Mut t on l Fuzz ng to D scover Softw re Bugs nd Vulner b l t es

Dyl n Wolff
Adv sed by Robert S gnor le
0 Sen or Honors Thes s
Boston College Co puter Sc ence Dep rt ent

Abstract:

Recent major vu

Contents:

1 Introduction

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 s 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 othe

Initially, for this project, I looked at two separate fuzzers that were fre 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 respectie

2.2 FuzzServer.py and FuzzClient.py

When run on the server machine, FuzzServer.py first reads in th

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 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 ar attempts to delete them. Once finished, the Mutator puts a "STOP" string on the queue as a poison pill for all other processes, and must be c

3 Fuz

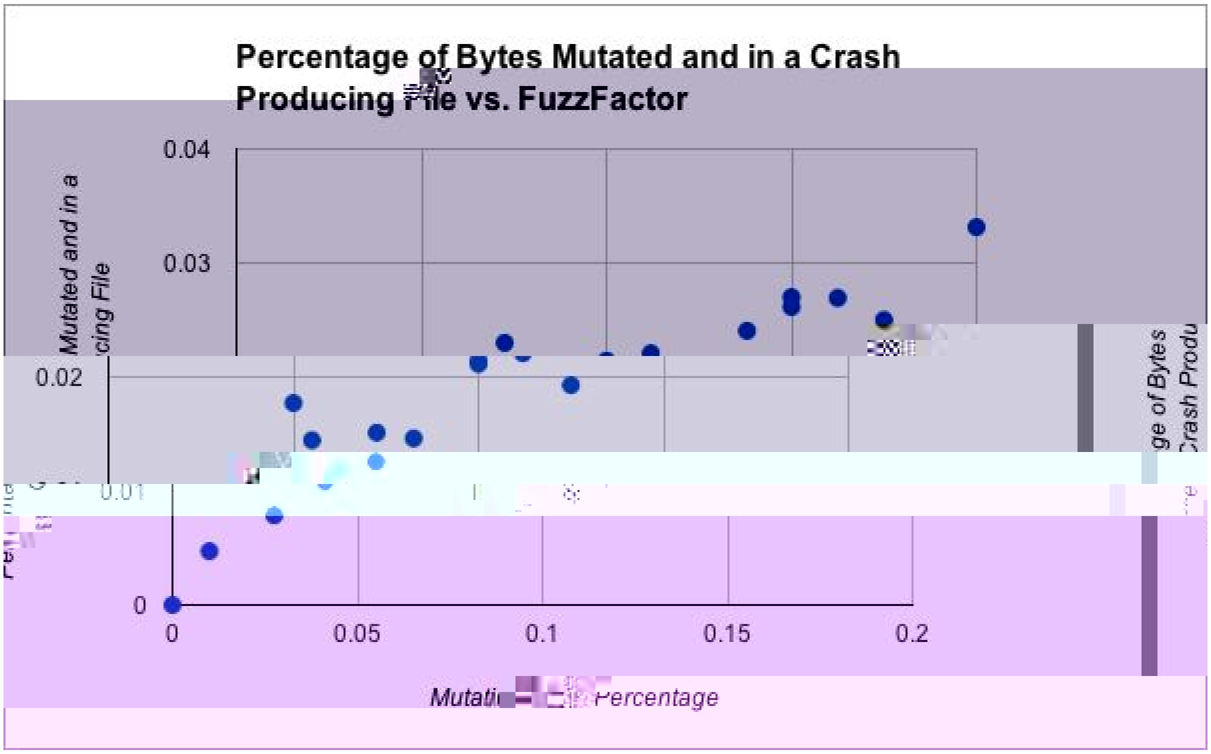
[15]. And VLC and other media players are extremely complex, dealing with many different file formats on many different platforms. This complexity makes

Another file seed file was separately found to cause crashes without any mutations prior to the initial fuz §

