Secure USB based File System for BMC Applications

Songjie Liang
Sorotek Consulting Inc
North Potomac, MD, USA
jeffliang1@gmail.com

Bharat S Rawal
IST Department
Pennsylvania State University at Abington
Abington, PA, USA
bsr17l@psu.edu

***Abstract*—A fundamental vision driving USB mass storage is commonly used in our day to day computing other storage applications and carrying personal data anywhere at any time. Portable storage, such as flash memory and USB disc, allows users transport several gigabytes to terabyte data in their pockets. For these applications, mass storage is secure and similar to a hard disk, and so the same file management system is used as in the Windows or Linux operating systems. USB mass storage because of linear block address behaves similarly to main memory. In this paper to overcome security issues of OS based mass storage devices, we introduce USB based File System for BMC Applications. This article presents the internals of a novel BMC based mass storage file system. At the end of this article, we present preliminary performance measurements using some of the interfaces. Specifically, we determine the time to create files, read and write files, and flush cache memory to the USB. BMC based USB mass storage file system allows applications to manage resources directly, and eliminates security threats and overhead associated with the conventional operating system or kernel.**

Keywords-Application Object, Bare PC, File System, Hardware Application Programming Interface (HAPI), USB, Secure File System

I. INTRODUCTION

File systems manage files and other data items stored on computer systems. File systems were initially constructed into a computer's operating system to assist access to records kept locally on resident storing media. As workstations grow into networked, some file storing abilities were divested from the individual user equipment to distinctive storing servers that stored large quantities of data on behalf of the user machinery [20]. When a file was required, the user machine simply demanded the file from the server. In this server-based design, the file system was prolonged to ease administration, and access to records kept distantly at the storage server over a network. Today, file storage is transferring toward a model in which files are stored on various networked workstations, rather than on a central storage server. The server-less construction poses new challenges to file systems. One particular task concerns managing files that are dispersed over many different processes in a way that permits files to be consistently stored and available even though changeable features of the computers. Also, at the same time preventing from accessing to the files by non-authorized users [20]. The development addresses these challenges and offers a way out that is operational for serverless dispersed file systems.

Conventional file management systems are typically tied to the underlying operating system. Some of the file commands to manage files internally invoke system calls and subsequently trigger an I/O interrupt. In such systems, the user application does not control the storage medium or the location of files on the medium, so it is not possible for the application to manage its own resources. Furthermore, since file system architectures such as FAT32 [12] and NTFS are open architectures, it is easy for intruders to corrupt or destroy an unsecured file system on mass storage. The question we address in this paper is: how can a file system be directly controlled and secured by an application programmer without going through an intermediary such as an operating system? To address some of the security concerns we introduce USB based File System (USB FS) for BMC Applications.

The exploding growth of IoT that demands for an intelligent on-line secondary storage combines in a Multiprogramming environment is immense. During the on-line communication, user owned off-line removable storage media such as USB and memory card becomes highly inconvenient. If all users are required to store as much information as they want in machine-accessible secondary storage, several needs become important. Rarely used data must penetrate to media with greater access times, to provide sufficient season on high-speed machines for further commonly employed files. Moreover, data must be easy to access when required, it must be safe from erroneous circumstances, and it should be available to other users on a readily controllable foundation when wanted. Finally, any attention whatever is not simple to a user's ability to manage this information should be invisible to him unless he specifies otherwise. The basic formulation of a file system designed to meet these needs is performed here. This technique presents the user with a simple means of addressing a predominantly significant amount of subsequent storage in a machine-independent fashion [17, 18].

The rest of this paper is organized as follows. Section 2 discusses related work; Section 3 talks on secure file system; Section 4 describes the architecture of the file management system; Section 5 gives design and implementation details of the file management system API; Section 6 presents discussion; Section 7 is for future research directions; and Section 8 summarizes our work.

II. Related Work

Wherever the USB file system API are illustrated in this paper using a BMC without kernel or OS running on the machine[11]. Many BMC applications such as Web servers, email servers, SIP servers, and VoIP clients have been developed previously. For example, a Web server based on the Bare Machine Computing (BMC) paradigm (originally referred to as the Dispersed Operating System Computing (DOSC) paradigm [7]) is described in [6]. The use of the BMC paradigm to build Web server clusters is the focus of [14, 15, and 16]. BMC applications are written in C/C++ and run as an application object (AO) by using their own interfaces to the hardware [8] and lean BMC device drivers. In contrast to [10] and [11], this paper provides details of new BMC interfaces to support a USB file management system and presents supplementary performance measurements using these interfaces. These interfaces directly work with the USB device, without the need for an external OS, kernel, or any other middleware.

