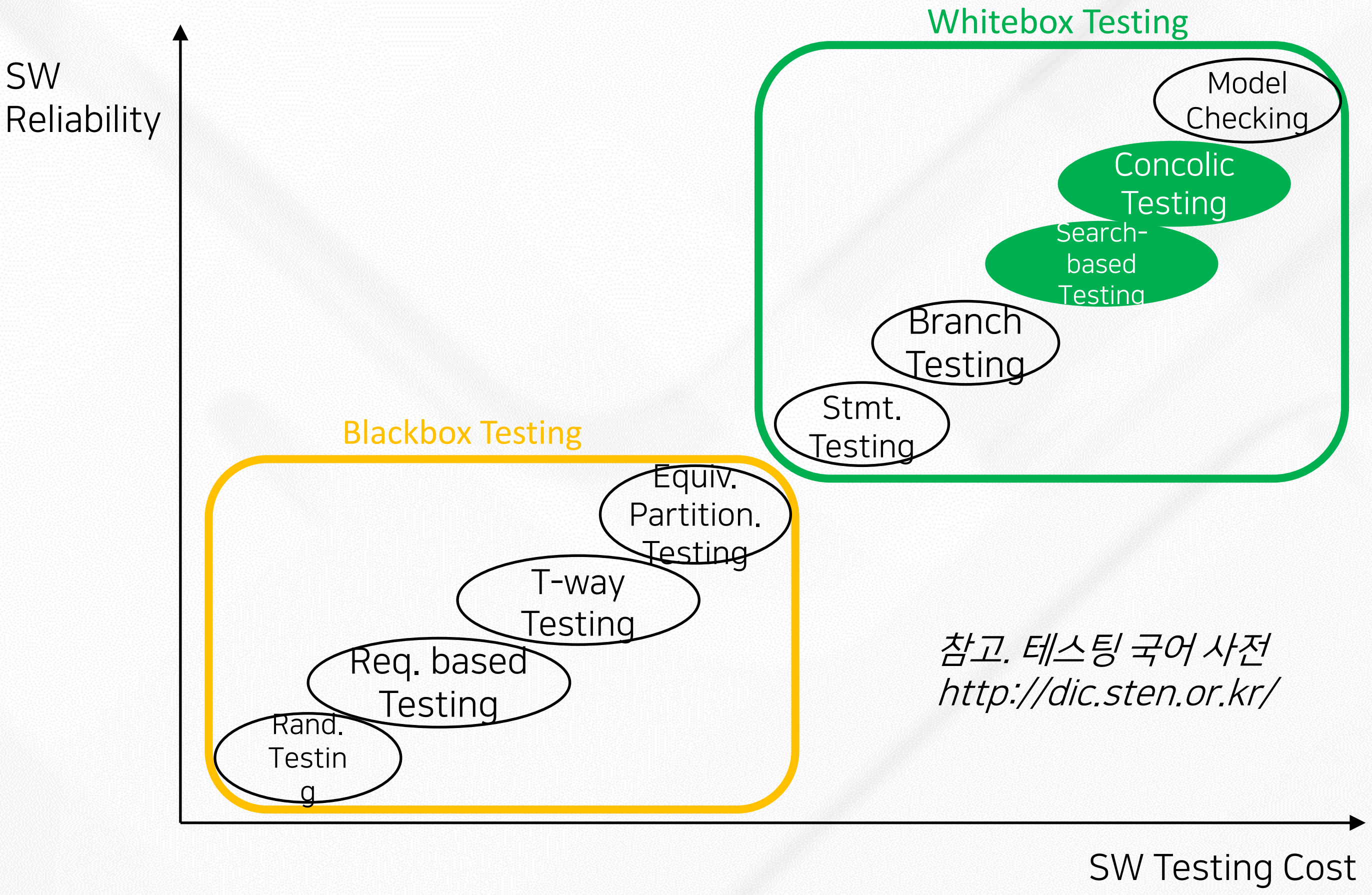
Automated Test Input Generation through Fuzzing

