FACULTY OF SCIENCE

DEPARTMENT OF COMPUTING

Digital Crime Scene Investigation for the Zettabyte File System

Andrew Li
andrew.li@students.mq.edu.au

Technical Report
5 June 2009

Abstract

Files stored on a computer are managed by the file system of the operating system. When a computer is used to store illegal data such as child pornography, it is important that the existence of the illegal data can be proven even after the data is deleted. In this study, a new functionality is added to the Zettabyte File System (ZFS) debugger, which digs into the physical disk of the computer without using the file system layer of the operating system. This new functionality enables digital crime scene investigators to retrieve any data from the disk, including deleted files. This paper presents an explanation of ZFS internals and describes the approach taken to arrive at the new ZFS debugger functionality. By using this new functionality, we find that the content and all the metadata (file size, owner, creation time, etc) of a deleted file can be retrieved directly from the disk without going through the file system layer of the operating system.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

1.1 Overview

With the development of modern information technology, computer related crime has become a threat to society because of the immense damage it can inflict. At the same time computer criminals have reached a level of sophistication which makes it difficult to track criminal data back to its original source.

Files in a computer are stored on digital storage such as a hard disk. A file system is a layer of the operating system that sits on top of the hard disk. It is like a filing cabinet for an operating system. It organises files in a way that enables the operating system to efficiently access files with minimal effort and translate the raw data on the disk to a format that can be understood by humans, like file name and directory name.

The job of a digital crime scene investigator is to carry out computer forensics examination pertaining to legal evidence found in computers and digital storage. It is the process of identifying, preserving, analysing, and presenting digital evidence in a manner that is legally accepted. The digital storage cannot be modified during the forensic examination, any modification performed on the digital storage evidence is considered as contaminated evidence which cannot be used in a court of law. This creates a need for a tool which can access the digital storage directly, without going through the file system layer in the operating system.

The Zettabyte File System (ZFS) is a new file system type developed by Sun Microsystems[1]. The ZFS file system debugger (ZDB) is part of the ZFS software suite that is used to diagnose and gather ZFS file system statistics. In this study, we present a new feature of the ZFS debugger which allows a digital crime scene investigator to access files directly from the hard disk without intervention of the operating system. Readers may think of the new feature of the ZDB as a tool that can grab a chunk of raw data from the hard disk, and translating it into files and directories which are human readable.

This study is focused on Solaris operating system because ZFS and ZDB primarily runs on the Solaris operating system.

1.2 Problem

Digital crime scene investigation is sometimes referred to as digital forensics, these two terms will be unsed interchangeably throughout the rest of this study. Digital forensics is the activity of finding out what has happened to a computer after an illegal activity has occurred on the computer. It is similar to the forensic science on humans, but digital forensics is performed on computers.

The first problem faced by digital forensics is evidence gathering. Digital investigators need to retrieve data from digital storage media without changing the original media. Under normal operations, data from disk will be retrieved from the operating system via the file system. This can induce non obvious changes to the disk because most file systems are not designed to be forensic aware. Take a simple example where a user is performing a read only operation on file like opening a file without changing its content. From the user's point of view, there has been no change made to the file system. But in the background, the operating system has already made changes to the disk via the file system. The attribute that has been changed is the access time of the file. In the field of digital forensics, any changes even a minor file attribute change is unacceptable, because it may hinge the decision of a whole court case. In our example, if the court case is about illegal hacking into a critical

[1] http://www.sun.com/software/solaris/zfs.jsp

computer system, then by examining the access time of the files will be critical evidence. In digital forensics, every detail counts. Our study will present an extension to the ZFS file system debugger (ZDB) which allow digital investigators to retrieve file metadata such as file size, owner, permission, change time and access time without modifying the data stored on disk.