Karne et al. introduce BMC, that there is no intermediary of any kind between the application and the hardware [7]. In contrast, similar approaches such as Exokernel [3], IO-Lite [13], Palacios and Kitten [9] require some form of an operating system or kernel. An example of a file management system that uses both a hard disk and flash memory is provided by the Umbrella file system [4,5]. In [1], cache systems are added at a driver level to improve the performance of removable storage devices, and in [2], an FAT32 file system for high performance clusters is presented. Unlike the BMC file management system described here, these storage systems are not managed directly by the application. Mazieres, David et al. split overall security into two pieces: file system security and key management [21]. Thus, SFS takes the strategy of meeting many key management tools; it presents compelling fundamentals of that users can immediately develop an extensive range of the underlying check mechanisms [21].

III. Ease of Use

Secure File System is a set of programs that control some encrypted disk sizes and operates underneath both batch commands and Windows. Individual volume performs as a regular DOS drive, though all information placed on it is encrypted at the individual sector level [22]. Encrypted sizes can be loaded and unloaded as required, and can be immediately replaced with a user-defined hotkey, or automatically emptied after a period of inactivity. They can also be converted back to standard DOS sizes, or have their contents damaged. The documentation comprises an in-depth investigation of various security perspectives of the software, as well as reasonably exhaustive design and programming details [22]. There are several security issues in file system below.

A. *Confidentiality*

Confidentiality refers to restricting access to information to a set of authorized principals, as well as preventing access to unauthorized ones. [22] describes three steps that are required to exercise to attain the information confidentiality – first we have to assure the principal's identification. These lists quickly become too big, so Role-Based Access Control (RBAC) is used instead [18, 22].

B. *Integrity*

Integrity provides administrators with means of detecting whether third parties have changed the information collected. The method would comprise computing the hash value of the data and encrypting that content in a way that can be guaranteed that the sender's identification can be checked[2,22]. Integrity can be rendered by cryptographic hash values digitally signed, to avoid undetected change of those hash values.

C. *Availability*

Availability refers to the possibility of obtaining the information requested when asked by those who require it. Usually, the availability depends on hardware, software, operating system, application services, etc., and it is not readily assured. Some of the latest Internet frauds and attacks, such as DoS and DDoS attacks, limit the availability of services to legitimate users.

IV. File Management System API (HAPI) Architecture

Fig. 2 shows an API call that in turn results in one or more driver calls (Fig. 1). A driver call sets up a USB transaction and a data structure that executes periodically in the application program. The HC controlled by the application program in turn issues all low level USB commands (setup, IN, OUT, etc.) to the USB device. An AO programmer can monitor the status and state of USB device in the data structures that are stored in user memory, which encapsulates the file management system within the AO. An AO may consist of one or more applications working as a single entity. Thus, the FAPI design is simplified and easier to implement for a BMC application than for an OS-based application.

The architecture of the BMC FAPI is shown in Fig. 1. Since there is no operating system or kernel, the AO programmer directly invokes driver calls via the API. The BMC driver communicates with the host controller for the USB device.

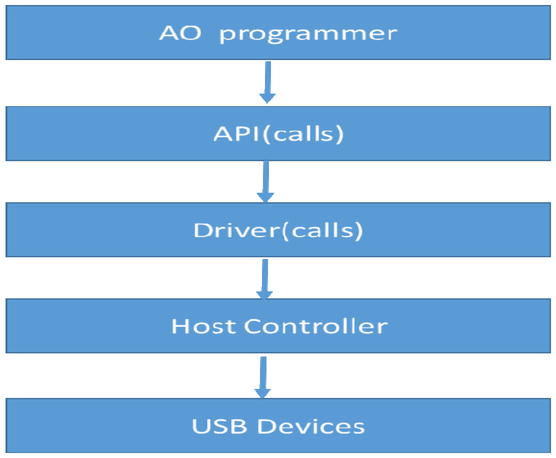


Figure 1. FAPI architecture

Fig. 2 shows standard file system calls in the Linux operating system and the equivalent BMC calls. Notice that only eight hardware API calls are used for BMC applications. Other calls needed in the Linux system are avoided because the AO programmer directly controls the file system.

