yFuzz:

1. Are there any consequences of using an older version of AFL, e.g., breadth of supported input?

All updated files in the AFL 2020 version compared to the 2018 version can be found at the link below. We found that most of the changes are related to bug fixes, android support, extension of metrics for output layout, Minor documentation improvements, specifying the CPU core id to execute the tool etc. https://github.com/google/AFL/blob/master/docs/ChangeLog

Upon reviewing the changes, we believe that yFuzz should work with the latest version of AFL. Our yFuzz code primarily instruments four files in AFL: afl--fuzz.c, config.h, Makefile, main.cl, as summarized in Table 1. We also checked for any further modifications to all ".c" files in the AFL 2020 version (over 1.0b AFL version). The findings are summarized in Table 2.

Table 1. Source codes modified for yFuzz tool

Afl-fuzz.c - main program
Config.h - configuration parameters are added
Makefile - make file to compile
Main.c - main file which has all switch case statements

Table 2. Modifications between AFL 2020 and AFL 2017: (changes by the original authors of AFL)

afl-analyze.c	In the *configure shared memory* a line of code (line no. 174 is changed in the newer version)
afl-as.c	None
afl-gcc.c	None
afl-gotcpu.c	None

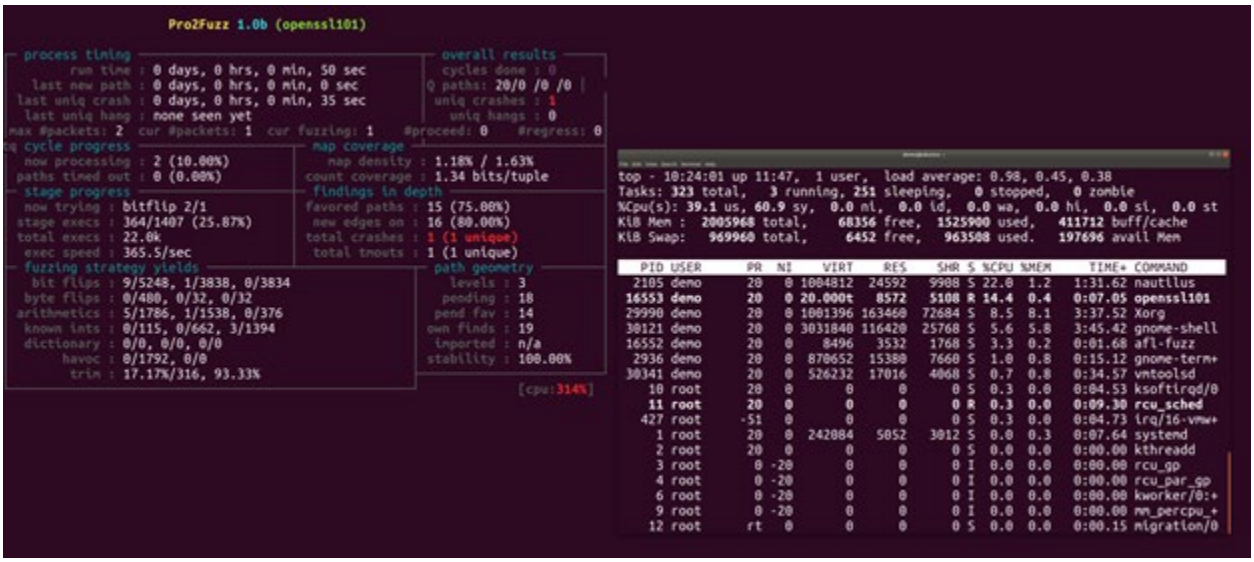
afl-showmap.c	In the *configure shared memory* a line of code (line no. 158 is changed in the newer version)
afl-tmin.c	In the *configure shared memory* a line of code (line no. 187 is changed in the newer version)
test-instr.c	None
Afl-fuzz.c	In the function "add_to_queue" a block of code (line 827) is changed to allow faster iterations but still our yFuzz tool with OpenSSL did not have any significant impact on the output. 2. More no.of flags have been added like version number and cpu core id 3. Extra metric for Output layout: 1. slowest_exec_ms - real time of the slowest execution in ms 2. peak_rss_mb - max rss usage reached during fuzzing in MB
Config.h	None
Makefile	None
Main.c	None