The second problem faced by a digital investigator is that files that maybe critical evidence may have been deleted by the time the investigator can gain access to the disk. It is a logical action for a computer criminal to remove any trail of evidence after committing a crime. In our extension of ZDB, a new feature has been added to allow investigators to retrieve files that have been deleted from the operating system's point of view.

The third problem is that ZFS is relatively new to the industry. Knowledge of ZFS may not be as widely understood as traditional file systems. In our study, we overcome this problem by studying digital forensics on traditional Unix file systems and cross examining it with ZFS.

1.3 Aim

The aim of this study is to provide a description of the design and implementation of our new feature in ZDB. Readers should be able to reproduce the functionality of the new ZDB feature using information from this study. The final program code is expected to be different, but the functionality of the new ZDB should be the same as the one we developed in this study.

The aim of the new ZDB feature is to provide a tool that can gather data from the disk without making any changes to the file system. The new ZDB will also need to be able to retrieve deleted data from the file system.

1.4 Assumed Knowledge

Target audiences are mainly but not limited to Unix System Administrators, System Engineers, Digital Forensic Investigators, Digital Security Specialists, Information Security Responders and System Programmers. To fully understand the work described in this study, it is desirable for readers to have the following prerequisite knowledge:

- Basic understanding of general file systems internals
- Basic understanding of digital storage in relations to how a file is stored on disk
- Basic idea of digital forensics
- Intermediate Unix commands and Unix internals
- Good C programming

1.5 Structure

The remainder of the paper is organized as follows. We firstly present related work in Section 2 to show what has been done in the field for ZFS forensics. Section 3 describes the file system internal operations for file deletion process on traditional Unix file systems and ZFS. Section 4 outlines the ZFS internal which describes the innards of the different layers of the ZFS file system. This section introduces basic knowledge of the ZFS architecture preparing readers with information to understand our work on the new ZDB feature in the following section. Section 5 describes the design and implementation of our new feature of ZDB. Section 6 presents future work which can further extend our new ZDB prototype into a real forensic tool that can be used in real life digital crime scene investigations. Finally Section 7 concludes the paper.

1.6 Acronyms

CSIRT - Computer Security Incident Response Team
DMU - Data Management Layer
DSL - Data Snapshot Layer
DVA - Data Virtual Address
EXT2 - Second extended file system
EXT3 - Third extended file system
FAT - File Allocation Table
FFS - Fast File System
MDB - Solaris Modular Debugger
POSIX - Portable Operating System Interface for Unix
RAID - Redundant Array of Inexpensive Disks
SPA - Storage Pool Allocator
TCT - The Coroner's Toolkit
TSK - The Sleuth Toolkit
UFS - Unix File System
ZAP - ZFS Attribute Processor
ZDB - ZFS File System Debugger
ZFS - Zettabyte File System
ZIL - ZFS Intent Log
ZPL - ZFS POSIX layer
ZVOL - ZFS Volume

2 RELATED WORK

2.1 Overview

Different file systems behave differently in the way they store files, delete files, and the way the file metadata (file owner, group, size, modified time, access control list, etc) is stored. File system forensic examination on different file systems has been explored in *File System Forensic Analysis* [Carrier, 2005] and *Forensic Discovery* [Farmer and Venema, 2005]. These studies presented detailed file system analysis on common file systems like Ext2, Ext3, and UFS, which have provided file system forensic concept for our new ZDB feature in the present paper.

File system examination on common Unix and Linux file systems can be done by using open source tools such as The Sleuth Kit[2] (TSK) and The Coroner's Toolkit[3] (TCT). These tools read the hard disk directly and translate the raw data into the file system structure that the tool understands. These tools can work on the Linux file systems Ext2 and Ext3, the Microsoft FAT file system, the Berkerly Fast File System (an extension of UFS, the Unix File System), the Hierarchical File System by Apple Computer and the Windows NT File System by Microsoft. We have tried applying these tools on ZFS, but it does not work because the ZFS structure is different to all the traditional file systems mentioned above. However, the basic principle from TSK and TCT have been shown to be useful as it provides the foundation for our new extension in ZDB. This will become clearer when TSK and TCT are cross examined with ZFS in Section 3.

