Mutational Fuzzing to Discover Software Bugs and Vulnerabilities

Dylan Wolff
Advised by Robert Signorile
2015 Senior Honors Thesis
Boston College Computer Science Department

Abstract:

Recent major vulnerabilities such as the Heartbleed bug have emphasized the importance of properly testing software. For both product testers and hackers, fuzzing has emerged as an important tool used to find these issues. This thesis investigates one branch of the field, mutational fuzzing, in which valid inputs are randomly altered to produce sample files. While the samples produced by this approach are not as efficient as the hand-crafted inputs used in generational fuzzing, a mutational fuzzer can easily be used on multiple applications with no prior knowledge of the input formats and only a small amount of setup. The fuzzer created for this thesis, written in Python, is a multiprocessing, distributed generic file fuzzer, and was used to test the popular media player VLC for bugs.

Contents:

1 Introduction

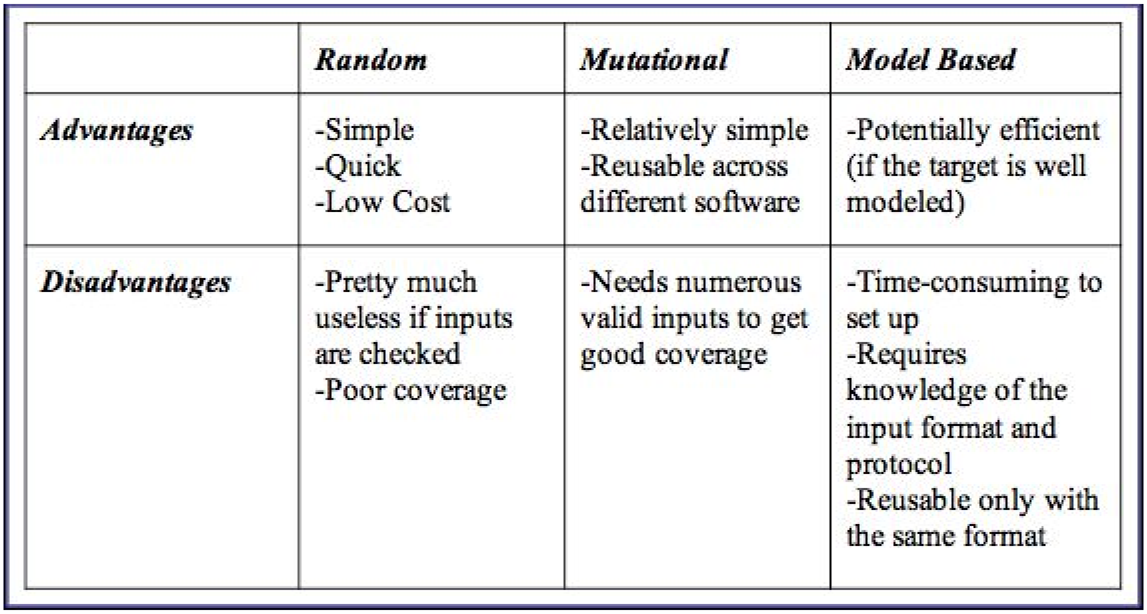
As technology becomes increasingly involved with our daily lives, breaches in the security of devices and applications that we rely on can have devastating implications. In recent months, serious vulnerabilities with seriously catchy names like “Shellshock” and “Heartbleed” have made headlines. These were important discoveries to be sure, but they are just two of countless important vulnerabilities in software that have been uncovered (and countless more that haven’t yet been found). The purpose of this thesis is to examine one branch of computer security, fuzzing, and how it is used by both security experts and attackers to test real world programs for these issues.

1.1 What is Fuzzing?

Fuzz testing is the process of feeding random, unexpected, or invalid data to a program’s inputs. This process is usually automated, with a debugger attached to the target program. If the program does crash, information about the state of the machine is logged by the debugger so that a programmer can later identify the faulty code that led to the crash. Fuzzing automates the process of vulnerability and crash discovery. This allows for a massive increase in the volume of tests that can be run compared to what a human could do, while also potentially catching subtle bugs that a human code auditor might miss when looking at lengthy and complex source code [15].

Unfortunately, fuzzing an entire input space is generally intractable.

Consider an image that is 47.4KB large. For each byte, there are 256 possible values so the input space for an image viewer accepting only images of this exact size is: 256^{47400}. This would take a computer running at 4GHz doing one possible iteration of this input space every cycle 10^{737} times the current age of the universe to finish. Thus the differences in types of fuzzers arise in how test cases are chosen from the vast input space of a program.



	Random	***Mutational***	***Model Based***
Advantages	-Simple -Quick -Low Cost	-Relatively simple -Reusable across different software	-Potentially efficient (if the target is well modeled)
Disadvantages	-Pretty much useless if inputs are checked -Poor coverage	-Needs numerous valid inputs to get good coverage	-Time-consuming to set up -Requires knowledge of the input format and protocol -Reusable only with the same format

Overview of different fuzzing strategies [7]

The two primary categories of fuzzers are mutational and generational. Random fuzzers can also be useful in a narrow set of applications, but most real-world software has some kind of input checking mechanism that makes productive random fuzzing impossible. Mutational fuzzing takes seed files that are valid inputs to a program and “mutates” random bits of data in these files to other values. These mutated files are then executed under the target program. Generational fuzzing relies on a deeper understanding of the application to create input files designed to test specific aspects of a program. Instead of randomly testing mutant files in the input space, using knowledge of the program and the structure of it's inputs, generational fuzzing aims to only tests minimum and maximum values within test cases (e.g. for a piece of code like: "if 3 <= x <=10:", the edge cases 3 and 10 are worth testing, as well as 2 and 11, but other values for x won't change the control flow of the program and are thus unlikely to produce any errors that these four values do not) . Unfortunately, code branching is still generally exponential with code length, so it is still unlikely that a significant portion of edge cases could be tested for all branches within a given program [9].

1.2 Why fuzzing?

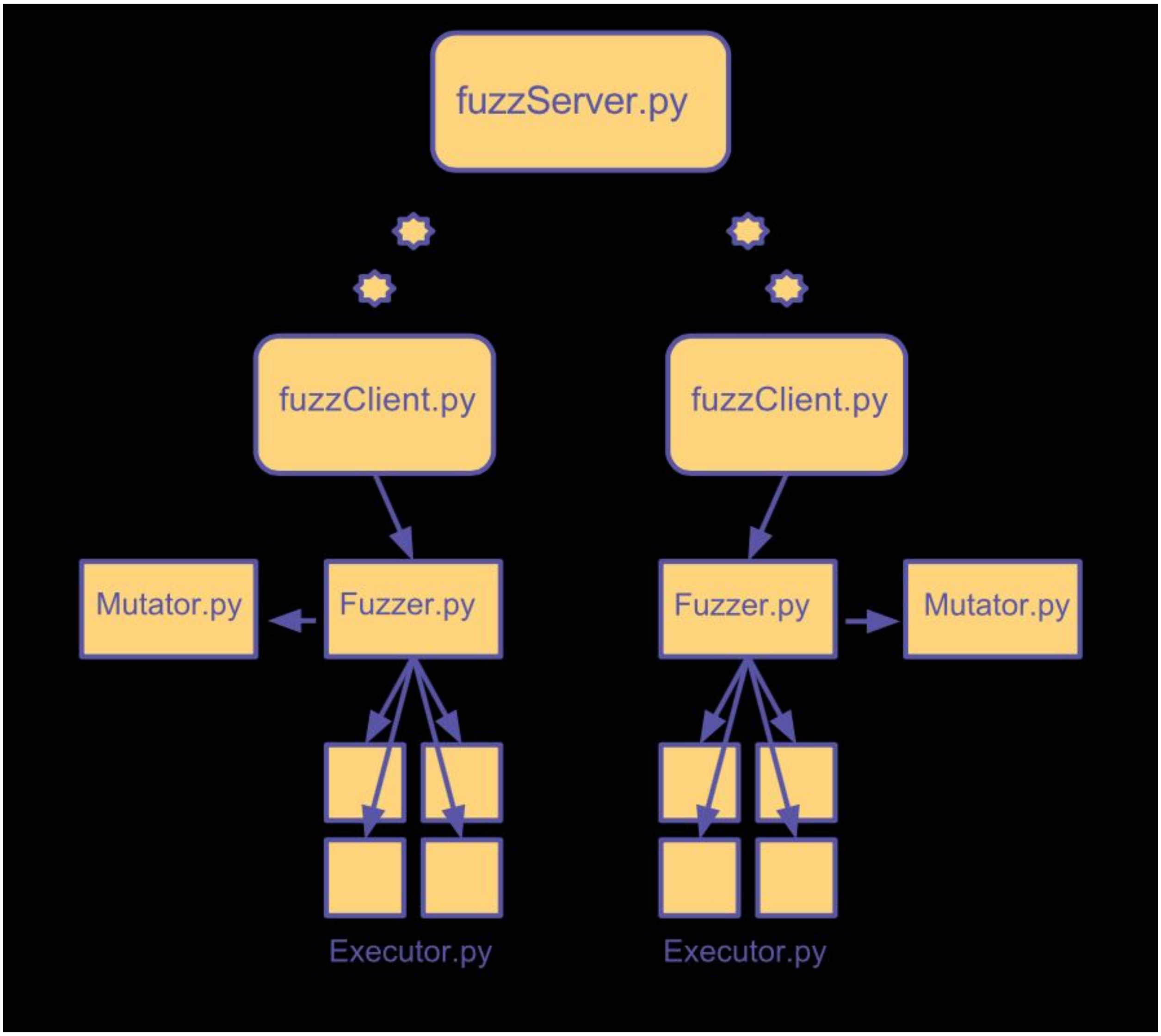
Most of us know enough not to download or open anything with weird file extensions, but vulnerabilities in applications can be exploited even by seemingly innocuous files like .wav, .doc, etc., the worst of which can even be used by an attacker to achieve arbitrary code execution on a machine. Fuzzing has uncovered such vulnerabilities in a wide variety of applications. Charlie Miller, one of the premier experts in fuzzing, was the first one to hack the iPhone [10]. He did so by exploiting a vulnerability he found while fuzzing. Other fuzzers have uncovered vulnerabilities in targets ranging from the Debian operating system to the image viewer ImageMagick [9]. Fuzzing has become so prominent that even "Verizon now requires that its vendors do their own fuzz testing before submitting equipment into the Verizon testing lab" [2].

Initially, for this project, I looked at two separate fuzzers that were free online. The first was the Basic Fuzzing Framework (BFF), the second Peach Fuzzer. In both cases I found the fuzzers to be incredibly complex and poorly documented. While they are both quality products, the sheer number of features in each is totally overwhelming. After far too much time spent wading through other people's code (ironically the exact kind of brutal legwork fuzzing itself aims to avoid) and the few tutorials I could find online, it became apparent that learning one of these prebuilt frameworks was a project in of itself, and certainly not one worth dedicating a serious amount of my thesis to. Instead, I have written my own fuzzer, with the intention of it being both effective and user friendly.

2 Fuzzer Structure

2.1 Environment and Structural Overview

The entire fuzzer is built in a virtual machine for Virtual Box running a 64 bit copy of Windows 7 Enterprise. This ensures that the host machine is unaffected by any adverse consequences of executing so many potentially damaging files during the fuzzing run. It also has the added benefit of making installing the fuzzer on multiple computers for a distributed run extremely simple. After installing Virtual Box and importing the WolffFuzz appliance on the new computer, the fuzzer is already fully operational. In order to maximize the amount of a program's input space that can be fuzzed in a given amount of time, the fuzzer is both distributed and multiprocessing, but for distributed fuzzing runs to work, port forwarding must be activated in the network settings of the server virtual machine (using the guest and host ip's respectively and the appropriate port number).



Overview of fuzzer structure. Depicted with two clients and four Executor processes per client.

2.2 FuzzServer.py and FuzzClient.py

When run on the server machine, FuzzServer.py first reads in the parameters for the fuzzing run from the config file, FuzzerConfig.txt, on the desktop. It then waits for client connections on a pre-selected port (12000 is the default). Upon connection to a client, it sends across the fuzzing parameters, followed by a portion of the sample files that are located in the servSamples folder on the desktop. Once all of the sample files have been distributed equally to each of the clients, the server waits once again on the selected port for client connections that signal that each client fuzzer has completed the run. From these connections, the server collects and consolidates the crashes of all clients into the ServCrashers folder on the desktop, and performs some cursory automated analysis to give the user the number of crashes considered exploitable, the number of unique crashes, and the total number of crashes.

