High-Speed Data Shredding using Python

David Leong
School of Infocomm, Republic Polytechnic
david_leong@rp.edu.sg

Eugene Yeo
School of Infocomm, Republic Polytechnic
91315@myrp.edu.sg

Abstract

In recent years, backup and restore is a common topic in data storage. However, there's hardly anybody mention about safe data deletion. Common data destruction methodology requires the wipe operation to fill the disk with zeros, then with random data, and then with zeros again. Three passes are normally sufficient for ordinary home users. On the down side, such algorithms will take many hours to delete a 2TB hard disk. Although current Linux utility tools gives most users more than enough security and data protections, we had developed a cross-platform standalone application that could expunge all confidential data stored in flash drive or hard disk. The data shredding software is written in Python, and it could overwrite existing data using user-defined wipe algorithm. This software project also explores the technical approaches to digital data destruction using various methodologies defined in different standards, which includes a selection of military-grade procedures proposed by information security specialists. The application operates with no limitations to the capacity of the storage media connected to the computer system, it can rapidly and securely erase any magnetic mediums, optical disks or solid-state memories found in the computer or embedded system. Not only does the software comply with the IEEE T10/T13 specifications, it also binds to the number of connectivity limited by the SAS/SATA buses.

Keywords*: data security, data destruction standards, data wiping algorithms.*

1. Introduction

Many old hard disks are usually filled up to the brim with data, and users may habitually think about buying a new and bigger disk when its capacity is almost full. To reduce the impact on their budgets, they could either choose to expand their local data storage capacity by reusing existing old hard disks, or they may decide to sell them via one of the many online auction sites, such as *eBay* or *Gmarket*.

In the course of time, users will have left numerous traces of personal and business data on the disk (Schneier, 1996). Of course, they would prefer to remove applications, confidential documents, letters, dismissal notices, emails, personal photo, video albums, financial statements, and access credentials from the disk (Morris, 2011).

A lot of users might already know how to delete a file or directory in their computers. All operating systems adopt the analogous approach by permitting users to click and select the file or directory icon and then press the [Delete] key on their keyboard to send the file off to the *recycle bin*. Some users might think that file deletion from a command line is much safer than deleting directly from the file icons, presuming that the deleted files will not be resurrected again in the *recycle bin*. For instance, *Windows PowerShell* has a one-liner at the command line that could make file deletion handy:

```
remove-item -path C:\test\*.* -force
```

Such file deletion techniques would permanently erase the file and folder from the context of their file system, and that is what many of the users might believe. Even if the user could manage to empty the files or folders from the *recycle bin*, these techniques simply remove entries in the file system journals. Without these entries, the file system has no way of locating the data. However, this does not mean that the data would just vanish in the hard disk. The content of the file is still resided somewhere in the hard disk platters. Old data do not disappear abruptly until the user starts writing new data over it.

To elaborate the analogy further, the `remove-item` command above simply deletes the *test* directory, its subdirectories, and all files it contain without prompting the user for confirmation. Strictly speaking, the command merely deletes the entry in the file allocation table associating with the file system. What really happens is that the operating system removes the *inode* from the *inode table*, thus freeing up some space in their local storage (Bach, 1986). Unfortunately, deleting files in this manner do not radically wipe out existing data from the hard disk. Physically removing the data stored in the space where the *inode* pointing to simply requires too much computational effort for the file system.

Contrary to many beliefs, reformatting of hard drives does not eradicate data from the computers either. For instance, when a hard disk is formatted, a modern operating system will just create a *superblock* to store information that protects the integrity of the file system (Deitel et al., 2004), and repartitioning just changes the entries in the partition table on the user's disk. The data may be lost somewhere in data nirvana, but it is definitely still reside on the disk media. Unskilled hackers are able to retrieve files (even those deleted a long time ago) using off-the-shelf data recovery tools that are easily available on the Internet. Many of the vulnerabilities were the result of bypassing the prevention mechanism (Schneier, 2000). Given this reality, factual response to data security is predominant.

In retrospect to expunging digital data in a way that is irrecoverable (NIST, 2006 & Wei, et al., 2011), users are highly recommended to perform *secure erase* procedure (also known as "disk wiping" or "data shredding") on old hard disks that may be eventually reused. The aspiration of our project is to develop a cross-platform data wiping application software using Python programming language. The software should be capable of sanitizing multiple hard disks or flash media devices simultaneously. The software should also be able to overwrite any existing data using predefined data destruction algorithms, which include a selection of military-grade procedures proposed by information security specialists (Gutmann, 1996; Roebuck, 2010).