The ZFS file system is still fairly new and there is no publicised forensic tool for the ZFS file system as yet. An initial proposal[4] for a new ZFS forensic tool has been posted to the Open Solaris Security Discuss Mailing List[5] in November 2007. The number of responses from the Open Solaris community has indicated that there is a need for a ZFS forensic tool.

The article *ZFS On-Disk Data Walk* [Bruning, 2008] uses the ZFS file system debugger (ZDB) and the Solaris Modular debugger (MDB) to walk through the ZFS file system layers. His study uses ZDB and MDB to trace a pointer from the disk to the actual physical file content of a file. The approach is similar to ours in that we perform all activities in ZDB by taking the active uberblock through the various layers of the file system, until the file content and metadata is pointed to by the uberblock is reached. This will become clear to readers as we explain our new ZDB extension in Section 4.

This section will firstly present related work on general file system forensics, followed by an examination on an open source file system forensic tool. Afterwards the initial proposal of a ZFS forensic tool is presented. Finally, the work produced by Max Bruning on file data recovery with ZDB and MDB is discussed.

2.2 File System Forensics

To develop a file system forensic tool, it is essential to have a good understanding of what is involved in file system forensics. In *File System Forensic Analysis* [Carrier, 2005] and *Forensic Discovery* [Farmer and Venema, 2005], the key concepts and fundamentals of file system forensics was explained by illustrating comprehensive examples of digital investigation on FAT, NTFS, EXT2, EXT3 and UFS file systems running on Solaris, Linux, and Windows systems. The authors have demonstrated how to tackle technically challenging concepts in digital crime scene investigation

[2] http://www.porcupine.org/forensics/tct.html
[3] http://www.sleuthkit.org/sleuthkit/
[4] http://blogs.sun.com/efi/entry/proposal_open_solaris_forensic_toolkit
[5] http://opensolaris.org/os/community/security/

scenarios by using detailed examples to discover hidden evidence, recovering delete data and validating the tools that is used for the digital investigation.

File system forensics can be performed on a live file system or on a disk image after the incident has occurred. There are trade offs for both approaches. A live file system can allow the investigator to capture live data such as running processes, network traffic and volatile memory state. However, the information collected depends heavily on the state of the operating system. If the system has been broken into, the evidence collected may not be trustworthy.

A typical file system forensic process is:

1. Preserve the evidence from the target hardware ensuring no further alterations is performed to the file system on the target machine
2. Gather evidence from the target file system. This step normally involves looking for hidden data. This can mean deleted data, data hidden in unusual places on the disk and, data that appears to be something that it is not in its natural form
3. Reconstruct evidence by analyzing the file system and trying to determine the series of events that have occurred on the target machine

The work carried out by Carrier (2005) and Farmer and Venema (2005) provided the foundation of the development in our work on the new ZDB feature. It enabled the new ZDB extension to have followed the fundamental principles and concepts of file system forensics. Ensuring the new ZDB can allow the digital investigator to appropriately use the tool in a file system forensic process.

2.3 Existing Open Source Forensic Tools

The Sleuth Kit[6] (TSK) is a C library and a collection of command line tools based on code from The Coroner's Toolkit[7] (TCT). TCT is a collection of programs by Dan Farmer and Wietse Venema (authors of *Forensic Discovery* [Farmer and Venema, 2005]) for a post mortem file system analysis of UNIX systems.

