Fuzzing

ICT Risk Assessment project

University of Pisa

Federico Minniti, Matteo Del Seppia

What is fuzzing? How does it work? What it is used for?

- Fuzzing is a software testing technique used for negative testing. It consists in providing to a target software, an executable or a library, pseudo-random inputs in order to discover potential bugs and vulnerabilities such as buffer overflows. The inputs can be generated starting from a corpus of files ("seeds'') or from scratch.

- Depending on the knowledge the fuzzer possesses about the target software, fuzzing is usually categorized in black-box, white-box, or coverage-guided (grey-box).

- The Fuzz tool, published by Barton Miller's research group in 1990, is considered the first notable example of fuzzing as it is recognized today. Fuzz aimed to identify potential security vulnerabilities in software by providing random command line inputs to programs.

Black-box fuzzing

- Black-box fuzzing generates inputs for a target program without knowledge about its internal structure. It generally relies on a static corpus that helps generating meaningful inputs for the target. Black-box fuzzing is still used when the target is large and slow or non-deterministic for the same input, or the input format is complicated and highly structured (such as a programming language).

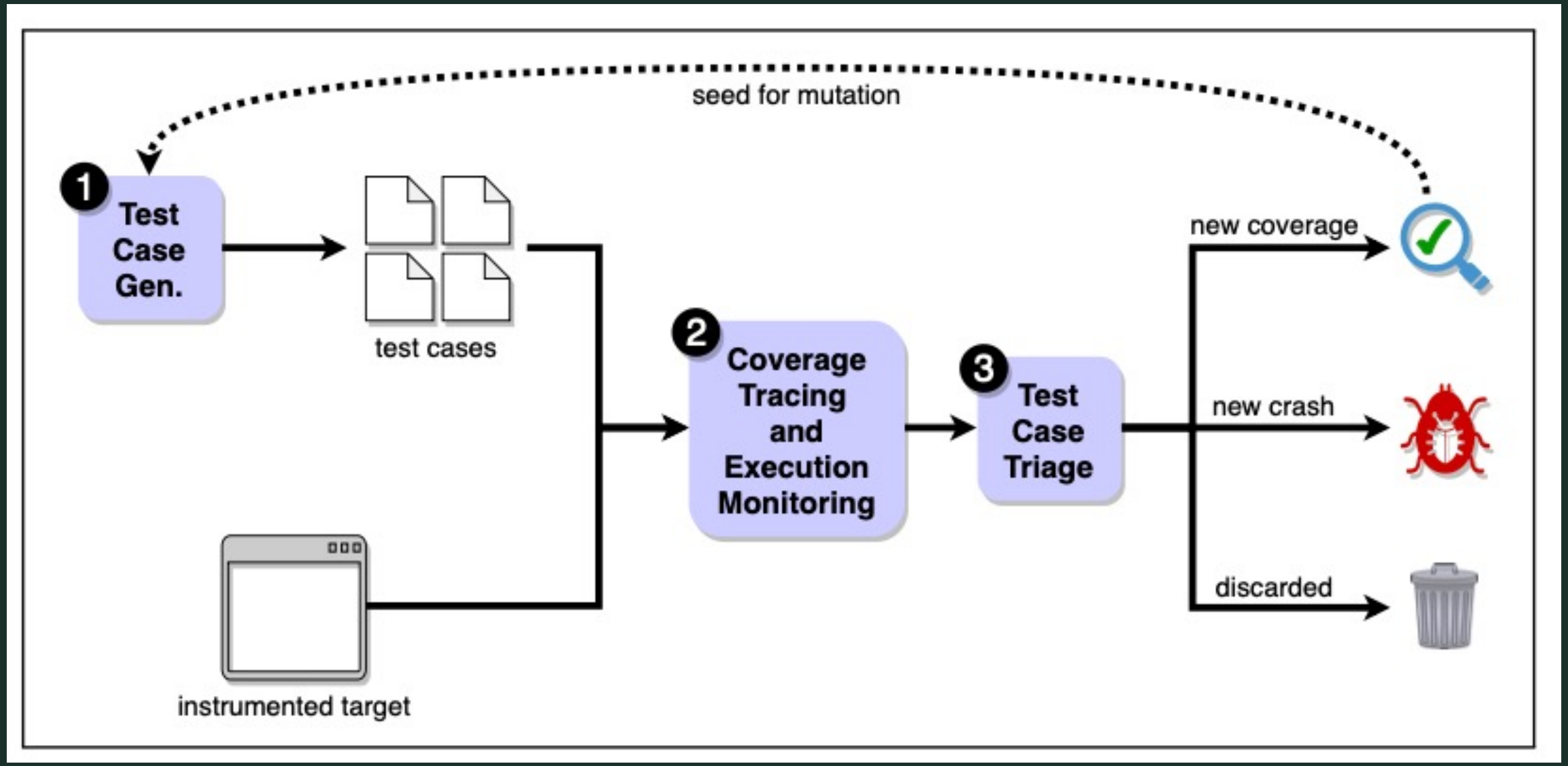
- The problem with black-box fuzzing is that we don't know how much code was covered during the fuzzing procedure. Also, certain branches may be very difficult to reach with randomly generated inputs without exploiting some knowledge of the source code.

