418 Final Report

Sungho Lee, Minwoo Oh

Overview

Fuzzing is an automated software testing technique that involves providing different types of inputs to a given program and monitoring the behavior of the program. The most prevalently used repository for fuzzing is AFL++, a fork of the original AFL (American Fuzzy Lop) developed by Michał Zalewski. In fuzzing, there are two main factors that determine how effective the fuzzing session is, namely **the number of program paths exposed in a given timeframe** and **the number of unique bugs exposed**. In order to leverage parallelism to perform better in both directions, several research groups from around the world have devised different methods of parallel fuzzing. In this report, we explore the effectiveness of the **seed-sharing version** of AFL++, with regard to 1/2/4/6 threads and conclude that parallelism improves both metrics and also leads to more stable sessions. We also talk briefly about the other types of implementations that are of interest.

Background

Fuzzing is a general term encompassing many different types of implementations under the definition listed above. We focused our direction towards the following: white-box (source code available), AFL++ (mutation, coverage based) default settings. These settings help us narrow the scope of the experimentation comparable to other forms of fuzzing such as black-box / grey-box fuzzing for white-box, PAFL / UniFuzz / EnFuzz for AFL++.

The benefit white-box testing brings compared to black-box / grey-box testing is apparent. If only the binary is available, the fuzzers (a program instance of fuzzing) can simply toss it an initial guess that the user provides, and mutate the input without knowledge of what kind of branches it is covering. All the fuzzers can do is hope that they get lucky and find a crash or hang. However, given the source code, we can do so much more - AFL++ provides a compile-time instrumentation so that between lines of assembly, markers called 'states' are inserted. These states are utilized by the fuzzer program during run-time to keep track of where the program is during its execution, enabling reasoning based on which states the program has reached so far. With the knowledge of coverage, the fuzzers have the potential of avoiding searching the same path over and over again.

Approach

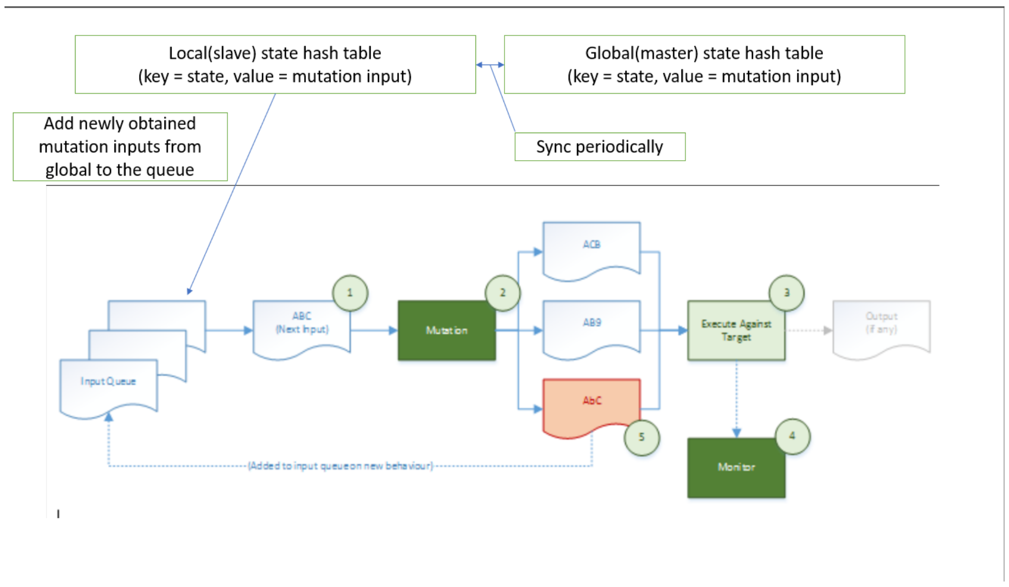

Local(slave) state hash table
(key = state, value = mutation input)
Global(master) state hash table
(key = state, value = mutation input)
Add newly obtained mutation inputs from global to the queue
Sync periodically
Input Queue
ABC (Next Input)
1
Mutation
2
ACB
AB9
AbC
5
Execute Against Target
3
Output (if any)
Monitor
4
(Added to input queue on new behaviour)

Diagram of one AFL++ fuzzing instance with seed-parallelism (Figure 1)

AFL++ follows the scheme below:

1. Load an initial user-supplied input to the input queue
2. Take the next input out of the input queue
3. Attempt to trim the test case to maintain the measured behavior of the program (1)
4. Mutate the input (1)
5. Run the input with the given program
6. If any of the mutated inputs generate a new state transition, then add the mutated input back into the input queue
7. **Only multi-thread : Periodically sync with main thread and add interesting inputs from main to the input queue** (1)
8. GOTO 2

(1) Methods deployed in the test to trim / mutate the input:

Calibration

A pre-fuzzing stage where the execution path is examined to detect anomalies, establish baseline execution speed, and so on. Executed very briefly whenever a new find is being made.

Trim L/S
Another pre-fuzzing stage where the test case is trimmed to the shortest form that still produces the same execution path. The length (L) and stepover (S) are chosen in general relationship to file size.