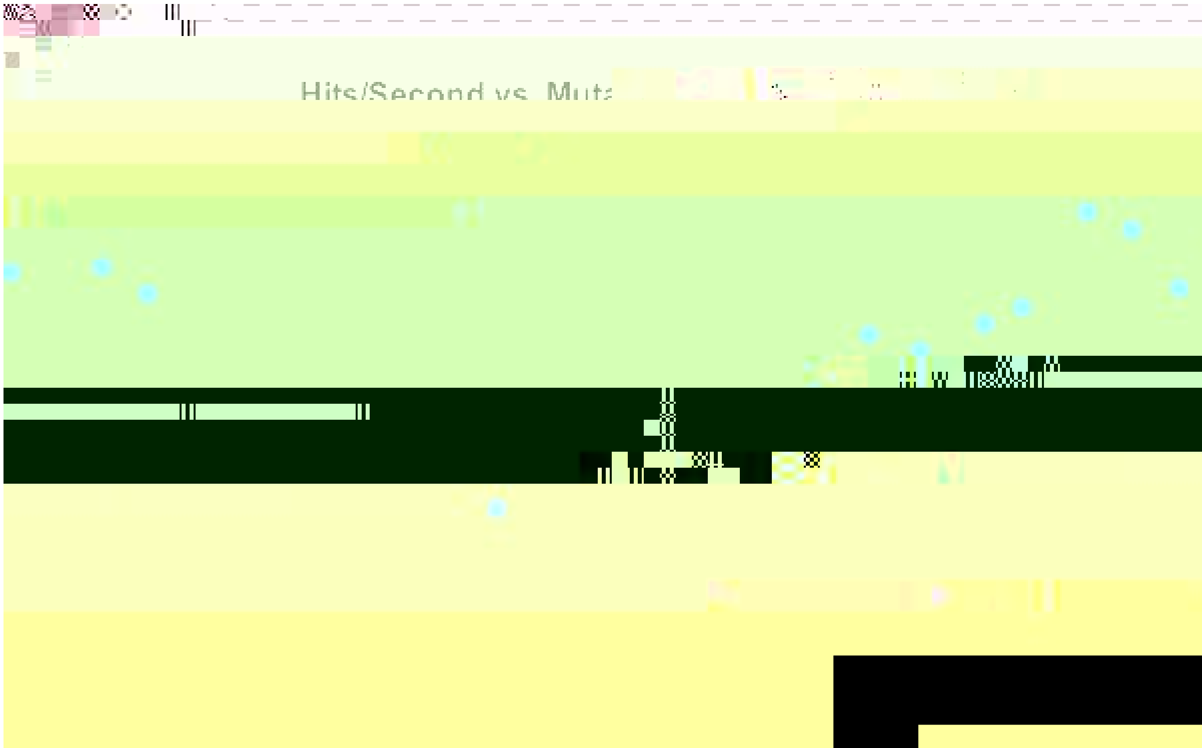
3.3 Optimizing Fuzzing Parameters

There are four parameters in FuzzerConfig.txt that have an extremely large impact on the efficacy of the run 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 increas



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 fo likely in that range.