White-box Fuzzing

- White-box fuzzing was extensively utilized by Microsoft in 2000s, which then published their discoveries under the name of SAGE (Scalable, Automated, Guided Execution) in 2012. Unfortunately, the tool itself was not published by Microsoft.
- SAGE begins with a good input for the program being tested and symbolically runs the program, keeping track of conditions encountered during the execution. These conditions help gather information about what inputs cause the program to take different paths. Next, the gathered conditions are negated and solved using a special tool called a constraint solver. The solutions provided by the solver are then used to create new inputs that make the program take different paths during execution, achieving a good (in theory 100%) code coverage.
- Even though SAGE has helped Microsoft in uncovering many bugs in its commercial software, this technique has its limits. First, we have to consider that a program can have a large number of branches (path-explosion problem), and testing all of them may be impossible. Then, the constraint solver may not be able to solve the constraints in a reasonable amount of time. Also, symbolic execution may produce false positives due to interactions with system calls, a problem that black-box fuzzing completely avoids (because you are actually running the program so there are no false positives). Finally, SAGE heavily relies on the quality of its initial input.

Coverage-guided fuzzing

Coverage-guided fuzzing takes the good sides of both white-box fuzzing and black-box fuzzing. It avoids symbolic executions used in SAGE to get rid of false positives, but runs a modified version of the target software that contains additional code called ''instrumentation'', which helps the fuzzer getting knowledge about code coverage. This of course has a cost in terms of time, since the program will be slower due to the additional code that is used to keep track of the explored paths, but it is a good trade-off between performance and obtaining coverage information, which is usually solved running the fuzzer with multiple threads or processes or in a distributed environment.


seed for mutation
1
Test Case Gen.
test cases
2
Coverage Tracing and Execution Monitoring
3
Test Case Triage
new coverage
new crash
discarded
instrumented target

Sanitizers

- Coverage-guided fuzzers, such as libFuzzer or HonggFuzz, typically incorporate code sanitization along with their compile-time instrumentation. These tools, introduced by Google ASan in 2012 and integrated into popular compilers like Clang or GCC, utilize shadow memory to identify memory corruption issues such as buffer overflows or user-after-free errors.

- Common sanitizers: ASan, MSan, TSan, UBSan

AFL

American fuzzy lop is a security-oriented fuzzer that employs compile-tim instrumentation and a evolutionary algorithm to automatically discover feasible, interesting test cases triggering new internal states of the targeted binary. Its nature based on coverage-feedback substantially improves the functional coverage for the fuzzed code with respect to black-box fuzzers. The compact synthesized corpora produced by the tool are also useful for seeding successive, more labor- or resource-intensive testing regimes.

Compile time instrumentation (__afl_setup)

```
__afl_setup
    if __afl_setup_failure != 0:
        __afl_return()
    (__afl_global_area_ptr == 0) ?
        __afl_setup_first() : __afl_store()


__afl_setup_first
    One time setup inside the target process
    - Get shared memory id from environment variable __AFL_SHM_ID
    - Map the shared memory and store the location
        in __afl_area_ptr & __afl_global_area_ptr.
    - __afl_forkserver()
```

Compile time instrumentation (__afl_forkserver)

```
__afl_forkserver
    Write 4 bytes to fd 199 (start message: Phone home and tell
    the parent that we're OK.)
    __afl_fork_wait_loop()

__afl_fork_wait_loop
    Wait for 4 bytes on fd 198 (listen for response; Wait for
    parent by reading from the pipe. This will block until the
    parent sends us something. Abort if read fails.) and then
    clone the current process
In child process
    __afl_fork_resume()
In parent
    Store child pid to __afl_fork_pid
    Write it to fd 199 and call waitpid which will write child
        exit status to __afl_temp
    Write child exit status contained in __afl_temp to fd 199.
    __afl_fork_wait_loop()

__afl_fork_resume
    (In child process)
    Closes the fds 198 & 199 (fuzzer <-> forkserver communication)
    resumes with execution
```