When run, FuzzClient.py immediately attempts to connect with the server at a port number and IP address specified in FuzzerConfig.txt on the desktop of its virtual machine. Upon connection, it parses the fuzzing parameters from the server, and writes the sample files it receives to the Samples folder on the desktop. Finally, it begins a new Fuzzer process before terminating.

2.3 Fuzzer.py

Fuzzer.py can be run manually by a user to proceed with a non-distributed fuzzing run. If this is the case, the fuzzing parameters are read in from the local

FuzzerConfig.txt file. Otherwise, the parameters are passed in as arguments when the Fuzzer process is created by a FuzzClient process as part of a distributed run. The Fuzzer process creates one Mutator process and the number of Executor processes specified by the fuzzing parameters. It passes to each child process a process/thread safe Queue to convey mutated file names from the Mutator to the Executor processes as well as a similarly synchronized Queue for the names of old mutated files that need to be removed. The Fuzzer process then sits in a loop checking if any of the Executors have died. If this is the case, the Fuzzer launches a new Executor process. The Fuzzer also checks the Queue for the string “STOP”. If “STOP”, the poison pill, is found, the Fuzzer waits until all Executors are finished with their current files before sending the contents of the Crashers folder to the server. If the fuzzing run is not distributed, the Fuzzer process calculates and prints the number of exploitable crashes, unique crashes, and total crashes before terminating.

2.4 Mutator.py

The Mutator process takes each seed file and mutates it the amount given by the mutation percentage parameter a certain number of times. This number is dictated by the iterations parameter in the FuzzerConfig.txt file. The Mutator first sorts the seed files by size, so that smaller files that are mutated faster can be made available more quickly for the initial executions of the target program. Each file is read into a string, which is then converted to a list. For each iteration of each file, the Mutator gets a random seed value. It then calculates a randomized write location between beginning and the end of the file and writes a byte value between 0 and 255 to that location in the list. These random writes to the list are performed a number of times equal to the file length in bytes multiplied by the mutation percentage. Once the writes are completed, the list is then converted back to a string, and written to a new file with a new name in the Mutated folder on the desktop. The mutated file's name, sample file name, and mutation seed are all then placed on the queue.

The mutation process described above is relatively memory intensive, as it carries the entire file in memory as a list. Another function, mutate2, was thus developed as an alternative that uses significantly less memory. mutate2 uses shutil.copy2() to copy the seed file to the mutated folder with a new name, and then the file.seek() function to write random bytes to random addresses in the new copied file. Unfortunately mutate2 is much slower than the first mutation function, so it is only called if the size of the file being mutated is such that it could potentially cause a memory error or does cause a memory error.

Whichever mutate function is chosen mutates all iterations of a given sample file before moving on to the next. At each iteration, the mutator logs its progress in a file such that it can be resumed if it were to crash. This process is repeated until all sample files have been mutated with the proper number of iterations. If the Mutated folder on the desktop exceeds 5 gigabytes, however, the Mutator process stalls to allow the executor processes to clear out the mutated files. This prevents the virtual machine from running out of its limited 25GB hard drive space. While stalling, the Mutator process checks a queue of names of already executed mutated files that Executor processes failed to delete and attempts to delete them. Once finished, the Mutator puts a “STOP” string on the queue as a poison pill for all other processes, and terminates.

2.5 Executor.py

The Executor process upon first launch gets a new filename off of the queue. If it finds “STOP” on the queue, it replaces it, writes “STOP” to its log file and terminates. If it gets a filename for a mutated file, it logs that it is entering execution of a file and executes that file under the target program with WinAppDbg attached. It then waits for a timeout in seconds, passed in as a parameter upon launch. If a crash is found, a callback function, my_event_handler(), creates a new folder for the crash, in which it puts the seed for the random mutations that caused the crash, the name of the sample file that caused the crash, and the output of the debugger in text files. After a timeout, the Executor kills the target program and logs that it has finished executing. It then attempts to remove the mutated file just executed. If it fails to do so because the target program hasn’t fully died yet, that file is put on a queue to be removed later. Finally, the Executor checks memory usage. If the memory usage of the computer exceeds 90% and the Executor process has been alive long enough to have executed at least two files, the process will self terminate. This is because certain programs (VLC included) cannot be terminated by any of the according WinAppDbg functions, nor Process_Terminate. These processes hang around, taking up memory, until the parent process is terminated. To prevent an accumulation of such processes to the point of a memory fault, the Executors must be culled if usage gets too high. Not killing Executors that have just begun is enough to ensure that the amount of memory saved by a kill is worth the overhead of restarting an Executor process.

2.6 reMutate.py

reMutate.py is used for post-processing. Specifically to use the crashsrc.txt files in each crash folder and the sample file to re-create a mutated file. It reads the specifications (seed, mutation percentage and sample file name) from a crashsrc.txt file dragged to the reMutate folder on the desktop. Using the name of the sample file, it finds the original sample in the Samples folder on the desktop. It then recreates the mutations of the mutator using the seed and mutation percentage, and creates a new file in the Mutated folder on the desktop.

Emphasis was placed in all parts of the fuzzer on being able to recover from a crash that is severe enough to bring down the fuzzer itself (or just if it needed to be paused or interrupted for any reason). This is accomplished by both the Executor and Mutator processes logging progress throughout the fuzzing run. When Fuzzer.py is run by a user, it prompts the user if a run needs to be resumed. If this option is selected, the Queues are read from serialized backup files and the Mutator and Executor processes read from their respective log files to pick up exactly from where they left off.

3 Fuzzing VLC

3.1 Why VLC?

VLC is listed as having been downloaded 60 million times on download.com, yet just last year, a vulnerability was discovered that allowed an attacker to gain arbitrary code execution on a machine by a user opening a malicious .asf file with the application [4]. With that many people using VLC, chances are, as an attacker, if you post a well disguised, harmful media file on the internet, a large percentage of the people who download it will have VLC. In general, there are also only a few programs that the average Joe Computerowner uses on a regular basis: web browsers, media players, email readers, document readers etc. [15]. Attacking programs like these not only has a high probability of the user having the application, but also of them using it. Even if you are smart enough to avoid downloading malicious content, VLC and other media players install plugins for popular web browsers (Firefox, Chrome, Internet Explorer, Safari etc.), harmful media files embedded in websites can be played without your explicit consent [15]. Furthermore, these programs are not only usually perceived as safe by their users, but often by the software engineers who write the applications themselves; the programmers for these type of applications who are not generally as focused on security as engineers working on something where it is more obviously important, like a financial transaction over the web

[15]. And VLC and other media players are extremely complex, dealing with many different file formats on many different platforms. This complexity makes it easier for programmers to make mistakes, and for those mistakes to go unnoticed. All of these factors combine to make VLC an enticing target for malicious hackers, and an important one for those concerned with cybersecurity. It is for these reasons that VLC was chosen as the target of the fuzzer created for this thesis.

3.2 Crashes from VLC

For the large scale test run of the fuzzer created for this thesis the parameters were the following:

Mutation Percentage: 0.01

Iterations per Sample File: 120

Number of Executing Threads per Client: 4

Number of Client Machines: 4

The 77 sample files for the run were downloaded from http://samples.mplayerhq.hu/. This is the same body of sample files that a security team at Google used to fuzz media libraries (some of which are also used by VLC) [9]. The version of VLC used for this run (and all other runs) was VLC media player 2.1.2 [15].

The run resulted in 253 crashes total, 25 of which are classified as exploitable by the debugger. From those crashes, there were 35 unique addresses at which faults occurred. Only 9 of the seed files caused crashes when mutated.

Another file seed file was separately found to cause crashes without any mutations prior to the initial fuzzing run, leaving 10 unique seed files and 36 unique crashes. Of 36 unique crashers, VLC crashed in 5 different dll's and 7 different functions: libavcodec_plugin (start), libsimple_channel_mixer_plugin (start), libvlccore (es_format_InitFromVideo, picture_Release, picture_Hold, picture_pool_Get, vout_ReleasePicture), libdtstofloat32_plugin (start), libfaad_plugin (start). Other faults occurred in ntdll (RtlFreeHeap, RtlUnhandledExceptionFilter) and msvcrt (memcpy), but a backtrace using GDB did not reveal what function or library in VLC caused the initial problem. Of the dll's, it appears that at least two of them, libfaad_plugin and libavcodec_plugin are used in other applications. The former is advertised on several websites for download as an open source AAC decoder while the latter has its own Wikipedia page, where it is listed as a component in 57 other applications [1, 8]!

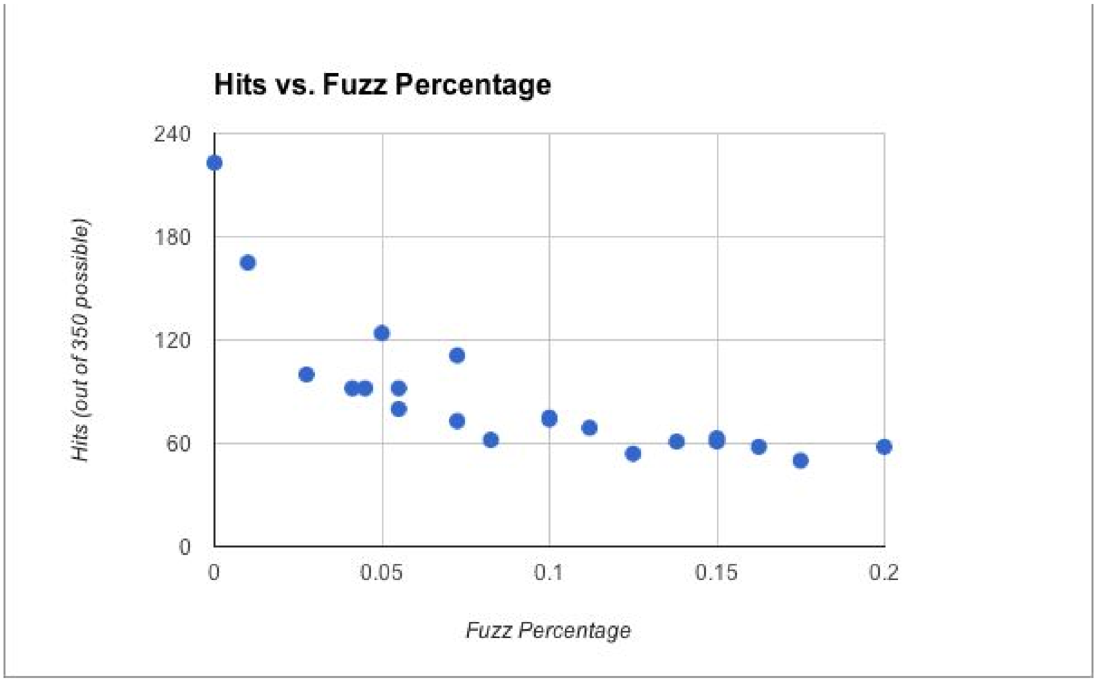
While it is left for future investigation to determine the causes of these crashes or their true exploitability, there is one bug that seems particularly dangerous. Specifically all the errors occurring with mutations of the issue1930.h264 file. The file produces the most unique instruction pointers upon crashing, but that instruction pointer is generally off in unallocated space. The other instruction pointer locations of crashes for mutations of this seed file are in the memcpy function, which notably does not check bounds on the target memory buffer. It seems likely that all of these "unique" crashes are really just the product of one error in which memcpy is sometimes overflowing a buffer and affecting a jump or return address. If this were found to be the case, the error would be highly exploitable.

