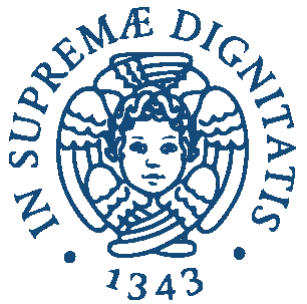


University of Pisa

ICT Risk Assessment Project:

Coverage-Guided Fuzzing with

AFL, LibFuzzer and Honggfuzz

Matteo Del Seppia
Federico Minniti

Academic year 2022-2023

Contents

Contents

1 A brief introduction to fuzzing

1.1 What is fuzzing

Fuzzing is an automated technique used for negative testing software. It consists in providing a target software pseudo-random inputs in order to discover potential bugs and vulnerabilities such as buffer overflows. The target may be an executable or a library API. The inputs can be generated starting from a corpus of files ("seeds") or entirely from scratch. Ideally, the goal of a fuzzing tool is to enter all the possible branches in the target's code, hence maximizing "code coverage". Also, depending on the knowledge the fuzzing tool ("fuzzer") possesses about the target software, fuzzing is usually categorized in black-box, white-box, or coverage-guided (grey-box).

Although fuzzing-like techniques were already known to software engineers as early as the 1980s ("The Monkey"), the emergence of public networks and the Internet played a significant role in increasing awareness about the importance of negative testing software for uncovering vulnerabilities and reliability issues. The appearance of worms employing buffer overflow attacks like the Morris Internet Worm in 1988 greatly contributed to the development of the software security field. 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 UNIX utility programs.

Several famous open source fuzzers, like Peach, were introduced in the early 2000s and continue to be utilized today. However, these fuzzers had a notable limitation in that they heavily relied on user-provided input files, which restricted their usability. The first fuzzers capable of dynamically increasing their effectiveness during runtime were EFS and SAGE. Significantly advancing the usability of fuzzing tools, American Fuzzy Lop (AFL) emerged as a major leap by utilizing compile-time instrumentation and evolutionary algorithms to discover test cases that increase code coverage. Shortly after the introduction of

AFL, LLVM libFuzzer was developed, with an emphasis on performance and fuzzing of smaller software components such as libraries. In 2018, Google Honggfuzz was released, prioritizing high performance and possessing the ability to detect and report hijacked signals from crashes, which can be intercepted and potentially hidden by a fuzzed program.

Fuzzing has proven to be an effective technique to tackle the difficulty, usually an impossibility, of generating every possible permutation of test data. Fuzzing forces programs to confront interesting corner cases that have the potential to expose critical problems and vulnerabilities. Also, fuzzing has been initially developed as a black-box technique, making it applicable to any application that accepts input, regardless of the programming language used. However, applications written in C/C++ are especially vulnerable to fuzzing due to their utilization of low-level memory management, which can easily conduct to potentially exploitable stack overflows or heap overflows.

1.2 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.

One example of black-box fuzzer is Peach, which generates test cases from user-defined templates, and is neither coverage-guided or feedback-driven.

1.3 White-box fuzzing

Black-box fuzzing has been widely used and has proven to be very effective, but its limit is that it does not reach a good code coverage. For example, the

probability of making the following program crash feeding it a random integer is only 1 over 2^{32}:

```
int foo(int x) {
  int y = x + 3;
  if (y == 13) abort(); // error
  return 0;
}
```

This intuitively explains the need for companies to design techniques that rely on their source code when looking for possible vulnerabilities in their software. 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.

This entire process is repeated using clever search techniques that try to explore as many different paths of the program as possible, even though in reality it may not be possible to explore all paths. At the same time, a runtime checker like Valgrind is used to verify multiple properties of the program.

Suppose we are testing the program above. We need to start from a good input, like $x = 1$. When we symbolically execute it with $x = 1$, the program goes through the else branch of the conditional statement, generating a new path constraint $x + 3 \neq 13$. Once this constraint is negated and solved, we get a solution $x = 10$, providing a new input that will go in the other branch of the conditional statement, this time crashing the program successfully only after one iteration!

Even though SAGE has helped Microsoft in uncovering many bugs in its commercial software, this technique has evident 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.

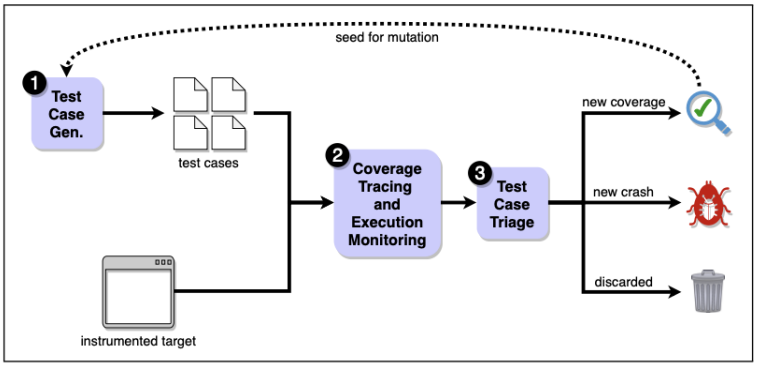
1.4 Coverage-guided fuzzing

1.4.1 What is 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.
The core components and processes of a coverage-guided fuzzer are:

- seed corpus: an initial set of input files is provided to the fuzzer. The corpus usually grows with time as new inputs that increase the code coverage are discovered;
- instrumentation: usually a custom compiler is used to compile the target, with the objective of injecting additional code to record information about executed code paths inside the program;
- mutation: the fuzzer generates inputs starting for the seed corpus files, mutating them with random bit flips, deletions, insertions and other transformations;

- execution tracking: the instrumented code tracks which parts of the program are traversed during the execution and records coverage information. Newly discovered paths are tracked and recorded;
- crash detection: if the program crashes or exhibits abnormal behavior, the fuzzer will log these information for further analysis;
- coverage feedback: the fuzzer keeps track of which paths have been covered and prioritizes inputs that can potentially explore new paths;
- mutation strategy: some fuzzers prioritize mutations that are likely to trigger new code paths, while other strategies involve combining multiple inputs to create new hybrids or generating inputs based on the feedback from previously executed test cases.
- bug triage: the detected crashes and abnormal behavior are triaged and analyzed.



1.4.2 A small digression on 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.
The shadow memory is a data structure that mimics the memory allocation of a program. It is typically represented as a bitmap indicating whether a specific

byte of memory is allocated to the program. Initially, all bits in the shadow memory are set to a specific value, indicating that the corresponding program memory has not been accessed or modified. Whenever the program performs a read or write operation, the corresponding bit in the shadow memory is checked. In case of:

- read: if the shadow memory indicates that the memory was not initialized, an error or warning is raised.
- write: when the program writes to memory, the corresponding bit in the shadow memory is set to a different value, indicating that the memory has been modified. When a buffer is allocated, for example, using a `malloc`, the starting address and size of the buffer are tracked. This information is utilized to mark the corresponding bits in the shadow memory as valid or allocated.

