AFL Fuzzing Improvement Project

Anthony Triolo, Daniel Garry, Jared Chester, Diane Tran
Rutgers University - CS 419

Git Repository: https://github.com/jmc07712/CS419-Project
Video Presentation: https://youtu.be/GivCb8GDI0Q

1.0 : FUZZING ABSTRACT

Fuzzing is one of the most widely used software vulnerability discovery techniques employed today. Fuzzing has remained highly popular due to its important advantage in that the testing is extremely simple, and free of preconceptions about system behavior and generally has a low barrier of deployment.Fuzz testing at a high level is simply an automated software testing technique that attempts to find software vulnerabilities by feeding malformed and semi-malformed inputs,"fuzz inputs", into a program; typically these inputs may be processed incorrectly and trigger some sort of unintentional behavior. Fuzz testing is achieved via a fuzzer, a program that performs the fuzz testing. There are a wide range of fuzzers available to the general public, however they can all be categorized into two main groups based on the granularity of semantics they observe in each fuzz run. These two groups are called black- and white-box fuzzers.

"black-box" fuzzer refers to those techniques that do not see the internals of the target program those that simply observe the input/output behavior of the program being tested. Testing of this nature is IO-driven or data-driven. Some also take the structural information about inputs into account to generate more meaningful test cases while maintaining the characteristic of not inspecting the code. Most fuzzers fall into this category. "white-box" fuzzers on the other hand generate test cases by analyzing the internals of the target program along with information gathered when executing. The Fuzzer we improved in this project, fits into neither of these categories.

2.0 : AFL FUZZING

AFL, American Fuzzy Lop, the Fuzzer we improved upon in our project is a mutation-based fuzzer, it generates new inputs by deterministically and non-deterministically modifying an input file, and by combining inputs to form entirely new ones. It is a greybox fuzzer which leverages coverage feedback to better test the target program, using lightweight program analysis to gather branch coverage for an input.

2.1 : Design Goals of AFL

AFL was built with the following goals in mind: speed, reliability and simplicity. AFL fuzzes achieves its speed goals by fuzzing targets at their native speeds, on top of this It uses instrumentation to ensure the amount of work done is kept to a minimum. Reliability is provided by avoiding automated testing strategies that are inherently unreliable, for example, symbolic execution. At its core, AFL is a traditional fuzzer which implements a variety of well-researched strategies when their use is needed most, this goes hand-in-hand with simplicity. AFL was designed to avoid the complexity present in most fuzzers . As such the only parts of the fuzzer that are configurable are the output file, the memory limit, and the ability to override the default, auto-calibrated timeout. On the off chance that it fails, AFL provides error messages that inform the user of what went wrong and why.

2.2 : Instrumentation

AFL instruments programs by injecting code following every conditional jump. Here each jump is assigned a unique identifier and counter, allowing the fuzzer to know for each input which branches are being hit and how many times. This form of coverage helps trace execution paths and provides considerably more insight into the execution path of the program than simple block coverage. This aids the discovery of fault conditions in the target code

2.3 : Input queue

AFL adds those test cases which produce new behavior into the input queue to be used in future rounds of fuzzing. To improve performance, AFL will occasionally select a smaller subset of these queue entries and designate them as “favorites”. This is done by assigning them a score equivalent to their execution time and file size. Those with the lowest scores are deemed as “favorites”. Favorites will be fuzzed statistically more often than their non-favorite counterparts. If there exists new, unfuzzed favorites in the queue, those who are not favored will be skipped 99% of the time. When there are no new favorites, if the current queue entry was previously fuzzed it will be skipped 95% of the time, if it has not undergone any fuzzing, it will be skipped 75% of the time. This approach to scheduling provides a good balance between test case diversity and queue cycling.

2.4 : Fuzzing Strategies

AFL employs deterministic fuzzing strategies early on and later progresses to purely random modifications. AFL begins with deterministically modifying the input files as they are more likely to produce more compact, simpler test cases. These strategies include:

Walking bit flips: in this stage, AFL performs sequential, ordered bit flips. The number of bits flipped per row can vary from one to four, however the stepover is always one. On average, Flipping a single bit will generate 70 new paths per one million inputs, flipping two bits in a row will generate 20 additional paths per million inputs and flipping four bits in a row will lead to an additional 10 paths per million inputs.