3.3 Optimizing Fuzzing Parameters

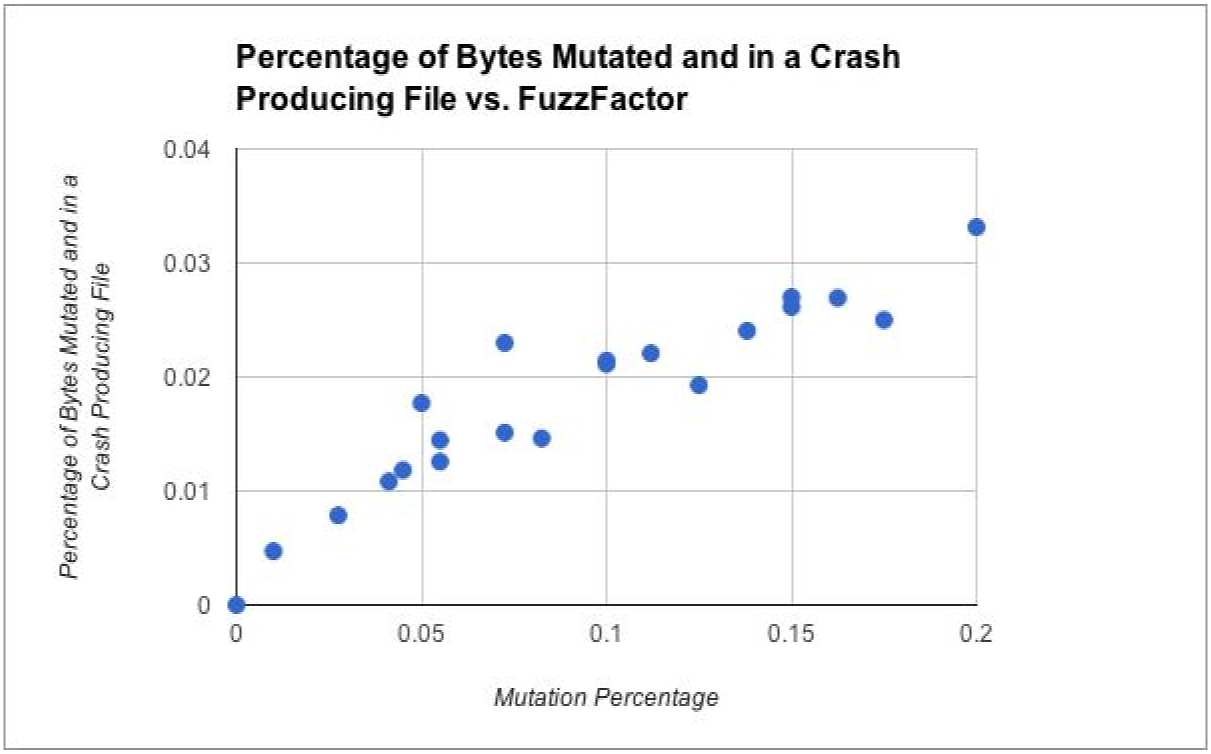
There are four parameters in FuzzerConfig.txt that have an extremely large impact on the efficacy of the run: Mutation Percentage, Number of Iterations per File, Timeout, and number of Executor processes. Using data gathered from the large scale fuzzing run of VLC for this fuzzer and prior research on this topic, we attempted to approximate optimal fuzzing parameters for future fuzzing runs.

3.3.1 Mutation Percentage

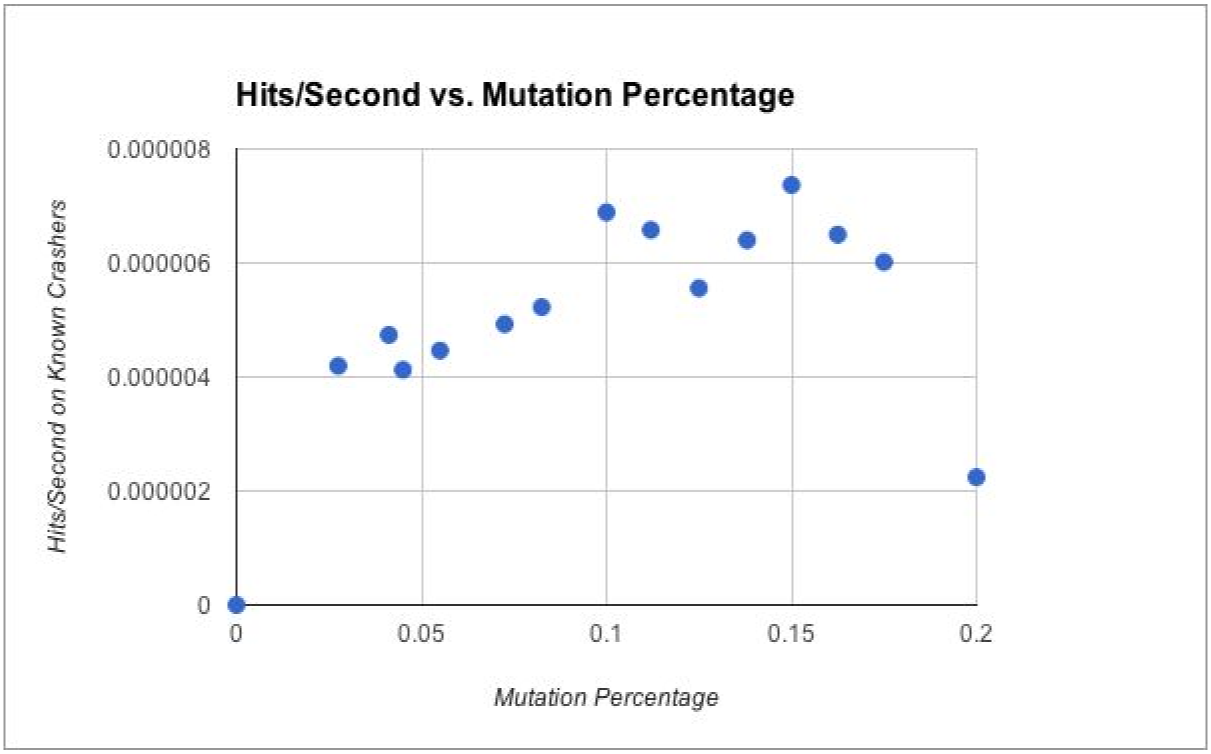
Fuzzing files has to walk the line between being structured enough to bypass input checks and other defenses that allow a given program to exit gracefully while also being invalid enough to cause crashes. There is virtually no previous research on this parameter, so, to test various mutation percentages, I took 7 different crash-inducing files (one crash causing mutation of each of the seed files that caused at least one crash). These files were then used as sample files for runs with varying mutation percentages. If the file became too mangled by the second round of mutations, then it would not pass the input checks that VLC performs and thus would not crash. Otherwise, the program should crash as it did before the mutation. The probability of undoing the mutation(s) from the first round and making the file non-buggy is extremely slim, so a file not crashing indicates with a relatively high probability that it was mutated to severely the second round (as it passed all input checks in the first round).



The graph of Hits vs. Fuzz Percentage above indicates that as the mutation percentage increases, the number of known crashers that pass crash (and thus pass input checks) declines. However, that decline is not particularly steep, and appears even to flatten out as the mutation percentage approaches 20%. Even as we are fuzzing less files, the average percentage of mutated bytes in files being executed and passing input checks is increasing approximately linearly with the fuzz factor as can be seen in the graph below.



More of these mutated bytes getting into executed files means that there is an increased chance that one of those bytes causes a crash, thus it would seem that a higher mutation percentage is better. However, an increase in mutation percentage has an impact on the execution time as well. It slows down the mutation process to the point that the Mutator process, rather than the Executor processes becomes the limiting factor in speed. Thus Hits/Second vs. Mutation Percentage graph below incorporates total execution time, finding the per second percentage of executed bytes. There are two peaks, at 10 and 15 percent, with a steep drop off after 15%. Thus the optimal fuzz factor for this setup is likely in that range.



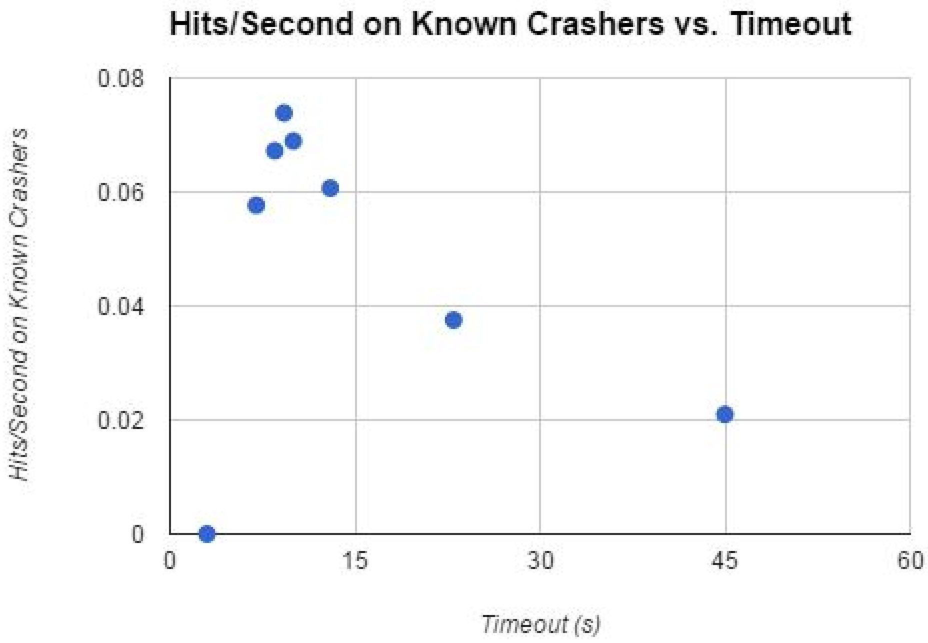
3.3.2 Number of Iterations per Sample File

The number of iterations may seem at first to be a relatively simple parameter to set. In order to fuzz as many files as possible, the number of iterations should be set as high as possible such that the fuzzing run can be completed in the time available. However, the number of files fuzzed is a combination of two factors: the number of iterations and the number of initial seed files. This begs the question, if we only have a set amount of time and thus can only make a certain number of mutant files, which should we prioritize, the number of seed files or the number of iterations? There has been substantial research done on seed selection for mutational fuzzing that helps to develop an answer.

The general consensus is that finding a minimum set of seed files that have the maximum code coverage is the optimal way to select seed files. Most notably, *Optimizing Seed Selection*, makes two strong conclusions to this effect: First that such a set “performed best” out of six methods for seed selection and that “Fuzzing with a reduced set is more efficient in practice” than an equivalent number of files from the full set [11]. Thus code coverage of the fuzzed files is an important metric when fuzzing. Miller reports an 1% increase in code coverage increases the percentage of bugs found by 0.92% [11]. So, when constructing a test set of a constant magnitude equal to the the product of the number of iterations and the number of seed files, we want to maximize code coverage. If we select our seed files with this in mind, with each additional seed file we are increasing code coverage. Each iteration, however, is highly likely to have the exact same code coverage as the seed file which it is mutating. Thus, given the choice, more seed files with different code coverage are preferable to more iterations. If we cannot find seed files with new code coverage however, extra iterations of the seed files is probably equally valuable and doesn’t require gathering any more files. Thus the answer to the question of how many seed files should be chosen relative to the number of iterations of mutations performed on each seed file is that maximizing the number of different (in terms of code coverage) seed files is the priority. Once code coverage has been maximized, each seed file should be iterated as many times as is possible given the allotted time for the run.