Moonzoo Kim

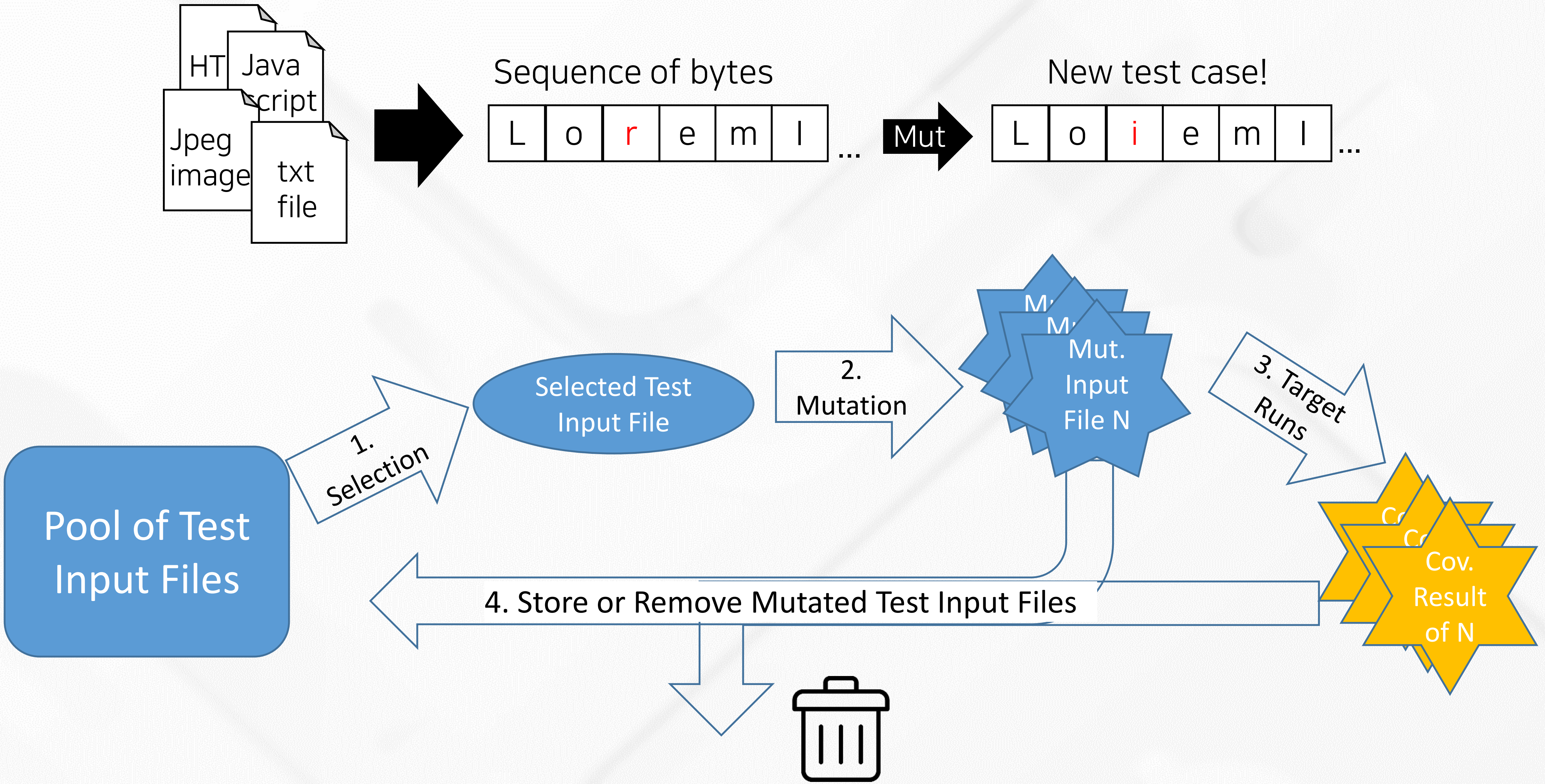
KAIST

Various SW Testing Techniques w/ Different Cost and Effectiveness



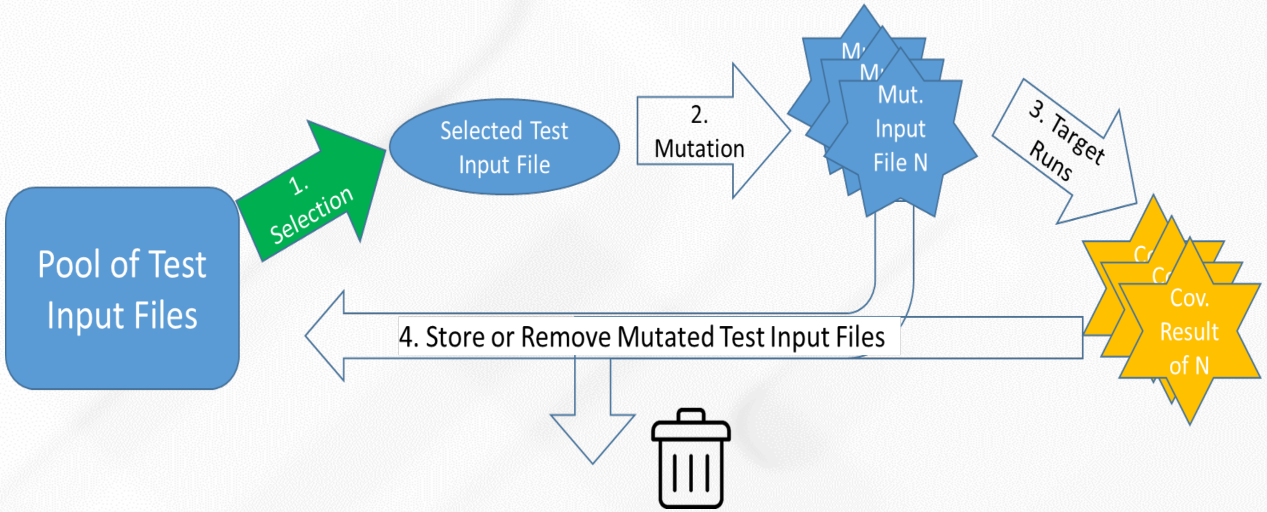
참고. 테스팅 국어 사전