Walking byte flips: similar to in the walking bit flip stage, here AFL performs 8,16 or 32 bit wide bit flips with a step over of one byte generating on average 30 additional paths per million input.

Addition and subtraction: here AFL attempts to trigger more complex conditions by subtly incrementing or decrementing existing integer values in the input file with a stepover of one byte. In this stage, AFL will perform 3 separate operations: addition and subtraction on individual bytes, looking at 16 bit values in both big and little endian and incrementing or decrementing depending on if the operation affects the most significant byte, and the third operation does the same as the second but with 32 bit integers.

After AFL has performed all the deterministic operations for a given input, it will move on to the purely random, non-deterministic stages:

Random stack modifications(HAVOC): an infinite loop of random operations that consist of stacked sequences of single bit flips, single byte sets, addition or subtraction of integers and bytes, block deletion, block duplication, block memset. This stage typically leads to more paths discovered than by all the previous deterministic stages.

Test case splicing: here AFL will take two different input files from the queue and splice them together at a random location and subsequently run it through the aforementioned HAVOC stage. This will generate on average 20 percent more execution paths than HAVOC.

3.0 : MODIFICATIONS TO AFL FUZZING

We put a considerable amount of time into researching pre-existing improvements made to AFL, namely through AFLfast. We found that most changes to AFL in AFLfast come from the amount of fuzz that is generated, the energy, for a seed and the order in which AFL chooses the inputs from the queue and how it designates “favorite” seeds that are chosen from the queue.

In traditional AFL, the amount of fuzzing for an input is computed depending on the execution time, transition coverage and creation time of the seed. Essentially, if a seed executes faster, covers more, and is generated later, then the amount of fuzz, that is the energy, is greater. AFL fast changed the computation of the amount of fuzz that is generated for a seed. It assigns low energy to seeds exercising high-frequency paths and high energy to seeds exercising low-frequency paths

Next, AFLfast changed the order in which AFL chooses the inputs from the queue and how it designates which seeds are favorites. Originally AFL chooses favorites as those seeds which are the fastest and smallest seeds for any of the control-flow edges it exercises. AFLfast chooses the seed exercising a control flow who has been chosen the least amount of times from the queue, and if there are several, then the seed which exercises a path exercised by the least amount of fuzz, and if there are still several, then the fastest and smallest seed. Secondly, AFL chooses the next favorite input which follows the current input in the queue. AFLfast chooses the next favorite seed as the one which has been chosen from the queue the least amount of times and if there are several, it chooses that which exercises a path exercised by the least amount of fuzz.

4.0 : OUR MODIFICATIONS

We had initially hoped to make modifications to the AFL source code to enact some meaningful change in the fuzzers performance. When considering the amount of research and subsequent testing that needed to be done, this ultimately proved to be a task too involved for the timeline given to us. We instead came to the conclusion that modifying certain elements of the operating system and taking advantage of some of AFL’s properties would give us the results we desired in a manner that was both time efficient and made sense given the goals of this project.

The nature of how AFL is built lends itself well to the use of a Virtual Disk. Thus the first route we took to improve the efficiency of AFL was using a Virtual Disk instead of the traditional SSD or HDD. Not only does the use of a RAM disk improve efficiency of read and writes, but it spares our drives from the wear and tear of constantly being written to and read from which is

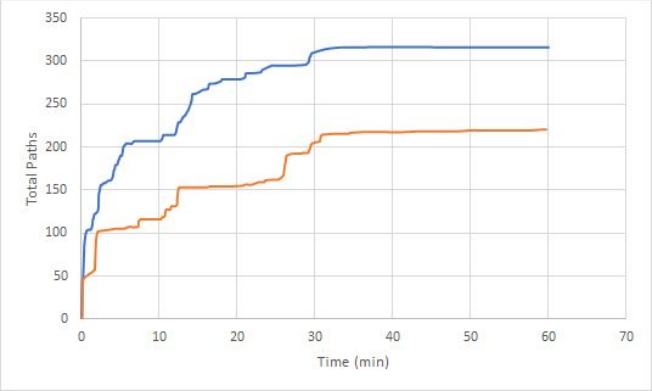
an unfortunate necessity of AFL fuzzing. This also has the added benefit of drastically improving the amount of paths discovered and the amount of test cases tested.