The file management system design is based on the disk map shown in Fig. 3, which conforms to the FAT32 specification and hardware API for the USB. The intention to use the existing FAT32 specification in our first Bare USB file system is the well-known characteristics of FAT32. The bare USB driver offers this hardware API to access the device directly within its C/C++ code [11].

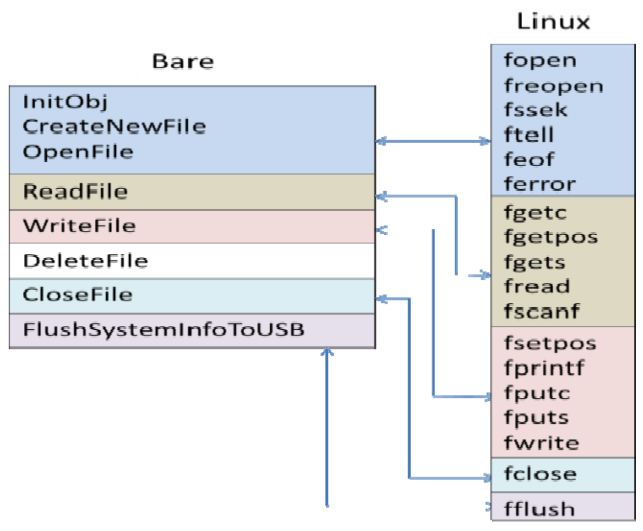


Figure 2. Linux API and BMC API

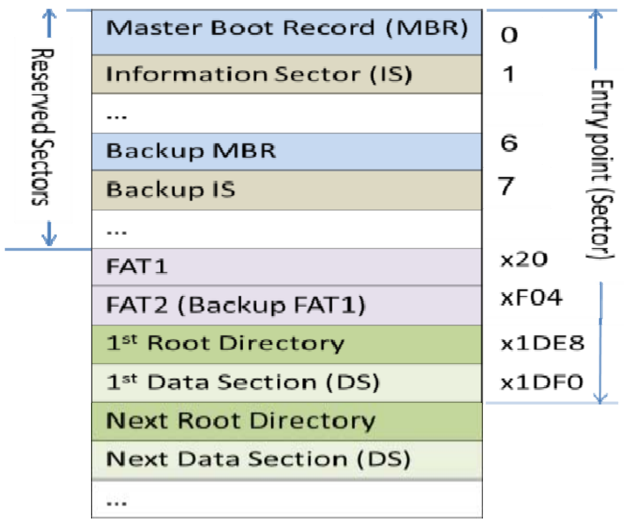


Figure 3. MBR memory layout

V. Design and Implementation

The eight FAPI interfaces are shown in Fig. 4. One can add more interfaces in BMC applications as they need and use them within its own object. We describe these interfaces below after discussing MBR calculations and caching.

A. *MBR Calculations*

The master boot record (MBR) spans one sector and contains many fields that describe characteristics of a given USB. During the initialization process (init()), we read the master boot record and information sectors into memory, and compute parameters needed for the file system. Some of these calculations are shown in Fig. 5. The parameters are needed for organizing a memory map of a USB device, which includes the master boot record, information sector, FAT1, and Root Directory, etc. as shown in Fig. 3. These calculations are performed once (when the USB is first plugged in to the system).

B. *Caching*

After computing parameters as shown in Fig. 5, the FAT and Root Directory entries (RD) from USB are cached into memory. Further updates to file characteristics are done in this cache memory. Similarly, first access to data on the USB is also cached into memory (limited to 8 sectors). The local cache tables maintain the system information (FAT and RD) and data sectors. An AO programmer has to explicitly issue the fflush() command to update the mass storage from the cache. Because the AO programmer directly controls the system, the design of BMC applications becomes simple and efficient.

C. *Interfaces*

As seen in Fig. 4, the FAPI design defines eight interfaces. They handle initializing objects, creating new files, opening or closing existing files, reading and writing files, deleting files, and flushing cache memory to the USB. We now provide more details of each interface.