The tools from TSK and TCT illustrate important concepts that we can employ in our design and implementation of ZDB. We will pick one utility from TSK as an example to demonstrate what the concept is. The `fsstat` utility is used to produce metadata about a UFS file system. It is possible for `fsstat` to do that because `fsstat` knows the UFS superblock structure which is defined in `tsk3/fs/tsk_ffs.h` in the TSK source code[8]. At a high level, when `fsstat` is given a UFS file system, it follows the below algorithm:

- Opens the file system
- Seek to an offset where the UFS superblock is located
- Reads the superblock into a superblock structure
- Print off the file system details

The detail implementation of the `fsstat` utility is shown below. Readers may wish to follow the program flow by referencing the TSK source code.

1. When `fsstat` gets executed on a UFS file system, it invokes the function `tsk_img_open()` in `tsk/img/img_open.c` to open the file system
2. The function `tsk_img_open()` from the previous step is a wrapper function that calls `tsk_fs_open ()` is defined in `tsk/fs/fs_open.c`. The `tsk_fs_open ()` function is a generic

[6] http://www.sleuthkit.org/
[7] http://www.porcupine.org/forensics/tct.html
[8] http://www.sleuthkit.org/sleuthkit/download.php

function that calls different subroutines to open the file system depending on the file system type. In this case, we have a UFS file system, the subroutine to call is `ffs_open()` in `tsk/fs/ffs.c`. Recall that UFS is also known as FFS

3. Function `ffs_open()` contains a file system structure `FFS_INFO` which contains a superblock structure `struct ffs_sb1` as defined in `tsk/fs/tsk_ffs.h`
4. The function `ffs_open()` now calls `tsk_fs_read_random()` in `tsk/fs/ffs.c` to read the superblock from the disk image and store the superblock and file system information in the `FFS_INFO` and `ffs_sb1` structure
5. `ffs_open()` now calls `ffs_fsstat()` in `tsk/fs/ffs.c` which prints out all the information about the file system with the details it obtained from the superblock

What `fsstat` is essentially doing is grabbing the raw data pointed to by the address offset of the superblock on disk, and interpreting the raw data blocks into a C structures.

By studying the TSK source code, we can get a basic idea of what our ZDB tool needs to do. The main characteristic that we have learnt from `fsstat in` TSK is that a file system forensics tool must understand the intended system of the file system it is investigating. In the above `fsstat` example, TSK is able to grab the raw data from the disk and translate it to a C structure. Once the superblock information is obtained from the disk, it can be used to track further information about the file system. The new extension we made to ZDB uses the same principle to turn raw data block into ZFS internal structures and using these structures to further track data back to the desired file content stored on disk. This is described and demonstrated in detail in Section 5 where the new ZDB feature is described.

2.4 ZFS Forensics Project Proposal

Evtim Batchev from Sun Microsystems has proposed to the Open Solaris community an initial project proposal for an Open Solaris Forensics Tools Project[9]. Evtim's proposal was initiated from a CSIRT meeting in University Of Oporto Portugal in September 2007. He then posted the proposal to the Open Solaris Security forum[10] in November 2007. The responses to the initial proposal were all positive indicating that there is a lack of ZFS specific forensics tools.

The proposal from Evtim Batchev has a much larger scope than our study here. His proposal is aimed at producing a complete forensic toolset which includes:

- Live system dissection tools based on the Solaris modular debugger (MDB) and dtrace. Dtrace is a dynamic tracing utility that allow users to examine deep inside a running kernel on a Solaris system.
- Live system monitoring and active data gathering tool sets
- Malware detection tool sets especially for loadable kernel module rootkits. A loadable kernel module rootkit is a kernel module that can be loaded dynamically into the system. When a loadable kernel module is inserted, it will allow root access (superuser privileges) to a non privileged login when a certain specific series of events occur.
- Open Solaris Forensics bootable DVD/CD/PenDriveIso including properly configured live gathering
- Tight integration of Solaris fingerprint database. A fingerprint for an operating system means a collection of unique behaviors which allow a user to probe a running system externally and be able to identify what version of Solaris the system is running
- Eventual creation and compilation of known malware database

[9] http://www.opensolaris.org/jive/thread.jspa?threadID=45379
[10] http://www.opensolaris.org/jive/category.jspa?categoryID=27