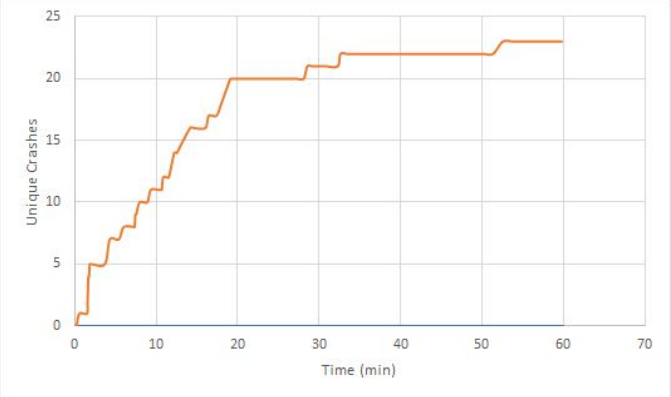
Another Avenue we pursued was the use of Dictionaries. One of AFL's features is that it supports using a dictionary of values when fuzzing. This is basically just a set of tokens that it can use when mutating a file instead of picking values at random. When AFL is given a dictionary, it is provided with more ways to mutate seed files using language from the program binary and thus increase the chance of finding a crash in the test program as well aid the fuzzer in determining the seed inputs that will create the most number of paths thereby increasing code coverage.

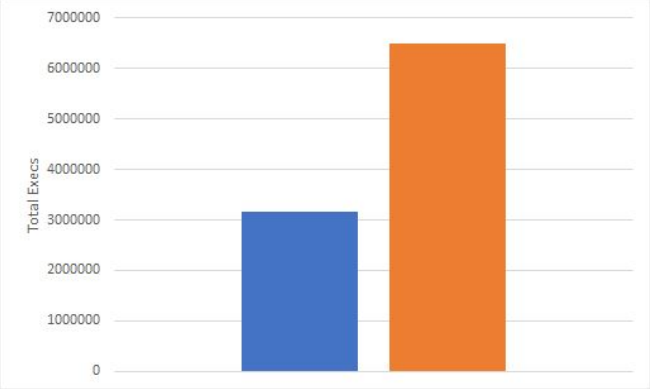
We used two suites in the evaluation of the aforementioned modifications, LAVA-M and the Google Fuzzer Suite. Due to the constraints of the project mentioned before we chose to test the efficacy of the changes made on two programs: json and Ssl1.1.0c - crl program. Both were tested twice, once without modifications as a control and once with to measure the effect of the changes, for an hour each. The LAVA-M suite was tested on an Ubuntu 18.0.4 VM running a 4 core Intel i7-4790k processor and 4 GB of DDR3 RAM

5.0 : RESULTS

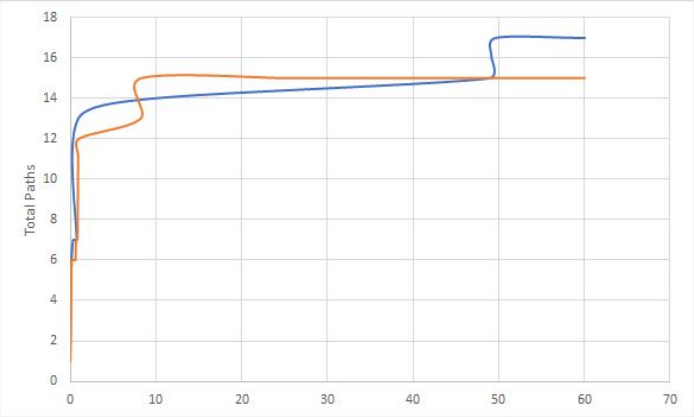
5.1 : base64 - Blue is normal, orange is ramdisk and dictionary