Havoc

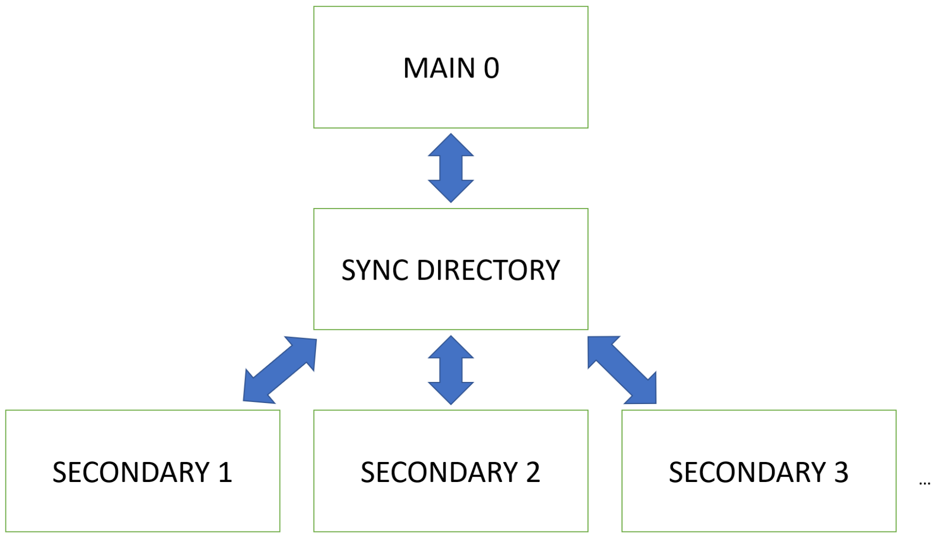
Fixed number of cycles with random tweaks from bit flipping, overwriting with random integers, deletion or duplication of some portion of the input

Splice

A last-resort strategy that kicks in after the first full queue cycle with no new paths. It is equivalent to ‘havoc’, except that it first splices together two random inputs from the queue at some arbitrarily selected midpoint.

Sync (Only for Parallel Fuzzing)

A stage used only when -M or -S is set. No real fuzzing is involved, but the tool scans the output from other fuzzers and imports test cases as necessary. The first time this is done, it may take several minutes or so.



Topology of Seed-Parallelism (figure 2)

The parallelism behind seed-sharing is quite simple. Each instance of the fuzzer is either classified as a "main" or a "secondary" instance. Each fuzzer has a hash table that keeps track of the states of the program that are covered. At some interval, the secondaries asynchronously add from the main's hash table unexplored states and the corresponding inputs to their own input queues. The main scans all secondaries and also updates its data structures as well in the same fashion. The topology of the system helps communication cost low because the number of connections is linear with respect to the number of secondaries. Also, because all communication is asynchronous and does not need the most recent updated hash table (some recent version will suffice), there is minimal stalling when the data structure is read from the secondary process and the main updates the same entry of the hash table. Only the main suffers from a lengthy synchronization, but that is minimal compared to the number of additional paths that all the secondaries receive from each other through main in the next sync step.

In terms of execution, the main and secondary continue to execute inputs they have in hand + new interesting inputs from other fuzzers, without knowledge of what the other processes are currently inspecting. The advantage of the parallelism and the speedup of the total operation here is quite difficult to quantify in the head due to the non-deterministic nature of fuzzing and the irregular nature of the synchronization, so a statistical analysis is suitable here. We expected the seed-sharing version to perform better early on, since the parallelism offloads the potential work that would be redone by the different instances. Also, other processes could lift processes that cannot find new states by their own mutation for some prolonged time out of the gutter, and the probability of all processes stuck in their own local states is much lower than one process being stuck. We expected that this safety net would create more stability for the session with increasing cores. (Fuzzers would be killed less often for not being able to find any more states)

Testing & Specification

Hardware & Software Specifications:

Ryzen 3600x CPU - 6 cores, 12 threads

VMware running on Windows 10 settings:

    Ubuntu 20.04.03 LTS
    4GB RAM
    CPU 6 cores, 2 threads each (gave all cores to the VM)

Test Parameters:

Test Program	: Readelf, compiler environment variable "CC=afl-gcc ./configure", binutils-2.25
User Input	: /bin/ps
Structure	: One main, Multiple secondary

Test Method:

Threads tested - 1, 2, 4, 6 threads
For each thread count, we tested 4 instances that reached 8 hours and averaged the statistics.

The following statistics were compiled:

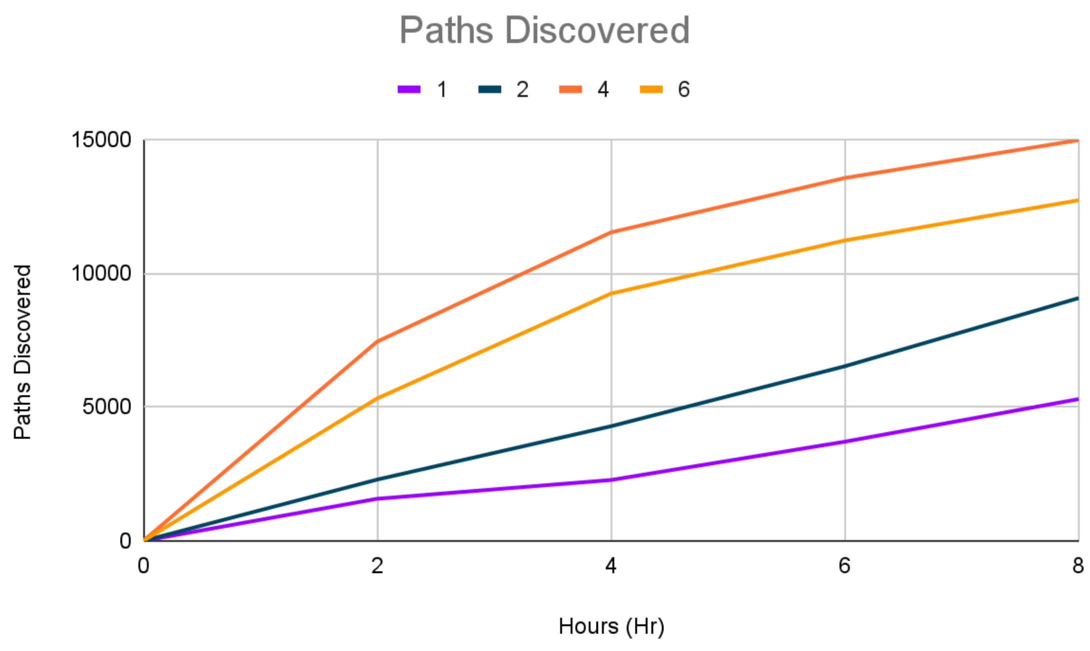
a. Paths found at 2 hour increments
b. Maximum Gap between Number of Paths Found
c. Bugs (crashes, hangs, etc.) found

a and c show the overall effectiveness of the fuzzers, so it is used as a good base metric for any fuzzer. b. shows anomalies between runs

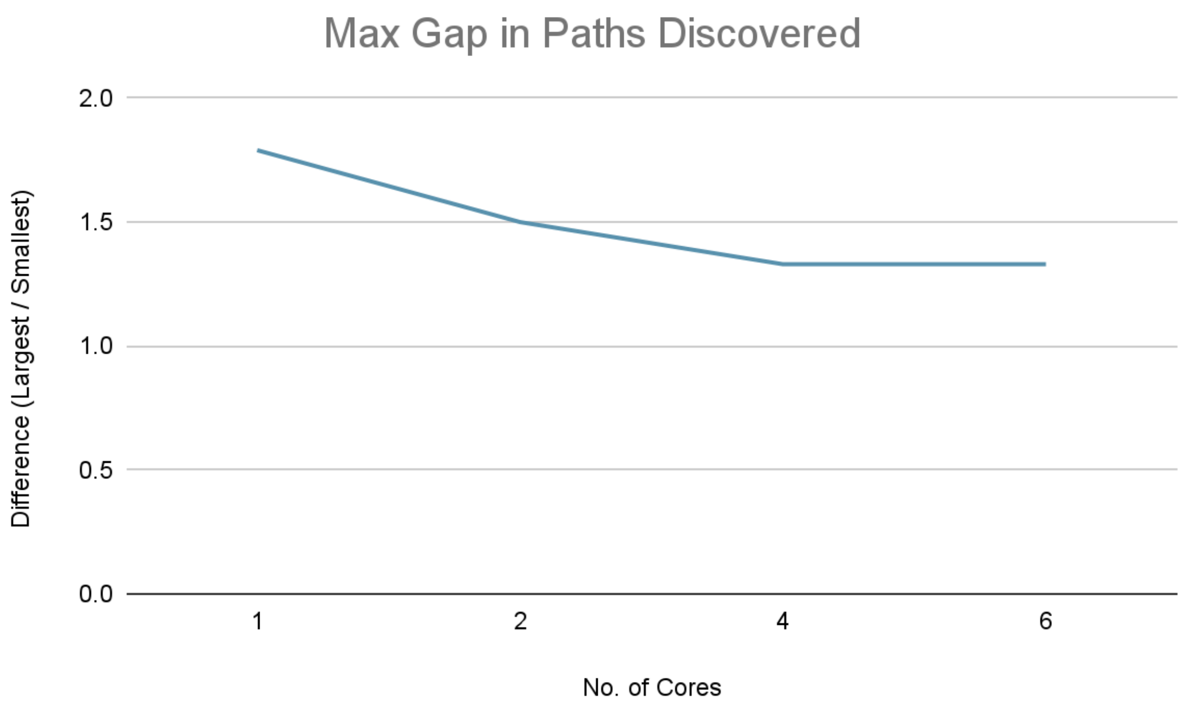
We expected that more cores would generally perform better at every metric, since sharing test inputs allows for individual processes to search for non-overlapping paths. We also expected the number of unique bugs found to be larger as the search space was widened. Furthermore, we expected that the maximum gap would be smaller due to sharing of paths. Although we were pessimistic about reaching linear improvement, we hoped that there would be some better-than-constant improvement.

On the side, we looked out for interesting behaviors regarding multiple fuzzers. Fuzzers are killed when they are stuck in a local minima and hanging. We expected that with more cores, fuzzers would be killed less from multiple fuzzers adding different inputs to each other's queue.

