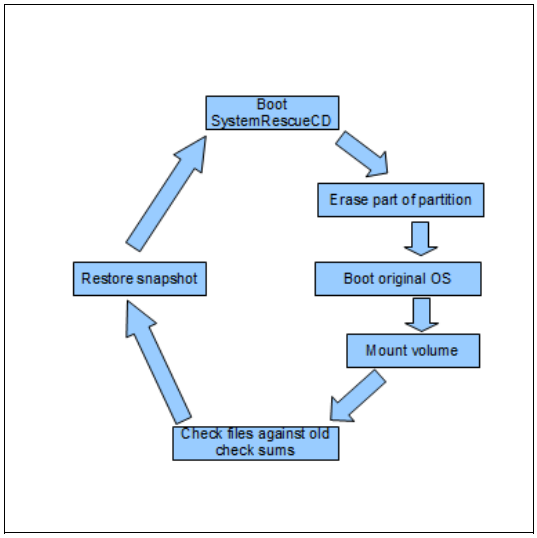
True Crypt Resilience Testing

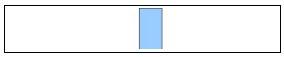
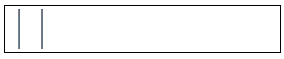
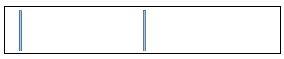
I use True Crypt for encrypting stuff. What do I encrypt? Nothing. Is that clear? Good. Anyway, what if the True Crypt volume gets some flipped bit? I only know some basics about encryption and so wasn't sure if a single flipped bit could cause the entire volume to become corrupted. Best try it out.

The tools used were VirtualBox, True Crypt, SystemRescueCD and a Python script I wrote myself. VirtualBox was very useful because I don't want to enter any commands erasing large chunks of partitions on my actual workstation, because then I am one mistake away from serious data loss. VirtualBox also allowed me to cause serious damage to the True Crypt volume and easily restore it using a snapshot. True Crypt was started up in the test system and made to encrypt a partition separate from the system partition upon which the OS ran. The actual partition is always encrypted but mounting it through True Crypt creates a sort of decrypted window through which one can access the data on that volume. The file system used was NTFS.

SystemRescueCD was used because it allowed me to use the Linux command line tool “dd” which is perfect for my purposes. The Python script was first used to generate check sums for all the files I put on the encrypted volume. These files were from a set of folders containing games which yielded a nice mixture of small and large files. Anyway, the script generated the check sums and put it on the system partition in the virtual machine. The process can be seen to the right.



I tested a variety of different patterns of corruption and found that True Crypt was very resilient even to massive corruption of the encrypted volume. Only one style of corruption made it impossible to mount the volume. Offsets are measured from the beginning of the partition, unless explicitly declared otherwise. Illustrations are not to scale and the last illustration only shows the first few kilobytes of the partition's 6 GB.

Erased 300 MB with an offset of 3000 MB from the beginning of the partition.	Corruption: Could not retrieve file system. 100% data loss.	
Erased 1 MB with an offset of 3000 MB.	Corruption: 1 out of 4624 files (0.02%). 63.7 MB out of 5542.3 MB (1.15%).	
Erased 1 MB with an offset of 100 MB and another 1 MB with an offset of 500 MB.	Corruption: 2 out of 4624 files (0.04%). 458.9 MB out of 5542.3 MB (8.28%).	
Erased 21 MB with an offset of 100 MB and another 21 MB with an offset of 3000 MB.	Corruption: 304 out of 4624 files (6.57%). 122.3 MB out of 5542.3 MB (2.21%).	
Erased 1 KB at 100 locations, each with an offset of 60 KB from one another.	Corruption: 318 out of 4624 files (6.88%). 6961.6 KB out of 5542.3 MB (0.12%).	