http://dic.sten.or.kr/

Fuzzing - Automated Test Input Generation via Random Mutation



Fuzzing Challenge #1

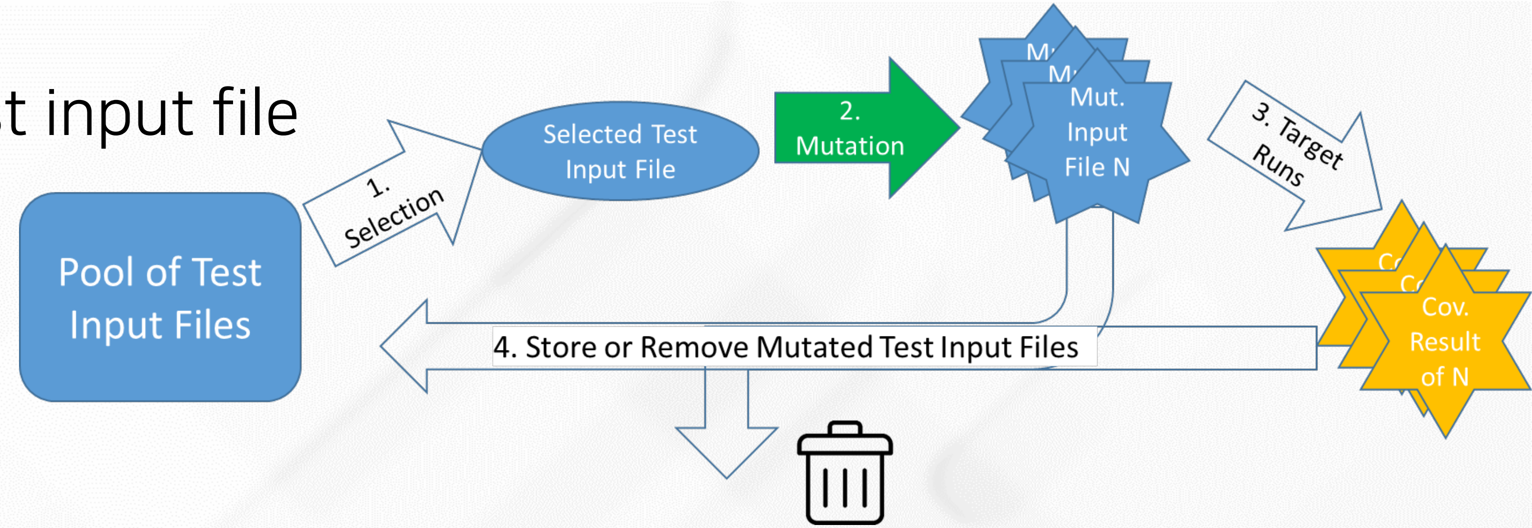
Which test input file to select to mutate?