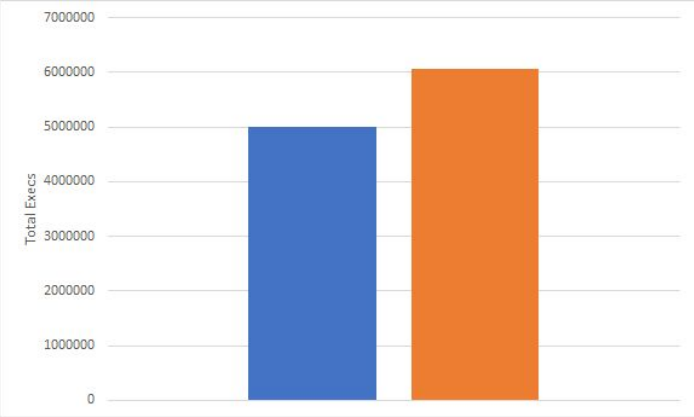




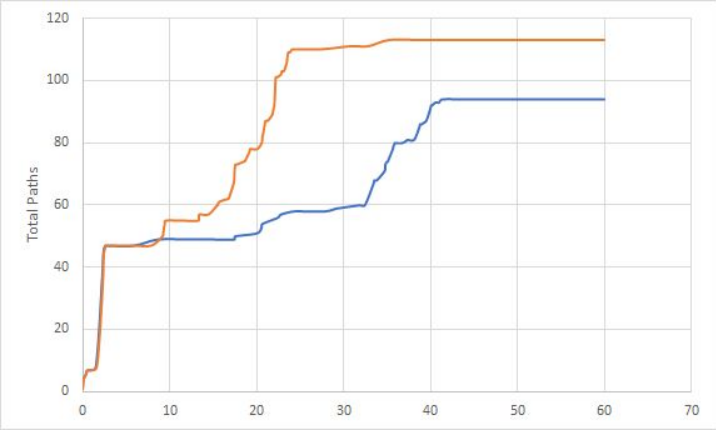


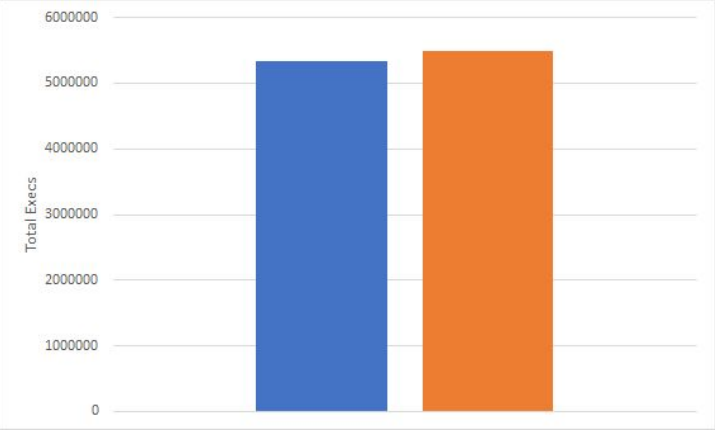
5.2 : md5sum - Blue is normal, orange is ramdisk