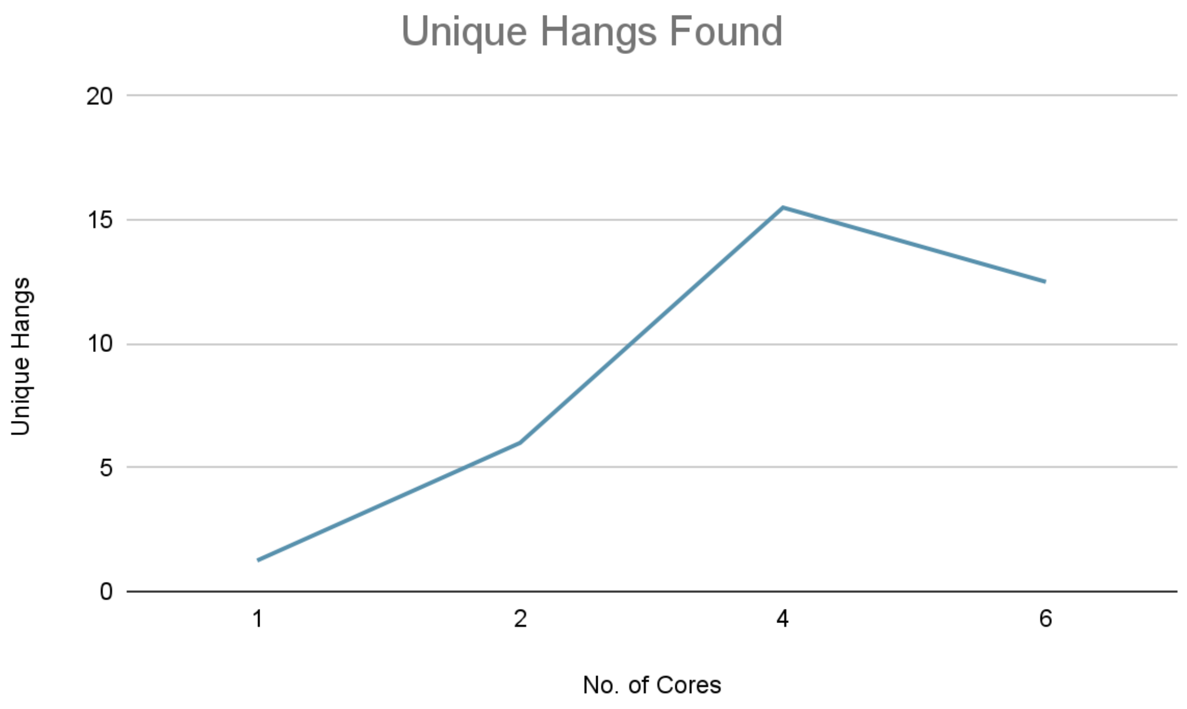
Results & Discussion



Average Paths Discovered Over a Fixed Period (figure 3)



Largest Gap of Paths Discovered Among All Instances of a given No. of Cores (figure 4)



Average Number of Unique Hangs Found (figure 5)

Just like how we expected, there was an increase in performance from 1 core to 2, 4, 6 cores in all four time periods. All three multi-core sessions experienced the same trend of performance boost; in the first 0 ~ 2 to 2 ~ 4 hour timeframes, all multi-core sessions successfully widened their gap with the single-core session, hitting the peak at around the 4 hour mark, hitting around linear speedup (1.88x / 5.07x / 4.06x). Thereafter, in the 4 ~ 6 hour timeframe the difference slowly started narrowing and in the 6 ~ 8 hour timeframe narrowed even more (1.71x / 2.82x / 2.40x). There was an anomaly of the 4-core performance being better than the 6-core performance. We speculate that this is due to resource allocation problems with the virtual machine for higher core counts.

Also, we observed that the variety of the number of paths found between different fuzzing sessions with an increasing number of cores decreased. This was expected because with sharing interesting inputs, the fuzzing session was less likely to fall into a local minima, causing abnormally low paths found and causing large varieties between individual sessions.

Lastly, we observed that with an increasing number of cores, the fuzzing session found more unique hangs. This behavior was also expected due to the search space of the fuzzing session widening from sharing inputs that could potentially be recalculated. We were surprised to see a super-linear performance boost (4.8x / 12.4x / 10x) with additional cores.

Conclusion

Expected performance boost regarding number of paths found was less than ideal because different instances of fuzzers do end up fuzzing the same inputs, and there is no deterministic strategy that given the same inputs different output spaces will be scanned. This strategy of parallel passing around transitions and inputs is synonymous to lab4, where the current state of the board was passed around once in a while to make more localized wire changes, but the algorithm itself didn't suggest a clear, deterministic improvement.

In terms of non-algorithmic bottlenecks, we did not think I/O was a huge issue because the sharing of transitions and inputs were done asynchronously, the shared resource was not being modified at the time of transition, and to reduce synchronization time only the main synced with everyone and spread the results out to each of the secondary fuzzers.

With regards to the variety of numbers of paths discovered, it was relieving to see that more cores showed smaller differences, since this meant that there exists a normal range of numbers of paths discovered for some given parameter. This fact enables researchers to set an acceptable range for a normal fuzzing session in the future for their own input parameters, only taking fuzz results from normal sessions to have confidence in their findings.

It was difficult to determine whether the outstanding result of unique hangs found was a fluke. This was due to the non-deterministic nature of fuzzing and insufficient time with testing which could have shown skewed results. While we were not able to find a fitting pattern for this result nor find an explanation for such weird behavior, it is possible that additional research on this topic could reveal more interesting ideas for practical bug-fixing.