The ultimate goal of our data destruction software is to annihilate all previous data entries and partitions, so as to make it impossible for malicious user to recover any readable data that was previously stored in the hard disk or non-volatile memory. In order to corroborate common server-board and mother-board architectures, the software shall support up to 4 PATA hard drives as well as 8 SATA/SCSI/USB hard drives or flash media devices. For versatility, the software executable should be small enough to be available in a bootable flash media device, or CD-ROM/DVD-ROM disc. It must be capable of supporting a variety of data storage devices with sector/block size of 512 bytes or greater. The software should also provide a built-in *Disk Viewer* utility for data read-back and verifications.

The remainder of this paper is organized as follows: Section 2 briefly describes our implementation of the Graphical User Interface (GUI). Section 3 describes the programming techniques used for extracting system information. Section 4 describes a variety of different data destruction methods and their wipe patterns. Section 5 describes the `numpy` method that generates pseudo-random numbers for scribbling existing data. Section 6 presents the generic file I/O methods for reading back raw data stored in a disk sector. Section 7 presents the prevailing technique to handle concurrency. Section 8 presents the `wx` methods used for displaying application status and gauge bars. Section 9 describes briefly our error logs and reports. Section 10 describes the techniques for optimizing wipe performance. Section 11 concedes our software limitations and describes future explorations, and Section 12 presents our conclusion.

2. Graphical User Interface

In the implementation process, the entire software development is divided into several software modules to simplify our development cycle; each was developed simultaneously by different individual. There are currently a handful of cross-platform Python frameworks that can used to develop graphical desktop applications, such as `PyGTK`, `wxPython` and `PyQT`. We have espoused `wxPython`. It is a set of Python bindings to the `wxWidgets` library, which is a prevailing cross-platform C++ application framework that can be used to create our GUI. What sets `wxPython` apart is the use of the same controls and themes as the rest of the system. Another benefit of using `wxPython` is that it can rapidly developed on one operating system platform and deployed to another with little or no changes to the source code (Precord, 2010).

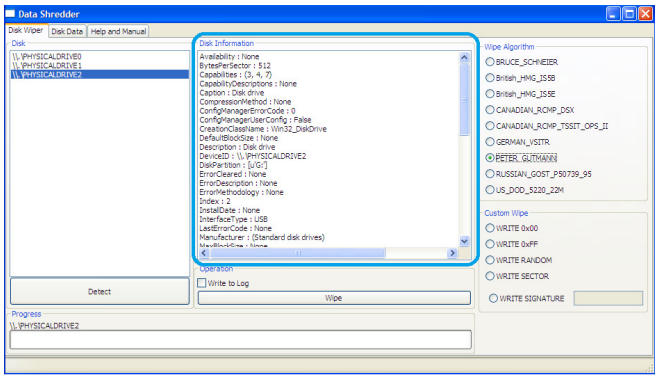

Figure 2.1. Selecting physical devices for data destruction

The main GUI screen of our data shredder software is shown in Figure 2.1. On the left pane, it shows a list of storage devices mounted onto the operating system. Detection and recognition of storage devices, as well as the presentation of device information are emphasized in the next section. The right pane shows a list of available data destruction methods to be selected by user. We will be describing their unique wipe patterns and number of wipe iterations in Section 4. In order to improve the software usability, we have included the status and progress bar to be elaborated further in Section 8. In addition, a checkbox is included to permit user to select whether reports should be generated after each wipe operation (see Section 9). This option is useful for security auditing as well as monitoring the performance of disk I/O.

3. Extracting Device Information

Once the software application is launched, it would scan through the computer system spontaneously to identify the type of operating system used. The [Detect] button in the main GUI will registers all data storage devices that are connected to the system. Figure 2.1 indicate a list of storage media mounted onto the operating system. The selection panel on the left pane displays the type of storage devices available, and user is able to select individual storage devices to view its device information. The information displayed includes the hard drive model, serial number, firmware revision, capacity (in gigabytes), total number of cylinders, heads, sectors, sector size, total number of sectors available, maximum value of the Logical Block Addressing (LBA), and many more.

```
1  def fetchDiskInfo():
2      c = wmi.WMI()
3      win32DiskDriveDict = {}
4
5      for diskDrive in c.query("SELECT * FROM Win32_DiskDrive"):
6          diskDriveDict = {}
7          diskDriveDict[u"DiskPartition"] = []
8          win32DiskDriveDict[diskDrive.DeviceID] = diskDriveDict
9
10         # Get Win32_DiskDrive Properties
11         for propertyName in sorted(list(diskDrive.properties)):
12             diskDriveDict[propertyName] = getattr(diskDrive, propertyName, '')
13
14         # Get Win32_DiskPartition Instance
15         for diskPartition in sorted(list(
16             diskDrive.associators("Win32_DiskDriveToDiskPartition"))):
17
18             # Get Win32_LogicalDisk Instance
19             for logicalDisk in sorted(list(diskPartition.associators("Win32_LogicalDiskToPartition"))):
20                 # Special Lookup for Disk Partition Letter
21                 diskDriveDict[u"DiskPartition"].append(logicalDisk.DeviceID)
22
23     return win32DiskDriveDict
```