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

AFLplusplus

Search

Site map

- Features
- Build & Install
- Documentation
- Tutorials
- Papers

Downloads

- Release 4.00c
- All releases
- Current devel
- License

Links

- Repo (GitHub)
- Donations
- Mailing list

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.

```
            american fuzzy lop ++2.65d (libpng_harness) [explore] {0}
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 43 sec      │  cycles done : 15    │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec       │  total paths : 703   │
│ last uniq crash : none seen yet                     │ uniq crashes : 0     │
│  last uniq hang : none seen yet                     │   uniq hangs : 0     │
├─ cycle progress ─────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 261*1 (37.1%)  │    map density : 5.78% / 13.98%        │
│ paths timed out : 0 (0.00%)      │ count coverage : 3.30 bits/tuple       │
├─ stage progress ─────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 14          │ favored paths : 114 (16.22%)          │
│ stage execs : 31/32 (96.88%)     │  new edges on : 167 (23.76%)          │
│ total execs : 2.55M              │ total crashes : 0 (0 unique)          │
│  exec speed : 61.2k/sec          │  total tmouts : 0 (0 unique)          │
├─ fuzzing strategy yields ────────┴───────────────┬─ path geometry ───────┤
│   bit flips : n/a, n/a, n/a                      │    levels : 11        │
│  byte flips : n/a, n/a, n/a                      │   pending : 121       │
│ arithmetics : n/a, n/a, n/a                      │  pend fav : 0         │
│  known ints : n/a, n/a, n/a                      │ own finds : 699       │
│  dictionary : n/a, n/a, n/a                      │  imported : n/a       │
│havoc/splice : 506/1.05M, 193/1.44M               │ stability : 99.88%    │
│   py/custom : 0/0, 0/0                           ├───────────────────────┘
│        trim : 19.25%/53.2k, n/a                  │          [cpu000: 12%]
└──────────────────────────────────────────────────┘
```

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

Search OSS-Fuzz

OSS-Fuzz on GitHub

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.

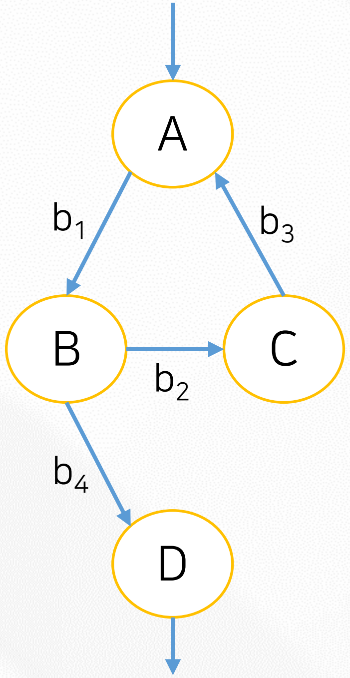
Trophies

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

Fuzzing Algorithm of AFL/AFL++

1. How <u>execution paths</u> of test input files are defined and analyzed
2. How to <u>select</u> test input files
3. How to <u>mutate</u> 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

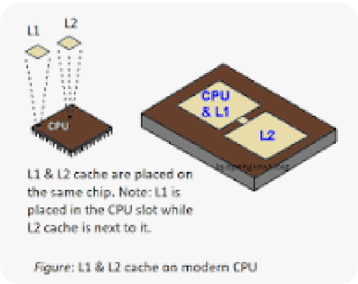
- tc2 covers a new path compared to tc1
  - the hit count for b_1, b_2, and b_3 of tc1 and tc2 cover different buckets.