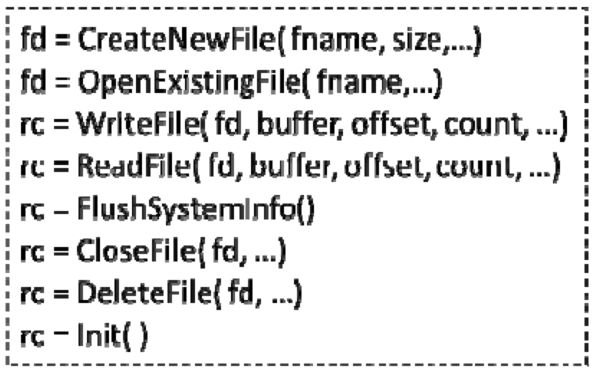

```
fd = CreateNewFile( fname, size,...)
fd = OpenExistingFile( fname,...)
rc = WriteFile( fd, buffer, offset, count, ...)
rc = ReadFile( fd, buffer, offset, count, ...)
rc = FlushSystemInfo()
rc = CloseFile( fd, ...)
rc = DeleteFile( fd, ...)
rc = Init( )
```

Figure 4. FAPI interfaces

D. *InitObj*

This interface reads the first two sectors from the USB to populate the Master Boot Record array (MBR) and Information Sector array (IS). Using the MBR, the FAT starting point, FAT size, the first root directory starting point, and the first data block starting point will be calculated. These values determine how many more sectors have to be read to populate the FAT array. The Root Directory (RD) array is computed using the FAT entry and other parameters and stored in cache memory. Furthermore, all program parameter initialization is done in addition to calculating the memory map addresses.

E. *CreateNewFile*

After verification of the file name, this interface creates a new entry in the FAT, and then in the RD. It also updates the Information sector with the file size in the cluster. It then takes a file name, file size, and file permissions, and creates a new file. It also returns a file pointer (fd), which is an index value for the RD. An AO programmer inputs an estimated file size to implement this interface. File growth is controlled by the AO programmer. This approach for creating files is different from that in conventional systems that have an operating system.

F. *OpenExistingFile*

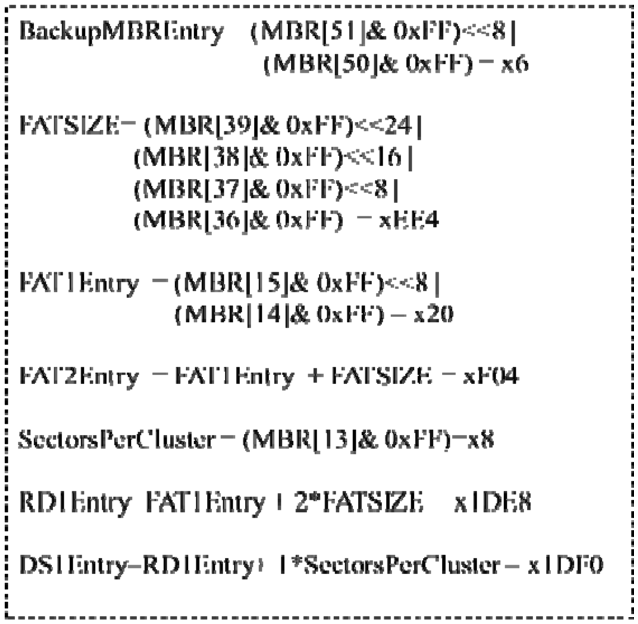
This interface opens an existing file using its file name. If the file exists, it returns the file pointer (fd), which is an index into the RD as mentioned before. It then updates the RD entry with appropriate parameters. If the file does not exist, it returns an error code.

G. *ReadFile*

This interface takes the file descriptor (fd), buffer, offset, and a number of bytes to read. It reads from the file if the file exists, and uses the file descriptor as an index value to read file parameters from its RD. It also uses offset as the starting point of the data to be read into the buffer. The offset value is the file offset from the beginning of the file. The offset is maintained by an AO programmer to manage file read operations. It reads the number of bytes specified by the count parameter into the buffer with a given offset. If the data exists in the cache, it simply reads from the cache memory; otherwise, it reads from the USB.

H. *WriteFile*

This interface takes the file descriptor (fd), buffer, offset and the number of bytes to write. It writes to the file if the file exists, and uses the file descriptor as an index value to read file parameters from its RD. It also uses offset as the starting point of the data to be written to the buffer. The offset value is the file offset from the beginning of the file. The offset is maintained by the AO programmer to manage file write operations. It writes a number of bytes as specified by the count parameter into the buffer with a given offset. This is quite different from a conventional file system, where the file system keeps track of file pointers. If the data exists in the cache, it simply writes to the cache memory; otherwise, it writes to the USB.



```
BackupMBREntry    (MBR[51]& 0xFF)<<8 |
                     (MBR[50]& 0xFF) = x6

FATSIZE= (MBR[39]& 0xFF)<<24 |
         (MBR[38]& 0xFF)<<16 |
         (MBR[37]& 0xFF)<<8 |
         (MBR[36]& 0xFF)  = xEE4

FAT1Entry  =(MBR[15]& 0xFF)<<8 |
            (MBR[14]& 0xFF) = x20

FAT2Entry  = FAT1Entry  + FATSIZE  = xF04

SectorsPerCluster= (MBR[13]& 0xFF)=x8

RD1Entry  FAT1Entry + 2*FATSIZE    x1DE8

DS1Entry=RD1Entry+  1*SectorsPerCluster = x1DF0
```