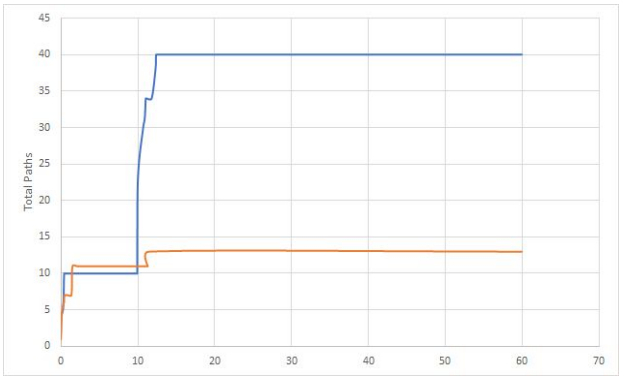


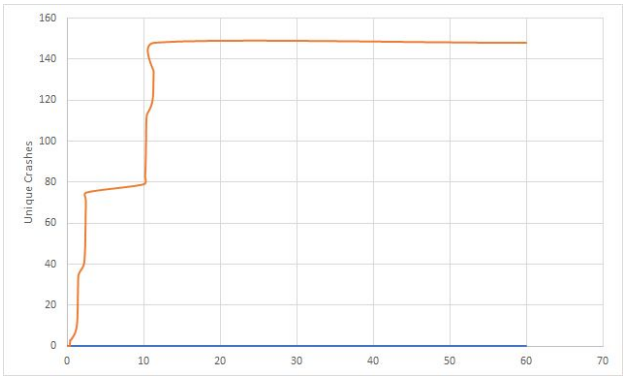
5.3 : Uniq - Blue is normal, orange is ramdisk

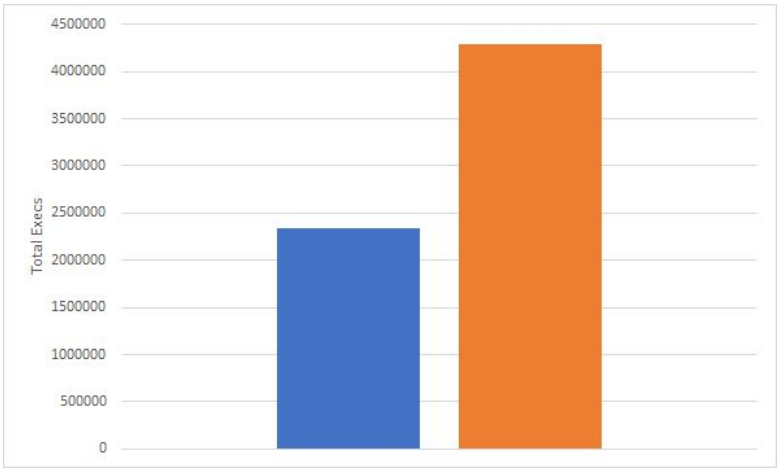




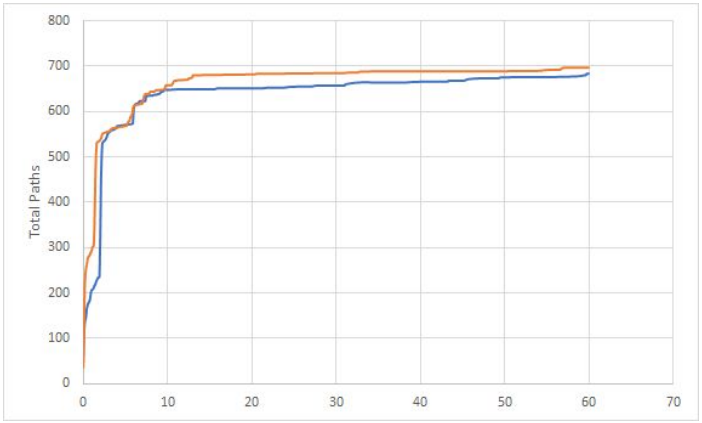
5.4 : Who - Blue is normal, orange is ramdisk and dictionary