Listing 3.1. Partial code to extract disk drive information from the Win32 system

For Windows platform, we have used the Windows Management Instrumentation (WMI) library to extract system information from the storage device. WMI is Microsoft's implementation of the Web-Based Enterprise Management (WBEM) and Common Information Model (CIM) standards. It is basically a set of extensions to the Windows Driver Model that provides an operating system interface to acquire system information. We have installed `WMI v.1.4.9` package on top of the `pywin32` extension. This module is a

lightweight wrapper that communicates to the WMI API. Listing 3.1 shows the partial code that extracts disk drive information using WMI.

For Linux platform, we used `fdisk` and `lshw` commands to extract detailed information of disk and hardware configuration in the computer system. `lshw` could exact memory configuration, firmware version, mainboard configuration, CPU version and speed, cache configuration, bus speed on DMI-capable x86 or IA-64 systems. It currently also supports PCI/AGP, CPUID, IDE/ATA/ATAPI, PCMCIA, SCSI and USB. Detailed usage of these commands can be found in the Linux `man` page. These Linux commands are executed using the Python's `subprocess` module, and we have used the `regular expression` module to search and extract vital disk drive information to be displayed on the center pane (see Figure 2.1).

4. Supporting Numerous Data Destruction Standards

The *Wipe Options* screen in Figure 4.1 consist more than 10 optional wipe methods incorporated into our data destruction software. The option allows the user to select various type of wiping algorithm that they desired. Beside customized settings, the software supports many standard wiping methods listed in the following sub-sections.

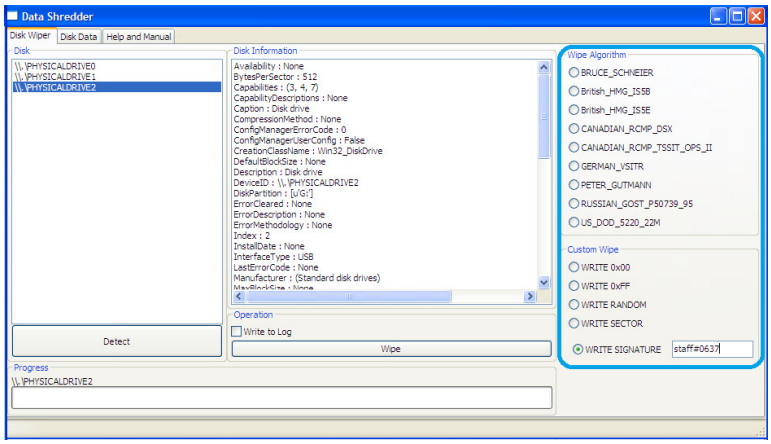

Figure 4.1. Supporting large variety of data wiping algorithms

4.1 British HMG IS5 - Baseline (1 pass + 1 verification pass)

This baseline scheme allows the data sectors in the storage device to be overwritten with zeroes. This wiping method also does a verification pass to ensure that the data written is correct.

4.2 British HMG IS5 - Enhanced (3 passes + 1 verification pass)

This enhanced scheme is a three pass overwriting algorithm. In the first pass, it overwrites all the data sectors in the storage device with `0x00`. In the second pass, it overwrites the entire data sectors again with `0xFF`. In the last pass, it overwrites all the data sectors in the storage device with pseudo-random numbers. This wiping method also does a verification pass after the third pass to ensure that the data overwritten are correct.

4.3 Russian GOST P50739-95 (2 passes)

This Russian standard allows the data sectors in the storage device to be overwritten with a single pass of zeroes (`0x00`), followed by another pass with pseudo-random numbers.

4.4 U.S. Standard, DoD 5220.22-M (3 passes)

The National Industrial Security Program Operating Manual, issued to the US Department of Defense, Department of Energy, and other US government agencies specifies standards for the clearing, and sanitising of data classified as confidential, secret, and top secret.