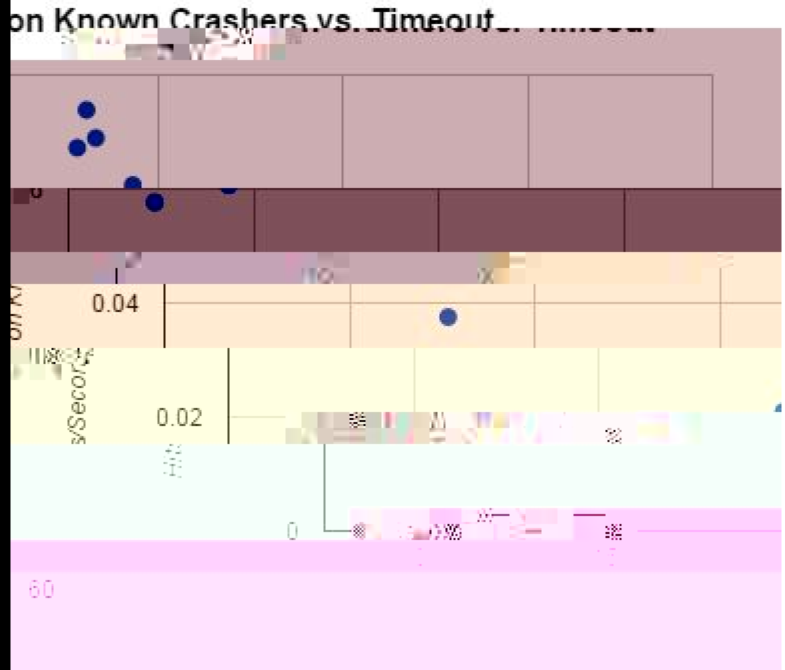
3.3.2 Number of Iterations per Sample File

The number of iterations mǺůay seem at ǵirst to be atively simple paȓraȋmeter to sȇet. In order to fuzz as many files as poss he number of iterations should be set as high as possible such that the ting run can be completed in the time available. However, the number es fuzzed is a combination of two factoȑrs: the number of itDĦÄu research done on seed selection

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 toH accommodate the extra startup time. Furthermore, the type of files affects the amount