Other sanitizers have been developed, like:

- LSan: LeakSanitizer is a run-time memory leak detector. It can be combined with AddressSanitizer to get both memory error and leak detection;
- MSan: MemorySanitizer is a detector of uninitialized reads;
- TSan: ThreadSanitizer is a tool that detects data races and deadlocks;
- UBSan: UndefinedBehaviorSanitizer is an undefined behavior detector, catching things like array subscript out of bounds (where the bounds can be statically determined), bitwise shifts that are out of bounds for their data type, dereferencing misaligned or null pointers and signed integers overflows;

Since sanitizers are designed to detect suspicious behavior during program execution, the combination of fuzzing with sanitization has been found very successful.

2 American Fuzzy Lop (AFL)

> American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. This substantially improves the functional coverage for the fuzzed code. The compact synthesized corpora produced by the tool are also useful for seeding other, more labor- or resource-intensive testing regimes down the road.

All file names cited are referred to the original google repository.

2.1 Compile-time instrumentation

AFL inserts its code-coverage instrumentation at compilation time. The instrumentation is done at assembly level. In other words, after the C/C++ source code of the target is compiled into assembly, the instrumentation code is added to the target and an instrumented assembly is generated. The instrumented assembly is then used to generate the target executable.
The companion tools that can be used as substitutes for GCC and Clang during the standard build process of the target code are `afl-clang`, `afl-clang++`, `afl-g++`, `afl-gcc`.
AFL's instrumentation is conducted injecting into the target's assembly additional instructions that allow monitoring and controlling the execution flow of the program. These modifications enable AFL to perform its fuzz testing efficiently. The main instrumentation logic is done in `afl-as.c`, and the codes to be instrumented, which is written in assembly, is in `afl-as.h` as string format.
The AFL instrumented code has the following basic units:

- `__afl_maybe_log`: It is typically inserted at strategic points within the program's execution flow, such as at the beginning and end of basic blocks or conditional statements. This function acts as a decision point where AFL determines whether to log the execution path taken by the program or not. One of the AFL's main goal is to maximize code coverage during fuzz testing by exploring as many different paths as possible within the

target program. The `__afl_maybe_log` function helps achieve this by selectively logging execution paths based on a coverage-guided strategy. AFL analyzes the instrumented code to identify interesting paths that have not been explored yet, and when such paths are encountered during runtime, `__afl_maybe_log` is called to record the path taken. By logging the execution paths, AFL can guide the generation of new test cases that exercise different parts of the program and potentially trigger previously unknown vulnerabilities. The logged path information is used to guide the fuzzing process and prioritize inputs that are likely to lead to unexplored execution paths.

- `__afl_setup`: is used to initialize shared memory pointer; in particular (pseudocode):

```
__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()
```

  Where:

  - `__afl_setup_failure` is a counter of the setup failures
  - `__afl_global_area_ptr` is a global pointer to shared memory
  - `__AFL_SHM_ID` is defined in config.h and the value of this variable is exactly the one returned by `shmget()` in `afl_fuzz.c`.
  - `__afl_area_ptr` is the pointer to the shared memory

Basically, the instrumented target gets a shared memory region ID from the environment variable and maps it in the binary via a call to `shmat`. The return value in `%eax` is the mapped address and is saved in `__afl_area_ptr`.

- `__afl_forkserver`: is used by the `__afl_setup` to activate the fork server. In simple words, the fork server code branches at the beginning of the main function, where a child process is forked if a new instance is needed for fuzzing. Going into details (pseudocode):

```
__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
```

Where:

– `__afl_fork_pid` is the cloned pid variable

– `__afl_temp` is a simple temp variable for different purposes

2.2 Genetic algorithm details

A general overview of the AFL algorithm can be written as follow:

Algorithm 1 Coverage-based Greybox Fuzzing

Input: Seed Inputs S

```
1: T✗ = ∅
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✗
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 $T✗$

Where:

- T is the list of inputs
- $T✗$ is the list of the crashing inputs
- p is the "energy" of the input t
- t' is the mutated input

2.2.1 Choosing the next input

To implement the `ChooseNext` method, AFL fetches test-cases from an ordered queue based on a hybric metric combining edge coverage (the percentage of edges in the program that have been traversed during fuzzing) with the edge

execution count (count of how many times the respective edge was executed in one run). The latter is bucketed to a power of two to avoid explosion of counters.

2.2.2 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.

2.2.3 Power scheduling

To allocate and distribute fuzzing time or resources among the test cases population AFL uses a mechanism known as *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 level* $e(s)$ of each test case s. In particular AFL uses the exploitation-based constant scheduling,

assigning the energy as a function α:

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

where:

- 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

2.2.4 Mutation engine

One of the most important things for a fuzzer is the mutation strategy. If the changes made to the input file are too conservative, the fuzzer will achieve very limited coverage. On the contrary, if the changes are too aggressive they will cause most inputs to fail parsing at a very early stage, wasting CPU cycles and producing messy test cases that are difficult to investigate and troubleshoot.
One interesting thing to notice is that the design of AFL provides a coverage-feedback loop: the test cases evolve to produce new input that are more likely to the discover new branches in the code, avoiding producing inputs that are just a waste of time.
The fuzzer also approaches every new input file by going through a series of progressively more complex, but exhaustive and deterministic fuzzing strategies, sequential bit flips and simple arithmetics, before diving into purely random behaviors. The reason for this is the desire to generate the simplest and most elegant test cases first; but the design also provides a very good way to quantify how much value each new strategy brings and if it is needed.
Now let's analyze in details the steps of the mutation engine:

1. **Walking bit flips**: the first and most rudimentary strategy employed by AFL involves performing sequential, ordered bit flips. The stepover is always one bit; the number of bits flipped in a row varies from one to four. Across a large and diverse corpus of input files, the observed yields are:

   - Flipping a single bit: 70 new paths per one million generated inputs.
   - Flipping two bits in a row: 20 additional paths per million generated inputs;
   - Flipping four bits in a row: 10 additional paths per million inputs;

   Note that the counts for every subsequent pass include only the paths that could not have been discovered by the preceding strategy.

   Of course, the strategy is relatively expensive, with each pass (e.g. flipping a single bit) requiring 8 new execution calls per every byte of the input file. Hence AFL stops after these three passes and switches to a second, less expensive strategy.

2. **Walking byte flips**: an extension of walking bit flip approach, this method relies on 8-, 16-, or 32-bit wide bitflips with a constant stepover of one byte. This strategy discovers around 30 additional paths per million inputs, on top of what could have been triggered with shorter bit flips. Now each pass (e.g. 8-bit wide bitflips) takes approximately one branch per one byte of the input file, making it surprisingly cheap, but also limiting its potential yields in absolute terms.

3. **Simple arithmetics**: to trigger more complex conditions in a deterministic fashion, in the third stage of AFL, it employs a technique to subtly increment or decrement existing integer values in the input file. The operation is done with a stepover of one byte, and the chosen range for the operation is -35 to +35.

   When it comes to the implementation, the stage consists of three separate operation:

   - First, AFL attempts to perform subtraction and addition on individual bytes in the input file. For example, if a byte in the input file has

the value 100, AFL may subtract 10 from it, resulting in 90, or add 10 to it, resulting in 110.

- Next, AFL looks at 16-bit values in the input file. It considers both endians and increments or decrements the values only if the operation would also affect the most significant byte. Otherwise, the operation would duplicate the results obtained from the 8-bit pass.
- Finally, AFL applies the same logic to 32-bit integers in the input file.

By incrementing or decrementing integer values in this manner, AFL aims to trigger different code paths and conditions within the program being tested. The range of values (-35 to +35) is experimentally chosen because fuzzing yields drop dramatically beyond these bounds. In particular, the popular option of sequentially trying every single value for each byte (equivalent to arithmetics in the range of -128 to +127) generally helpy very little and is skipped by AFL.

The yields, in terms of exploring different paths, vary depending on the specific file format. For example, AFL may find approximately 2 additional paths per million inputs in a JPEG format, while in an `xz` format it may discover around 8 additional paths per million inputs.

It's worth noting that this technique comes at a cost. On average, it requires about 20 executions calls per one byte of the input file. However, the cost can be reduced by limiting the range to +/- 16, which still provides a significant impact on path coverage while improving efficiency.

4. **Known integers**: in its last deterministic approach, AFL employs a predefined set of integers specifically chosen for their high likelihood of triggering edge conditions in typical code.

   This set of integers includes values such as -1, 256, 1024, MAX_INT-1, and MAX_INT (where MAX_INT represents the maximum value of an integer data type in the programming language). These integers are hardcoded into AFL as "interesting" values due to their predisposition to uncover potential issues.

During the fuzzing process, AFL uses a stepover of one byte to sequentially overwrite existing data in the input file with one of these known "interesting" values. It performs these writes in both endians (little-endian and big-endian) and covers different data widths, including 8-bit, 16-bit, and 32-bit writes.

By replacing existing data with these known integers, AFL aims to explore particulars conditions and trigger specific code paths that might exhibit different behavior or reveal vulnerabilities. The goal is to uncover bugs or unexpected behaviors that arise when the program encounters these particular values. Some examples can be:

- edge conditions and exceptional behavior in the program
- boundary conditions and potential integer overflow scenarios
- issues related to numeric bounds or comparisons

The yields for this stage, in terms of discovering new paths, typically range from 2 to 5 additional paths per one million fuzzing attempts. However, it's important to note that the average cost for this stage is approximately 30 execution calls per one byte of the input file.

5. **Stacked tweaks**: when AFL has exhausted deterministic strategies for a particular input file, it enters this phase where applies a sequence of randomized operations to the input file in an ongoing loop. The operations consist of the following:

   - *Single-bit flips*: AFL randomly flips individual bits within the input file. This operation can introduce small changes to the file's contents and potentially trigger different code paths.
   - *Attempts to set "interesting" bytes, words, or dwords*: AFL tries to set specific bytes, words (16 bits), or dwords (32 bits) within the file to predefined "interesting" values. These values are chosen for their likelihood of triggering edge conditions or exceptional behavior in the program. Both endians (little-endian and big-endian) are considered during this operation.

- *Addition or subtraction of small integers to bytes, words, or dwords*: AFL performs random addition or subtraction of small integers to bytes, words, or dwords within the file. These operations can introduce variations in numeric values and potentially affect the program's behavior.
- *Completely random single-byte sets*: AFL randomly sets individual bytes within the file to completely random values. This operation introduces a high degree of randomness and can explore unforeseen code paths.
- *Block deletion*: AFL randomly selects and deletes blocks of data within the file. This operation simulates the removal of certain data regions and tests how the program handles missing or incomplete information.
- *Block duplication via overwrite or insertion*: AFL duplicates blocks of data within the file by either overwriting existing data or inserting additional copies. This operation can test how the program handles duplicated or expanded data regions.