Genetic algorithm

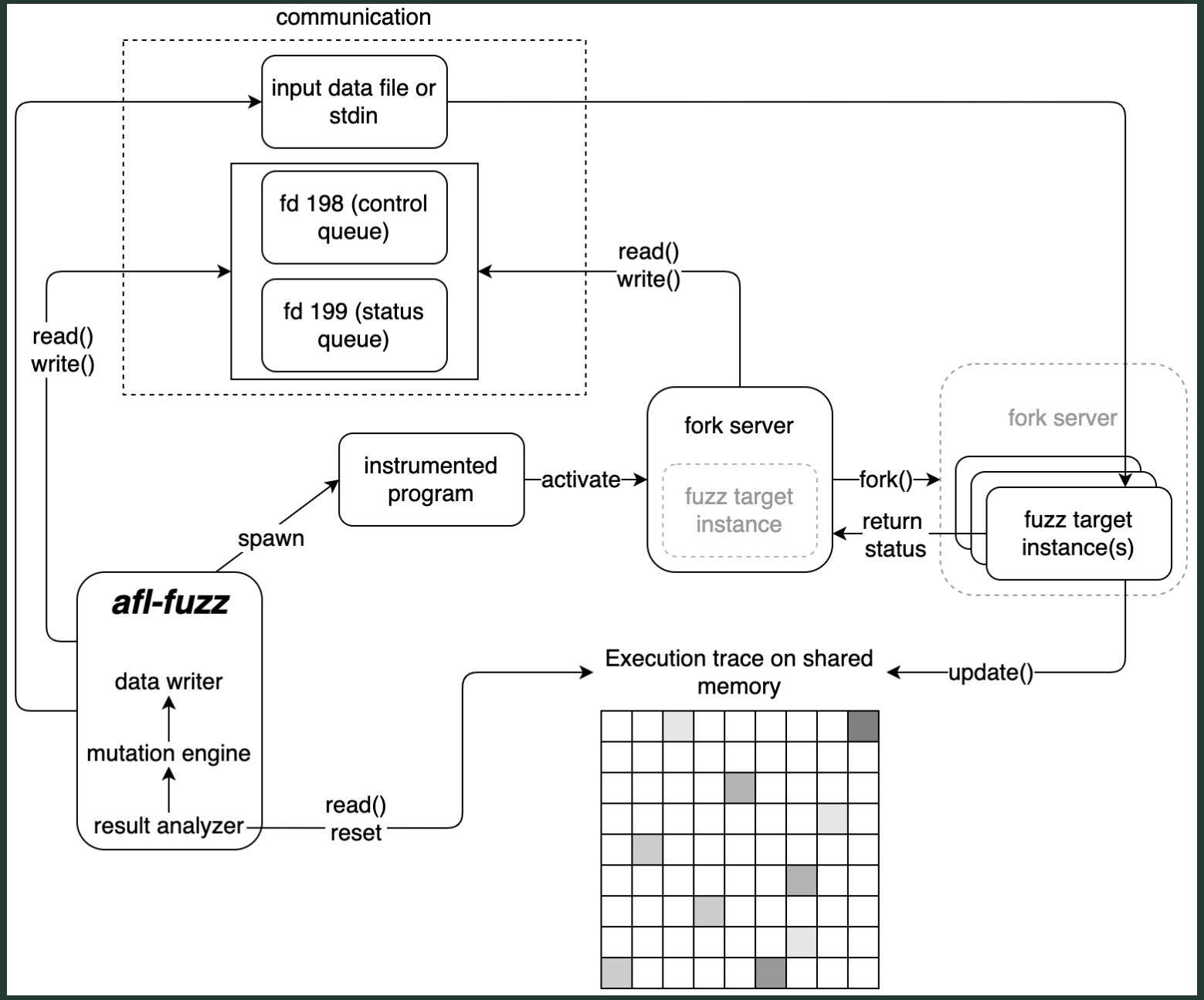
Algorithm 1 Coverage-based Greybox Fuzzing

Input: Seed Inputs S

```
1:  T_x = ∅
2:  T = S
3:  if T = ∅ then
4:      add empty file to T
5:  end if
6:  repeat
7:      t = chooseNext(T)
8:      p = assignEnergy(t)
9:      for i from 1 to p do
10:         t' = mutate_input(t)
11:         if t' crashes then
12:             add t' to T_x
13:         else if isInteresting(t') then
14:             add t' to T
15:         end if
16:     end for
17: until timeout reached or abort-signal
```

Output: Crashing Inputs T_x

AFL - Communication details



AFL - Selecting interesting inputs

- The goal of AFL is to improve the test cases (input data provided to the target) so they can explore different paths during the program's execution, increasing the chances of discovering potential bugs or unexpected behavior. To do so, AFL takes each input seed and randomly mutates it to create new variants of the input data. During the program execution, AFL uses coverage instrumentation to track which parts of the program (branches, functions, etc.) are being executed by each test case. The goal is to find inputs that trigger previously unexplored code paths; if a new path is discovered it means the test case has revealed some previously untested behavior, and AFL will add it to the queue.
- During the fuzzing process, AFL saves inputs that explore new ''buckets'' (containing hitcounts). Buckets are 8 bit counters in a bitmap, where each byte (in theory) represents a certain edge. An input is considered ''interesting'' (saved to the queue) if it explores at least one new bucket (one new edge).
- Notice that the shared bitmap has a limited size, which means it can only track a certain number of edges. This limitation introduces the possibility of collisions, where multiple edges end up sharing the same byte in the bitmap. Since collisions are possible, AFL's coverage tracking is only approximated in reality.

AFL – Power scheduling

- The objective of a power schedule is to optimize the time spent on fuzzing by prioritizing the most promising seeds that are likely to result in higher coverage increase in a shorter amount of time. In other words, it aims to maximize the efficiency and effectiveness of the fuzzing process
- The power schedule is responsible for determining the energy levele(s) of each test cases. In particular AFL uses the exploitation-based constant scheduling, assigning the energy as a function α:

$$e(s) = \alpha(t_{ex}, btc_{ex}, ct_s)$$

- t_{ex} is the execution time
- btc_{ex} is the block transition coverage of the execution
- ct_s is the test case creation time

It is also important to notice that, the energy value does not depend on:

- the number of times that the seed has previously been chosen from the queue T
- the number of test cases that exercise the same execution path exercised by s

AFL - Mutation strategies

- Walking bit flips
- Walking byte flips
- Simple arithmetics
- Known integers
- Stacked tweaks
- Test case splicing