3.3.3 Timeout

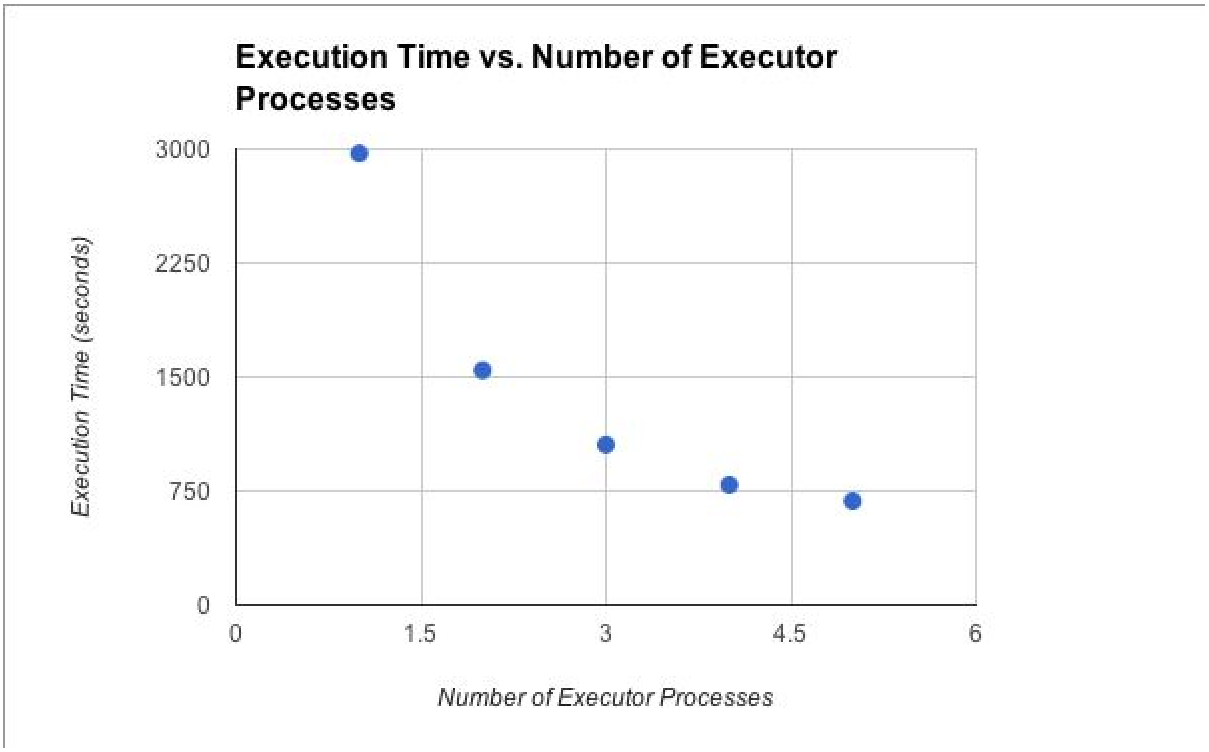
The timeout parameter sets how long the Executor process will wait before killing an executing target process. There is virtually no previous research done on this setting because it is unique to the target application and environment for the fuzzing run; the slower the computer or the larger the application, the longer it takes for each execution, and thus the longer the timeout must be to accommodate the extra startup time. Furthermore, the type of files affects the amount of time to execute the entire file. Media files can range from seconds to hours long. This makes it difficult to determine a firm timeout for applications that run these files. A subset of unique crash inducing files gathered from the initial large test run on VLC were tested with a mutation percentage of zero and varying timeouts to determine the optimal timeout for a fuzzing run on VLC. The tests were run on an iMac11,2 host (3.06 GHz Intel Core i3 Dual Core Processor, with 4GB of 1333MHz Ram), and the virtual machine was set to 4 cores and 2GB of RAM. The graph plots crash rate (crashes/total files) divided by the total time to execute the entire run against the timeout setting.



The graph above suggests that the 45 second timeout used in the initial test run was not optimal for finding crashes, but rather a timeout closer to 9.25s. Because files are being randomly mutated, they are generally equally perturbed at the beginning and at the end of the file. Thus if a given amount of perturbation has a probability *p* of causing a crash for a given block of the file, then the block executed first has approximately a probability of *p* to cause a crash. For the next block to cause a crash, the first block must not crashed, which gives a probability of roughly *(1-p)*p*. The probability of the next block causing a crash is *(1-p)*(1-p)*p*. And so on. Thus, we are far less likely to see crashes later in the execution when compared to the beginning. For media files, it also seems likely that the beginning of the execution is when most of the heavy lifting for the application takes place; once the video begins to play, most of the data being used from the file is just dictating the colors of the pixels

3.3.4 Number of Executor Processes

This is probably the easiest parameter to figure out. The short and obvious answer is: as many as possible. The data in the graph below is clear and expected: there are very good returns for introducing a little bit of parallelism, but there are diminishing returns as the number of parallel processes increase. Between 4 and 5 executor processes, the improvement begins to flatten out. At 6 Executors or more, the fuzzer begins to become unreliable and unstable. It seems then that running 4 Executors, one for each core in the Virtual Machine, is the optimal number for a fuzzing run, as it retains nearly the greatest productivity at lower risk of failure than with 5 or more Executors.



4. Conclusions and Further Work

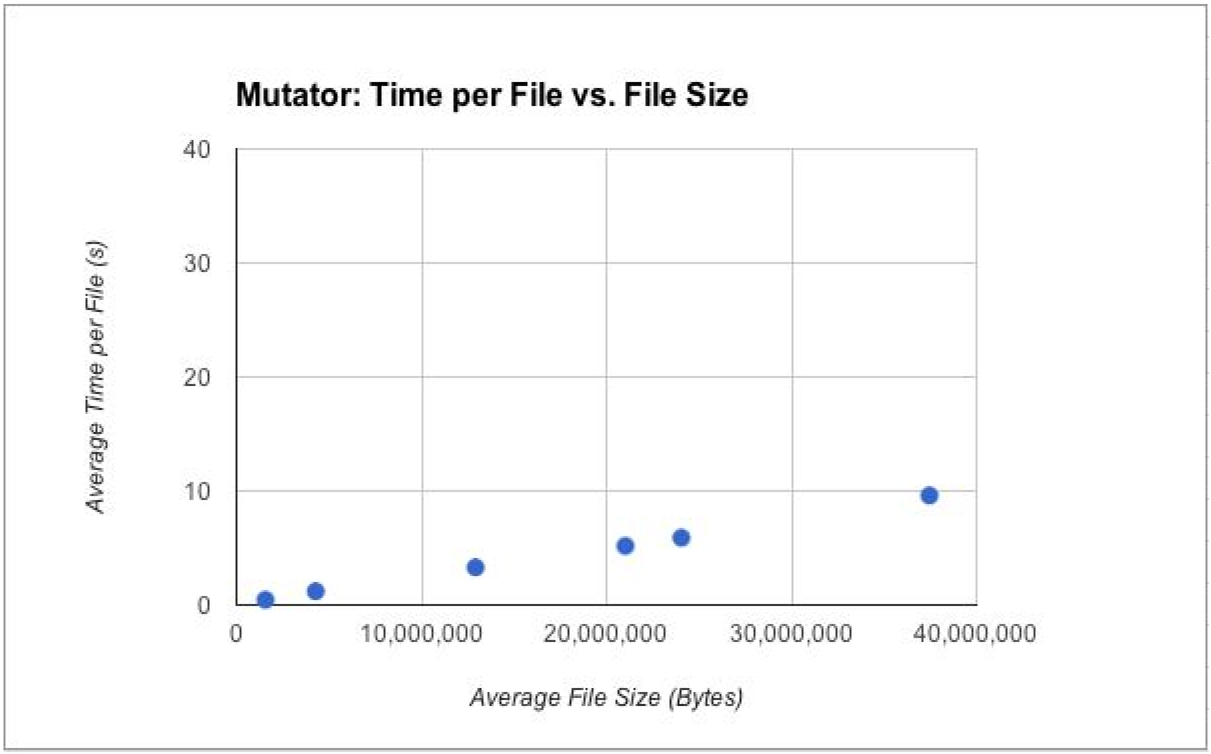
The fuzzing run on VLC yielded a lot of important information. First and foremost, at least ten unique crashes, one of which appears to be highly exploitable. These crashes were then useful in determining optimal parameters for future fuzzing runs on VLC. Specifically that the optimal timeout is around 9.25s, the optimal fuzz factor is around 15%. Further testing revealed that running four Executor processes for a fuzzing run is likely to be ideal. The crash files found in this paper exposed bugs in a variety of libraries used by VLC and other programs, and could have serious ramifications if any of them are found to be exploitable. I intend on sending a full bug report to VLC for any of these issues that haven't yet been resolved.

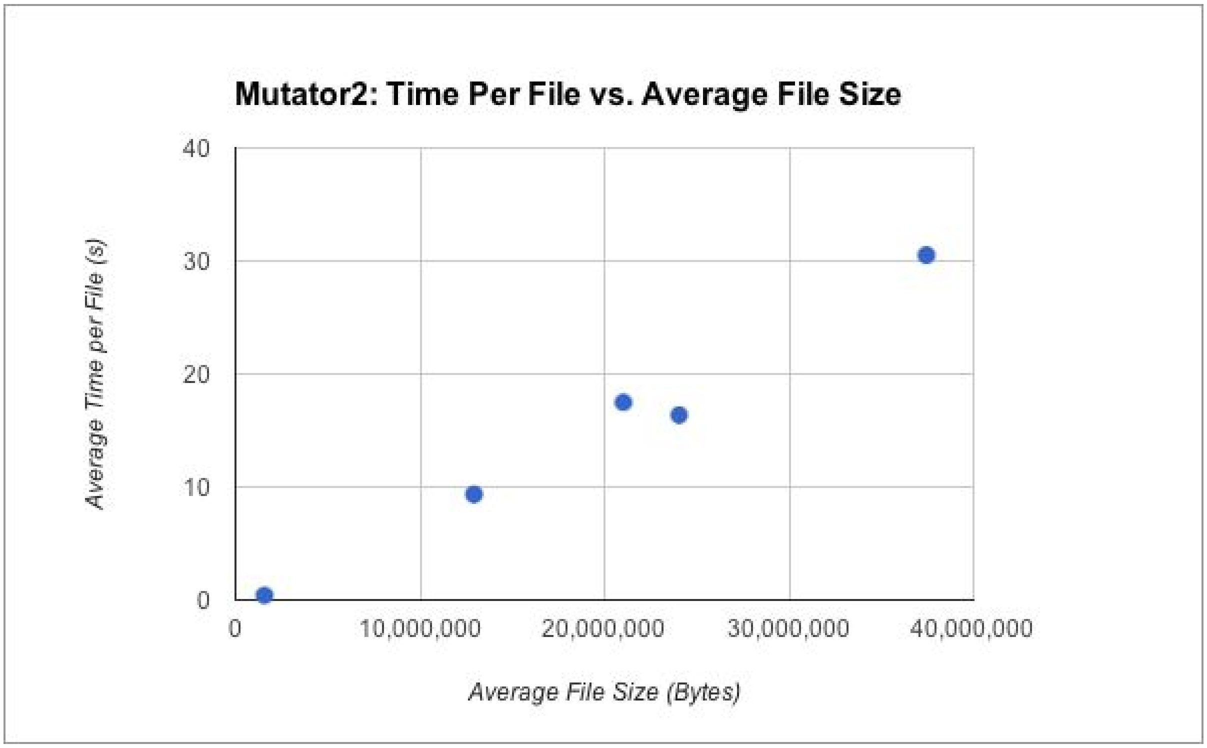
Given more time, there are several aspects of this project I would have liked to explore further. First of all, another large scale fuzzing run on VLC with the optimized parameters found in this paper could help to confirm the conclusions I drew from the first run, as well as potentially uncover more bugs. Reverse engineering the errors found during the first fuzzing run of VLC and discovering whether any of them are actually exploitable would complete the vulnerability discovery process and assist developers on the project. Also, because the fuzzer will crash occasionally during long fuzzing runs, it would be helpful for the FuzzServer to be checking in on FuzzClients (through a ping-ack protocol for example) to see when they die. If a client dies, the server could notify the human in charge of the fuzzing run that a machine has gone down to minimize downtime for that computer.