- *Block memset*: AFL sets a block of data within the file to a specific value, typically zero. This operation can help uncover issues related to the handling of initialized or zeroed memory regions.

To optimize the execution path yields, AFL has determined through extensive testing that it's beneficial for each operation to have roughly equal probability. Additionally, the number of stacked operations is chosen as a power-of-two between 1 and 64. This ensures a balanced exploration of different operations while avoiding excessive complexity.

Furthermore, for block operations (block deletion, duplication, and memset), the block size is typically capped at around 1 kB. This limitation helps maintain a balance between effectiveness and performance.

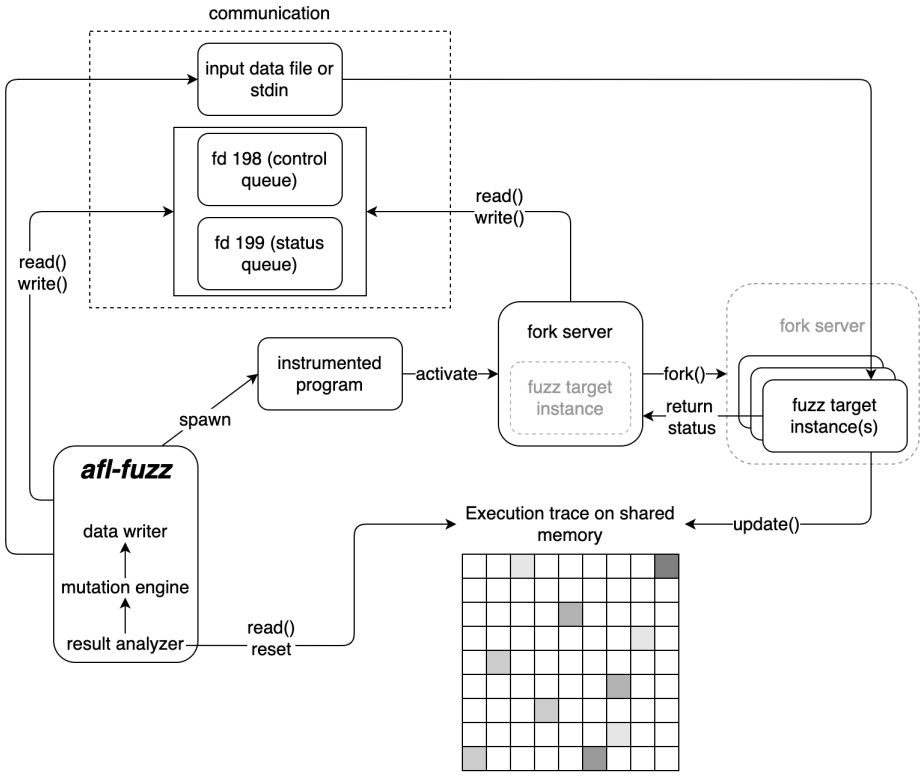
Overall, the absolute yield for the stacked tweaks stage is typically comparable to or higher than the total number of execution paths discovered during earlier deterministic stages.

6. **Test case splicing**: assume that we have a corpus of input files in the AFL queue. Test case splicing is a last-resort strategy that AFL uses when other techniques have been exhausted.

   Let's say we have two distinct input files in the queue: File A and File B. These files should have at least two locations where they differ. To apply test case splicing, AFL randomly selects a location in the middle of both files and creates a transient input file by combining the beginning of File A with the end of File B. This creates a new input file, referred to as the spliced file. The spliced file is then passed through a short run of the *stacked tweaks* phase, which involves applying a randomized sequence of operations as described earlier.

   The purpose of test case splicing is to create a hybrid input file that combines different characteristics from both File A and File B. By doing so, AFL aims to explore new execution paths that may not have been triggered by either File A or File B alone. Typically, test case splicing can discover around 20% additional execution paths that were unlikely to be found using previous operations alone. This strategy leverages the diversity and variability within the corpus of input files to uncover new code paths and potentially identify hidden bugs or vulnerabilities. It's important to note that for test case splicing to be effective, a good and varied corpus of input files is necessary. AFL will hopefully generate this kind of corpus automatically during the fuzzing process.

2.3 Going into communication details



Now it is useful to link all the points in order to have a global vision of the process and which are the interactions between the components of AFL.

1. `afl-fuzz` spawns the instrumented program that has to be fuzzed.
2. The instrumented program creates a fork server that is responsible for:
   - communicating with the main `afl-fuzz` executable over pipes;
   - spawning new fuzz target instances with `fork()` calls;
3. The fork server acts as a separate process responsible for executing the target program on the generated test inputs. So before the target instance of the program is executed, the fork server has to read the input data that has to be passed to the target.

- `afl-fuzz` program has two ways to pass the new input to the fuzz target:
  - passing the data over the standard input;
  - creating a new file with the new data and having the fuzz target to read the data from the just created file
- Communication via File Descriptors: The fork server communicates with the fuzzed process using file descriptors 198 and 199. File descriptors are integer values that represent open files or communication channels.
- The file descriptors are inherited from father process to child each time that and new process is forked. So the fuzz target gets the input data from its standard input file descriptor so it does not need to explicitly open any files. Technically the standard input of the child process is backed by a file that is always modified by the `afl-fuzz` process when new input it generated.

4. After input data are passed to the fuzz target either over standard input or over a newly created file that the fuzz target reads, the fuzz target executes itself by writing an execution trace to a shared memory and finishes running.

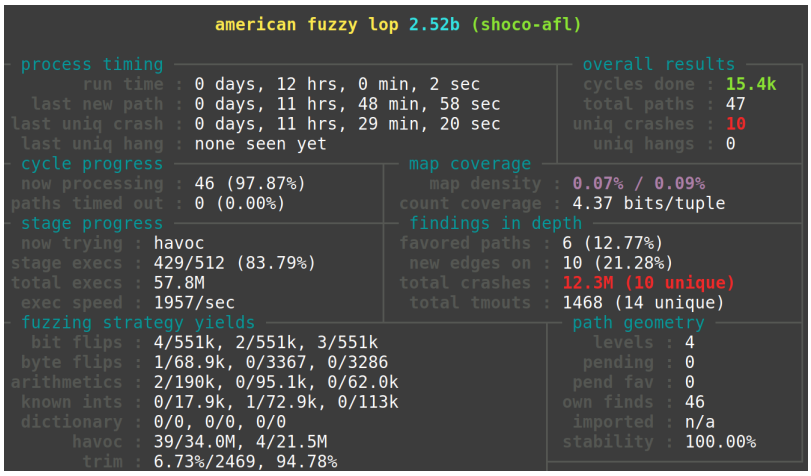
5. `afl-fuzz` reads the trace left by the fuzz target from the shared memory and creates a new input by mutating old ones. What to mutate is controlled by the new and historical information that the instrumentation data from the fuzz target provides.

6. Finally the new inputs are passed again to the fork server, so `GOTO 3`

(The fork server is actually a small part of the instrumented fuzz target. The new program instance that fork() call spawns still technically holds a reference to the fork server, but those parts are just active at different times, visualized as gray in the figure. If the fuzz target is in *persistent mode*, it doesn't get restarted for every new input.)

2.4 AFL's GUI

It's worth giving a look at AFL's graphical interface to understand what information offers to the user.



Going from top to bottom and from left to right, the interface is made of several components:

- the timing section: it tells how long the fuzzer has been running and how much time has elapsed since its most recent finds. It is broken down into "paths", which are test cases that trigger new execution patterns, crashes, and hangs;
- results section: it reports the count of queue passes done so far, that is the number of times the fuzzer went over all the interesting test cases discovered so far, fuzzed them, and looped back to the very beginning. Every fuzzing session should be allowed to complete at least one cycle; and ideally, should run much longer than that. The remaining fields in this part of the screen should the number of paths discovered so far and the number of unique crashes;
- cycle progress section: it tells how far along the fuzzer is with the current queue cycle: it shows the ID of the test case it is currently working on, plus the number of inputs it decided to ditch because they were persistently timing out;

- map coverage section: the first line in the box tells how many branch tuples we have already hit, in proportion to how much the bitmap can hold. The number on the left describes the current input; the one on the right is the value for the entire input corpus. The other line deals with the variability in tuple hit counts seen in the binary. In essence, if every taken branch is always taken a fixed number of times for all the inputs we have tried, this will read "1.00". As we manage to trigger other hit counts for every branch, it will start to move toward "8.00" (every bit in the 8-bit map hit), but will probably never reach that extreme.

- stage progress section: it gives an in-depth peek at what the fuzzer is actually doing right now in terms of mutations strategy, together with the execution count progress indicator for the current stage, a global execution counter, and a benchmark for the current program execution speed;

- findings in depth section: it includes the number of paths that the fuzzer likes the most, and the number of test cases that actually resulted in better edge coverage (versus just pushing the branch hit counters up). There are also additional, more detailed counters for crashes and timeouts.

- fuzzing strategy yields section: it keeps track of how many paths have been netted, in proportion to the number of executions attempted, for each of the fuzzing strategies discussed earlier on.

- path geometry section: the first field in this section tracks the path depth reached through the guided fuzzing process. In essence: the initial test cases supplied by the user are considered "level 1". The test cases that can be derived from that through traditional fuzzing are considered "level 2"; the ones derived by using these as inputs to subsequent fuzzing rounds are "level 3"; and so forth. The next field shows you the number of inputs that have not gone through any fuzzing yet. The same stat is also given for "favored" entries that the fuzzer really wants to get to in this queue cycle (the non-favored entries may have to wait a couple of cycles to get their chance). Next, we have the number of new paths found during this fuzzing section and imported from other fuzzer instances when doing parallelized

fuzzing; and the extent to which identical inputs appear to sometimes produce variable behavior in the tested binary. The last bit measures the consistency of observed traces. If a program always behaves the same for the same input data, it will earn a score of 100

- CPU load section: it shows the apparent CPU utilization on the local system. It is calculated by taking the number of processes in the "runnable" state, and then comparing it to the number of logical cores on the system.

3 LLVM LibFuzzer

> LibFuzzer is an in-process, coverage-guided, evolutionary fuzzing engine.

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). The implementation of this harness is user-defined:

```
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.

3.1 LLVM SanCov

LLVM SanitizerCoverage is a simple code coverage instrumentation integrated into LLVM compilers. It can be used to insert user-defined functions on function, basic block and edge levels. A default implementation of this callbacks is also provided and can be turned-on at compile time. The possible compiler flags are:

- `-fsanitize-coverage=trace-pc`: this flag is used to trace basic code coverage information for a certain input. Basically, the compiler will insert instrumentation into the target to mark the basic blocks "hit" during

execution. The invoked callback is `__sanitizer_cov_trace_pc`, which updates a bitmap used to mark blocks being hit.

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

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

//Clang output with -O0 -fsanitize-coverage=trace-pc
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
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
```

- `-fsanitize-coverage=trace-pc-guard`: with this flag the compiler will insert the `__sanitizer_cov_trace_pc_guard(&guard_variable)` callback on every edge. Guards are defined by LLVM as unique identifiers of basic blocks, expressed as `uint32_t`. Since every edge is identified by its own guard variable, the runtime is able to track edge-specific state information which is much richer than a simple hit bitmap. Basic blocks can now be identified uniquely and associated to additional information, at the discretion of who implements the callback. A constructor callback `__sanitizer_cov_trace_pc_guard_init` will also be called by the compiler to initialize the backing array storing guards numbers:

```
//Clang output with -O0 -fsanitize-coverage=trace-pc-guard
test(int):                                # @test(int)
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        movabs  rax, offset .L__sancov_gen_
        mov     dword ptr [rbp - 8], edi # 4-byte Spill
        mov     rdi, rax
        call    __sanitizer_cov_trace_pc_guard
        mov     ecx, dword ptr [rbp - 8] # 4-byte Reload
```

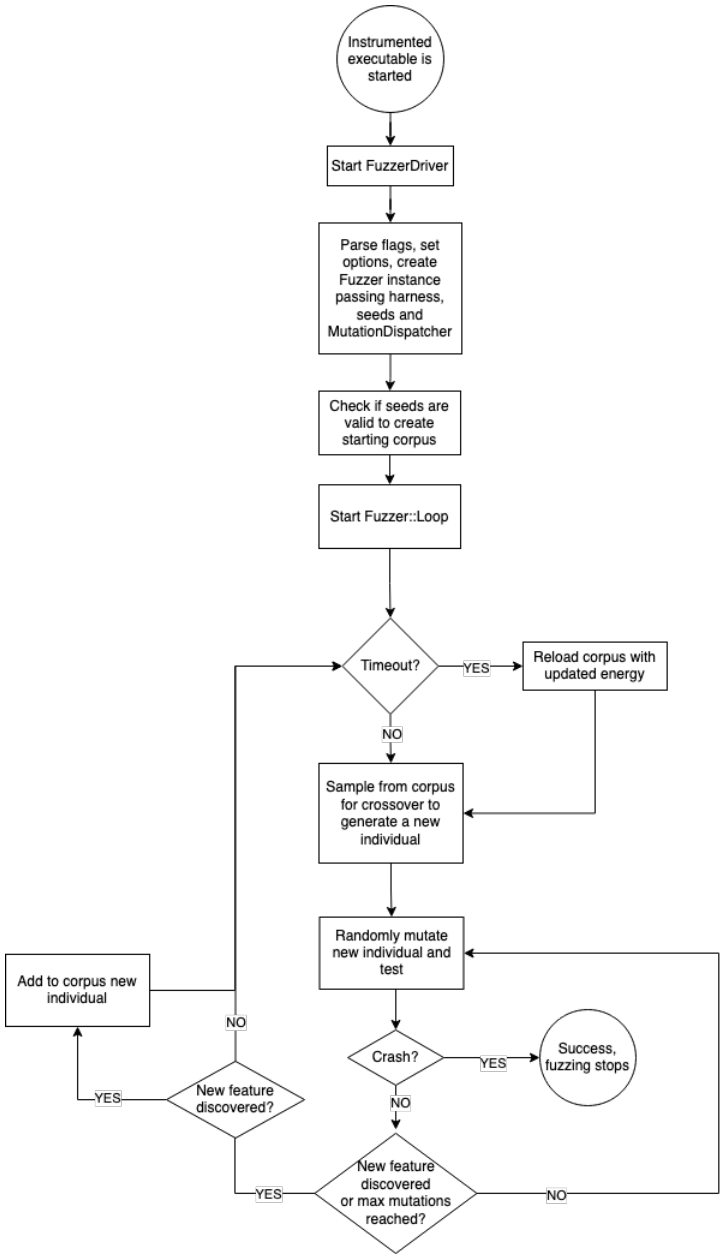
```
        mov     dword ptr [rbp - 4], ecx
        mov     edx, dword ptr [rbp - 4]
        add     edx, 1
        mov     eax, edx
        add     rsp, 16
        pop     rbp
        ret
main:                                   # @main
        push    rbp
        mov     rbp, rsp
        sub     rsp, 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
```

There are multiple invocations of `__sanitizer_cov_trace_pc_guard` and they all use constant values like `.L__sancov_gen_.1`. These correspond to specific elements in the array that was passed into the constructor callback. They can be thought as corresponding to names for each edge and this is how we can track which edges are executed and in which order.

3.2 How LibFuzzer works

During compilation, LibFuzzer instruments the code of the target API with its custom SanCov callbacks to keep track of code coverage information. Not only that, it creates a fuzzer executable, runnable with multiple options and flags regarding the corpus, grammar of mutations, dictionary, corpus minimization, etc. Notice that during compilation there is the possibility to instrument the target also with ASan or MSan or similar, in order to discover more bugs related to memory management.
The code mentioned comes directly from LibFuzzer's source, included in the LLVM compiler infrastructure at: `llvm-project/compiler-rt/lib/fuzzer/`.



Here are the high-level steps LibFuzzer does when a fuzzer executable is run:

- When the fuzzer starts, it just builds a `FuzzerDriver` object instance, passing as arguments the command line arguments and the `LLVMFuzzerTestOneInput` provided by the user.

- FuzzerDriver parses the flags and sets the options of the fuzzer. If the intent of the user was fuzzing the target callback (the alternative could be running a set of inputs and then exiting, possibly collecting information about coverage), three objects are created and then used: an object representing the corpus, a `MutationDispatcher`, and a `Fuzzer` object that takes as arguments the other two plus the harness and the options. After, the fuzzing process is started entering on the fuzzer loop.

- Before entering the fuzzing loop, the method `ReadAndExecuteCorpora` is run in order to be sure that the provided corpus doesn't make the target crash and it is actually useful, meaning that the inputs flow inside the target touching at least some basic blocks (otherwise there may be some problem with the target or the code may not be instrumented for code coverage). Note that the if the initial corpus is not present or does not seem to be valid LibFuzzer will insert a default seed containing only `\n`.

- When the fuzzing loop begins, the initial population of generated inputs (consisting only of the starting seeds) is periodically refreshed. During each iteration, a new individual (input) is created from the most recently loaded population using the `MutationDispatcher`, first by crossover and then using one or multiple mutations. The new individual is mutated and tested until either LibFuzzer founds some unique features generated by it or it reaches the limit for max mutations. In this context, a feature refers to any signal (such as new basic blocks touched, new functions reached, system calls, signals, etc.) that indicates a change in the program's behavior when a specific input is provided. It is a broader concept with respect to the code coverage provided by LLVM. If the new individual has generated new coverage it is added to the input corpus, hence it may be picked again for mutation in the next iterations.

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 exploit some mutations that are similar to AFL but also different ones; in particular some mutations are:

- Erase byte: a byte from the current sequence is removed. Erasing a byte, effectively removing one piece of data from the original data, results in a smaller size. The byte to be erased is usually chosen randomly from the data

  Example: 01010010 ~~11011010~~ 11010000 → 01010010 11010000

- Insert byte: it is the opposite operation of EraseByte. Within the selected sequence, a random position is chosen where the new byte will be inserted. After a random byte value is generated and is then inserted at the chosen position in the sequence, increasing its size by one byte.

  Example: 01010010 11011010 → 01010010 11011010 *11010000*

- Insert repeated bytes: is an InsertByte but now are added at least 3 ramdom bytes

- Single bit flip: already detailed for AFL

- Change byte: a random byte is selected in order to substitute it with random one. It can be also seen as a single bit flip repeated on some bits of the randomly selected byte.

  Example: 01010010 *11010000* → 01010010 10010101

- Shuffle bytes takes a data sequence (usually a byte array or string) and randomly rearranges its bytes in a new order. For example, suppose you have the input data "ABCDEFG", after applying the operation, it could become "DABGFCE" or any other random permutation of the original bytes.

- Change ASCII integer: find ASCII integer in data, perform random math ops and overwrite into input For example:

  1. A function searches for occurrences of ASCII integers within the provided data sequence. If the input data contains the string "Hello123World", the function will find the ASCII integer "123" within it.
  2. After, the fuzzing process performs random mathematical operations on it. These math operations could include addition, subtraction, multiplication, division, etc. The purpose of these random math operations is to generate new input variations from the original ASCII integer. Suppose a random addiction of 50.
  3. Finally, the fuzzing process overwrites the original occurrence of the ASCII integer in the input data with the result of the math operations. The "123" become "173", and this value replace the original one.

- Change binary integer: do the same of Change ASCII integer, but now instead of search occurrences of ASCII integers, it has a function that search occurrences of binary integer.

  So the steps are: find binary integer in data, perform random math operations and overwrite into data sequence.

- Copy part: randomly select a start position of the data sequence and a size. Then is extracted the part of the sequence that has as length the size extracted, starting from the start position.

  Example: start = 1 size = 2, 01010010 *11011010 11010000* $\rightarrow$ 11011010 11010000

- Crossover: combine two inputs from the corpus to produce a single offspring, as in classical crossover;

- Add word to persistent AutoDict: first of all it is useful to define an automatically generated dictionary (AutoDict): it is a dictionary that keeps track of words observed in previously processed inputs. This dictionary helps LibFuzzer recognize common patterns that have occurred before.

Thus, LibFuzzer replaces part of input with one that previously increased coverage in past executions.

- Add word to temporary AutoDict: this one is similar to the previous mutation, but now the automatically generated dictionary has just the most recent words that have increased coverage, instead of having all the previous ones.

3.3 LibFuzzer's GUI

LibFuzzer's GUI is very similar to a live log. Here's a brief example:

```
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-
```

The early parts of the output include information about the fuzzer options and configuration, including the current random seed.
After that, fuzzing starts and each line contains:

- an event code, like INITED (the fuzzer has completed initialization, which includes running each of the initial input samples through the code under test), NEW (the fuzzer has created a test input that covers new areas of the code under test), REDUCE (the fuzzer has found a better (smaller) input that triggers previously discovered features), RELOAD (the fuzzer is performing a periodic reload of inputs from the corpus directory);
- `cov`: total number of code blocks or edges covered by executing the current corpus;

- `ft`: total number of features covered;
- `corp`: number of entries in the current in-memory test corpus and its size in bytes;
- `lim`: current limit on the length of new entries in the corpus;
- `rss`: current memory consumption;
- `exec/s`: current executions per second (this sometimes is 0 even if everything is fine);
- `L`: for NEW and REDUCE events, the size of the new input in bytes;
- `MS`: for NEW and REDUCE events, the count and list of the mutation operations used to generate the input.

4 Honggfuzz

> Honggfuzz is a security oriented, feedback-driven, evolutionary, easy-to-use fuzzer.

HonggFuzz usage is very similar to AFL, while its internal behavior and structure borrows some characteristics from LibFuzzer and others from AFL. For example, it can provide input to the fuzzed program via `stdin` or external files like AFL, but its coverage-feedback core and sanitization mechanisms are based on its own LLVM LibFuzzer's customized callbacks. Moreover, Honggfuzz leverages by default the address sanitization mechanism provided by LLVM ASan, and other mechanisms of sanitization like MSan can be turned on as well.
Not only that, 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 and threads to take advantage of the full potential of the CPU cores. The interaction between processes and threads happens via signal monitoring and shared memory as in AFL, but its parallelism is way more optimized with respect to AFL. In Honggfuzz there are multiple concurrent threads mutating and executing on their own and updating concurrently a shared dynamic corpus (which is the only source of concurrency overhead), while in AFL it's only the main process that takes care of the evolution of the fuzzed inputs, waiting for its child processes to execute the inputs and return their status.
Borrowing some of its core components from LLVM LibFuzzer, and combining this with its clever usage of the CPU, it is very fast in persistent fuzzing mode, where instead of fuzzing an entire instrumented program you test only a single API.
Moreover, Honggfuzz uses low-level interfaces to monitor processes (e.g. ptrace under Linux and NetBSD), and it will discover and report hijacked/ignored signals from crashes.

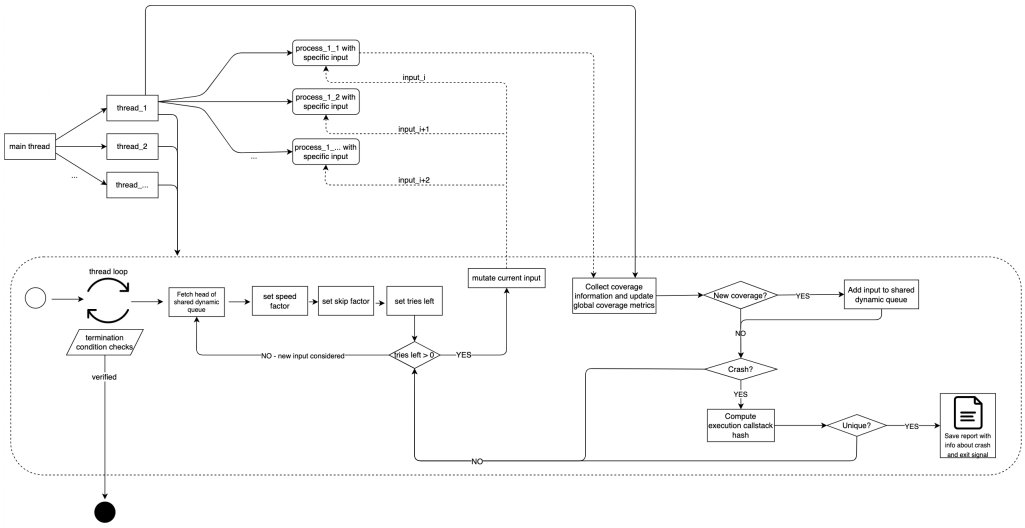
4.1 How Honggfuzz works

4.1.1 The honggfuzz_t data structure

The Honggfuzz data structure represents the main configuration and state information of the fuzzing process. It is a crucial data structure that holds various settings, statistics, and pointers necessary for the fuzzing processes to function correctly and work together towards a good evolution of the dynamic corpus. In particular, it contains information about:

- Threads: Since the fuzzing engine is usually multi-threaded, this structure contains thread-specific data and also general details about the activated threads;
- IO: in that structure information about the input and output data of the fuzzing process are stored but also the pointers or data structures related to the generated new input test cases (dynamic corpus queue, and more);
- General execution parameters and options;
- Mutation options and parameters
- Timing data about the fuzzing process;
- Mutex data structures: in order to access, to shared data during the fuzzing process across the different threads;
- More interface variable/data;

4.1.2 Execution flow



First of all the `honggfuzz_t` data structure is allocated globally in the main process in order to have all the information needed during the Honggfuzz execution. As already mentioned, the structure is provided with objects for acquiring R/W locks, allowing for correct synchronization between threads.
The main thread create a certain number of threads in order to increase the speed; where the number of threads created is contained in the `honggfuzz_t` data structure (`threads` data structure of `honggfuzz_t`). When threads starts all of these start to check if the termination condition is verified; in particular it goes to check if the number of mutation reaches the max number of mutation, so the max number of iterations (`mutate` data structure of `honggfuzz_t`).
If the termination condition is not satisfied, the thread fetches the input, and to do that it takes the head of the shared dynamic queue (contained in the *io* data structure of `honggfuzz_t`) in order to mutate that input. But in order to mutated the input we need first to:

1. set the `speedFactor`
2. set the `skipFactor`
3. set `triesLeft`

In the Honggfuzz power schedule, the energy of an input is represented by the variable `run->triesLeft`. When an input is fetched from the head of the dynamic corpus queue, it is immediately assigned a certain number of "tries" based on its past performance compared to previous inputs. The number of tries depends on the opposite of a penalty factor called `skipFactor`, which can be positive or negative.

If the penalty is positive, the input is skipped. Otherwise, the opposite of the penalty value is used as the energy value for the input. To compute the `skipFactor`, Honggfuzz first calculates a `speedFactor`, which assigns an initial penalty to the input based on its execution speed compared to the average execution speed of previously seen inputs. Then, Honggfuzz combines this penalty from the `speedFactor` with additional penalties from other performance metrics, such as coverage induced by the input, input age, size, and reference count (indicating how many good mutations this input has generated so far).

This approach assigns more energy to inputs that are fast, new, small, have good coverage, and tend to produce mutations likely to be valuable additions to the dynamic corpus.

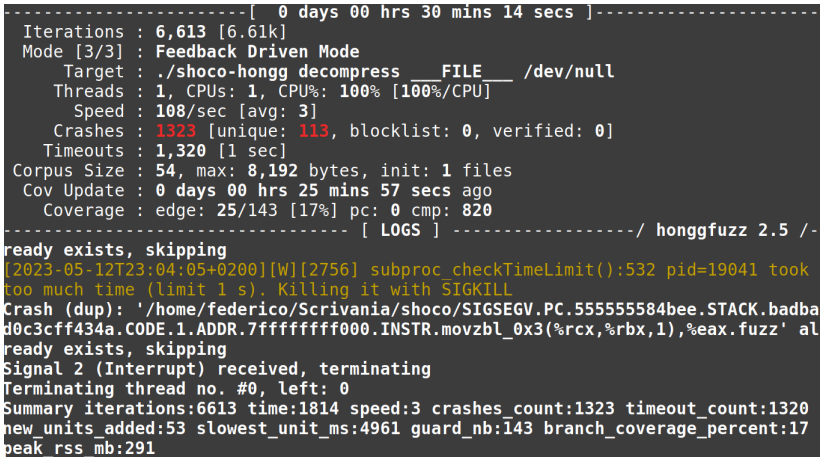
Furthermore, the number of mutations performed on the current input depends on the `speedFactor` and the `mutate` data structure (of `honggfuzz_t`). Thus, a certain number of random mutations is applied to the input. The kind of mutations employed by Honggfuzz in this step are more or less the same of the ones used by LibFuzzer. One interesting difference is the lack of crossover; similarly to AFL, Honggfuzz uses test-case splicing as a last resort strategy.

After the mutation step, a new process running the target program is created by the thread, and the mutated input is then passed to the process to start the execution.

During the process, signals and coverage information are registered. When the process is terminated, the global coverage metrics are updated (the global ones are in the `feedback` and `cnts` data structures contained in `honggfuzz_t`). If any new coverage information is induced by the current input, it is added to the shared dynamic queue of inputs (insertion is ordered by coverage). If the process crashes, the thread will analyze the exit status to gather more information about what went wrong and will compute the hash of its execution call stack to check

whether it's unique or not. Since the crash is identified as unique using only the call stack hash, Honggfuzz does not differentiate between crashes if their call stacks are equal. If the process does not crash or the crash is not unique, the thread returns to stage of input fetching for mutation (if there is still available energy for the current input the next input will still be the current).
Finally, if the crash is unique, a report with information about the crash and exit signals is saved. Otherwise, the thread returns to the input fetching stage.

4.2 Honggfuzz's GUI



Honggfuzz's graphical interface is way less communicative than AFL's one. From top to bottom it reports:

- the number of executions;
- the current fuzzing mode, which should be "Feedback Driven Mode" for our means;
- the target command;
- CPU utilization, threads running;
- the current and average number of executions per second;
- a report line about the crashes so far;
- the number of timeouts;

- the current size of the dynamic corpus, with the max size in bytes as well for each input;
- the time passed from the last update in coverage;
- coverage information: ratio of visited/unvisited edges, trace PCs with hardware instrumentation, traced compares;
- the last lines written to the execution logs;

5 Fuzzing practice

After examining the fundamental characteristics of the three fuzzers, a limited number of tests were conducted. The tests were initially carried out on programs and libraries with established vulnerabilities, and are then followed by more tests on two minor text-compression utilities sourced from Github. Notably, the last two programs had unidentified vulnerabilities.

5.1 Fuzzer-test-suite (Google)

The Google Fuzzer Test Suite is a testing framework developed by Google to compare the performance and behavior of different fuzzers over the same programs. This suite is designed to automatically identify vulnerabilities and potential flaws in software by subjecting it to a rigorous and systematic fuzzing process. The suite supports natively AFL, Honggfuzz and LibFuzzer as fuzzing engines. The compilation and instrumentation of target programs and libraries is automatized by the suite through a set of command line tools and flags. We decided to test the following programs using the suite:

- `boringssl-2016-12-02`: an open-source cryptography library developed by Google. It's designed for providing cryptographic functions and protocols for various software projects while prioritizing simplicity, reliability, and security.
- `c-ares-CVE-2016-5180`: c-ares is a lightweight open-source library that performs asynchronous DNS requests. It simplifies network programming by handling DNS resolution in the background, allowing applications to continue running without waiting for DNS responses.
- `libxml2-v2.9.2`: a widely-used open-source software library for parsing and manipulating XML and HTML documents.
- `openssl-1.0.1f` and `openssl-1.0.2d`: another widely-used open-source cryptography library that provides essential tools and libraries for secure communication and data protection.

- `re2-2014-12-09`: an open-source software library developed by Google for efficient and safe regular expression matching. It provides a fast and memory-efficient alternative to traditional regular expression engines, ensuring predictable performance and preventing certain types of worst-case scenario attacks.

5.1.1 Basic fuzzer-test-suite behavior

The suite commands are rather easy. To choose the fuzzer to test, an environment variable `$FUZZING_ENGINE` must be set appropriately. The default fuzzing engine is LibFuzzer. From the root folder of the project and the environment variable set, you can decide to build one of the target programs included in the suite using executing the script `./{target-program}/build.sh`. This will take care of downloading, compiling and linking the appropriate files for testing and creating the executable to be fuzzed. The suite will turn on address sanitization by default. The fuzzing process can be started with the following commands:

```
#LibFuzzer
./target-executable
#Honggfuzz
honggfuzz -i input_folder -o output_folder -- ./target-executable
#AFL
afl-fuzz -i input_folder -o output_folder -- ./target-executable
```

5.1.2 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

The test suite by Google indicates also the type of bugs each target is susceptible to raise. Taking in consideration the aggregate results from all three fuzzers, we managed to uncover all the bugs but the one in boringssl-2016-02-12.

The statistics seem to indicate that LibFuzzer is the most consistent across the trio. Despite its single-threaded core, LibFuzzer manages to achieve comparable coverage to AFL but within a shorter timeframe. This could be attributed to LibFuzzer's distinctive energy-based system, which markedly differs from the other two fuzzers. By employing energy as a parameter to establish a probability distribution for corpus sampling and incorporating default crossover, LibFuzzer seemingly produces superior individuals and achieves broader coverage in less time, even though its executions per second are much lower in comparison.

On the contrary, Honggfuzz consistently achieves significantly wider coverage in less time with respect to the other two, possibly due to its efficient multi-threaded and multi-process architecture characterized by minimal inter-thread communication. However, this greater speed does not necessarily correspond to

an increased frequency of actual crashes. Honggfuzz may excel particularly in scenarios involving large programs necessitating fast and comprehensive coverage, a task unsuitable for LibFuzzer given its non-performance-oriented nature.

5.2 Shoco

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

1. `shoco_compress`: takes input and output buffers and their size, fills the output buffer with the compressed representation of the input buffer and returns the size of the compressed data. If the length argument is 0, the input 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, whatever comes first.

2. `shoco_decompress`: takes input and output buffers and their size, fills the output buffer with the decompressed representation of the input buffer and returns the size of the decompressed data.

In order to do that Shoco makes available its two primitives:

1. `size_t shoco_compress(const char * const shoco_restrict original, size_t strlen, char * const shoco_restrict out, size_t bufsize)`

2. `size_t shoco_decompress(const char * const shoco_restrict original, size_t complen, char * const shoco_restrict out, size_t bufsize)`

5.2.1 LibFuzzer

In order to fuzz Shoco's primitives with LibFuzzer it is necessary to write the following two code functions.

`target_compress.c`