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

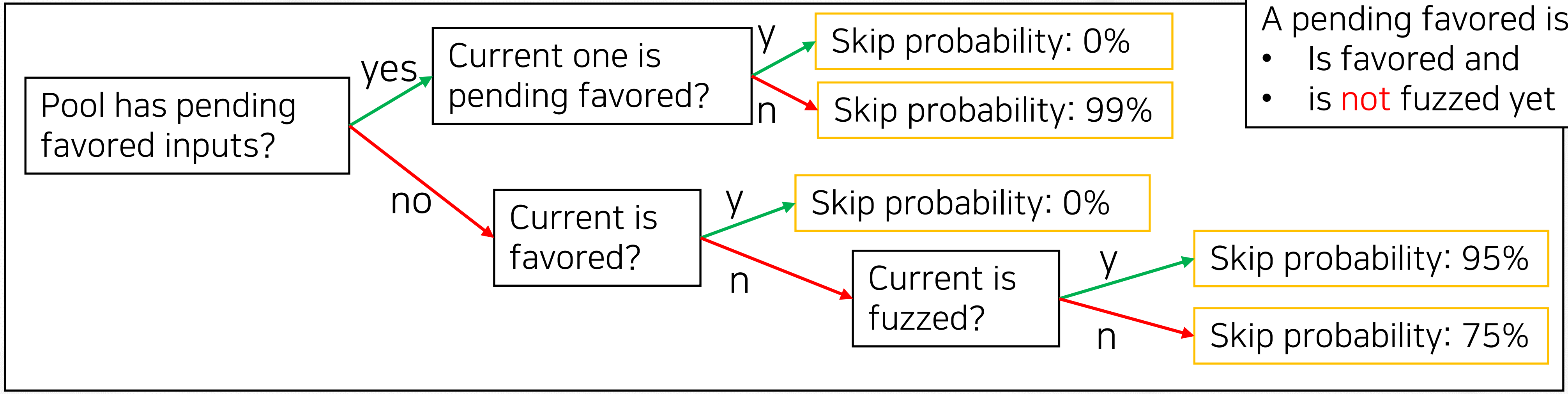
Input 410 • Exec_time: 10 • Length: 10	Input 324 • Exec_time: 5 • Length: 12	…	Empty

64K favored inputs corresponding to the 64K branches

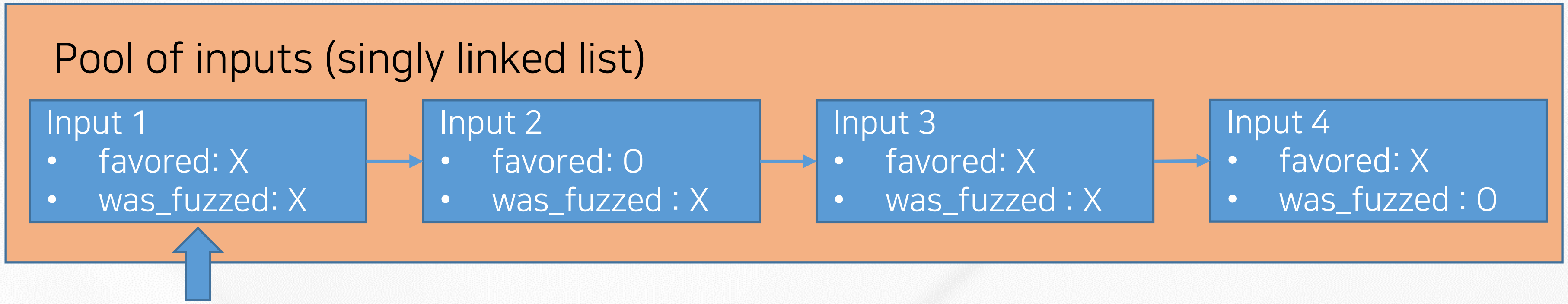
- AFL selects favored inputs at:
  - the initialization phase after calculating performance score based on the initial seed inputs, and
  - the beginning of each fuzzing cycle

Selecting Input from Pool (1/4)