Under this standard, data may be cleared by writing any bit pattern to the entire disk once. Disks are sanitised by writing a different bit pattern to the disk on each of three passes (Schneier, 1996). However, drives containing top secret data are not permitted to be sanitised in this manner; they must be physically destroyed, or the disks subjected to degaussing, scrambling completely the magnetic patterns used to store data on the disk, rendering the drive itself inoperable.

U.S. Department of Defense specifies three passes extended character rotation overwrite algorithm in the DoD 5220.22-M specification. This Total Privacy shredding method conforms to these overwriting standards as well as method 'd' of the Cleaning and Sanitation Matrix (DoD, 2006).

4.5 Canadian RCMP DSX Method (3 passes + 3 verification passes)

The DSX method is a three pass overwriting algorithm. In the first pass, it overwrites all the data sectors in the storage device with zeroes. In the second pass, it overwrites the entire data sectors again with ones. In the third pass, it overwrites all the data sectors in the storage device with pseudo-random numbers. After each pass, the values overwritten in the data sectors are verified to ensure integrity (RCMP, 2003).

4.6 Canadian RCMP TSSIT OPS-II (7 passes + 1 verification pass)

This method is a seven passes overwriting algorithm with three alternating patterns of zeroes and ones followed by the last pass with random characters. This wiping method also does a verification pass to ensure that the data overwritten in the final pass is correct.

4.7 German VSITR (7 passes)

Similar to previous method, the German standard overwrites each data sector in the storage device with three alternating patterns of zeroes and ones, followed by the seventh pass with random character. However, no verification on the overwritten data is needed.

4.8 Bruce Schneier's Algorithm (7 passes)

This method offers a seven pass overwriting algorithm. The first pass with all ones, the second pass with all with zeroes and then five more passes with a cryptographically secure pseudo-random sequence (Schneier, 1996).

4.9 Peter Gutmann's Algorithm (35 passes)

Peter Gutmann suggested this method to ensure that the recovery of data can be made as difficult as possible for an attacker by offering the 35 overwrite passes algorithm (Gutmann, 1996). Table 4.1 shows a series of wipe patterns in binary and hexadecimal notations. This algorithm is slow, but extremely reliable. It requires a lot of patience on the part of the user.

Pass	Overwritten data	
	Binary Notation	Hex Notation
1	*Random*	*Random*
2	*Random*	*Random*
3	*Random*	*Random*
4	*Random*	*Random*
5	01010101 01010101 01010101	55 55 55
6	10101010 10101010 10101010	AA AA AA
7	10010010 01001001 00100100	92 49 24
8	01001001 00100100 10010010	49 24 92
9	00100100 10010010 01001001	24 92 49
10	00000000 00000000 00000000	00 00 00
11	00010001 00010001 00010001	11 11 11
12	00100010 00100010 00100010	22 22 22
13	00110011 00110011 00110011	33 33 33
14	01000100 01000100 01000100	44 44 44
15	01010101 01010101 01010101	55 55 55
16	01100110 01100110 01100110	66 66 66
17	01110111 01110111 01110111	77 77 77
18	10001000 10001000 10001000	88 88 88

Pass	Overwritten Data	
	Binary Notation	Hex Notation
19	10011001 10011001 10011001	99 99 99
20	10101010 10101010 10101010	AA AA AA
21	10111011 10111011 10111011	BB BB BB
22	11001100 11001100 11001100	CC CC CC
23	11011101 11011101 11011101	DD DD DD
24	11101110 11101110 11101110	EE EE EE
25	11111111 11111111 11111111	FF FF FF
26	10010010 01001001 00100100	92 49 24
27	01001001 00100100 10010010	49 24 92
28	00100100 10010010 01001001	24 92 49
29	01101101 10110110 11011011	6D B6 DB
30	10110110 11011011 01101101	B6 DB 6D
31	11011011 01101101 10110110	DB 6D B6
32	*Random*	*Random*
33	*Random*	*Random*
34	*Random*	*Random*
35	*Random*	*Random*
-	-	-

Table 4.2. A series of wipe patterns in Peter Gutmann's algorithm

4.10 Custom Setting

This option allows users to define their own method to erase a drive. Users may select any of the 4 different predefined bit patterns (0x00, 0xFF, pseudo-random numbers or LBA) to be used, or they could opt for custom signature up to a maximum of 30 ASCII characters.