We came to the conclusion that with the current scheme of parallelization, while simple to implement and test, it is difficult to make any more improvements to achieve more deterministic, fine-grain parallelization of fuzzing. Therefore, we searched online for different paradigms of parallelization that we could learn from. We discuss some of the new ideas proposed by other researchers in the “Future Ideas & Other Implementations” section to overcome the limitations imposed by AFL.

Problems

We initially proposed to implement a version of parallel fuzzing called PAFL, a paper by a Chinese research team whose code for their remarkable results were not available online. The end goal was to figure out what they claimed was indeed true and test it on other programs as well. However, it turned out to be that the AFL++ codebase was too difficult to modify as someone who had not been working with it previously, and the PAFL description was a bit vague and so it was difficult to define the data structures used for the exact implementation. Therefore we had to scrap that approach.

Also, because the processors used have a 6-core processor with 12-threads, the computer would also freeze at some points when more than 8 fuzzing instances were running because the fuzzer scheduler would prioritize unused cores first and then move on to doubly schedule a core. This would make no cores available for user input interrupts and the VM would freeze or when the click was registered, the fuzzing session, thinking that is hanging when in reality it just wasn't getting time from the CPU, would kill itself. Due to this hardware limitation, we were able to test up to only 6 cores. It would have been nice to use a processor that had more than 8 true cores instead of a 6 core one.

Lastly, we faced problems regarding fuzzing itself, where the fuzzer would give up testing when it identified itself as hanging, and therefore unable to find new instances, and kill itself. This behavior was more commonly seen in single-threaded instances, where it could not expand its search space in larger increments. We could not find a workaround on this matter, so we just tested more times to have the fuzzing sessions reach the 8 hour point specified.

Future Ideas & Other Implementations

There aren't a whole lot of people working on fuzzing, but some of the notable ones we found were PAFL, UniFuzz, and 256 parallel fuzzing. PAFL proposes a different kind of scheme for fuzzing where in addition to synchronizing the seeds, the main node distributes the seeds to the secondary nodes based on their trace. Specifically, currently covered state space is divided and assigned to secondary fuzzers instances, and those fuzzer instances only fuzz seeds that cover assigned states. Furthermore, it confines each node to a state space to some offset to limit the overlap of state space between fuzzer instances. Therefore, this approach improves upon the current parallel fuzzing by enhancing the load distribution between cores. UniFuzz (Zhou et al.) suggested a dynamic centralized input queue implementation of fuzzing where the paper claimed to achieve near linear speedup, but we were not able to research further in time. 256 parallel fuzzing contributed to the overall picture because it showed us that 256 parallel instances of fuzzing on a single machine is possible without too much trouble on the system and so it blazed the trail for future improvements to the project.

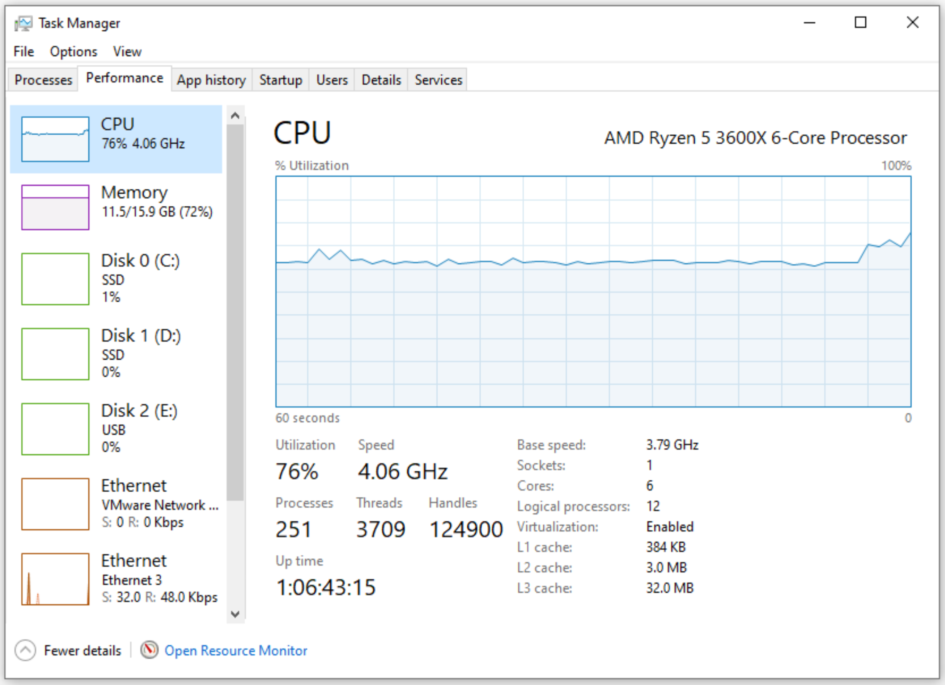
That being said, fuzzing *is* an active area of research within the security field, and there exist several papers with different takes on fuzzing. As a consequence, there is no centralized way of benchmarking, which UniFuzz (Li et al.) attempts to solve via creating a platform to evaluate different fuzzers. Meanwhile, Enfuzz focuses on the fact that many fuzzers suffer from lack of robustness due to different implementations, and suggests a way to run different fuzzers simultaneously with a global state and seed pool to take advantage of the different strengths of different fuzzers.