Figure 5. Memory maps calculations

I. *DeleteFile*

This interface deletes the FAT entry, updates IS entries and removes the root directory entry for that file. It requires the file descriptor (fd) as an input.

J. *CloseFile*

This interface closes an open file. It needs the file descriptor (fd) as an input parameter. It also updates the appropriate directory and FAT entries.

K. *FlushSystemInfo*

This hardware API flushes cache memory containing all system information (FAT, RD, etc.) to the USB. An AO programmer has to invoke this call explicitly to make the mass storage current with respect to internal cache. Data storage is cached out to mass storage every 3 seconds or if the cache buffer is 1/3 full. Again, the AO programmer directly manages the file system via this interface instead of requiring an operating system or kernel to control files as in a conventional application. Each root directory in an FAT32 system consists of 128 files constituting 4096 bytes or 8 sectors. The data for these files are stored on the USB immediately following the root directory. Additional or next root directories and data entry points are calculated as shown in Fig. 6. The last root directory entry points to the next root directory entry to manage a large number of files.

VI. DISCUSSION

File - A file virus appends itself to an executable file, making it run the virus code first and then skip to the start of the initial program. These bugs are named parasitic, since they do not give any extra records at the system, and the primary program is still entirely working [23, 24]. Bare machine code only allows executing code related BMC.

Boot - A boot virus keeps the boot sector, and runs before the OS is placed. These are further known as memory viruses while running they remain in memory and do not arrive in the file system [23]. BMC do not use Hardwick.

Macro - These viruses exist as macros (script) that are run automatically by individual macro-capable applications such as MS Word or Excel. These viruses can live in word processing documents or spreadsheet files [23]. BMC file system does not use macro-capable programs so issues macro can avoid.

Source code viruses scan for source code and contaminate it to spread [23]. Alteration in source code may alter the file size. BMC system prevents the execution of the program if file size changes.

Polymorphic viruses switch every time they spread. They do not change their underlying properties, but just their impression. The virus examiner's software is hard to recognize them [23]. BMC objects are whitelisting, which allows only listed methods and application to compile and execute. Non-listed program will never get administered by BMC application.

Encrypted viruses progress in encrypted form to elude discovery. In tradition, they are self-decrypting, which then enables them to infect other files [23]. BMC only executes if the code is produced and converted into the BMC format only.

Stealth viruses try to circumvent discovery by altering parts of the code that could be applied to identify it. For instance, the read ( ) system call could be changed so that if a corrupted file is created and the infected part gets jumped, the reader would see the original unadulterated file [23]. BMC does not use OS based system call, and BMC uses its BMC flavored drivers.

Tunneling viruses try to bypass disclosure by injecting themselves into the interrupt handler string, or into device drivers [23]. BMC uses very few interrupts such as timer and BMC flavored hardware irrupts.

VII. FUTURE WORK

Parallel file systems are gaining in attractiveness in high-performance computing as well as a cloud-based service. We have demonstrated various Split-protocol applications high-performance and cloud-based computing. As a future work, we are working on developing a distributed file system based on Split-protocol.

VIII. CONCLUSION

We demonstrated the feasibility of building a novel USB based file management system and API for BMC applications. BMC application bypasses the operating system and it makes to run OS based handler code. It only requires a few calls with comparison to the file management calls associated with a Linux API, so it is of small size code and provides total control to the application programmer. We also discussed the advantages of this USB file system against various viruses. The preliminary performance has been done, but more detailed performance studies are necessary to compare the BMC file system with a Linux file system. The present work serves as a basis for developing secure and high performed USB mass storage systems for use by BMC applications. This approach also has a broader impact on computing of the future.

ACKNOWLEDGMENT

Parts IV and V were done by S. L. at Towson University under Prof. R. K. Karne's supervision. He greatly appreciates Prof. Karne's guidance.