AFL – Stacked tweaks

Single-bit flips

Attempts to set "interesting" bytes, words, or dwords

Addition or subtraction of small integers to bytes, words, or dwords

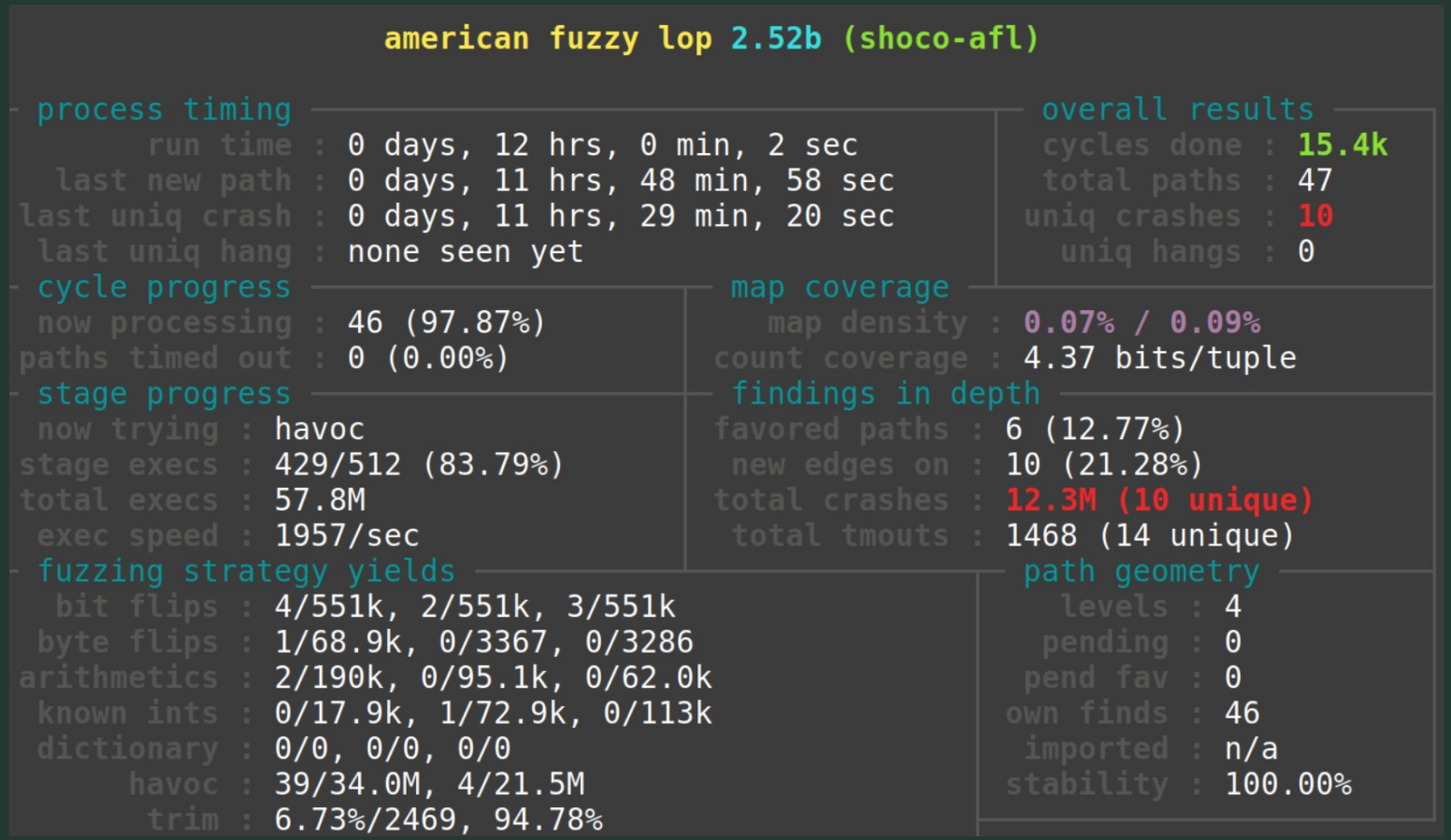
Completely random single-byte sets

Block deletion

Block duplication via overwrite or insertion

Block memset

AFL interface



LLVM LibFuzzer

LLVM LibFuzzer is a fuzzing engine that is directly integrated with the library being tested. The fuzzing process starts from a set of input samples, randomly selecting one and applying random mutations to it. The modified input is then passed into the target program through a fuzz target, referred to as a ''harness'' LLVMFuzzerTestOneInput:

```
int LLVMFuzzerTestOneInput (const uint8_t* Data, size_t Size) {
    DoSomethingInterestingWithMyAPI (Data, Size );
    return 0;
}
```

Unlike AFL, which focuses on fuzzing standalone programs that typically receive input from standard input, libFuzzer operates within the same process and executes the harness multiple times with different inputs. This makes it easier to fuzz libraries and their APIs, as long as the fuzz tester understands the APIs being fuzzed. To track code coverage, LibFuzzer relies on a LLVM module pass called SanitizerCoverage (SanCov) to statically instrument the program. Hence, LibFuzzer can only be used when the source code of the target is available.