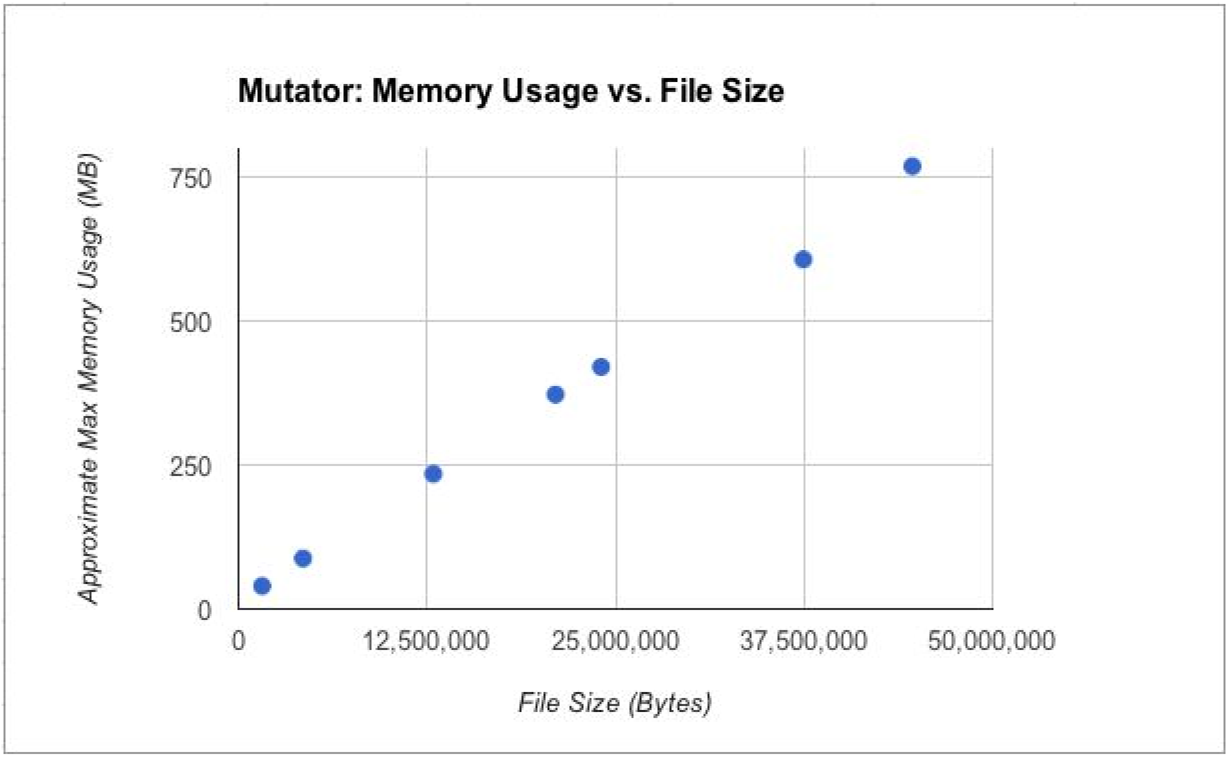
5 References

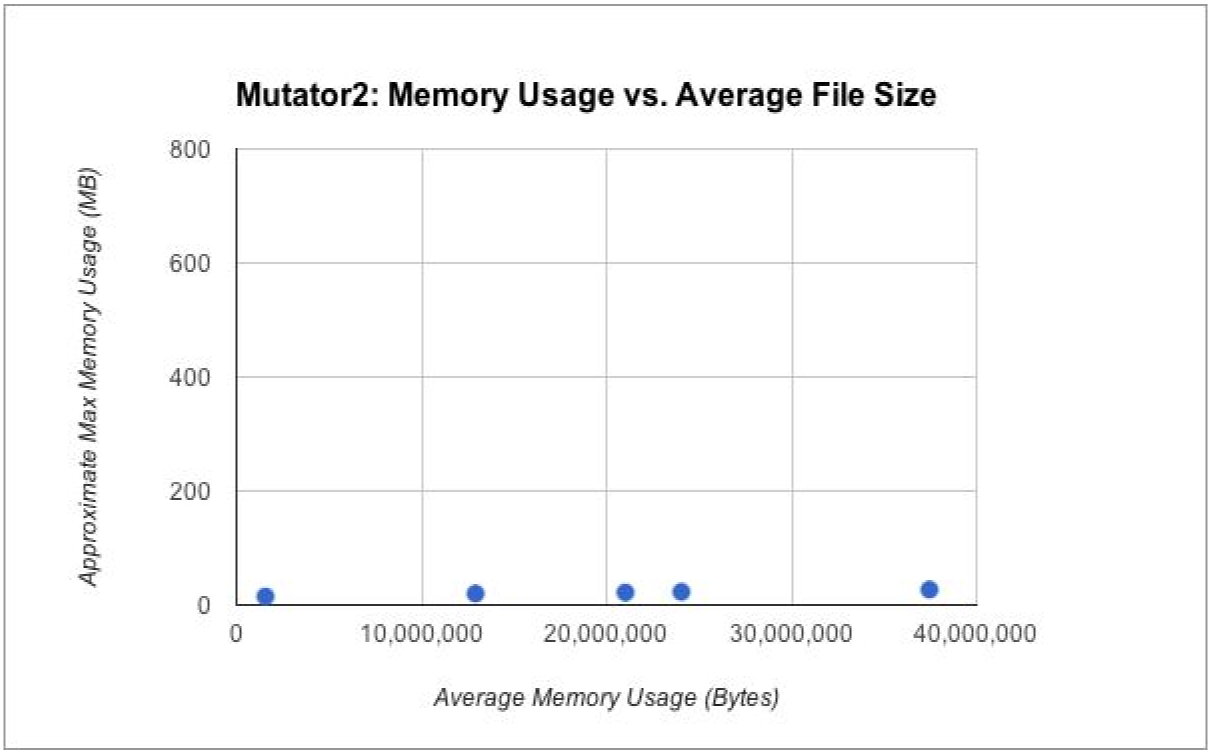
1. Amorim, Roberto, Edwards, John, Gan, Michael, Brooker, Marc and Naylor, David. "RareWares: AAC Decoders." RareWares., http://www.rarewares.org/aac-decoders.php.
2. Bekrar, Sofia, Chaouki Bekrar, Rolan Groz, and Laurent Mounier. 2012. "A Taint Based Approach for Smart Fuzzing." *IEEE Computer Society*.
3. Codenomicon. 2012. "Fuzz Testing: Improving Medical Device Quality and Safety." *MDISS Technical White Paper Series*.
4. Constantin, Lucian. "Critical Vulnerability in Affects Latest VLC Media Player, last modified 1/31/132014.
5. ffmpeg. "Index of Samples.", http://samples.mplayerhq.hu/.
6. Garg, Parul. "Fuzzing -- Mutation Vs. Generation." Infosec Institute., last modified 1/4/12, accessed 12/9/14, 2014, http://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/.
7. Lewis, Colleen, Barret Rhoden, and Cynthia Sturton. 2007. "Using Structured Random Data to Precisely Fuzz Media Players." Graduate, UC Berkeley.
8. "Libavcodec." Wikipedia., http://en.wikipedia.org/wiki/Libavcodec.
9. Liu, Xiuwen and Redwood, Owen. "Offensive Security: Spring 2014 Lectures." Offensive Computer Security., accessed 12/9, 2014, http://www.cs.fsu.edu/~redwood/OffensiveComputerSecurity/lectures.html.
10. Miller, Charlie. 2010. Las Vegas, NV, USA, CodenomiCON Ltd, 8/9/2010.
11. Rebert, Alexandre, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. "Optimizing Seed Selection for Fuzzing." *USENIX Security Symposium*.
12. Royal, Martin and Peter Pokorny. 2012. "Dumb Fuzzing in Practice." Undergraduate Capstone, Cameron University, http://www.cameron.edu/uploads/8d/e3/8de36a6c024c2be6dff3c34448711075/5.pdf;.
13. Samiux. 2013. *HOWTO : CERT Basic Fuzzing Framework (BFF) on Ubuntu Desktop 12.04 LTS*. Samiux's Blog.
14. Seth. 2010. *Memorystatusex*. http://stackoverflow.com/questions/2017545/get-memory-usage-of-computer-in-windows-with-python
15. Thiel, David. 2008. "Exposing Vulnerabilities in Media Software." Amsterdam, The Netherlands, iSEC Partners, Inc, 3/25/08.
16. VideoLAN Organization. 2014. *VLC Media Player*. Vol. 2.1.2. https://www.videolan.org/vlc/releases/2.1.2.html
17. Vilas, Mario. 2009. *Winappdbg*. Vol. 1.5. http://winappdbg.sourceforge.net/

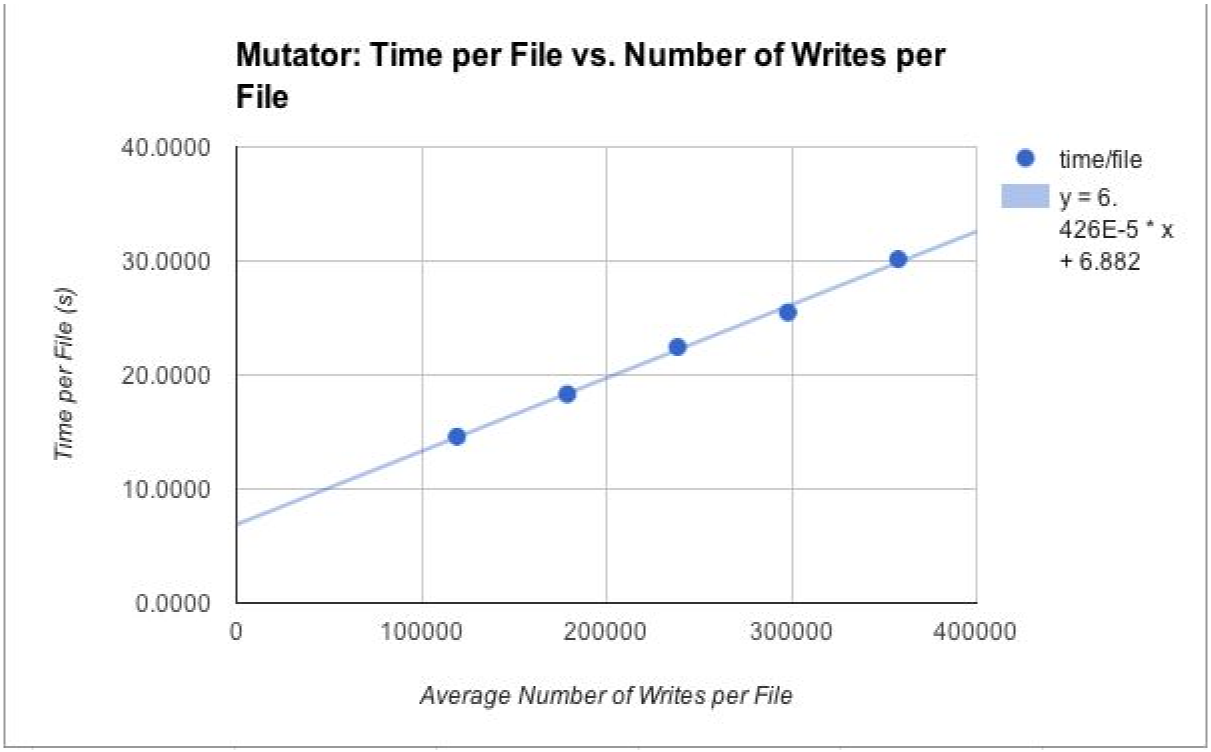
Appendix A: Comparison of Mutator Function Performance

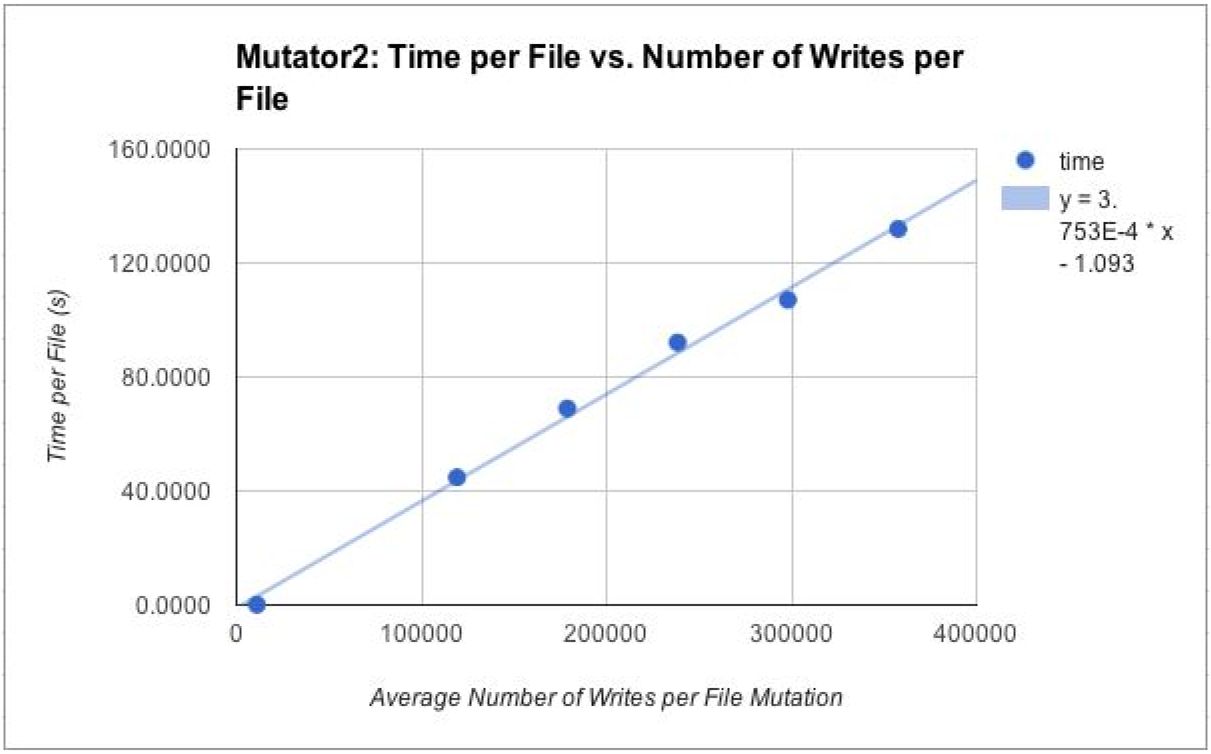





Mutator: Memory Usage vs. File Size
Approximate Max Memory Usage (MB)
750
500
250
0
0
12,500,000
25,000,000
37,500,000
50,000,000
File Size (Bytes)