If we are to expand this project, one direction would be to test the fuzzer to see if it performs well parallely on certain types of programs (where the input is expected to be of some type) or just well in general. This direction would give us more confidence on the results we procured in this paper. Another direction would be to expand the parallelism to more than the number of cores tested, to see if the added parallelism scales. Lastly, it would be good if the PAFL or UniFuzz implementation is available so that there would be more directions of comparison.

References

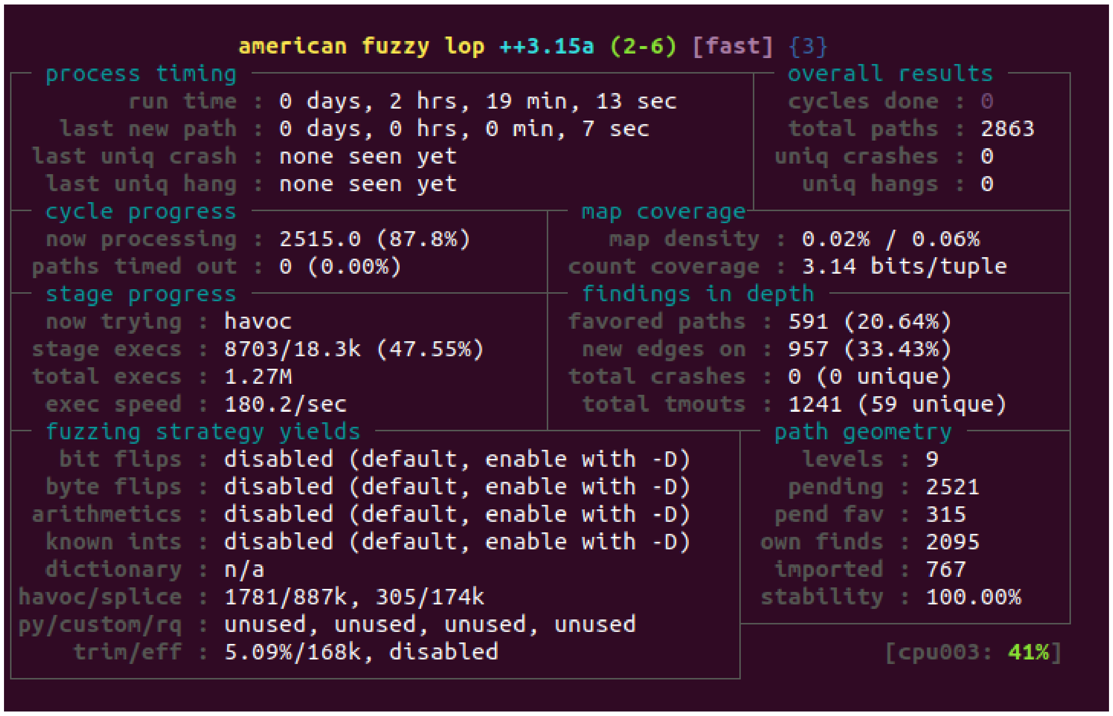
- Source Code & General Knowledge
    - https://www.evilsocket.net/2015/04/30/Fuzzing-with-AFL-Fuzz-a-Practical-Example-AFL-vs-binutils/
    - https://github.com/google/AFL
    - https://github.com/AFLplusplus/AFLplusplus
- Other Implementations
    - https://gamozolabs.github.io/fuzzing/2018/09/16/scaling_afl.html
    - J. Ye, B. Zhang, R. Li, C. Feng and C. Tang, "Program State Sensitive Parallel Fuzzing for Real World Software," in *IEEE Access*, vol. 7, pp. 42557-42564, 2019, doi: 10.1109/ACCESS.2019.2905744. https://ieeexplore.ieee.org/document/8668503
    - Chen, Yuanliang, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao and Zhuo Su. "EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers." USENIX Security Symposium (2019). https://www.semanticscholar.org/paper/EnFuzz%3A-Ensemble-Fuzzing-with-Seed-Synchronization-Chen-Jiang/8754951ba8bbb42ff10e100fa853a5ce86af5ab1
    - Yifan Wang, Yuchen Zhang, Chenbin Pang, Peng Li, Nikolaos Triandopoulos and Jun Xu. "Facilitating Parallel Fuzzing with Mutually-exclusive Task Distribution." 2021. https://arxiv.org/pdf/2109.08635.pdf
    - Xu Zhou, Penfei Wang, Chenyifan Liu, Tai Yue, Yingying Liu, Congxi Song, Kai Lu and Qidi Yin. "UniFuzz: Optimizing Distributed Fuzzing via Dynamic Centralized Task Scheduling." 2020. https://arxiv.org/pdf/2009.06124.pdf
- Pictures & Diagrams
    - https://blog.nettitude.com/uk/fuzzing-with-american-fuzzy-lop-afl
- On Fuzzing
    - Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8-13, 2020, Virtual Event, USA. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409729
    - Yuwei Li et al. "UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers." 2020. https://arxiv.org/abs/2010.01785