LLVM SanCov (-fsanitize-coverage=trace-pc)

```
int main() {
    //this is a basic block
    test(1);
    return 0;
}
```

```
main:                                   # @main
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    call    __sanitizer_cov_trace_pc
    mov     dword ptr [rbp - 4], 0
    mov     edi, 1
    call    test(int)
    xor     ecx, ecx
    mov     dword ptr [rbp - 8], eax # 4-byte Spill
    mov     eax, ecx
    add     rsp, 16
    pop     rbp
    ret
```

```
int test(int a) {
    //this is a basic block
    return a + 1;
}
```

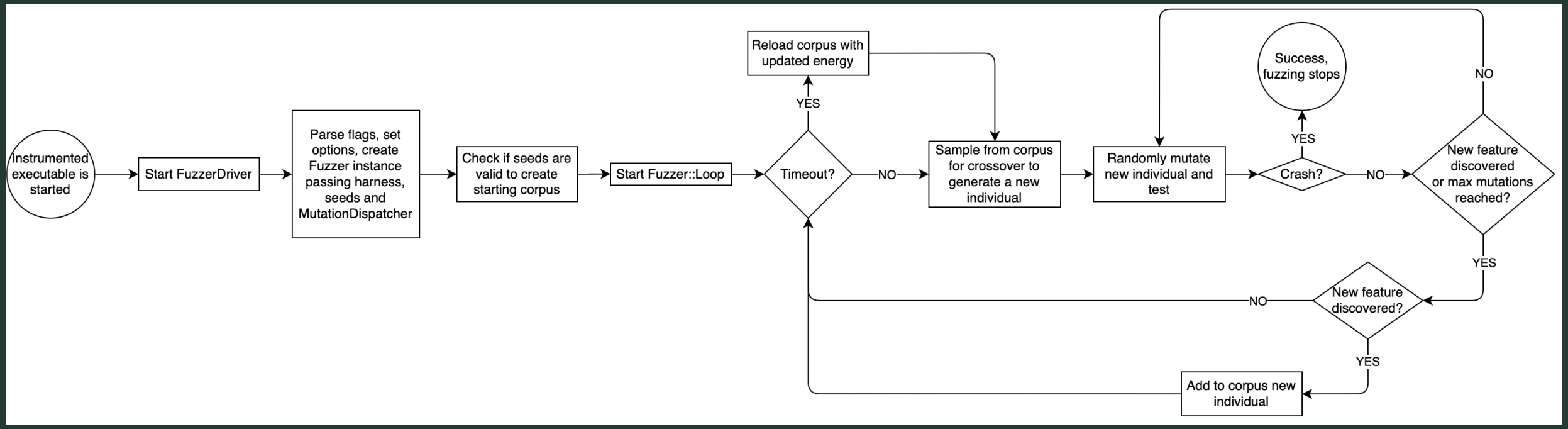
```
test(int):                              # @test(int)
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     dword ptr [rbp - 8], edi # 4-byte Spill
    call    __sanitizer_cov_trace_pc
    mov     eax, dword ptr [rbp - 8] # 4-byte Reload
    mov     dword ptr [rbp - 4], eax
    mov     ecx, dword ptr [rbp - 4]
    add     ecx, 1
    mov     eax, ecx
    add     rsp, 16
    pop     rbp
    ret
```

LLVM SanCov (-fsanitize-coverage=trace-pc-guard)

```
main:                                   # @main
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        movabs  rdi, offset .L__sancov_gen_.1
        call    __sanitizer_cov_trace_pc_guard
        mov     dword ptr [rbp - 4], 0
        mov     edi, 1
        call    test(int)
        xor     ecx, ecx
        mov     dword ptr [rbp - 8], eax # 4-byte Spill
        mov     eax, ecx
        add     rsp, 16
        pop     rbp
        ret
```

```
sancov.module_ctor_trace_pc_guard:      # constructor for sancov module
        push    rax
        movabs  rax, offset __start___sancov_guards. //pointer to start guard number
        movabs  rcx, offset __stop___sancov_guards.  //pointer to end guard number
        mov     rdi, rax
        mov     rsi, rcx
        call    __sanitizer_cov_trace_pc_guard_init
        pop     rax
        ret
.L__sancov_gen_:
        .zero   4
.L__sancov_gen_.1:
        .zero   4
```

How LibFuzzer works



LibFuzzer – Coverage-feedback details

The fuzzing process of LibFuzzer is evolutionary because each individual of the population is sampled for crossover and mutation according to a weighted probability distribution based on ''energy''. An individual will have a higher energy if it is fast to be tested and it has generated rare features during its test. This way, LibFuzzer privileges interesting inputs that will run quick and are more likely to reach dark spots of the target program. The list of features tracked during the fuzzing process changes as initially rare features become more and more common. This way, the exploration at any point in time really focuses about rare features at that moment, discarding stale information about features that with the current corpus can be reached easily.