Mutator2: Memory Usage vs. Average File Size
Approximate Max Memory Usage (MB)
800
600
400
200
0
0
10,000,000
20,000,000
30,000,000
40,000,000
Average Memory Usage (Bytes)


Mutator: Time per File vs. Number of Writes per File
40.0000
30.0000
20.0000
10.0000
0.0000
Time per File (s)
0
100000
200000
300000
400000
Average Number of Writes per File
● time/file
y = 6. 426E-5 * x + 6.882


Mutator2: Time per File vs. Number of Writes per File
160.0000
120.0000
80.0000
40.0000
0.0000
Time per File (s)
0
100000
200000
300000
400000
Average Number of Writes per File Mutation
● time
y = 3. 753E-4 * x - 1.093

Appendix B: Source Code

```
# FuzzClient.py
# Dylan Wolff 5/8/15
# FuzzClient immediately makes a connection with a FuzzServer and receives fuzzing files and parameters
#       for the run. It then starts a Fuzzer process for the run itself

from socket import *
from multiprocessing import Process
import Fuzzer, os

#function that receives a binary file from the clientSocket and returns it
def recFile(clientSocket, filesize):
        s = ''
        while len(s) != filesize:
                s = s + clientSocket.recv(filesize - len(s))

        return s

if __name__ == '__main__':

        path = '/Users/Fuzzer/Desktop/'

        #Delete files from any previous runs
        print "Discarding old sample files, hope you saved everything..."
    for root, dirs, files in os.walk(path + "Samples"):
      for f in files:
        os.unlink(os.path.join(root, f))

  #Get server info parameters from the config File
        f = open(path + 'FuzzerConfig.txt', 'r')
        sl = f.readlines()
        serverPort = int(sl[11].strip())
        serverIP = sl[13].strip()
        f.close()

        print "Connecting to " + str(serverPort) + " at " + serverIP
        clientSocket = socket(AF_INET, SOCK_STREAM)
        clientSocket.connect((serverIP, serverPort))



        #get parameters for the fuzzing run and put in config file
        reply = clientSocket.recv(4096)

        #Peel out info from the string to give to the Fuzzer process
        sl = reply.split('\n')
        print sl
        timeout = float(sl[1].strip())
        fuzzFactor = float(sl[3].strip())
        program = sl[5].strip()
        iterations = int(sl[7].strip())
        numExecs = int(sl[9].strip())

  #put the parameters in the config file in case we need to resume
        print  "rec config"
        clientSocket.send("sup")
        print "ack"
        f = open(path + 'FuzzerConfig.txt', 'w+')
        f.truncate()
        f.write(reply + '\n#Server Port Number\n' + str(serverPort) + '\n#Server IP Address\n' + serverIP +
'\n#Number of Client Fuzzers\n' + '1')
        f.close()

        #receive files until we get the done snding files token of all 'a's
        while True:

                #Get the reply from the server
                reply = ''
                while len(reply) != 4096:
                    reply = reply + clientSocket.recv(4096-len(reply))

                #if all files have been set/received, break
                if reply == 4096*'a':
                    break
                #send acknowledgement to server
                clientSocket.send("sup")

                #peel out the information about the file
		s = reply.split('|')

		#receive a file of length specified by s[1]
		guy = recFile(clientSocket, int(s[1]))
		#send acknowledge
		clientSocket.send("sup")

		#write the recevied file to the samples folder
		f = open(path + 'Samples/' + s[0], 'wb')
		f.write(guy)
		f.close()

	clientSocket.close()



	#Now begin the fuzzing run
	process = Process(target=Fuzzer.Fuzzer, args=(False, timeout, fuzzFactor, program, iterations, 
numExecs, serverPort, serverIP, path, True))
	process.start()





# FuzzServer.py
# Dylan Wolff 5/8/15
# FuzzServer waits for client connections. Once a connection is made, it sends across fuzzing parameters and
#	sample files from the servSamples folder to each client. Finally it again waits for client connections to
#	receive the results from a distributed fuzzing run. The default port for the server to wait on is 12000

import os, uuid, shutil, Executor, pickle, Mutator, time
from socket import *
from multiprocessing import Process, Queue

#Function that sends a file from the ServSamples folder to a client given a connection socket and the filename
def sendFile(filename, targetSocket):
	f = open(path + 'ServSamples/' + filename, 'rb')
	s = f.read()
	totalBytes = len(s)

	targetSocket.send(filename + '|' + str(totalBytes) + '|' + 'x'*(4096 - len(filename) - len(str(totalBytes)) - 2))
	#First send over the information about the file, padding the rest of the packet with xs so its the
```

```
expected
    # number of bytes

    #first send across the size of the file so the client knows how much to expect
    targetSocket.recv(3)
    #ack after every send to prevent consecutive messages from being merged

    targetSocket.send(s)
    #Actually send the file across
    targetSocket.recv(3)
    #ack




def distributeSamples(numClients, numExecs, path, timeout, program, fuzzFactor, iterations, serverPort,
serverIP):
    #this function distributes the sample files evenly across all client fuzzers

    clientConfigInfo = '#Timeout: \n' + str(timeout) + '\n#Fuzz Factor: \n' + str(fuzzFactor) + '\n#Name of
Program Being Fuzzed: \n' + program + ' \n#Number of Mutation iterations per File: \n' + str(iterations)
+'\n#Number of Executing Threads per Client: \n' + str(numExecs)

    #set up serversocket
    serverSocket = socket(AF_INET, SOCK_STREAM)
    serverSocket.bind((serverIP, serverPort))
    serverSocket.listen(numClients)

    #get list of files to send to clients
    samples = os.listdir(path + "ServSamples")
    #samples per client is integer division, so remainder is used for lefotvers
    samplesPerClient = len(samples)/numClients
    remainder = len(samples)%numClients

    #while we haven't sent files to all of the clients
    clientSent = 0
    while clientSent < numClients:
        clientSent = clientSent + 1
        connectionSocket, addrs = serverSocket.accept()

        connectionSocket.send(clientConfigInfo)
        connectionSocket.recv(3)
        #give the client the parameters for the fuzzing run, then wait for ack
        #figure out the appropriate number of sample flies to send over
        samplesSent = 0
        if remainder > 0:
            extra = 1
            remainder = remainder - 1
        else:
            extra = 0

        #send them over
        while samplesSent < samplesPerClient + extra:
            sendFile(samples[0], connectionSocket)
            samples.remove(samples[0])
            samplesSent = samplesSent + 1

        #send the finished sending files message (super bootleg, but whatever)
        connectionSocket.send(4096*'a')

def recFile(clientSocket, filesize):
    #this function receives a file from a given socket knowing the filesize

    s = ''
    while len(s) != filesize:
        s = s + clientSocket.recv(filesize - len(s))

    return s

def aggregateCrashes(path, numClients, serverPort, serverIP):
    #this function takes in all the crashes from all the clients

    #set up serversocket
    serverSocket = socket(AF_INET, SOCK_STREAM)
    serverSocket.bind((serverIP, serverPort))
    serverSocket.listen(numClients)


    clientsReported = 0
    directory = str(uuid.uuid4())


    while clientsReported < numClients:
        #while we haven't received files from all of the clients

        os.makedirs(path + '/servCrashers/' + directory)
        #make a directory for our first expected crash folder

        clientsReported = clientsReported + 1
        #tally the client for reporting in
        connectionSocket, addrs = serverSocket.accept()
        #accept the connection

        #receive files until we get the done snding files token
        while True:
            print "waiting to receive"
            reply = ''
            while len(reply) != 4096:
                reply = reply + connectionSocket.recv(4096-len(reply))

            if reply == 4096*'a':
                #if we get the end of folder token, then we create a new folder for the next batch
                directory = str(uuid.uuid4())
                os.makedirs(path + 'servCrashers/' + directory)
            elif reply == 4096*'b':
                #if we then receive the end crashes token, we delete that folder, and get the next client and their crashes
                shutil.rmtree(path + 'servCrashers/' + directory)
                break
            else:
                #otherwise, we begin receiving files, and putting them in that folder.
                connectionSocket.send("sup")

                s = reply.split('|')
                #initial message has the filename and the size in it
                guy = recFile(connectionSocket, int(s[1]))
                connectionSocket.send("sup")

                f = open(path + 'servCrashers/' + directory + '/' + s[0], 'wb')
                f.write(guy)
                f.close()

    serverSocket.close()
    print "All Clients Finished Fuzzing"

def fuzzReport(path, iterations):
    # This function does some autmatic analysis of a distributed fuzzing run

    samples = os.listdir(path + 'ServSamples')
    totalFiles = len(samples) * iterations
    crashes = os.listdir(path + 'ServCrashers')
    totalCrashes = len(crashes)

    print "\n\nOf " + str(totalFiles) + " files, " + str(totalCrashes) + " were crashes"

    count = 0
    c = 0
    rips = []
    uniques = 0
    for crash in crashes:
        try:
            f = open(path + 'ServCrashers/' + crash + '/crashlog.txt', 'r')
        except:
            continue
        lines = f.readlines()
        f.close()
        if 'Exploitable' in lines[2]:
            #check if crashlog says its Exploitable
            count = count + 1
        for line in lines:
            #check if RIP is unique to this crash
            if 'rip=' in line:
                rip = (line.split('=')[1]).split(' ')[0]
                if rip not in rips:
                    rips.append(rip)
                    uniques = uniques + 1




    print "\n " + str(count) + " of which are considered Exploitable"
    print "\n " + str(uniques) + " of which are unique (by fault address)"
```

```
if __name__ == '__main__':

        #Get parameters for fuzzing run
        path = '/Users/Fuzzer/Desktop/'



        f = open(path + 'FuzzerConfig.txt', 'r')
        sl = f.readlines()
        timeout = float(sl[1].strip())
        fuzzFactor = float(sl[3].strip())
        program = sl[5].strip()
        iterations = int(sl[7].strip())
        numExecs = int(sl[9].strip())
        serverPort = int(sl[11].strip())
        serverIP = '10.0.2.15'
        #The server is always the local VM IP address for NAT Configuration

        numClients = int(sl[15].strip())
        f.close()


        response = raw_input("Are you in the middle of a run and need to receive files? (y or n)\n")
    if (response == 'y'):
        response = raw_input("how many clients need to still report in? \n")
        aggregateCrashes(path, int(response), serverPort, serverIP)
        fuzzReport(path, iterations)
    else:

        print "Discarding old aggregate crashes"
        for root, dirs, files in os.walk(path + "ServCrashers"):
                for f in files:
                        os.unlink(os.path.join(root, f))
                for d in dirs:
                        shutil.rmtree(os.path.join(root, d))

        distributeSamples(numClients, numExecs, path, timeout, program, fuzzFactor, iterations, serverPort,
serverIP)

        aggregateCrashes(path, numClients, serverPort, serverIP)
```

```
                fuzzReport(path, iterations)


# Fuzzer.py
# Dylan Wolff 5/8/15
# Fuzzer.py launches Mutator and Executor processes, monitors them in case they die, and, at the end
#   a fuzzing run, either sends the crashes back to the server, or does a cursory analysis itself,
#   depending on whether the run was distributed or not

from winappdbg import Debug, HexDump, win32, Thread, Crash
from socket import *
import os, uuid, shutil, Executor, pickle, Mutator, time, Fuzzer, ctypes, multiprocessing



class Fuzzer():


    def __init__(self, resume, timeout, fuzzFactor, program, iterations, numExecs, serverPort, serverIP, path, distributed):



        path = 'C:\Users\Fuzzer\Desktop/'

        #Delete remaining files from previous run if not resuming
        if not resume:
            print "Deleting old files, hope you saved everything..."
            for root, dirs, files in os.walk(path + "State"):
                for f in files:
                    os.unlink(os.path.join(root, f))
            for root, dirs, files in os.walk(path + "Crashers"):
                for f in files:
                    os.unlink(os.path.join(root, f))
                for d in dirs:
                    shutil.rmtree(os.path.join(root, d))
            for root, dirs, files in os.walk(path + "Mutated"):
                for f in files:
                    os.unlink(os.path.join(root, f))
        if resume:
            # Get the Queues back from serialized file
            print "Resuming previous run..."


            try:
                qnfilein = open(path + "State/qn", 'rb')
                qnList = pickle.load(qnfilein)
                qnfilein.close()
            except:
                qnList = []

            try:
                qfilein = open(path + "State/q", 'rb')
                qList = pickle.load(qfilein)