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



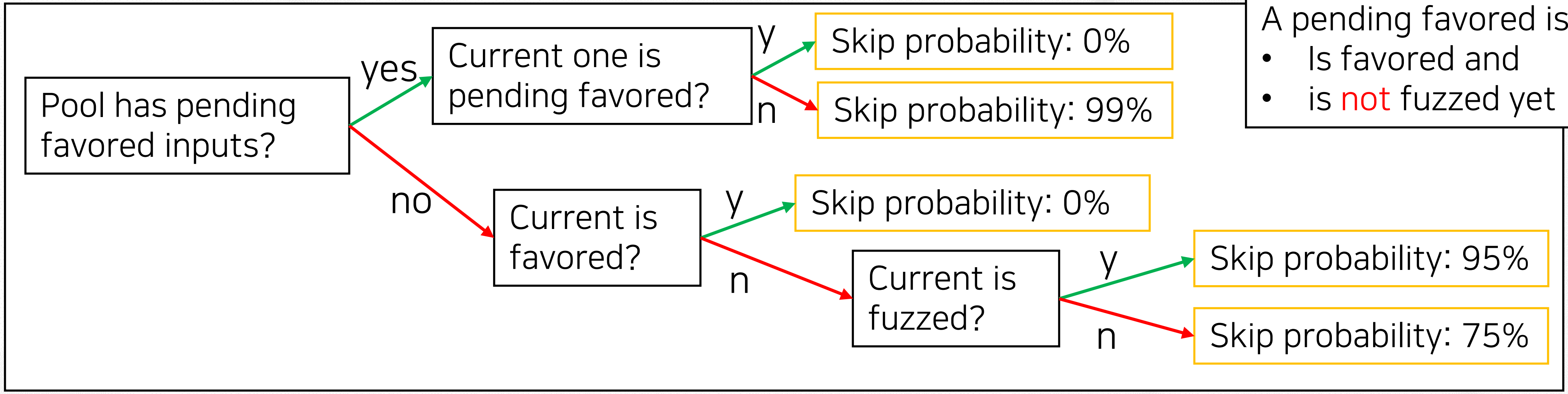
A pending favored is an input that:
- Is favored and
- is not fuzzed yet



Skip with 99% chance

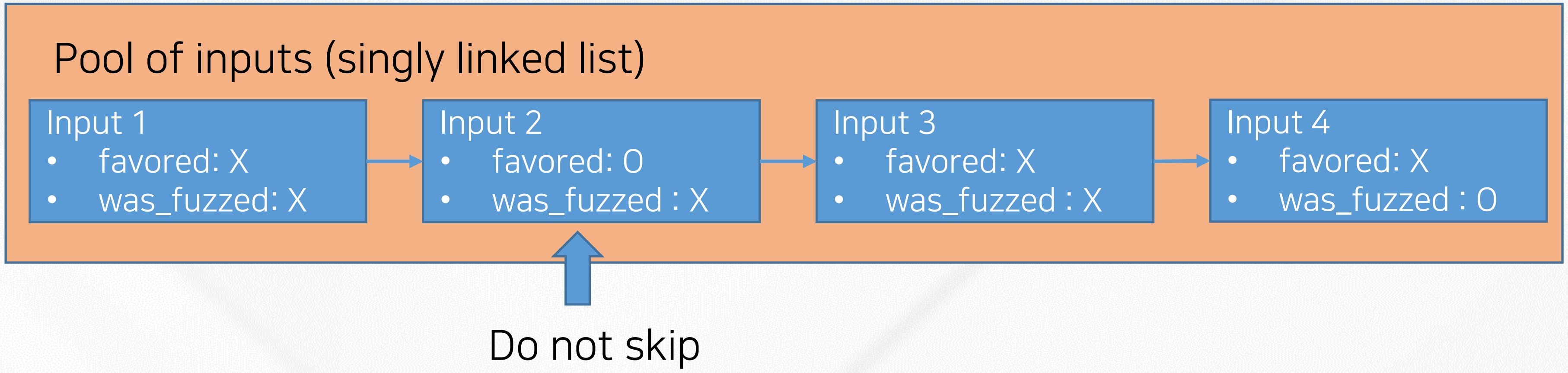
Selecting Input from Pool (2/4)