p


on Known Crashers vs. Timeout
0.04
0.02
0
60

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, bu

t

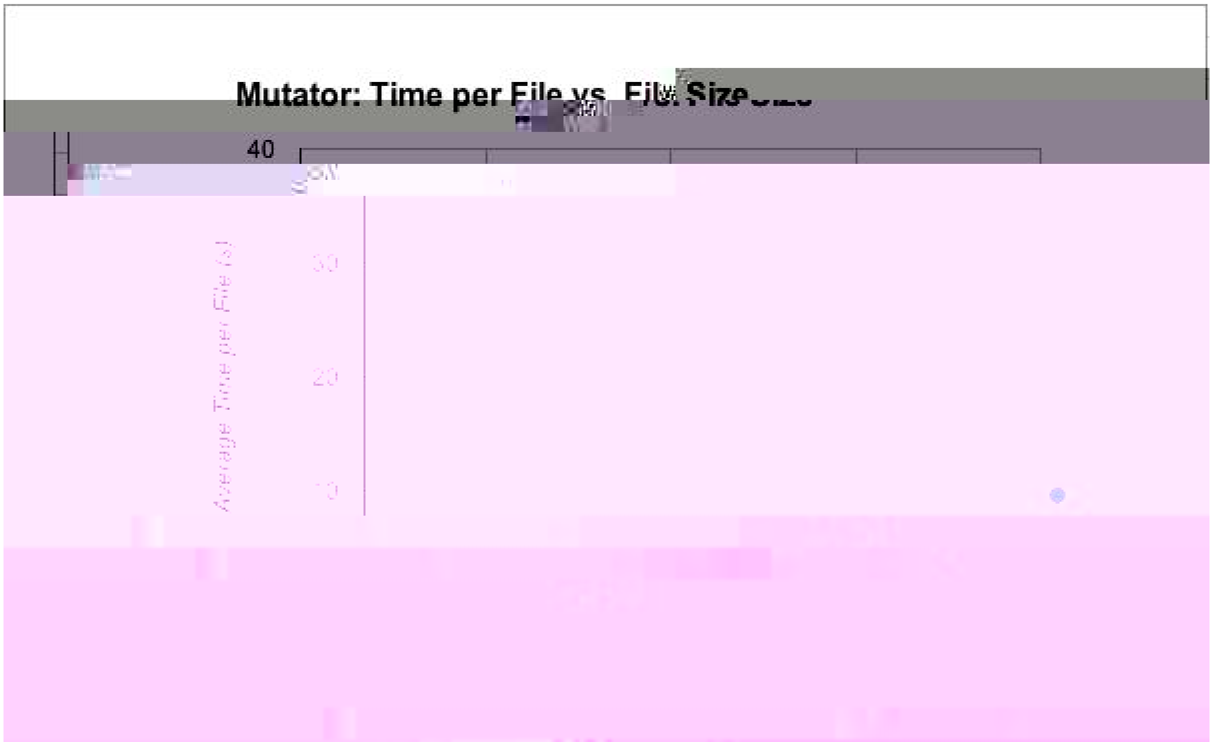
4. Conclusions and Further Work

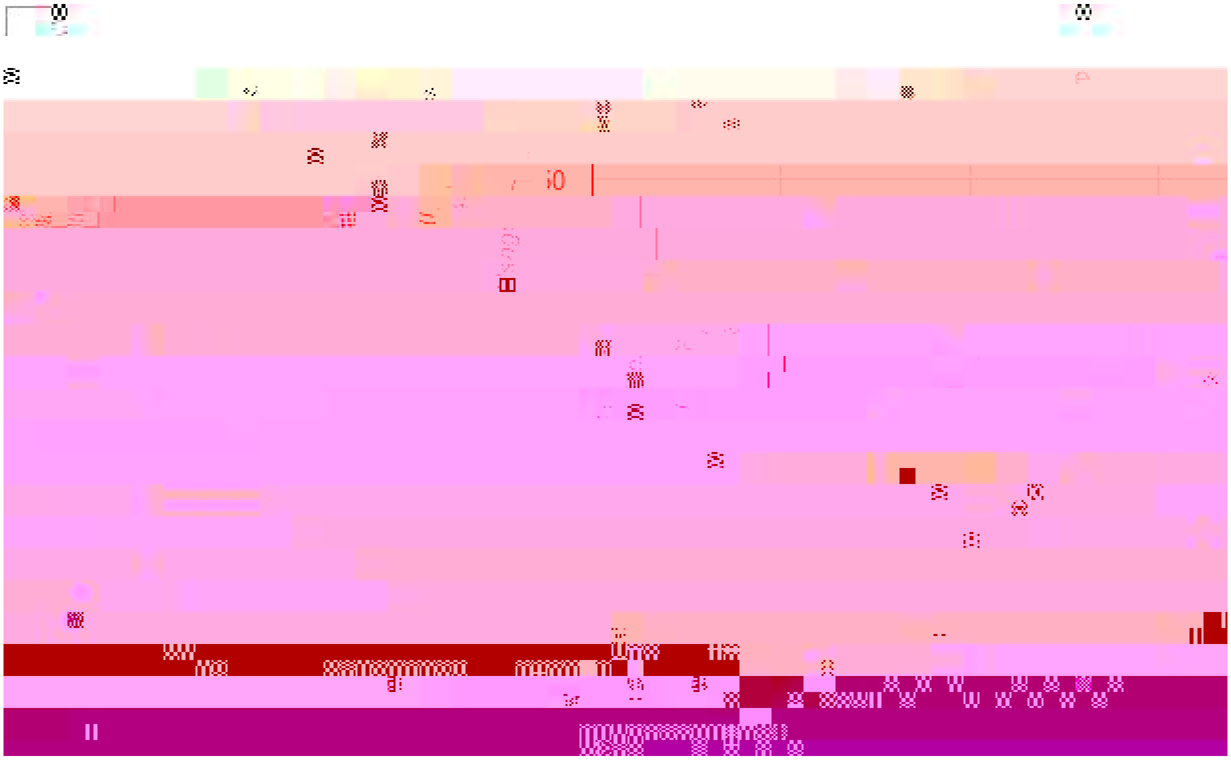
The fuzzing run on VLC yielded a lot of important information. First and foremost, at least ten unique crashes, one of which appears to be highly ex ϴ ,