```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;
}
```

`target_decompress.c`

```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;
}
```

After that it is necessary to edit the make file in order to create the LibFuzzer's instrumented executable file.

`Makefile`

```
# 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
```

In order to instrument a target API with LibFuzzer, it is necessary to compile with `clang`. LLVM's documentation reports that the flag `-g` for the debug symbols and `-O1` for code optimization are needed. The `-fsanitize` flag must be assigned at least the `fuzzer` parameter to use LibFuzzer, but other parameters like `address` or `memory` can be added in order to activate address sanitization or memory sanitization (which are mutual exclusive). After that, the source files necessary for the API and the file containing the harness are specified, together with the name of the instrumented executable.
In order to run the fuzzing process with LibFuzzer you need to run the following commands:

```
make libfuzzer
time ./libfuzzer-decompress -print_final_stats=1 2> fuzz.log
time ./libfuzzer-compress -print_final_stats=1 2> fuzz.log
```

The `time` command is used to determine how long the fuzzing process takes to run.
`-print_final_stats=1` is a LibFuzzer option in order to print statistics at exit
`2>` is useful to redirect the output on a log file

5.2.2 AFL

For AFL, things are simpler because you just need to run the makefile specifying the AFL compiler (`CC`); this is possible thanks to the fact that the shoco makefile has a parametric compiler that you can specify.

```
make CC=afl-clang
afl-fuzz -i input -o output -m none -- ./shoco-afl compress
@@ /dev/null
afl-fuzz -i input -o output -m none -- ./shoco-afl decompress
@@ /dev/null
```

Now, as you can notice, you have also to specify command line arguments: in particular the functionality (`compress` or `decompress`), a placeholder for AFL to insert the generated test cases during fuzzing (`@@`) and finally `/dev/null` that is a special file in Unix-like systems that discards all data written to it. This indicates that the output of the decompression will be ignored, as the goal is to

focus on code coverage and crash discovery rather than the actual output.

5.2.3 Honggfuzz

For Honggfuzz things are very similar to AFL, so you can do as follows:

```
make CC=hfuzz-clang
honggfuzz -i input -o output -- ./shoco-hongg decompress
___FILE___ /dev/null
```

Also the command structure is very similar, but a difference that you can notice is the placeholder `___FILE___`, this is the same of `@@` of AFL, and it is for the input file that will be fuzzed. The fuzzer will replace this placeholder with the actual input files from the input directory during the fuzzing process.

5.2.4 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)

Regarding crashes LibFuzzer is the only one that is able to find them both for the compress and also for the decompress. Another interesting thing is that Honggfuzz for the decompress is able to find a huge number of unique crashes; in particular consider that for that reason Honggfuzz is stopped after 30 minutes! So on average it found 3.8 crashes every minute.

Regarding the coverage AFL reaches the best coverage of the three. In LibFuzzer for the coverage there is not a value because it crashed immediately at the beginning.
Finally for the type of crashes is important to highlight the fact that Honggfuzz for the Shoco decompress found both the *global-buffer-overflow* just found by LibFuzzer and AFL, but also a *stack-overflow*.

5.3 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; for example the string "the" is compressed into a single byte. Just to get a comparison with other common libraries, zlib will usually not be able to compress text shorter than 100 bytes.
Smaz publishes two APIs:

- `int 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.

- `int 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.

5.3.1 LibFuzzer

`target-compress.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;
}
```

`target-decompress.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;
}
```

`Makefile`

```
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
```

Some flags were present in the original `smaz` Makefile and we decided to maintain them.

5.3.2 AFL-Honggfuzz

`smaz` is a library with no executables; we have written two simple C main functions that just open the file whose name is passed through the first command line argument and call either `smaz_compress` or `smaz_decompress`.

`main-compress.c`

```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;
}
```

main-decompress.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_decompress(content, length, buffer, 8192);
    // free the allocated memory
    free(content);
    return 0;
}
```

`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
```

5.3.3 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

Even if `smaz_compress` seems to be rather safe, LibFuzzer managed to identify a *heap-buffer-overflow* in `smaz_decompress`. Notably, LibFuzzer managed to identify the bug immediately, while the other two continued to increase coverage for some time but did not identify the bug, while AFL and Honggfuzz can not spot any crash. Another interesting thing to notice is that for smaz AFL reach a better coverage respect the other two, as happened for shoco.

6 Reference

AFL

- https://github.com/google/AFL
- https://afl-1.readthedocs.io/en/latest/about_afl.html
- https://tunnelshade.in/blog/afl-internals-compile-time-instrumentation/
- https://www.core.gen.tr/posts/003-afl-internals/
- https://mboehme.github.io/paper/CCS16.pdf

LIBFUZZER

- https://clang.llvm.org/docs/SanitizerCoverage.html
- https://github.com/llvm/llvm-project/tree/main/compiler-rt/lib/fuzzer

HONGGFUZZ

- https://github.com/google/honggfuzz
- https://honggfuzz.dev/

ALTRO

- Fuzzing for Software Security Testing and Quality Assurance by Ari Takanen, Jared D. Demott, Charles Miller, Ed. Artech House 2008