Appendix


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
Open Resource Monitor

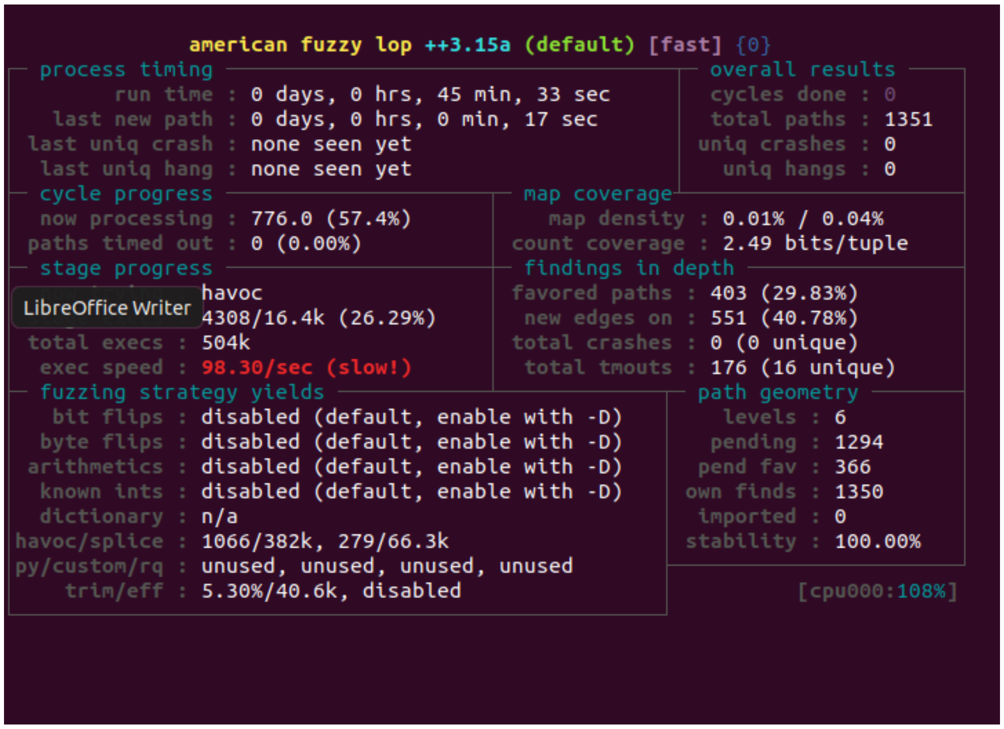
```
CPU                                   AMD Ryzen 5 3600X 6-Core Processor
% Utilization                                                     100%
60 seconds                                                           0

CPU        76% 4.06 GHz
Memory     11.5/15.9 GB (72%)
Disk 0 (C:) SSD 1%
Disk 1 (D:) SSD 0%
Disk 2 (E:) USB 0%
Ethernet   VMware Network ... S: 0 R: 0 Kbps
Ethernet   Ethernet 3 S: 32.0 R: 48.0 Kbps

Utilization  Speed            Base speed:          3.79 GHz
76%          4.06 GHz         Sockets:             1
                              Cores:               6
Processes  Threads  Handles   Logical processors:  12
251        3709     124900    Virtualization:      Enabled
                              L1 cache:            384 KB
Up time                       L2 cache:            3.0 MB
1:06:43:15                    L3 cache:            32.0 MB
```

CPU usage as seen from the host device



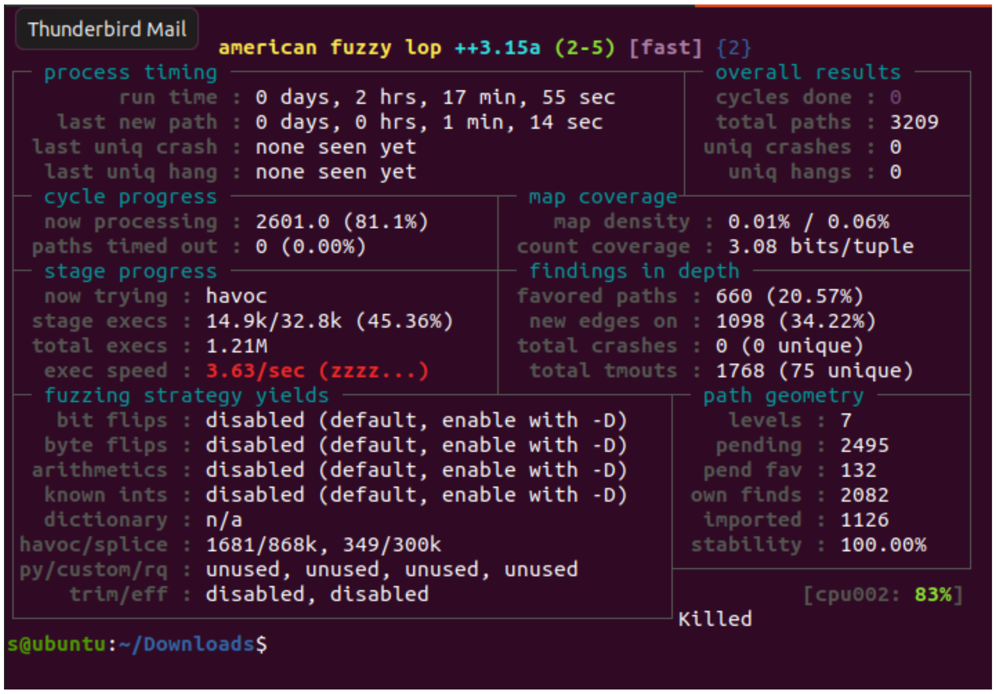
```
              american fuzzy lop ++3.15a (2-6) [fast] {3}
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 2 hrs, 19 min, 13 sec      │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 7 sec        │  total paths : 2863  │
│ last uniq crash : none seen yet                      │ uniq crashes : 0     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ─────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 2515.0 (87.8%)  │    map density : 0.02% / 0.06%        │
│ paths timed out : 0 (0.00%)       │ count coverage : 3.14 bits/tuple      │
├─ stage progress ─────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc               │ favored paths : 591 (20.64%)          │
│ stage execs : 8703/18.3k (47.55%) │  new edges on : 957 (33.43%)          │
│ total execs : 1.27M               │ total crashes : 0 (0 unique)          │
│  exec speed : 180.2/sec           │  total tmouts : 1241 (59 unique)      │
├─ fuzzing strategy yields ────────┴─────────────┬─ path geometry ────────┤
│   bit flips : disabled (default, enable with -D) │    levels : 9          │
│  byte flips : disabled (default, enable with -D) │   pending : 2521       │
│ arithmetics : disabled (default, enable with -D) │  pend fav : 315        │
│  known ints : disabled (default, enable with -D) │ own finds : 2095       │
│  dictionary : n/a                                │  imported : 767        │
│havoc/splice : 1781/887k, 305/174k                │ stability : 100.00%    │
│py/custom/rq : unused, unused, unused, unused     ├────────────────────────┘
│    trim/eff : 5.09%/168k, disabled               │          [cpu003: 41%]
└──────────────────────────────────────────────────┘
```