Increasing the number of passes will increase the security of the wipe process. However, it is unlikely that any customized method would be regarded as sufficient to sanitize the drive. The primary purpose is to provide a quick and unique data destruction solution for users with unclassified data.

5. Generating Pseudo-random Numbers

It is interesting to note that the US Department of Defense had stipulates physical destruction of magnetic media containing highly confidential data in DoD 5220.22-M NISPOM in 2006 (DoD, 2006). The algorithms mentioned in Sections 4.1 – 4.10 overwrite the sectors once or multiple times with specific data patterns. In order to generate genuinely random data, and to make it impossible to subtract this data from the read signals, most approaches use random number generation.

```
1  import time
2  from subprocess import call
3
4  BLOCKSIZE = 8192
5  DEVICE = "/dev/sda1"     # Storage device to be deleted
6
7  print "Device to delete: " + DEVICE + "\r\nBlock size: " + str(BLOCKSIZE)
8  print time.strftime("%d %B %Y %H:%M:%S", time.localtime()) + " 1st pass"
9  call("dd if=/dev/zero of=" + DEVICE + " bs=" + str(BLOCKSIZE), shell=True)
10 print time.strftime("%d %B %Y %H:%M:%S", time.localtime()) + " 2nd pass"
11 call("dd if=/dev/urandom of=" + DEVICE + " bs=" + str(BLOCKSIZE) , shell=True)
12 print time.strftime("%d %B %Y %H:%M:%S", time.localtime()) + " 3rd pass"
13 call("dd if=/dev/zero of=" + DEVICE + " bs=" + str(BLOCKSIZE) , shell=True)
14 print time.strftime("%d %B %Y %H:%M:%S", time.localtime()) + DEVICE + " deleted"
```

Listing 5.1. An example of Python script that could wipe a SATA hard drive in 3 passes

Linux operating system gives application developers a high level of security based on special device files such as `/dev/urandom`, which creates simple random data. `/dev/zero` file gives developers any number of null bytes (`0x00`). The Python script shown in Listing 5.1 first fills the disk with zeros, then with random data, and lastly with zeros again. For home users, running this script should be more than sufficient. However, military and corporate data will typically require at least 7 or more passes in order to elude any possibility of data remenance (Gutmann, 1996; Gutmann, 2001).

Instead of using special Linux device files, Python provides `random.random` and `numpy.random` libraries for cross-platform developers to generate random numbers. Both of these libraries use the *Mersenne twister sequence* to generate pseudo-random numbers, and both methods are completely deterministic. The reasons we have adopted the `numpy.random` library is because it contains a few additional probability distributions commonly used in scientific research, as well as a couple of convenience functions for us to generate random data. However, note that these libraries are definitely unsuitable for any serious cryptographic usage.

6. Viewing Data Sectors

Our application provides a built-in *Disk Viewer* utility for data read-back verifications. For auditing purpose, we have indulged functionality for user to view the content of any sectors before and after data wiping. For instance, the content of Volume Boot Record (VBR) from a 512 MB USB flash drive, as shown in Figure 6.1, can be viewed by our data shredder software. However for security reasons, we did not provide functionality for user to edit data at specific byte position. A better and more reliable approach to security auditing is to compute and verify the hash value of the device after each wipe operation.