                qfilein.close()
            except:
                qList = []
        else:
            qList = []
            qnList = []

        q = multiprocessing.Queue()
        for item in qList:
            q.put(item)
        qn = multiprocessing.Queue()
        for item in qList:
            qn.put(item)

        # Give a time estimate in seconds for the run
        try:
            print "Running", numExecs, " Executor processes. Estimated time = ",
((timeout*iterations*len(os.listdir(path + '/Samples/')))/numExecs)
        except:
            pass



        #begin the Mutator process
        process = multiprocessing.Process(target=Mutator.Mutator, args=(path, fuzzFactor, iterations, q, qn,
resume))
        process.start()

        #start appropriate number of Executor processes
        eProcesses = [None] * numExecs
        for i in range(numExecs):
          eProcesses[i] = multiprocessing.Process(target=Executor.Executor, args=(timeout, program, path, q, qn,
i, resume))
          eProcesses[i].start()



        # sit in a loop monitoring Executor processes and checking if the fuzzing run is over
        while True:

          time.sleep(1)
          for i in range(numExecs):
            if not eProcesses[i].is_alive():
              eProcesses[i] = multiprocessing.Process(target=Executor.Executor, args=(timeout, program, path, q,
qn, i, True))
              eProcesses[i].start()

          if q.qsize() == 1:
            top = q.get()
            q.put(top)
            if top == "STOP":
              break

        #Wait until all Executors are done. Run each again just to be totally sure everything is done running
        for i in range(numExecs):
          while eProcesses[i].is_alive():
            time.sleep(1)
          eProcesses[i] = multiprocessing.Process(target=Executor.Executor, args=(timeout, program, path, q, qn,
i, True))
          eProcesses[i].start()

        for i in range(numExecs):
          while eProcesses[i].is_alive():
            time.sleep(1)
        print "Finished fuzzing run"

        #delete any remaining mutated files
        fails = 0
        for i in range(qn.qsize()):
            try:
                s = qn.get(False)
            except:
                continue
            try:
                os.remove(path + 'Mutated/' + s)
            except:
                fails = fails + 1
        print "unable to delete ", fails, " mutated files"

        if(distributed):
            print "sending results to server"
            self.sendCrashes(path, serverPort, serverIP)
        else:
            self.fuzzReport(path, iterations)

def fuzzReport(self, path, iterations):
        #this function does a cursory automatic analysis of the crashes produced by a run
        samples = os.listdir(path + 'Samples')
        totalFiles = len(samples) * iterations
        crashes = os.listdir(path + 'Crashers')
        totalCrashes = len(crashes)

        print "\n\nOf " + str(totalFiles) + " files, " + str(totalCrashes) + " were crashes"

        count = 0
        rips = []
        uniques = 0
        for crash in crashes:
            f = open(path + 'Crashers/' + crash + '/crashlog.txt', 'r')
            lines = f.readlines()
            if 'Exploitable' in lines[2]:
                #check if crashlog says its exploitable
                count = count + 1
            for line in lines:
                #check for a unique RIP
```

```
                if 'rip=' in line:
                    rip = (line.split('=')[1]).split(' ')[0]
                    if rip not in rips:
                        rips.append(rip)
                        uniques = uniques + 1

        print "\n " + str(count) + " of which are considered Exploitable"
        print "\n " + str(uniques) + " of which are unique (by fault address)"

    def sendFile(self, fpath, fname, targetSocket):
        # this function sends a single file given its name and path across a given socket 4096 bytes at a time

        f = open(fpath + '/' + fname, 'rb')
        s = f.read()
        totalBytes = len(s)
        targetSocket.send(fname + '|' + str(totalBytes) + '|' + 'x'*(4096 - len(fname) - len(str(totalBytes)) - 2))
        #first send across the size of the file so the receiver knows how much to expect as well as the name, fill up
the packet with x's
        targetSocket.recv(3)
        #ack after every send to prevent consecutive messages from being merged
        targetSocket.send(s)
        #sendfile
        targetSocket.recv(3)
        #ack

    def sendCrashes(self, path, serverPort, serverIP):
        #this function sends all of the crashes in the Crashers folder back to the main server for analysis

        # set up connection with server
        clientSocket = socket(AF_INET, SOCK_STREAM)
        clientSocket.connect((serverIP, serverPort))



        #get all crash folders
        crashes = os.listdir(path + 'Crashers')

        for crash in crashes:

            #for each folder (note that windows has a hidden file it sometimes auto creates called desktop.ini.
            # It's obviously not a folder, so it produces an error if we were to look inside it)
            if crash != 'desktop.ini':
                files =  os.listdir(path + 'Crashers/' + crash)

                for f in files:
                    #for each file in crash folder, send it across
                    self.sendFile(path + 'Crashers/' + crash, f, clientSocket)



                #4096 a's is the end of a crash folder signal
                clientSocket.send(4096*'a')
        print "Totally done sending files to server"
        #4096 b's is the end of all the crashes from this guy signal
        clientSocket.send(4096*'b')



        clientSocket.close()



    if __name__ == '__main__':
        # This is for either non-distributed runs, or to resume a run
        response = raw_input("Would you like to resume a previous fuzzing run? (please type either 'y' or 'n' followed by the enter key) \n")
        if (response != 'n'):
            responseDist = raw_input("Is the run distributed? (please type either 'y' or 'n' followed by the enter key) \n")
        else:
            responseDist = 'n'

        path = 'C:\Users\Fuzzer\Desktop/'

        #Get Fuzzing parameters from Config File
        f = open(path + 'FuzzerConfig.txt', 'r')
        sl = f.readlines()
        timeout = float(sl[1].strip())
        fuzzFactor = float(sl[3].strip())
        program = sl[5].strip()
        iterations = int(sl[7].strip())
        numExecs = int(sl[9].strip())
        serverPort = int(sl[11].strip())
        serverIP = sl[13].strip()
        f.close()
        process = multiprocessing.Process(target=Fuzzer.Fuzzer, args=((response != 'n'), timeout, fuzzFactor,
program, iterations, numExecs, serverPort, serverIP, path, (responseDist == 'y')))
        process.start()

# Executor.py
# Dylan Wolff 5/8/15
# Receives files to execute under the target program (program_name) from a synchronized queue (q) with
#   a debugger attached (using the Debug class from winappdbg). A callback function (my_event_handler)
#   records the debugger output upon a crash to a unique folder in the Crashers folder.

from winappdbg import Debug, HexDump, win32, Thread, Crash
from time import time
import os, uuid, shutil, Queue, subprocess, ctypes, pickle

class Executor():
    def __init__(self, timeout, program_name, path, queue_in, qn, my_pid, resume):
        self.timeout = timeout
        self.queue_in = queue_in
        self.program_name = program_name
        self.path = path
        self.filename = None
        self.mutator_specs = None
        self.resume = resume
        self.my_pid = my_pid
        self.qn = qn
        self.enterLoop()


    def enterLoop(self):
        #Function in which the Executor sits in a loop executing files under the target program
        fuzzed = 0
        while True:


            if not self.resume:
                #if we didn't just die, get next file from the queue
                try:
                    obj = self.queue_in.get()
                except:
                    #if we can't get a file off of the queue, try again
                    continue
                #poison pill
                if obj == "STOP":
                    self.queue_in.put("STOP")
                    fileout = open(self.path + "State/" + str(self.my_pid), 'w')
                    fileout.truncate()
                    fileout.write("STOP" + " | " + "pls")
                    fileout.close()
                    return

                #otherwise we prepare to debug
                self.mutator_specs = obj['mutator_specs']
                self.filename = obj['filename']

                #then log as a new start
                fileout = open(self.path + "State/" + str(self.my_pid), 'w+')
                fileout.truncate()
                fileout.write(self.filename + " | " + self.mutator_specs + " | " + str(False))
                fileout.close()

            else:
                #we've resumed, so set this back
                self.resume = False
                #if we are recovering from a crash of the executor, load the last file tried
                filein = open(self.path + "State/" + str(self.my_pid), 'r')
                s = filein.read()
                filein.close
                params = s.split(" | ")

                #if we are supposed to be stopped, just return
                if params[0] == "STOP":
                    return

                #if we had already finished executing the previous file before death, this is True
                finishedPrevFile = (params[2] == "True")

                if finishedPrevFile:
                    #if we did finish the file we were on, then continue loop at top to look at queue
                    continue
                else: #if we haven't finished the previous file, do so
                    self.mutator_specs = params[1]
                    self.filename = params[0]

            print "Executing ", self.filename
            #run the file
            x = [self.program_name, self.path + "Mutated/" + self.filename]
            self.simple_debugger(x)

            #then log as done
            fileout = open(self.path + "State/" + str(self.my_pid), 'w+')
            fileout.truncate()
            fileout.write(self.filename + " | " + self.mutator_specs + " | " + str(True))
            fileout.close()

            #try to remove the old mutated file
            try:
                os.remove(self.path + "Mutated/" + self.filename)
            except:
                #if we can't because of a zombie executing process, put in on a queue for later
                self.qn.put(self.filename)
                try:
                    #serialize qn and add to a file in case of crash
                    fileout = open(self.path + "State/qn", 'w+')
                    pickle.dump(self.qndump(), fileout)
                    fileout.close()
                except:
                    pass

            #if we are runnning short on memory because the executor processes are failing to kill children,
            # killing the parent executor will definitely kill the kids and free up memory so that the OS
            # doesn't start randomly killing processes
            stat = MEMORYSTATUSEX()
            ctypes.windll.kernel32.GlobalMemoryStatusEx(ctypes.byref(stat))

            if 90 < int(stat.dwMemoryLoad) and fuzzed != 0:
                #kill if this executor has run more than one process since last death
                print self.my_pid, " committing suicide to save memory"
                os.kill(os.getpid(), 9)

            fuzzed = fuzzed+1
```

```
    def my_event_handler(self, event ):
        # This callback function logs crashes of the target program.
        # Credit to the WinAppDbg website for the extremely detailed tutorial used to construct this function

        # Get the process ID where the event occured.
        pid = event.get_pid()

        # Get the thread ID where the event occured.
        tid = event.get_tid()

        # Find out if it's a 32 or 64 bit process.
        bits = event.get_process().get_bits()

        # Get the value of EIP at the thread.
        address = event.get_thread().get_pc()

        # Get the event name.
        name = event.get_event_name()

        # Get the event code.
        code = event.get_event_code()

        # If the event is an exception...
        if code == win32.EXCEPTION_DEBUG_EVENT and event.is_last_chance() and code not in
(win32.ERROR_SEM_TIMEOUT, win32.WAIT_TIMEOUT):

            # Get the exception user-friendly description.
            name = event.get_exception_description()

            # Get the exception code.
            code = event.get_exception_code()

            # Get the address where the exception occurred.
            try:
                address = event.get_fault_address()
            except NotImplementedError:
                address = event.get_exception_address()
        crash = Crash(event)
        crash.fetch_extra_data(event, takeMemorySnapshot = 2)

        #Log the crash in a new unique crash folder in the Crashers directory
        folder = str(uuid.uuid4())
        os.makedirs(self.path + '/Crashers/' + folder)
        f = open(self.path + '/Crashers/' + folder + '/crashlog.txt', 'w')
        f.write(crash.fullReport(bShowNotes = True))
        f.close()
        f = open(self.path + '/Crashers/' + folder + '/crashsrc.txt', 'w')