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



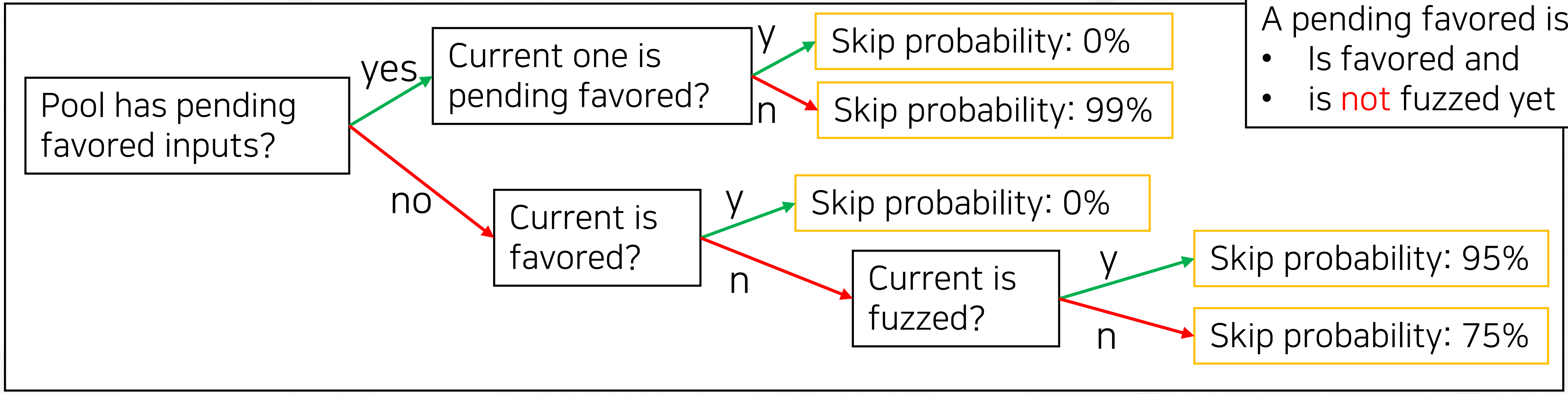
A pending favored is an input that:

- Is favored and
- is not fuzzed yet

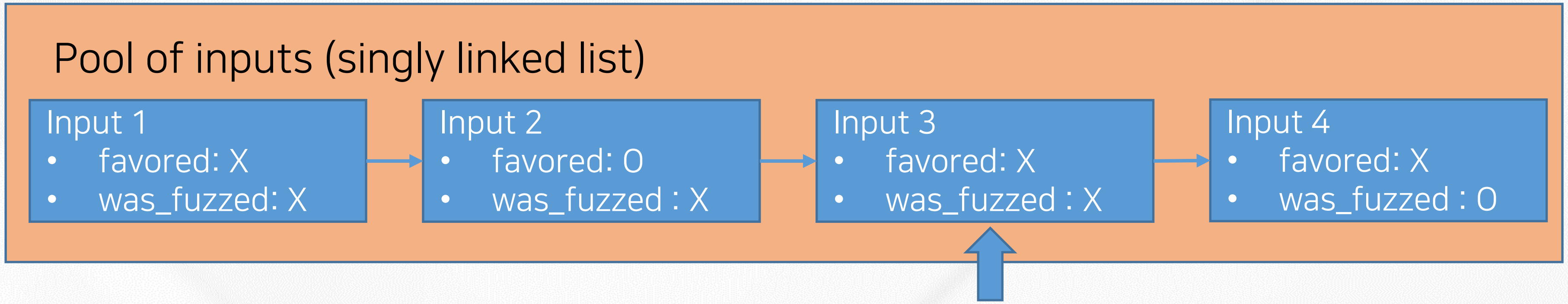


Selecting Input from Pool (3/4)

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



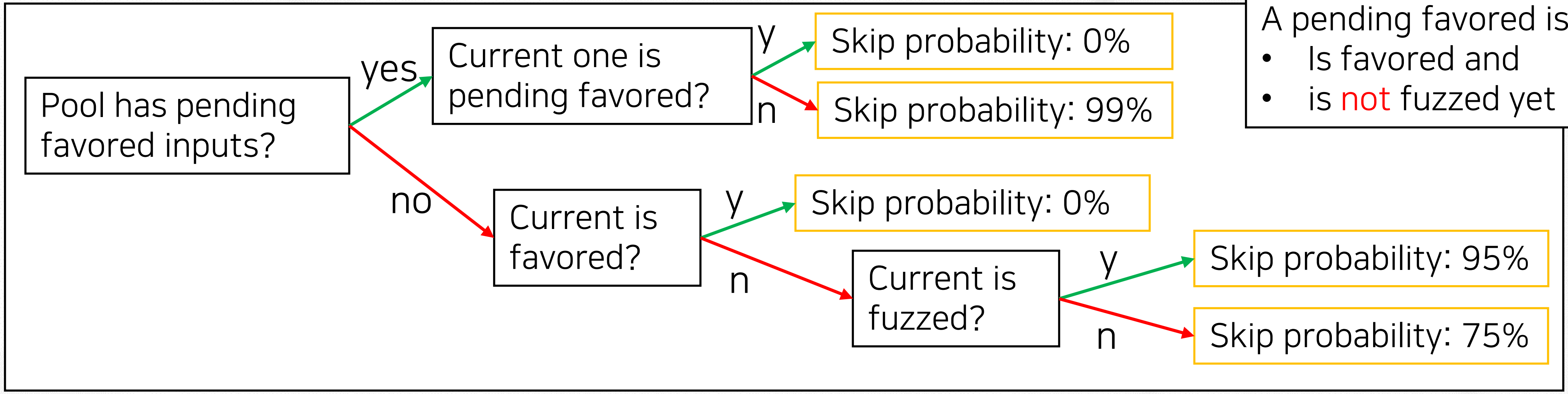
A pending favored is an input that:
- Is favored and
- is not fuzzed yet