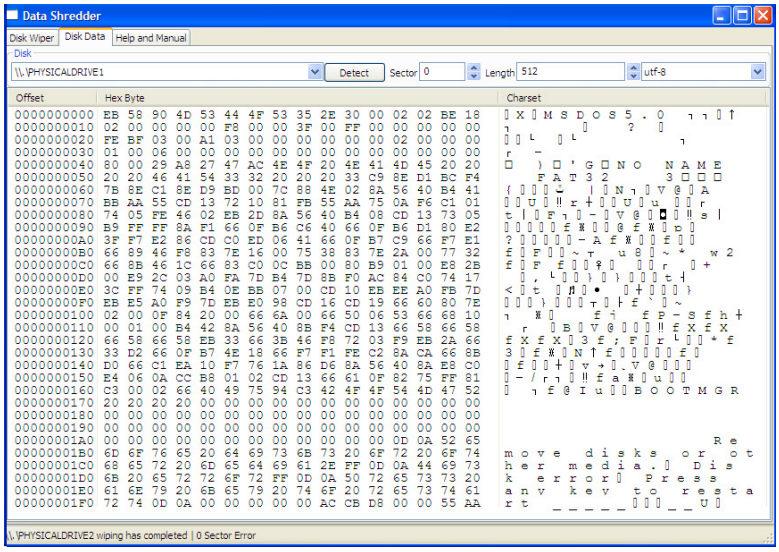

Figure 6.1. Display content of FAT32 VBR in hexadecimal and ACSII encoding

For Linux operating system, reading back the content from a storage device can be rather straight forward using the `dd` command. An example of Python script for LBA/sector search and data read-back can be accomplished in Listing 6.2 below.

```
1   MAX = 1000
2   DEVICE = "/dev/sda1"
3   from subprocess import Popen, PIPE
4
5   for i in range MAX:
6       cmd = "dd if=" + DEVICE + " skip=" + str(i)
7       cmd += " bs=512 count=1 2>/dev/null | hexdump -v -e '%_p'"
8       p= Popen(cmd, stdout=PIPE)
9       output = p.communicate()[0]
10      if output:
11          print "Sector " + i + ": " + output
```

Listing 6.2. An example of Python script to capture the content in specific data sectors

However, rather than evoking `dd` command from `popen()`, it is more effective to read raw data directly from disk sectors using methods available from the Python's built-in file objects. The following are 3 lines of Python code to acquire the sector number, seek to the appropriate LBA and read 512 bytes of data from the precise offset.

```
sector = self.view.sector.GetValue()
self.disk.seek(512*sector)
data = self.disk.read(512)
```

7. Managing Concurrency in Data Wiping

Due to slow disk I/O incurred on individual storage device during a wipe process, we have implemented our wipe operations using concurrency approach. In Python, multiple processes and threads are supported by the `multiprocessing`, `thread`, or the newer `threading` module. Threading is generally a technique to decouple tasks that are not sequentially

dependent. We have adopted the `threading` module to fork multithreads so that our application could handle its wipe operations on multiple storage devices concurrently.

```
#!/usr/bin/env python
import threading

# Thread that will trigger the wiping algorithm
class WiperThread(threading.Thread):
    def __init__(self, parent, disk, totalSector, algorithm, **kwargs):
        self.parent = parent
        threading.Thread.__init__(self)
        self.algorithm = algorithm
        self.disk = disk
        self.totalSector = totalSector
        self.kwargs = kwargs
        self.start()

    def run(self):
        """Execute The Algorithm"""
        print "running %s"%self.algorithm
        self.algorithm(self.disk,self.totalSector,**self.kwargs)
        wx.CallAfter(self.parent.diskWiped,self.disk)
        print "COMPLETE %s"%self.algorithm
```

Listing 7.1. Partial code for forking multithreads to handle concurrent wipe operations

8. Displaying Progress Status and Gauge Bars

Status bar is a common component found at the bottom of the main windows content area. We have created our `ProgressStatusBar` class by creating a subclass of `StatusBar`. We use it to display short messages or status of our wipe operation. For each storage device discovered by the data shredding application, we create a `Gauge` bar to show the progress during a long-running wipe operation, as illustrated in Figure 8.1. A timer is also used for updating the `Gauge` bar.

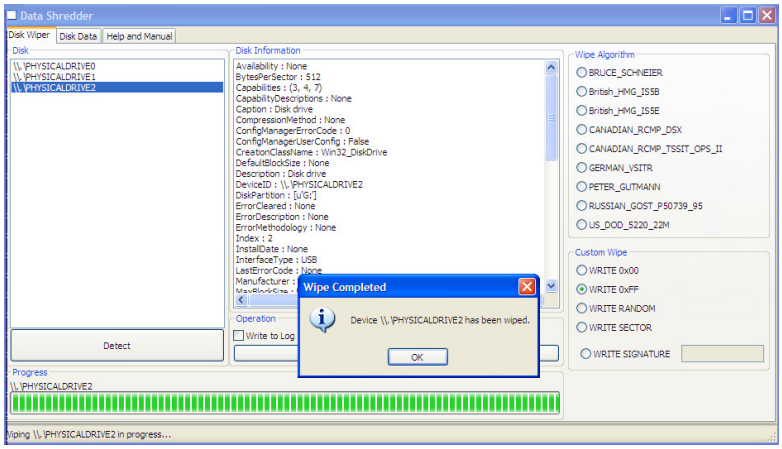

Figure 8.1. Display of progress bar at the end of a wipe operation

The calculations to predict the estimated completion time for the wipe operation and the percentage of task completion as indicated in the gauge bar are listed as follows:

```
Estimate total wipe time (in seconds) = (no_of_passes *
total_sectors) / (((curr_round - 1) *
total_sectors) + curr_sector_offset) * (curr_time - start_time)

Percentage of task completed (in %) = (((curr_round - 1) *
total_sectors) + curr_sector_offset) /
(no_of_passes * total_sectors) * 100
```