        f.write(self.mutator_specs)
        f.close()




def simple_debugger(self, argv ):
    # This function creates a Debug object and executes the target program and file under it
    #   Again, credit to the WinAppDbg website for tutorials on how to use their stuff

    # Instance a Debug object, passing it the event handler callback.
    debug = Debug( self.my_event_handler, bKillOnExit = True )

    maxTime = time()+self.timeout
    # Start a new process for debugging.

    #Execute the program
    currentProcess = debug.execv(argv)

    #wait for the timeout
    while debug and time() < maxTime:
        try:

            debug.wait(1000)



    # Continue if the timeout hasn't been reached
```

```
            except WindowsError, e:
                if e.winerror in (win32.ERROR_SEM_TIMEOUT, win32.WAIT_TIMEOUT):
                    continue
                raise

            try:
                debug.dispatch()
            finally:
                debug.cont()

        # stops debugging, killsall child processes according to WinAppDbg documentation
        #   In practice, this doesn't always work
        debug.stop()



        #if the target process is still alive, kill it. Equivalent to PROCESS_TERMINATE
        try:
            currentProcess.kill()
        except:
            pass

    def qndump(self):
        #this dumps the queue of undeleted files to a list
        qnList = []
        while self.qn.qsize() != 0:
            qnList.append(self.qn.get())

        for item in qnList:
            self.qn.put(item)

        return qnList


class MEMORYSTATUSEX(ctypes.Structure):
    #Taken directly from stackoverflow. See paper bibliography for details.
    #   Gets information about total system memory usage
    _fields_ = [
        ("dwLength", ctypes.c_ulong),
        ("dwMemoryLoad", ctypes.c_ulong),
        ("ullTotalPhys", ctypes.c_ulonglong),
        ("ullAvailPhys", ctypes.c_ulonglong),
        ("ullTotalPageFile", ctypes.c_ulonglong),
        ("ullAvailPageFile", ctypes.c_ulonglong),
        ("ullTotalVirtual", ctypes.c_ulonglong),
        ("ullAvailVirtual", ctypes.c_ulonglong),
        ("sullAvailExtendedVirtual", ctypes.c_ulonglong),
    ]

    def __init__(self):
        # have to initialize this to the size of MEMORYSTATUSEX
        self.dwLength = ctypes.sizeof(self)
        super(MEMORYSTATUSEX, self).__init__()


# Mutator.py
# Dylan Wolff 5/8/15
# Mutator.py is initialized with various fuzzing parameters. It then begins mutating files
#       in the samples folder according to those parameters according to these parameters. These
#       files are mutated by either the mutate, or mutate2 functions according to file size and
#       memory availability. The mutate function uses larger amounts of memory, but is faster
#       when compared to the mutate2 function. Mutated files are placed in the Mutated directory

import random, math, os, Queue, shutil, time, pickle, multiprocessing, copy

class Mutator():

        def __init__(self, path, fuzzFactor, iterations, q, qn, resume):
                self.q = q
                self.path = path
                self.fuzzFactor = fuzzFactor
                self.iterations = iterations
                self.qn = qn

                if resume:
                        self.resumer()
                        #call a fucntion to resume a fuzzing run that is already in progress
                else:
                        #otherwise, we know we have to mutate all sample files

                        #get list of names of all sample files
                        samples = os.listdir(self.path + "Samples")
        #sort them by size small to large
        samples.sort(key = lambda sample: (os.path.getsize(self.path + "Samples/" + sample)))

        #mutate each sample
        for sample in samples:
            self.mutate(self.path, sample, self.fuzzFactor, 0, self.iterations)

        self.q.put("STOP")



def log(self, sample, iters):
    #this function logs the progress of the mutator in case a fuzzing run is interrupted
    fileout = open(self.path + "State/Mutator", 'wb+')
    fileout.truncate()
    fileout.write(sample + " | " + str(iters) + "\n")
    fileout.close
    return

def resumer(self):
    #this function resumes the mutator if a fuzzing run has been interrupted.

    currentFiles = []

    #Find the files all Executor processes will resume at
    currentStates = os.listdir(self.path + "State")
    for thread in currentStates:
        if thread != "Mutator":
            filein = open(self.path + "State/" + thread, 'rb')
            s = filein.read()
            filein.close()
            name = s.split(" | ")[0]
            currentFiles.append(name)

    # Get any Mutated files that weren't deleted after execution
    qnlist = []
    while self.qn.qsize() != 0:
        qnlist.append(self.qn.get(False))
        #
        # Next Resume mutating files from where the Mutator left off
        #
        samples = os.listdir(self.path + "Samples")
        samples.sort(key = lambda sample: (os.path.getsize(self.path + "Samples/" + sample)))


        filein = open(self.path + "State/Mutator", 'rb')
        s = filein.read()
        filein.close()
        params = s.split(" | ")
        #params[0] is name, [1] is number of iters done

        #flip through sorted list of seed files until current is found, discarding all others
        while params[0] != samples[0]:
            samples.pop(0)

        #finish the iterations for the current seed file
        currentsample = samples.pop(0)
        if int(params[1]) != self.iterations-1:
            try:
                self.mutate(self.path, currentsample, self.fuzzFactor, int(params[1]) + 1,
self.iterations)
            except:
                pass

        # do all the rest of the seed files
        for sample in samples:
            try:
                self.mutate(self.path, sample, self.fuzzFactor, 0,  self.iterations)
            except:
                pass

        self.q.put("STOP")

    def mutate2(self, path, fullfilename, fuzzFactor, start, iterations):
        # A mutate function that uses less memory

        ext = "." + fullfilename.split('.')[-1]
        filename = fullfilename[0:len(fullfilename)-(len(ext))]
        filesize = os.path.getsize(path + "Samples/" + filename + ext)
        #get the filesize

        numwrites = int(math.ceil(fuzzFactor * filesize))
        #get the number of writes to do (size/factor)


        for i in range(start, iterations):

            shutil.copy2(path + "Samples/" + filename + ext, path + "Mutated/" + filename + str(i) + ext)
            #copy the sample into the new folder with a new name
            fileout = open(path+ "Mutated/" + filename + str(i) + ext, 'r+b')
            #open the file

            random.seed() #seed with system time to get a clean random number
            randSeed = str(random.random()) #keep track of our seed
            random.seed(randSeed) #seed the thing so that we can recreate mutated file

            for j in range(numwrites):

                rbyte = random.randrange(256)
                #get our random byte
                randloc = random.randrange(filesize)
                # get our random location to write to
                fileout.seek(randloc, 0)
                #find a random location in the file
                fileout.write(chr(rbyte))
                #write the byte as an ascii character to that locale


            self.log(fullfilename, i)
            #close the file
            fileout.close()

            self.q.put({'mutator_specs': filename + ext + '~' + str(randSeed) + '~' + str(fuzzFactor), 'filename' : filename + str(i) + ext})

    def checkTotalMutatedSize(self, path):
        # returns the size of the Mutated folder on the desktop
        totalsize = 0
        currentMutes = os.listdir(path + "Mutated")

        for f in currentMutes:
            totalsize = totalsize + os.path.getsize(path + "Mutated/" + f)

        return totalsize



    def mutate(self, path, fullfilename, fuzzFactor, start, iterations):
        # This function mutates files quickly, but uses a lot of memory
        ext = "." + fullfilename.split('.')[-1]
        filename = fullfilename[0:len(fullfilename)-(len(ext))]

        #Check how much Hard Drive Space we are taking up with the mutated files. If it exceeds ~5
Gigs,
            #we start running out of virtual HD space. Since the Executors need to catch up
anyway,
            # no harm in sitting around and waiting for them before continuing
        filesize = os.path.getsize(path+ "Samples/" + filename + ext)

        while self.checkTotalMutatedSize(path) + filesize >= 5000000000:
            print "HD Capacity Risk, mutator stalling for more space to be cleared"

            for i in range(self.qn.qsize()):
                #while stalling, the mutator checks a queue of old mutated files that need
                # to be deleted and attempts to delete them
                try:
                    s = qn.get(False)
                except:
                    continue
                try:
                    os.remove(path + 'Mutated/' + s)
                except:
                    time.sleep(1)
                    try:
                        os.remove(path + 'Mutated/' + s)
                    except:
                        qn.put(s)
```

```
if filesize > 110000000:
    print "Extremely Large File, switching mutator functions to conserve memory"
    self.mutate2(path, fullfilename, fuzzFactor, start, iterations)
    return

#read in the sample to a string
try:
    filein = open(path+ "Samples/" + filename + ext, 'rb')
    raw = filein.read()
    filein.close()
except:
    print "Memory Error, switching mutator functions"
    del raw
    self.mutate2(path, fullfilename, fuzzFactor, start, iterations)
    return



#get its length
filesize = len(raw)



numwrites = int(math.ceil(filesize * fuzzFactor))
#get the number of writes to do (size/factor)



for i in range(start, iterations):
    # for each mutation iteration

    #copy the file into a list, if we get a memory error because the file is too big, use the version of mutate that doesn't use memory
    try:
        new = list(raw)
    except:
        print "Memory Error, switching mutator functions"
        del raw
        del new
        self.mutate2(path, fullfilename, fuzzFactor, start, iterations)
        return

    random.seed() #seed with system time to get a clean random number
    randSeed = str(random.random()) #keep track of our seed
```

```
            random.seed(randSeed) #seed the thing so that we can recreate mutated file


            for j in range(numwrites):

                rbyte = random.randrange(256)
                #get our random byte
                randloc = random.randrange(filesize)
                #get our random location

                #write the byte as an ascii character to that locale in the list
                new[randloc] = chr(rbyte)


            self.log(fullfilename, i)

            #write to the new file
            fileout = open(path+ "Mutated/" + filename + str(i) + ext, 'w+b')
            fileout.write("".join(new))
            fileout.close

            #print "on the q"
            self.q.put({'mutator_specs': filename + ext + '~' + str(randSeed) + '~' + str(fuzzFactor), 'filename' : filename + str(i) + ext})


            #serialize q and add to a file in case of crash
            fileout = open(self.path + "State/q", 'wb')
            pickle.dump(self.qdump(), fileout)
            fileout.close()

    def qdump(self):
        #this dumps the queue of mutated files to a list
        qList = []
        while self.q.qsize() != 0:
            qList.append(self.q.get())

        for item in qList:
            self.q.put(item)

        return qList
```

```
# Dylan Wolff
# 5/8/15
# reMutate.py is a script that reads in any number crashsrc.txt file from the remutateFolder on
#       the desktop and recreates the mutated file according to specifications within. The original
#       the original sample files need to be placed in the Samples folder, and, upon completion, the
#       recreated mutated file will be in the Mutated folder.

import random, os, math

path = 'C:\Users/Fuzzer/Desktop/'
#path hardcoded to be Desktop of VM

fs = os.listdir(path + 'remutateFolder')
j = 0
for f in fs:
        j = j + 1
        a = open(path + 'remutateFolder/' + f, 'r')
        s = a.read()
        samplename, sampleseed, fF = s.split('~')
        #pull the name of the original sample file, the random seed that gave rise to the mutations,
        # and the mutation percentage
        a.close()

  fuzzFactor = float(fF)

  # now use the seed and the same mutation procedure as the mutate function in Mutator.py
  # to re-generate the mutated file
        random.seed(sampleseed)

        sample = open(path + 'Samples/' + samplename, 'rb')
        s = sample.read()
        sample.close()

        filesize = len(s)
        numwrites = int(math.ceil(filesize * fuzzFactor))

        new = list(s)
        for i in range(int(numwrites)):
                rbyte = random.randrange(256)
                randloc = random.randrange(filesize)
                new[randloc] = chr(rbyte)

        #write to the new file in the Mutated directory
        fileout = open(path + "Mutated/" + str(j) + samplename, 'w+b')
        fileout.write("".join(new))
        fileout.close
```