2.5 ZDB Data Walk

The study *ZFS On-Disk Data Walk* [Bruning, 2008] has a similar aim as our study. Bruning in his study used the combination of the ZFS file system debugger (ZDB) and the Solaris modular debugger (MDB) to examine data on a ZFS file system. The approach is quite clumsy in that it first uses ZDB to get a ZFS internal structure directly from the disk, then feed that data into MDB to get the values stored inside the ZFS structure. Now using the values from the ZFS structure obtained from MDB, feed it back into ZDB to get the next layer of internal ZFS structures. This process repeats a number of times until the target file content has been reached.

The work by Bruning described the interaction between different layers of ZDB in great details. It helped provide deep insight for our study in developing the new extension of ZDB.

2.6 Summary

There are not many well published file system forensic tools for ZFS, because ZFS has only been introduced into the industry for around four years. From the Open Solaris Forensics Tools Project by Evtim Batchev, we learnt that there is a necessity for a ZFS forensic tool which allows digital investigators to perform file system investigation on ZFS.

By exploring existing file system forensic tools and techniques on common Unix file systems, we gained an understanding of what is required in a file system forensic tool. The new ZDB feature must allow a digital investigator to reliably gather data from a system running ZFS. This can be done by following two main principles:

- Data on a file system cannot be modified during a digital crime scene investigation
- The ZFS forensic tool must be able to read and understand the ZFS internal structures on the disk

3 FILE DELETION

3.1 Overview

This section is devoted to describing how a file system handles file deletion. We will firstly describe some fundamentals of Unix file systems to illustrate how files are stored on disk. We will then look at an early version of Unix to see how the operating system stores a file internally with C structures. The reason for studying an early version of Unix is because the code is simple to understand and the idea behind it has not changed significantly in modern Unix file systems. After gaining a basic foundation of how files are stored in the operating system's file system by studying an early version of Unix, we will move on to modern file systems like UFS and ZFS. The file deletion process of these file systems will be examined and compared in detail. We will also illustrate how to use existing tools to retrieve deleted data on UFS. The ability to retrieve deleted files on ZFS is limited with existing tools. This section will explain the limitations of the existing tools by trying to apply similar techniques on ZFS as on UFS.

3.2 Introduction to Unix File System

The main purpose of a file system is to efficiently store and retrieve data on digital media like hard disks. To attain this goal, file systems have to store the actual data as well as some management data like file and directory names, permission, access time, allocation tables etc. This data is commonly referred to as metadata.

Files are stored on the hard disk and are divided into disk tracks and sectors. A file system is the layer above the hardware which organises files logically for the operating system. The disk is logically divided into partitions. In each partition, the cylinder group describes how the disk blocks are organised.