9. Generating Reports

Our data shredder application is capable to generate 2 different reports. The first report is inaugurated by the user's option to log all successful wipe operations, whereas the second report will records any errors encountered during the wipe process. Some users may find it auspicious to keep track of their wipe frequency, thus our application provides them with an option to generate a detailed log file after each wipe process. However, the error reports are mandatorily generated to indicate precisely the wipe time and location of the bad sectors, with conjecture that the *P-list* and *G-list* provided in the hard disk defects table may no longer be accurate.

```
1  class WriteLogFile:
2      def __init__(self):
3          self.model = Model()
4
5      ......
6
7      def writeLog(self, endTime):
8          if self.writeBoolean:
9              diskModel = self.diskInfo[4].split(":")
10             curtime = datetime.datetime.now()
11             curTimeStr = curtime.strftime("%Y-%m-%d %H %M %S")
12             totalTime = endTime - self.startTime
13
14             FILE = open(self.directory+"/"+diskModel[1].lstrip()+" "+curTimeStr+".txt","w")
15
16             for info in self.diskInfo:
17                 FILE.write(info+"\n")
18
19             FILE.write("Wipe Option : "+self.algorithm+"\n")
20             FILE.write("Time Taken : "+str(totalTime)+" seconds\n")
21             FILE.close()
22             print "Log File Created!"
23         else:
24             print "No Log File is Created!"
```

Listing 9.1. Partial code to generate log files

Our application is designed to operate on working hard disk drives with a possibility of multiple bad sectors. The data shedder software will queries the storage device directly for drive parameters, read a drive in 64 sectors (32 KB) block and would skip the entire 64 sectors block if an error is found. Each storage device will preserve its individual log and error files. These files are distinct by *log* and *err* file extensions.

10. Performance Testing and Optimization

Performance testing of our initial prototype accomplished single pass with wipe time that is 5 times longer than the original version developed by CBL Data Recovery (S) Pte Ltd. We

have conducted our analytical studies and identified the performance bottleneck to be at the disk I/O. Wiping data sector-by-sector is too sluggish.

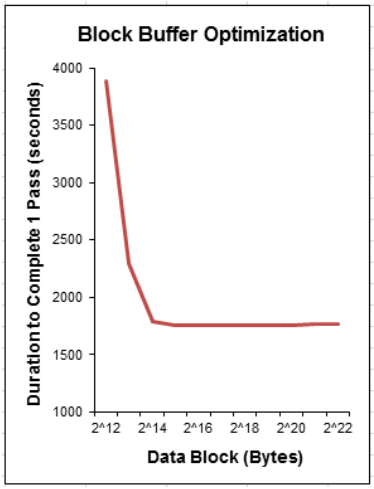

Figure 10.1. Performance of Disk I/O with various data block sizes

The experimental result of applying *block buffer* optimization in our newer version of the data shredding software is shown in Figure 10.1. The wipe performance of the software improves drastically when the buffer size escalates. However, buffer size larger than 32 KB does not yield further improvement to the overall disk I/O performance. From these results, we adopted the block wiping technique with 32 KB buffer in all our later implementations.

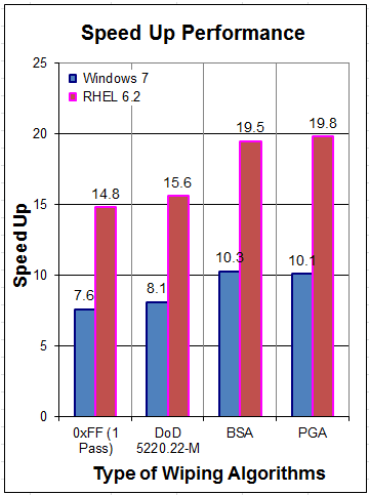

Figure 10.2. Speed up performance of selected data destruction methods on a 512 MB flash drive using Windows 7 and Red Hat Enterprise Linux 6.2.

As indicated in Figure 10.2, adopting block buffer optimization could improve the overall wipe performance of the DoD 5220.22-M method on a 512 MB flash drive by at least 8

times, when compared with the original version of the CBL Data Shredder software that was written in C++ and runs exclusively on only Microsoft Windows and MS-DOS.