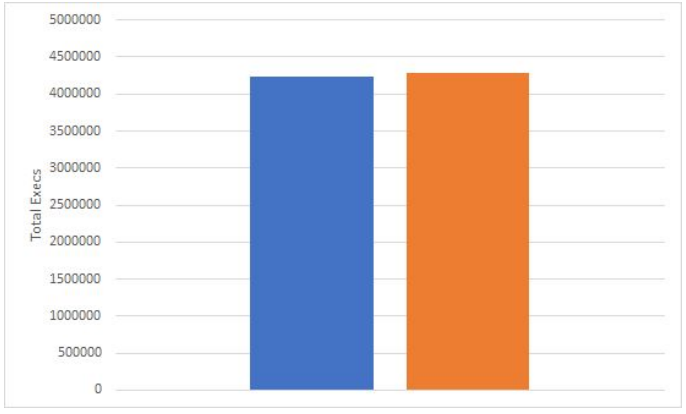




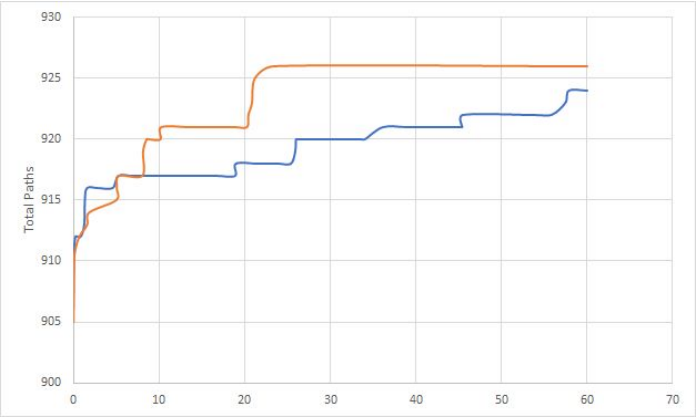


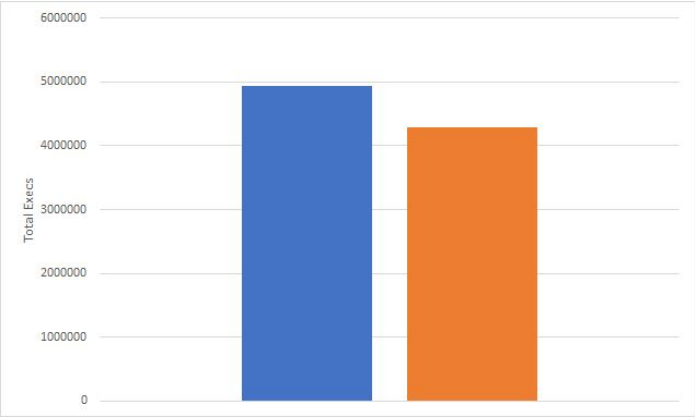
5.5 Json - Blue is normal, orange is ramdisk and dictionary





5.6 : ssl-crl - blue is normal, orange is ramdisk and dict





6.0 : EXPLANATION OF RESULTS

Before explaining the results seen in the graphs, it is first important to establish the variables we looked at to gauge whether or not AFL was improved upon. The first and most obvious one is total crashes. The entire point of AFL is to try to find bugs within a program's code that cause fatal crashes, so that when the program is released, these bugs are already fixed. If we were able to increase the number of crashes found by using our modifications, then this would be a significant improvement to AFL. Another variable we looked at was total paths found. A "path" in AFL is a new test case that is discovered during fuzzing that AFL considers "interesting" enough to want to run the full fuzzing process on. If more paths are found, then the probability that a test case is found that causes a crash increases. Lastly, we looked at total executions, which is the number of test cases run against the test program. By testing more cases, we get a larger coverage of tests, which once again increases the likelihood of finding a test that causes a crash.

The first test program we ran was the "base64" program within the LAVA-M test suite. We ran this test both with the normal configuration, as well as on a virtual disk. With the virtual disk configuration, we also provided the program with a dictionary made up of the object dump, as well as all strings, that were in the program binary. If we look at the first graph, which shows the number of total paths over time for both configurations, we see that the number of total paths actually decreases in the new configuration. One reason for this is that for each path, a new section of modifications is done using the provided dictionary. This means that we are spending more time on each individual path, so we get through fewer overall paths, which can lead to fewer "interesting" paths being discovered. This is not such a big deal, since we are providing the fuzzer with more meaningful operations, and therefore creating more meaningful tests, since the dictionary provides the fuzzer with the actual language of the program. By doing

this, we do see an increase in total executions, which means that the total number of tests run against the test program has increased, since we are overall generating more test cases. The biggest improvement seen is in the total number of crashes. In the hour that we ran the program, the original configuration of AFL found no crashes at all. With our improved configuration, it found 23 unique crashes. This is an extremely important improvement, since it means that by providing a dictionary, we were able to create more meaningful test cases that ended up resulting in crashes. This is due to the fact that the fuzzer does not have to rely on a well created seed to generate meaningful cases, but is provided additional information and options for creating test cases that are more closely related to the program binary.