REFERENCES

[1] Y. H. Chang, P. Y. Hsu, Y. F. Lu, and T. W. Kuo "A Driver-Layer Caching Policy for Removable Storage Devices", ACM Transactions on Storage, Vol. 7, No. 1, Article 1, June 2011, p1:1-1:23.

[2] M. Choi, H. Park, and J. Jeon, "Design and Implementation of a FAT File System for Reduced Cluster Switchign Overhead", 2008 International Conference on Multimedia and Ubiquitous Engineering.

[3] D. R. Engler and M.F. Kaashoek, "Exterminate all operating system abstractions", Fifth Workshop on Hot Topics in operating Systems, USENIX, Orcas Island, WA, May 1995, p. 78.

[4] Intel Corporation, Enhanced Host Controller Interface Specification for Universal Serial Bus, March 2002, Rev 1, http://www.intel.com/technology/USB/download/ehci-r10.pdf

[5] J. A. Garrison and A. L. N. Reddy, "Umbrella File System: Storage Management across Heterogeneous Devices", Vol. 5, No. 1, Article 3, March 2009, p3:1-3:24.

[6] L. He, R. K. Karne, and A. L. Wijesinha, "The Design and Performance of a Bare Web Server", International Journal of Computers and Their Applications, IJCA, Vol. 15, No. 2, June 2008, pp. 100-112.

[7] R. K. Karne, K. V. Jaganathan, T. Ahmed, and N. Rosa, "DOSC: Dispersed Operating System Computing", OOPSLA '05, 20th Annual ACM Conference on Object Oriented Programming,Systems, Languages, and Applications, Onward Track, ACM, San Diego, CA, October 2005, pp. 55-61.

[8] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "How to run C++ Applications on a bare PC", SNPD 2005, Proceedings of SNPD 2005, 6th ACIS International Conference, IEEE, May 2005, pp. 50-55.

[9] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, R. Brightwell, "Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing", Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), April 2010.

[10] R. K. Karne, S. Liang, A. L. Wijesinha, and P. Appiah-Kubi, "A Bare Mass Storage USB Driver", International Journal of Computer and Applications (accepted for publication, June 2012).

[11] S. Liang, R. K. Karne, and A. L. Wijesinha, "A Lean USB File System for Bare Machine Applications", Proceedings of the 21st International Conference on Software Engineering and Data Engineering, ISCA, June 2012, pp. 191-196.

[12] [Microsoft Corp, "FAT32 File System Specification", http://microsoft.com/whdc/system/platform/firmware/fatgn.rnspx, 2000.

[13] V. S. Pai, P. Druschel, and Zwaenepoel. "IO-Lite: A Unified I/O Buffering and Caching System," ACM Transactions on Computer Systems, Vol.18 (1), ACM, Feb. 2000, pp. 37-66.

[14] Perisoft Corp, Universal Serial Bus Specification 2.0, http://www.perisoft.net/engineer/USB20.pdf.

[15] B. Rawal, R. K. Karne, and A. L. Wijesinha. "Mini Web Server Clusters for HTTP Request Splitting," 2011, IEEE International Conference on High Performance, Computing and Communications, Banff, Canada, p94-100.

[16] Total Phase Inc., USB Analyzers, Beagle, http://www.totalphase.com.

[17] Universal Serial Bus Mass Storage Class, Bulk Only Transport, Revision 1.0, 1999, http://www.USB.org.

[18] Daley, Robert C., and Peter G. Neumann. "A general-purpose file system for secondary storage." In Proceedings of the November 30--December 1, 1965, fall joint computer conference, part I, pp. 213-229. ACM, 1965.

[19] FreeOTFE - Free disk encryption software for PCs and PDAs. Version 5.21. Project website:

http://sourceforge.net/projects/freeotfe.mirror/, 2014.

[20] Adya, Atul, William J. Bolosky, Gerald Cermak, John R. Douceur, Marvin M. Theimer, and Roger P. Wattenhofer. "Serverless distributed file system." U.S. Patent 7,062,490, issued June 13, 2006.

[21] Mazieres, David, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. "Separating key management from file system security." In ACM SIGOPS Operating Systems Review, vol. 33, no. 5, pp. 124-139. ACM, 1999.

[22] https://www.cs.auckland.ac.nz/~pgut001/sfs/Accessed on10/5/2016

[23] https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/15_Security.html Accessed on10/5/2016

[24] http://antivirus.about.com/cs/glossary/g/filevirus.htm Accessed on 10/5/2016