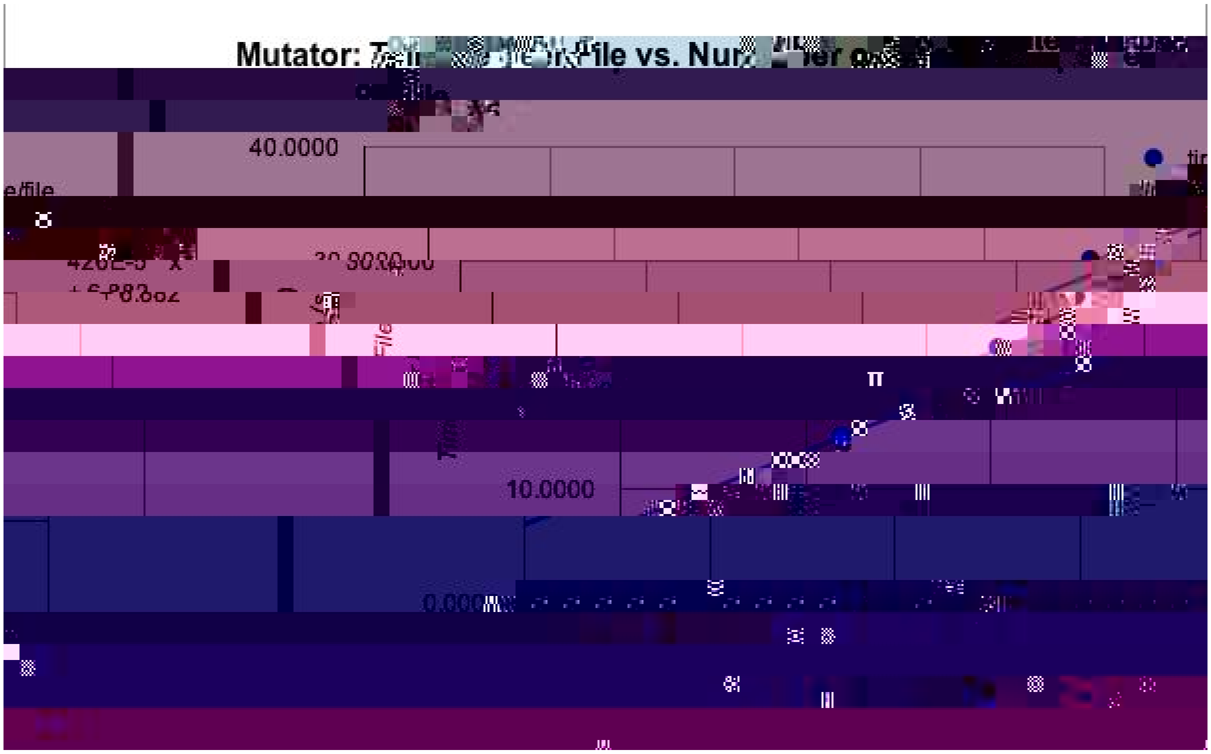
5 References

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, Roland 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