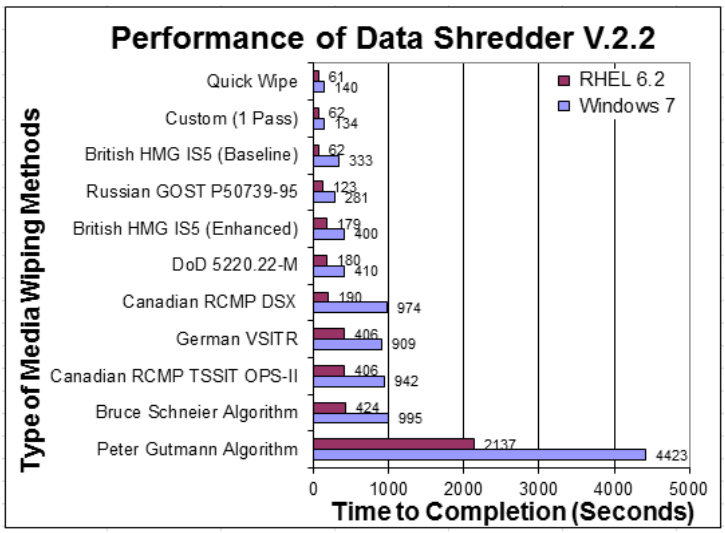

Figure 10.3. Wipe performance of different data destruction methods on a 512 MB flash drive using Windows 7 and Red Hat Enterprise Linux 6.2.

Beside incrementing the size of our write buffer, we have also exploited Pypy's *dynamic translation* (Just-In-Time compiler) method to improve our runtime performance. The comparison of different data destruction methods on a 512 MB flash drive using Windows 7 and Red Hat Enterprise Linux (RHEL) 6.1 is shown in Figure 10.3.

11. Potential Impact and Exploitation

Our data shredder software works well on solid state drives, non-volatile memory (NVRAM), USB flash drives, CD-RW/DVD-RW drives, as well as PATA/SATA/SCSI hard disk drives resided in laptops, desktop PCs, workstations, servers, and low-cost storage appliances. However, we are uncertain that our application works effectively on magnetic tape drives, Fibre Channel drives, large JBODs, NAS or SAN storage systems. Further investigations are desired if these categories of storage devices are to be addressed in the near future.

12. Conclusion

In this paper, we have shown that advanced data destruction software can be developed in a prolific and innovative manner using Python. We have explored a variety of different Python libraries to assist us in creating a decent user interface, extracting system information, managing concurrency, generating reports and optimizing disk I/O performance. Our cross-platform application software offers a wide range of military-grade procedures recommended by information security specialists. Our application also operates with no limitations to the capacity of the storage media connected to the computer system, and it could rapidly and securely expunge any existing data stored in the magnetic media, optical disks and solid-state memories.

References

Bach, Maurice (1986): *The Design of the UNIX Operating System*, Prentice Hall, Inc.

Deitel, Harvey et al. (2004): *Operating Systems (3rd Edition),* Pearson Prentice Hall.

DoD (2006): *DoD 5220.22-M National Industrial Security Program Operating Manual (NISPOM)*, U.S. Department of Defense.

Gutmann, Peter (1996): *Secure Deletion of Data from Magnetic and Solid-State Memory*, In SSYM'96: Proceedings of the 6th Conference on USENIX Security Symposium, Berkeley, CA, USA, USENIX Association.

Gutmann, Peter (2001): *Data Renanence in Semiconductor Devices*, In SSYM'01: Proceedings of the 10th Conference on USENIX Security Symposium, pages 4-4, Berkeley, CA, USA, USENIX Association.

Morris, James (2011): *Document Shredding and File Deletion*, Enirtak, Inc.

NIST (2006): *Guidelines for Media Sanitization*, NIST Special Publication 800-88, U.S. National Institute of Standards and Technology.

Precord, Cody (2010): *wxPython 2.8 Application Development Cookbook*, Packt Publishing Ltd.

RCMP (2003): *Hard Drive Secure Information Removal and Destruction Guidelines*, Information Technology Security Guide, Lead Agency Publication G2-003, Royal Canadian Mounted Police.

Roebuck, Kevin (2011): *Data Wiping and Destruction*, Tebbo.

Schneier, Bruce (1996): *Applied Cryptography (Second Edition)*, John Wiley & Sons, Inc.

Schneier, Bruce (2000): *Secret & Lies: Digital Security in a Networked World*, Wiley Publishing, Inc.

Wei, Michael et al. (2011): *Reliably Erasing Data from Flash-Based Solid State Drives*, In FAST'11: Proceedings of the 9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, USENIX Association.