LibFuzzer mutations

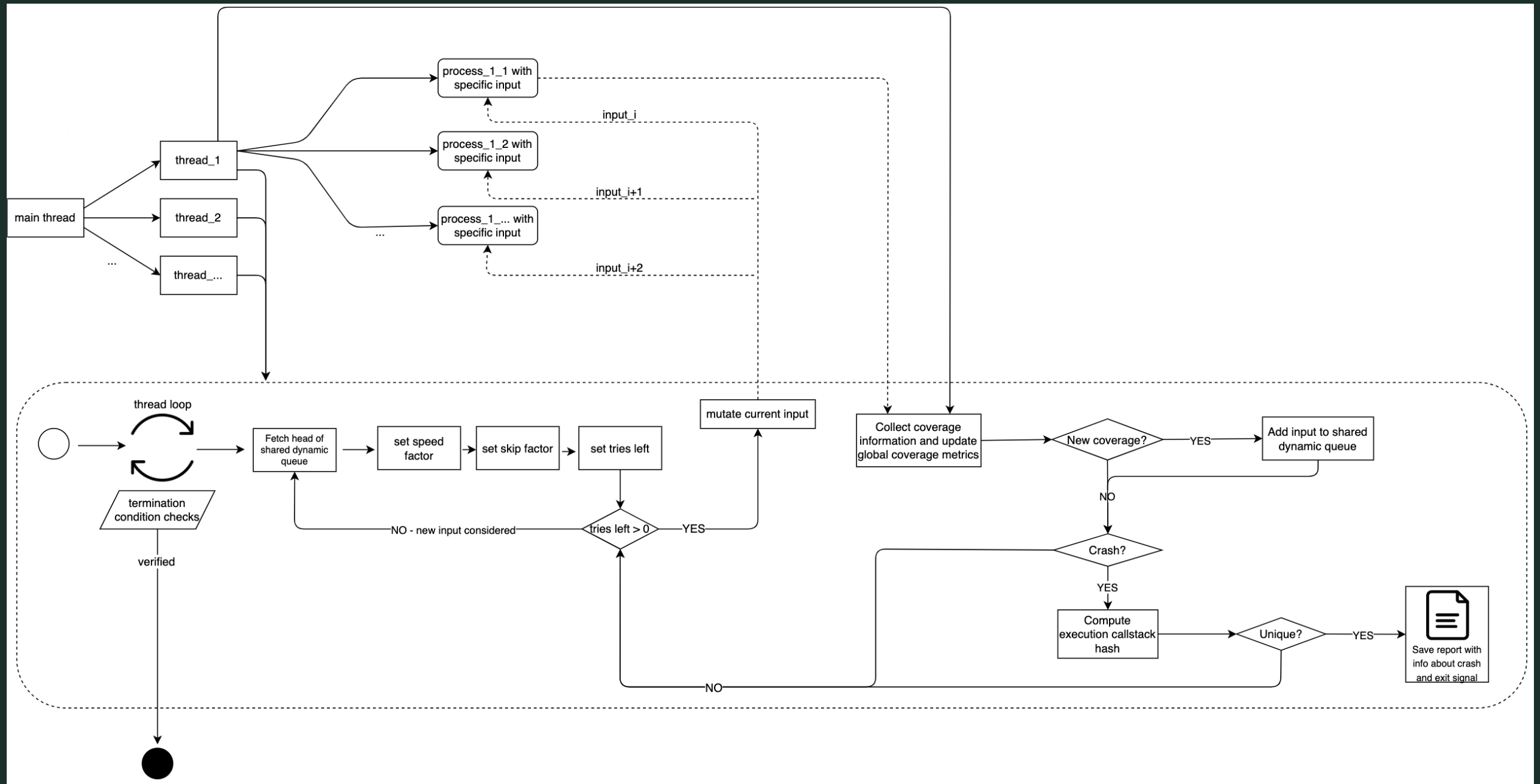
- Insert Byte
- Erase Byte
- Insert Repeated Bytes
- Change Byte
- Single Bit Flip
- Shuffle Bytes
- Change ASCII Integer
- Change Binary Integer
- Copy Part
- Crossover
- Add word to persistent AutoDict
- Add word to temporary AutoDict

LibFuzzer interface

```
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 83668586
INFO: Loaded 1 modules   (49 inline 8-bit counters): 49 [0x55a5f733c030, 0x55a5f733c061),
INFO: Loaded 1 PC tables (49 PCs): 49 [0x55a5f733c068,0x55a5f733c378),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 6 ft: 6 corp: 1/1b exec/s: 0 rss: 30Mb
#3      NEW    cov: 7 ft: 9 corp: 2/3b lim: 4 exec/s: 0 rss: 31Mb L: 2/2 MS: 1 InsertByte-
#11     NEW    cov: 10 ft: 12 corp: 3/5b lim: 4 exec/s: 0 rss: 31Mb L: 2/2 MS: 3 CrossOver-EraseBytes-InsertByte-
#12     NEW    cov: 10 ft: 15 corp: 4/8b lim: 4 exec/s: 0 rss: 31Mb L: 3/3 MS: 1 CrossOver-
#13     NEW    cov: 11 ft: 16 corp: 5/11b lim: 4 exec/s: 0 rss: 31Mb L: 3/3 MS: 1 ChangeByte-
#17     NEW    cov: 13 ft: 20 corp: 6/15b lim: 4 exec/s: 0 rss: 31Mb L: 4/4 MS: 4 ShuffleBytes-InsertByte-EraseBytes-CopyPart-
#23     NEW    cov: 13 ft: 21 corp: 7/19b lim: 4 exec/s: 0 rss: 31Mb L: 4/4 MS: 1 CrossOver-
#39     REDUCE cov: 13 ft: 21 corp: 7/18b lim: 4 exec/s: 0 rss: 31Mb L: 2/4 MS: 1 EraseBytes-
#42     NEW    cov: 13 ft: 23 corp: 8/22b lim: 4 exec/s: 0 rss: 31Mb L: 4/4 MS: 3 ChangeBit-ShuffleBytes-CrossOver-
#43     NEW    cov: 13 ft: 24 corp: 9/26b lim: 4 exec/s: 0 rss: 31Mb L: 4/4 MS: 1 ChangeByte-
#59     REDUCE cov: 13 ft: 24 corp: 9/25b lim: 4 exec/s: 0 rss: 31Mb L: 1/4 MS: 1 EraseBytes-
#70     NEW    cov: 14 ft: 25 corp: 10/27b lim: 4 exec/s: 0 rss: 31Mb L: 2/4 MS: 1 ChangeBinInt-
#74     REDUCE cov: 15 ft: 26 corp: 11/31b lim: 4 exec/s: 0 rss: 31Mb L: 4/4 MS: 4 ChangeBit-ChangeBit-InsertByte-CrossOver-
```