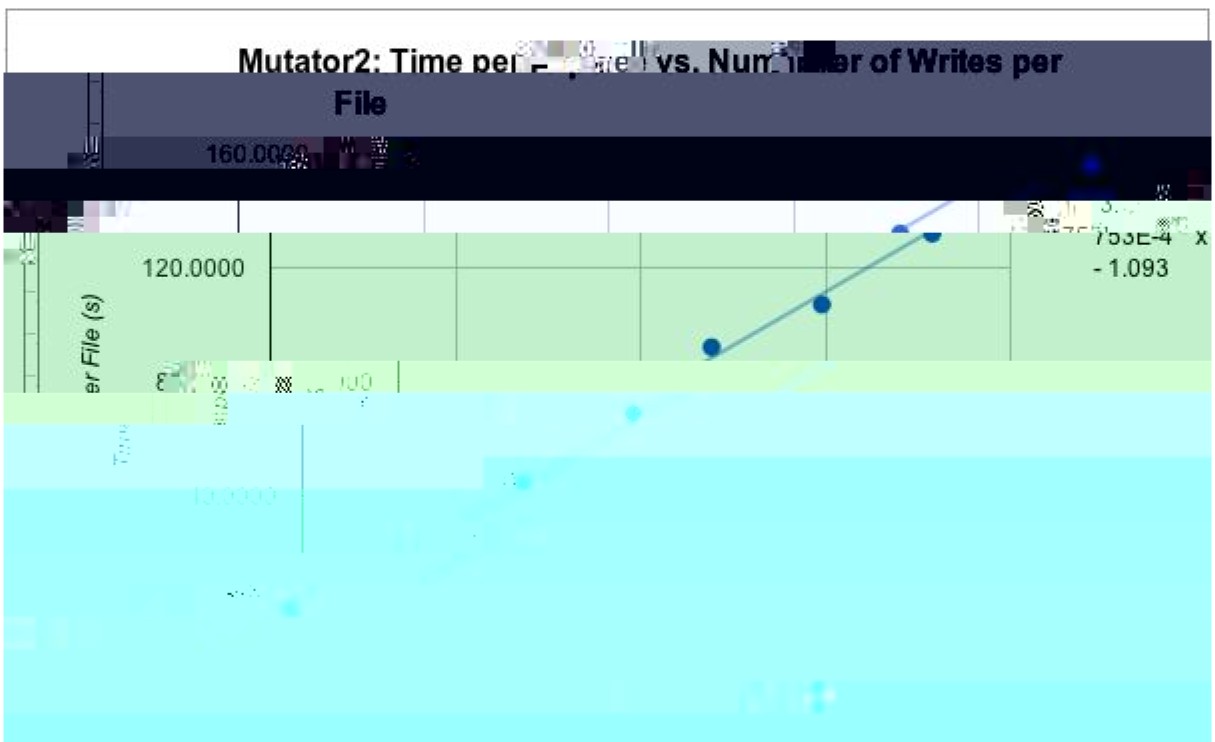
Appendix A: Comparison of Mutator Function Performance







40.0000
10.0000



Mutator2: Time per
of Writes per
File
160.0000
120.0000
- 1.093

Appendix B: Source Code

```
# FuzzClient.py
# Dylan Wolff 5/8/15
# FuzzClient immediately makes a co
```

cl

```
s = reply.split('|')

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

```
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
```

```
#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 me
```

```
#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
```

```
        print "All Clients Finished Fuzzing"

def fuzzReport(p
```

```
if __name__ == '__main__':

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



    f = open(path + 'Fuzz=
```

```
        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\Fuz
```

```
resume))
        process.start()

        #start appropriate number of Executor processes
        eProcesses = [None] * numE
```

```
                  i


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 "e
```

```
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 fo
```

```
process = multiprocessing.Process(target=Fuzzer.Fuzzer, args=((response != 'n'), timeout, fuzzFact
```

```
#poison pill
if obj == "STOP":
    self.queue_in.put("STOP")
    fileout = open(self.path + "State/" + str(self.my_pid), 'w')
    fileout.tru
```

```
        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
```

```
def my_event_handler(self, event):
```

```
        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
        #   A
```

```
            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):        я
    #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_ulo      ή n                    t
```

```
("ullAvailPhys", ctypes.c_ulo
```

```
        #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()
    f
```

```
#
# Next Resume mutating files from where the Mutator left off
#
samples = os.listdir(self.path + "Samples")
samples.sort(key = l
```

```
        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
```

```
        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
```

```
if filesize > 110000000:
        print "Extreme
```

```
random.seed(randSeed) #seed the thing so
```

```
# 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 SampIn
```

```
        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
```