Skip with 75% chance

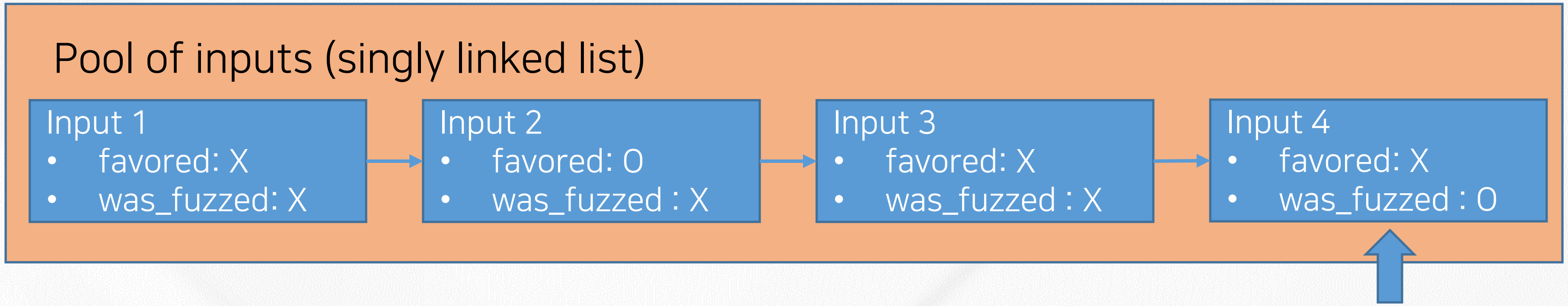
Selecting Input from Pool (3/4)

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



A pending favored is an input that:

- Is favored and
- is not fuzzed yet



Skip with 95% chance

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:

1	0	1	0	0	0	0	1

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

0	1	1	0	0	0	0	1

1	1	0	0	0	0	0	1

1	0	0	1	0	0	0	1

1	0	1	1	1	0	0	1

1	0	1	0	1	1	0	1

1	0	1	0	0	1	1	1

1	0	1	0	0	0	1	0

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:

byte 1								byte 2							
1	0	1	0	0	0	0	1	0	1	1	1	0	0	1	1

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

byte 1								byte 2							
1	0	1	0	0	0	1	0	0	1	1	1	0	0	1	1

*add 1 to byte 1

byte 1								byte 2							
1	0	1	0	0	0	1	1	0	1	1	1	0	0	1	1

*add 2 to byte 1

... (up to 35)

byte 1								byte 2							
1	0	1	0	0	0	0	0	0	1	1	1	0	0	1	1

*subtract 1 from byte 1

byte 1								byte 2							
1	0	0	1	1	1	1	1	0	1	1	1	0	0	1	1

*subtract 2 from byte 1

... (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:

byte 1								byte 2							
1	0	1	0	0	0	0	1	0	1	1	1	0	0	1	1

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

byte 1								byte 2							
1	0	0	0	0	0	0	0	0	1	1	1	0	0	1	1

*overwrite -128 to byte 1

byte 1								byte 2							
1	1	1	1	1	1	1	1	0	1	1	1	0	0	1	1

…

*overwrite -1 to byte 1

byte 1								byte 2							
1	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0

*overwrite -128 to byte 2

byte 1								byte 2							
1	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1

…

*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:

a	b	c	d	e	f	g	h

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

i	n	t	d	e	f	g	h
a	i	n	t	e	f	g	h
a	b	i	n	t	f	g	h
a	b	c	i	n	t	g	h
a	b	c	d	i	n	t	h
a	b	c	d	e	i	n	t

Fuzzing Inputs – havoc

V+Lab

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.