Honggfuzz

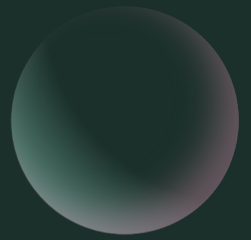
Honggfuzz’s engine is highly customizable and supports several hardware-based (CPU: branch/instruction counting, Intel BTS, Intel PT)and software-based feedback-driven fuzzing modes, more than any other fuzzer.It has a single supervising process that automatically spawns processes andthreads to take advantage of the full potential of the CPU cores. The interaction between processes and threads happens via signal monitoring and sharedmemory as in AFL, but its parallelism is way more optimized with respect toAFL. In Honggfuzz there are multiple concurrent threads mutating and execut-ing on their own and updating concurrently a shared dynamic corpus (whichis the only source of concurrency overhead), while in AFL it’s only the mainprocess that takes care of the evolution of the fuzzed inputs, waiting for its childprocesses to execute the inputs and return their status


main thread
thread_1
thread_2
...
thread_...
process_1_1 with specific input
process_1_2 with specific input
process_1_... with specific input
...
input_i
input_i+1
input_i+2
thread loop
termination condition checks
verified
Fetch head of shared dynamic queue
set speed factor
set skip factor
set tries left
tries left > 0
NO - new input considered
YES
mutate current input
Collect coverage information and update global coverage metrics
New coverage?
YES
Add input to shared dynamic queue
NO
Crash?
YES
Compute execution callstack hash
Unique?
YES
Save report with info about crash and exit signal
NO

Honggfuzz interface

```
------------------------[  0 days 00 hrs 30 mins 14 secs ]----------------------
  Iterations : 6,613 [6.61k]
  Mode [3/3] : Feedback Driven Mode
      Target : ./shoco-hongg decompress ___FILE___ /dev/null
     Threads : 1, CPUs: 1, CPU%: 100% [100%/CPU]
       Speed : 108/sec [avg: 3]
     Crashes : 1323 [unique: 113, blocklist: 0, verified: 0]
    Timeouts : 1,320 [1 sec]
 Corpus Size : 54, max: 8,192 bytes, init: 1 files
  Cov Update : 0 days 00 hrs 25 mins 57 secs ago
    Coverage : edge: 25/143 [17%] pc: 0 cmp: 820
---------------------------------- [ LOGS ] -----------------/ honggfuzz 2.5 /-
ready exists, skipping
[2023-05-12T23:04:05+0200][W][2756] subproc_checkTimeLimit():532 pid=19041 took
too much time (limit 1 s). Killing it with SIGKILL
Crash (dup): '/home/federico/Scrivania/shoco/SIGSEGV.PC.555555584bee.STACK.badba
d0c3cff434a.CODE.1.ADDR.7fffffff000.INSTR.movzbl_0x3(%rcx,%rbx,1),%eax.fuzz' al
ready exists, skipping
Signal 2 (Interrupt) received, terminating
Terminating thread no. #0, left: 0
Summary iterations:6613 time:1814 speed:3 crashes_count:1323 timeout_count:1320
new_units_added:53 slowest_unit_ms:4961 guard_nb:143 branch_coverage_percent:17
peak_rss_mb:291
```

Fuzzing Experiments



Fuzzer test suite (Google)

The Google Fuzzer Test Suite is a testing framework developed by Google tocompare the performance and behavior of different fuzzers over the same pro-grams. This suite is designed to automatically identify vulnerabilities and po-tential flaws in software by subjecting it to a rigorous and systematic fuzzingprocess. The suite supports natively AFL, Honggfuzz and LibFuzzer as fuzzingengines. The compilation and instrumentation of target programs and librariesis automatized by the suite through a set of command line tools and flags

Fuzzer test suite - results

Crashes (unique)			
Target	LibFuzzer	AFL	Honggfuzz
c-ares-CVE-2016-5180	1	4 (1)	4 (1)
openssl-1.0.1f	1	6 (1)	0
openssl-1.0.2d	1	18 (1)	1
re2-2014-12-09	0(1 DBG)	0	9 (1)
libxml2-v2.9.2	1	4 (1)	0
boringssl-2016-02-12	0	0	0

Coverage (edges)			
Target	LibFuzzer	AFL	Honggfuzz
c-ares-CVE-2016-5180	33	32	22
openssl-1.0.1f	595	467	2973
openssl-1.0.2d	750	992	906
re2-2014-12-09	2379	2560	2714
libxml2-v2.9.2	2077	2588	4397
boringssl-2016-02-12	880	628	818

Type of crash	
Target	Type
c-ares-CVE-2016-5180	heap-buffer-overflow in ares_create_query
openssl-1.0.1f	heap-buffer-overflow in tls1_process_heartbeat
openssl-1.0.2d	Assertion failed
re2-2014-12-09	DFA out of memory (debug)
libxml2-v2.9.2	heap-buffer-overflow in xmlDictComputeFastQKey
boringssl-2016-02-12	None

Shoco

Shoco is a C library to compress and decompress short strings, freely availableon GitHub. It is very fast and easy to use. The default compression model isoptimized for english words, but you can generate your own compression modelbased on your specific input data. It is mainly composed by two functionalities:

- shoco_compress: takes input and output buffers and their size, fills theoutput buffer with the compressed representation of the input buffer andreturns the size of the compressed data. If the length argument is 0, theinput buffer is assumed to be null-terminated. If it’s a positive integer,parsing the input will stop after this length, or at a null-char, whatevercomes first
- shoco_decompress: takes input and output buffers and their size, fills theoutput buffer with the decompressed representation of the input buffer andreturns the size of the decompressed data.