2. Memory cost of ASAN:

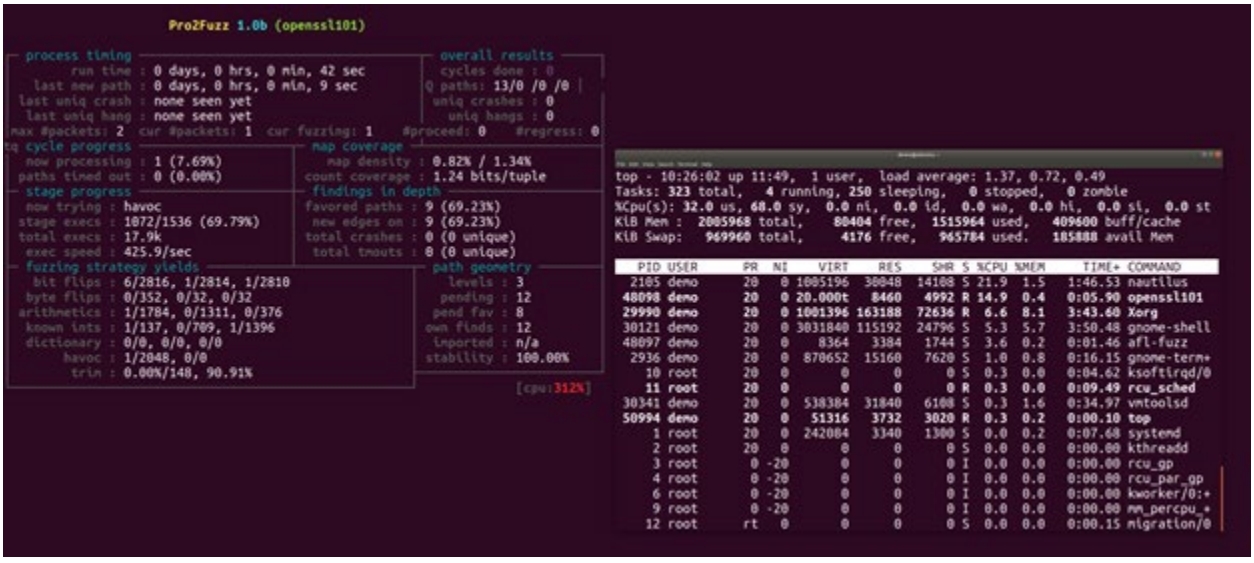
It is possible that ASAN will consume lots of memory space. In the worst case, it may consume 20 Tb memory space, depending on different machine configurations and programs. Please refer to GitHub discussion: https://github.com/google/sanitizers/issues/731.

However, during the test of yFuzz on OpenSSL, we didn't observe much differences in memory usage as ASAN was switched on/off. Dynamic memory usage stays around about 0.4% monitored by using *top* command in another terminal, and it does not vary much. We have saved the screenshots below.

Turn on ASAN:



Turn off ASAN:



3. Whether ASAN is required for yFuzz?

ASAN aims to detect memory-based bugs. Without ASAN, some bugs cannot be detected by AFL and yFuzz, such as a classic off-by-one-error. For more details, please refer to: https://fuzzing-project.org/tutorial2.html

P.S. To turn ASAN off, delete the configuration "AFL_USE_ASAN = 1" in the ***build*** file, which is located in the program directory (in VM it is ./yfuzz/testcase/openssl101).

4. Please provide feedback as desired on any of the issues/TODOs in the README.

We have spoken with Yurong Chen, the student who developed yFuzz and now works at FB Research. He mentioned the readme is a deprecated note during the development of yFuzz and can be safely ignored at this point. All issues noted there have already been resolved in the current yFuzz version shared at this point.

CustomPro:

1. How old is TEMU? 2009?

   Temu was last tested in 2009. Any questions regarding the BitBlaze project can be sent to bitblaze@gmail.com. More info on the source code updates can be found at http://bitblaze.cs.berkeley.edu/temu.html.

2. [Low, requires test] What is the impact of TEMU's age on support for newer versions of input software?

   While we have not not used DECAF in our project so far, we performed a preliminary study to compare TEMU's and DECAF's capabilities.

   Capabilities of TEMU: **TEMU** is able to perform whole-system dynamic taint analysis. Marking certain information sources (e.g., keystrokes, network inputs, reads for certain memory locations, and function call outputs) as tainted, **TEMU** keeps track of the tainted information propagating in the system.

Capabilities of DECAF: **DECAF** ensures precise tainting by maintaining bit-level precision for CPU registers and memory, and inlining precise tainting rules in the translated code blocks. Thus, the taint status for each CPU register and memory location is processed and updated synchronously during the code execution of the virtual machine.

3. [low] Why do generated binaries have a dependency on libdyninstaAPI_RT.so?

We were able to find only limited information about Dyninst's design, since it is not an open-source project. Based on the information provided in the Dyninst manual, Page 63, it appears that this library must be added to the environment before using the Dyninst API: https://github.com/dyninst/dyninst/blob/master/dyninstAPI/doc/dyninstAPI.pdf.

To locate the runtime library that Dyninst needs to load into your program, an additional environment variable must be set. The variable `DYNINSTAPI_RT_LIB` should be set to the full pathname of the run time instrumentation library, which should be:

```
NOTE: DYNINST_LIB should be set to the location where Dyninst library
files were installed

$(DYNINST_LIB)/libdyninstAPI_RT.so (UNIX)

%DYNINST_LIB/libdyninstAPI_RT.dll (Windows)
```