Multi-thread execution - bottom right imported paths



```
            american fuzzy lop ++3.15a (default) [fast] {0}
─ process timing ──────────────────────────────────────┬─ overall results ─┐
       run time : 0 days, 0 hrs, 45 min, 33 sec        │  cycles done : 0  │
  last new path : 0 days, 0 hrs, 0 min, 17 sec         │ total paths : 1351│
last uniq crash : none seen yet                        │uniq crashes : 0   │
 last uniq hang : none seen yet                        │  uniq hangs : 0   │
─ cycle progress ─────────────────┬─ map coverage ─────┴───────────────────┤
 now processing : 776.0 (57.4%)   │    map density : 0.01% / 0.04%         │
paths timed out : 0 (0.00%)       │ count coverage : 2.49 bits/tuple       │
─ stage progress ─────────────────┼─ findings in depth ────────────────────┤
LibreOffice Writer havoc          │ favored paths : 403 (29.83%)           │
             4308/16.4k (26.29%)  │  new edges on : 551 (40.78%)           │
total execs : 504k                │ total crashes : 0 (0 unique)           │
 exec speed : 98.30/sec (slow!)   │  total tmouts : 176 (16 unique)        │
─ fuzzing strategy yields ────────┴─────────────┬─ path geometry ─────────┤
  bit flips : disabled (default, enable with -D) │    levels : 6          │
 byte flips : disabled (default, enable with -D) │   pending : 1294       │
arithmetics : disabled (default, enable with -D) │  pend fav : 366        │
 known ints : disabled (default, enable with -D) │ own finds : 1350       │
 dictionary : n/a                                │  imported : 0          │
havoc/splice : 1066/382k, 279/66.3k              │ stability : 100.00%    │
py/custom/rq : unused, unused, unused, unused    ├────────────────────────┘
    trim/eff : 5.30%/40.6k, disabled             │         [cpu000:108%]
─────────────────────────────────────────────────┘
```

Single-thread execution - bottom right imported paths



```
Thunderbird Mail
             american fuzzy lop ++3.15a (2-5) [fast] {2}
─ process timing ──────────────────────────────────────┬─ overall results ─┐
       run time : 0 days, 2 hrs, 17 min, 55 sec        │  cycles done : 0  │
  last new path : 0 days, 0 hrs, 1 min, 14 sec         │ total paths : 3209│
last uniq crash : none seen yet                        │uniq crashes : 0   │
 last uniq hang : none seen yet                        │  uniq hangs : 0   │
─ cycle progress ─────────────────┬─ map coverage ─────┴───────────────────┤
 now processing : 2601.0 (81.1%)  │    map density : 0.01% / 0.06%         │
paths timed out : 0 (0.00%)       │ count coverage : 3.08 bits/tuple       │
─ stage progress ─────────────────┼─ findings in depth ────────────────────┤
 now trying : havoc               │ favored paths : 660 (20.57%)           │
stage execs : 14.9k/32.8k (45.36%)│  new edges on : 1098 (34.22%)          │
total execs : 1.21M               │ total crashes : 0 (0 unique)           │
 exec speed : 3.63/sec (zzzz...)  │  total tmouts : 1768 (75 unique)       │
─ fuzzing strategy yields ────────┴─────────────┬─ path geometry ─────────┤
  bit flips : disabled (default, enable with -D) │    levels : 7          │
 byte flips : disabled (default, enable with -D) │   pending : 2495       │
arithmetics : disabled (default, enable with -D) │  pend fav : 132        │
 known ints : disabled (default, enable with -D) │ own finds : 2082       │
 dictionary : n/a                                │  imported : 1126       │
havoc/splice : 1681/868k, 349/300k               │ stability : 100.00%    │
py/custom/rq : unused, unused, unused, unused    ├────────────────────────┘
    trim/eff : disabled, disabled                │         [cpu002: 83%]
─────────────────────────────────────────────────┘Killed
s@ubuntu:~/Downloads$
```

Dead fuzzer - killed due to not being able to find a new path for a long time