I feel confident that the resilience of a True Crypt volume is only slightly lower than that of an underlying file system. That is to say, you would get pretty much the same result performing the tests above on an unencrypted NTFS partition. The resilience can hardly be greater than the file system(that isn't what encryption software does) and there is also a header in a True Crypt volume that is needed for decryption to be possible. If that header is damaged you have 100% data loss.

For those interested in the details of the encryption I can say that the encryption scheme used was AES with Ripemd160 for hashing. The password was *gi4jt9gsJnsgpo5mytXv*, so it was not *ABC* which could theoretically make the encrypted volume more coherent. I know that's not how it works, but I don't expect you to take my word for it. I should admit that I was perfectly willing to believe that the correct decryption of block X was contingent upon the correct decryption of block X-1. I was equally willing to believe that True Crypt did not merely scramble the bits and bytes within a block but could also jumble blocks around. If that had been the case, the large 21 MB erasures would have caused massive damage. It could be argued that the catastrophic loss of data after the 300 MB erasure indicates that some jumbling occurs but I doubt it.

If this document teaches you nothing else, at least it might give you an idea of what I mean when I say something should be tested.

Appendix A – Source code for the Python script verifying files after corruption

```
import os, zlib

oneKB = 1024;
oneMB = 1024*oneKB;
oneGB = 1024*oneMB;

def sum(file):
	try:
		file_handler=open(file, 'rb')
	except IOError:
		print("--* File " + file + " not found *--")
		return "NOT FOUND"
	file_reference = file_handler.read()
	generated_digest = str(zlib.adler32(file_reference))
	file_handler.close()
	return generated_digest

def formatBytes(size):
	if(size > 10*oneGB):
		newSize = size/oneGB;
		return str(round(newSize, 1)) + " GB";
	elif(size > 10*oneMB):
		newSize = size/oneMB;
		return str(round(newSize, 1)) + " MB";
	elif(size > 10*oneKB):
		newSize = size/oneKB;
		return str(round(newSize, 1)) + " KB";
	else:
		return str(size) + " B";
```

```
def checkSums():
	inf = open("C:\Temp\hashsums.txt", 'r')
	recorded = { }
	for line in inf.readlines():
		divided = line.split(' => ')
		thisSum = divided[1].strip()
		thisSize = divided[2].strip()
		recorded[divided[0]] = (thisSum,thisSize)
	inf.close()

	allFiles = 0
	badFiles = 0
	badBytes = 0
	allBytes = 0
	for a,(b,c) in recorded.items():
		currSum = sum(a)
		if(currSum.__eq__("NOT FOUND")):
			badFiles += 1
			badBytes += int(c)
		elif(currSum.__ne__(b)):
			print(a + " has been corrupted.")
			#'" + currSum + "' != '" + b + "'")
			badFiles += 1
			badBytes += int(c)
		allFiles += 1
		allBytes += int(c)
	allSize = formatBytes(allBytes)
	badSize = formatBytes(badBytes)
	byteCorruptionPercent = str(round(100*(badBytes/allBytes), 2)) + "%"
	fileCorruptionPercent = str(round(100*(badFiles/allFiles), 2)) + "%"
	print("Corruption:")
	print(str(badFiles) + " out of " + str(allFiles) + " files which is " + fileCorruptionPercent + ".")
	print(badSize + " out of " + allSize + " which is " + byteCorruptionPercent + ".")


def makeSums(dir):
	out = open("C:\Temp\hashsums.txt", 'w')
	for dir_path, sub_dirs, files in os.walk(dir):
		for file in files:
			absPath = dir_path + os.sep + file
			generated_digest = sum(absPath)
			print(file + " => " + generated_digest)
			size = os.stat(absPath).st_size
			out.write(absPath  + " => " + generated_digest + " => " + str(size) + "\n")
	out.close()

#makeSums("D:\Programming")
checkSums()
```

Appendix B – Command equivalent to that I used to carry out run 5

```
#!/bin/bash
for i in {1..100}
do
	dd if=/dev/random of=/dev/sdb bs=1K seek=$(($i*60)) count=1
done
```