When doing testing on the “md5sum” program, we only used a virtual disk, as we noticed the performance when using a dictionary was not advantageous. This is due to the way that AFL fuzzes when using dictionaries. The number of executions done using the dictionary is based on the length of the number of dictionary entries and the length of the seed. The dictionary for this program was much longer, and the seed files were long as well. So while we provided the fuzzer with more ways to modify the seed, it took more time to run through one path. So, the fuzzer ended up spending most of its time on the original seed file, thus preventing it from attempting these modifications on more interesting paths. Had we been able to run this longer, we may have seen improvements either to the total paths or number of crashes. We ended up seeing that the number of total paths found decreased slightly. This could be luck, since different cases are fuzzed each time, so the original configuration may have just fuzzed a better path earlier on. If we had run the program for longer, we would most likely see the two become even, if not see an improvement using the ramdisk. We did, however, see an improvement in the total number of executions. Because the read and write speeds of RAM are much faster, when using the virtual disk, the fuzzer spends less time reading and writing to/from files and more time executing the newly generated tests against the program.

For the same reason as with md5sum, when running the “uniq” program, we only used a virtual disk and did not use a dictionary. With our configuration, we saw an increase in both total paths and total executions. Once again, by increasing the number of total paths, we increase the likelihood of finding a path that generates an input that will cause a crash. While a crash was not found in the short term, it may be found in the long term. The increase in total executions shows how the virtual disk increases the overall speed of the fuzzer, allowing it to test more generated cases over time.

When running the “who” program, we saw a decrease in total paths, but a huge increase in both crashes and total executions just like with “base64”. Since we once again used a dictionary, the fuzzer spent more time on each path by modifying the original path file using the provided dictionary files, and because of this, explored fewer paths overall which leads to fewer paths found. However, the most important metric, total crashes, saw an increase from zero to 144. Once again we see that by providing the fuzzer with files that use the actual language of the program binary, thus leading the fuzzer to produce more meaningful and less random test cases, we find cases that actually cause the program to crash. Also, we see again that the

addition of the virtual disk allows the program to run faster and execute more test cases against the program.

For both of the google fuzzer suite programs, we saw a clear increase in total paths found. However, we did not find any crashes in either run of the fuzzer. This is most likely due to the fact that these are real world programs that do not have many known bugs to begin with. These are also much more complicated programs, and so a dictionary of objects and strings from the binary does not offer any improvement, especially due to the fact that these programs provided an extensive amount of seed files, so many of the generated paths are already meaningful based on the provided files. The increase in paths shows that by using a virtual disk, we can speed up the actual fuzzing process in order to do more executions, and spend less time reading and writing from/to files.

7.0 : CONCLUSIONS

Naturally, our approach to improving AFL only scratches the surface of what is actually feasible. As it stands, there have been a wide range of modifications that have been made to the base release of AFL since it was released in 2017. Modifications have been made that increase the speed and or the code coverage of the fuzzer through modifying the underlying algorithms AFL uses or the environment in which it runs- the OS and hardware- as we did. Strides have also been taken in modifying AFL to work without the source code, such as code emulation and code instrumentations, both dynamic and static.

A few notable fuzzers which are inspired by AFL include:

FairFuzz - an extension for AFL, that targets rare branches.
PerfFuzz- an extension for AFL, that looks for test cases which could significantly slow down the program.
Neuzz - fuzzing with neural networks
AFLSmart - another Graybox fuzzer. As input, it gets specification of input data in the format used by the Peach fuzzer
AFLGo - is an extension for AFL meant for getting to certain parts of code instead of full program coverage. It can be used for testing patches or newly added fragments of code.

20 POINT DISTRIBUTION:

10 - ANTHONY TRIOLO
4 - JARED CHESTER
4 - DANIEL GARRY
2 - DIANE TRAN