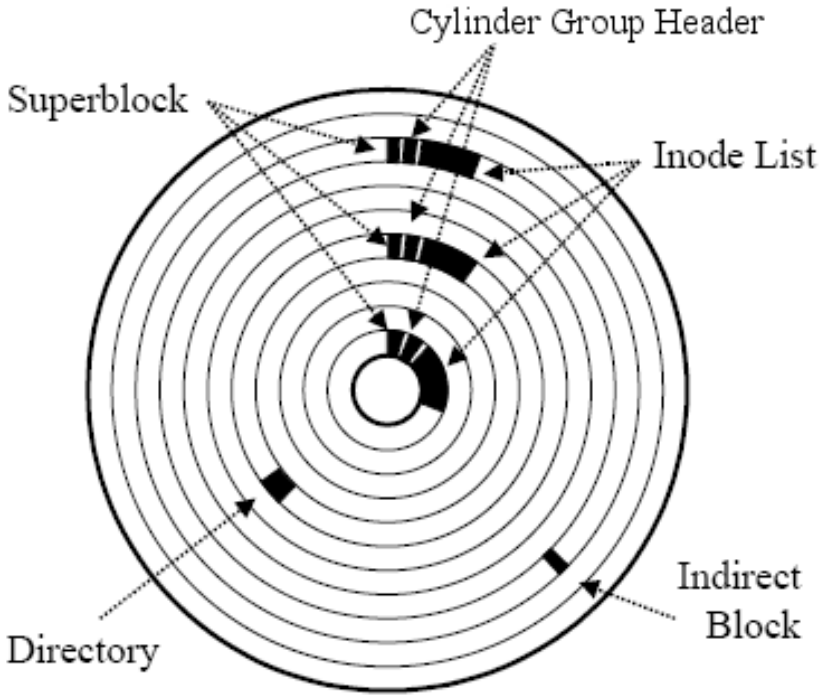

Figure 3-1 Disk layout of a Unix file system

Within the cylinder group, a superblock describes the structure of the file system layout. Within the superblock, contains a list of inodes. An inode describes the file properties and contains references to the data block which contains the content of a file. Figure 3-2 shows the structure of an inode.

A file is considered deleted when the user cannot see the file content from the operating system. The actual file content may still exist in the disk of the computer. This section will presents how files are stored in a very early version of Unix and how the same concept has carried onto modern Unix file systems. After explaining how a file is stored in a file system, we will move onto more specific examples which examine how UFS and ZFS handles file deletion.

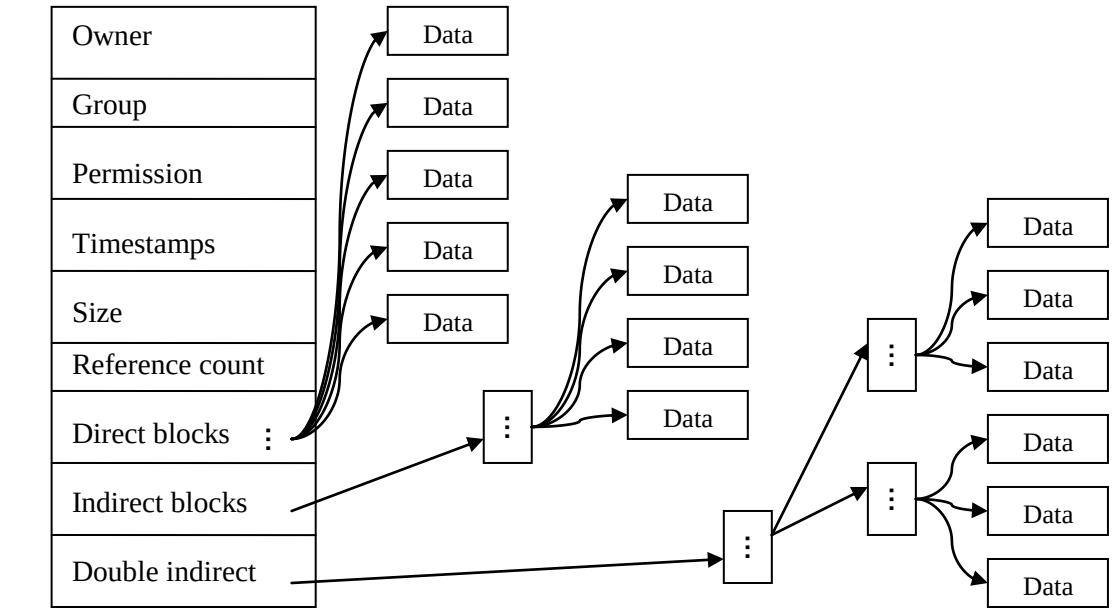

Figure 3-2 Inode structure

3.3 Historic Unix File System

The Sixth Edition Unix Operation system was the first version of the Unix operating system which the source code was made publicly available. In *A commentary on the Sixth Edition Unix Operation system* [Lions, 1977], every section of the Unix source code was explained in detail. Below is an extract from Chapter 18 of *A commentary on the Sixth Edition Unix Operation system* which describes how Unix handles a file.