Shoco – LibFuzzer extension

```c
#include "shoco.h"
#include <stdint.h>

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  char buffer[8192] = {0};
  char* bytes = (char*)data;
  shoco_compress(bytes, size, buffer, 8192);
  return 0;
}
```

```c
#include "shoco.h"
#include <stdint.h>

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  char buffer[8192] = {0};
  char* bytes = (char*)data;
  shoco_decompress(bytes, size, buffer, 8192);
  return 0;
}
```

```
# Makefile's extension for libfuzzer
libfuzzer:
        clang -g -O1 -fsanitize=fuzzer,address target-compress.c
        shoco.c -o libfuzzer-compress

        clang -g -O1 -fsanitize=fuzzer,address target-decompress.c
        shoco.c -o libfuzzer-decompress
```

Shoco - results

Crashes (unique)			
Target	LibFuzzer	AFL	Honggfuzz
shoco (compress)	1	0	0
shoco (decompress)	1	10 (1)	113≫

Coverage (edges)			
Target	LibFuzzer	AFL	Honggfuzz
shoco (compress)	/	146	53
shoco (decompress)	15	47	25

Type of crash	
Target	Type
shoco (compress)	heap-buffer-overflow in shoco_compress
shoco (decompress)	global-buffer-overflow in shoco_decompress, stack-overflow (honggfuzz)

Smaz

Smaz is a simple compression library suitable for compressing very short strings.Its main feature point the ability to compress strings as short as a few bytes. Its APIs are:

- smaz_compress(char *in, int inlen, char *out, int outlen): Compress the buffer 'in' of length 'inlen' and put the compressed data into 'out' of max length 'outlen' bytes. If the output buffer is too short to hold the whole compressed string, outlen+1 is returned. Otherwise the length of the compressed string (less then or equal to outlen) is returned.
- smaz_decompress(char *in, int inlen, char *out, int outlen): Decompress the buffer 'in' of length 'inlen' and put the decompressed data into 'out' of max length 'outlen' bytes. If the output buffer is too short to hold the whole decompressed string, outlen+1 is returned. Otherwise the length of the compressed string (less then or equal to outlen) is returned.

Smaz - LibFuzzer extension

```c
#include <stdio.h>
#include <stdint.h>
#include "smaz.h"
int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    char* input = (char*)data;
    char buffer[8192];
    smaz_compress(input, size, buffer, 8192);
    return 0;
}
```

```c
#include <stdio.h>
#include <stdint.h>
#include "smaz.h"
int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    char* input = (char*)data;
    char buffer[8192];
    smaz_decompress(input, size, buffer, 8192);
    return 0;
}
```

```
libfuzzer:
    clang -g -O1 -ansi -Wall -pedantic -W -fsanitize=fuzzer,address
    target-compress.c smaz.c -o fuzzer-libfuzzer-compress

    clang -g -O1 -ansi -Wall -pedantic -W -fsanitize=fuzzer,address
    target-decompress.c smaz.c -o fuzzer-libfuzzer-decompress
```

Smaz – AFL/Honggfuzz extension

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#include "smaz.h"
int main(int argc, char** argv) {
    char* filename = argv[1];
    char *content;
    long length;
    FILE *fp;

    // open the file for reading
    fp = fopen(filename, "rb");
    if (fp == NULL) {
        return 1;
    }

    // find the length of the file
    fseek(fp, 0, SEEK_END);
    length = ftell(fp);
    fseek(fp, 0, SEEK_SET);

    // allocate memory to hold the content of the file
    content = (char *)malloc(length + 1);

    // read the content of the file into the string
    fread(content, 1, length, fp);

    // close file
    fclose(fp);

    char buffer[8192];

    smaz_compress(content, length, buffer, 8192);
    // free the allocated memory
    free(content);
    return 0;
}
```

Smaz – AFL/Honggfuzz extension (Makefile)

```
afl:
	afl-clang -o fuzzer-afl-compress -O2 -Wall -W
	-ansi -pedantic smaz.c main-compress.c

	afl-clang -o fuzzer-afl-decompress -O2 -Wall -W
	-ansi -pedantic smaz.c main-decompress.c

honggfuzz:
	hfuzz-clang -O2 -ansi -Wall -W -pedantic smaz.c
	main-compress.c -o fuzzer-honggfuzz-compress

	hfuzz-clang -O2 -ansi -Wall -W -pedantic smaz.c
	main-decompress.c -o fuzzer-honggfuzz-decompress
```

Smaz - results

Crashes (unique)			
Target	LibFuzzer	AFL	Honggfuzz
smaz (compress)	0	0	0
smaz (decompress)	1	0	0

Coverage (edges)			
Target	LibFuzzer	AFL	Honggfuzz
smaz (compress)	35	59	31
smaz (decompress)	6	27	11

Type of crash	
Target	Type
smaz (compress)	/
smaz (decompress)	heap-buffer-overflow in smaz_decompress