A file is described by a `struct file` defined in `/usr/include/file.h`. The following shows the `struct file` definition as in the Sixth Edition Unix:

```
5507: struct  file
5508: {
```

```
5509:          char    f_flag;
5510:          char    f_count;        /* reference count */
5511:          int     f_inode;        /* pointer to inode structure */
5512:          char    *f_offset[2];   /* read/write character pointer */
5513: } file[NFILE];
```

Most modern operating systems have made enhancements to the `struct file`, but the basic idea is the same. When a file is created, the file will be assigned to an element in the `file` array; an inode is then assigned to the file which will be stored in the variable `f_inode` and the reference count variable `f_count` will increase. When `f_count` is zero, the file is considered deleted. The actual file content is left untouched on the disk.

3.4 Modern Unix File Systems

In modern Unix file systems like UFS and EXT, files are still handled by a file structure as in the Sixth Edition Unix. In modern Unix like Solaris 10, the file system code has moved to a more modular approach. An additional layer of abstraction has been on top of the original file system layer. The main reason is because this layer handles all lower level operations, so that the file system code will only have to talk to this virtual layer, and the virtual layer handles the interface to the lower layer to actually perform operations on the data blocks. This provides a more flexible solution, in a sense that when a newly invented file system type is added to the operating system, the file system code only has to deal with the virtual layer meaning that the file system code does not have to worry about the low level routines which deal with hardware devices. ZFS is a good example that has benefited from this approach.

When a file is deleted from the file system, the user will not be able to see the file content. What happens inside the file system when a file gets deleted is system dependant. In UFS and the Berkeley Fast File System [McKusick, M., 1984.], the connection between directory entry, file attributes and data blocks are cleared when a file is deleted. In the Linux second extended file system [Card, R. 1994], the directory entry is marked as unallocated, but the connection between directory entry, file attributes and file data block are preserved. In any case, the actual data stored on the disk track and disk sector are still on the disk until the file system reallocates the under laying data blocks. This means it is possible for the content of a deleted file to remain on the disk for long periods of time if the system is not heavily used.

The UFS design states that files are allocated close together to avoid disk fragmentation and increase disk seek performance [Chapter 15, McDougall, R. & Mauro, J. 2006]. To get an idea of what happens in the operating system when a file is deleted, we will examine how Solaris 10 performs a file deletion under Unix File System (UFS). At a high level, when a file gets deleted, the file system goes through the following process:

- User deletes a file from the operating system
- The file deletion generates a system call to initiate the file removal
- The virtual file system layer receives the instruction from the operating system about the file removal
- The virtual file system layer now instructs the specific file system code to handle the deletion, in this case the file system being used is UFS
- After the UFS file system code has removed the file, the inode of that file will be free

The detailed implementation of the above process is described below. All function call routine names and source file name are mentioned below. Readers may wish to follow the process by tracing through the OpenSolaris source code[11] to get a more hands on experience.

[11] http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/

1. Any file deletion on Solaris begins with the `unlink()` system call. The `unlink()` system call is defined in `unlink.c`[12]
2. The `unlink()` system call then calls `vn_remove()` which is defined in `vnode.c`[13]. The `vn_` portion of the system call refers to virtual node, this is the virtual layer as mentioned previously
3. The routine `vn_remove()` now calls `vm_removeat()` which is also defined in `vnode.c`
4. `vn_removeat()` now calls `VOP_REMOVE()`, which is a pointer to the `ufs_remove()` function defined in `ufs_vnops.c`[14].
5. `ufs_remove()` calls `ufs_delete_drain()` in `ufs_thread.c`[15]
6. `ufs_delete_drain()` calls `ufs_delete()` which is defined in the same source file `ufs_thread.c`
7. The function `ufs_delete()` will now decrease the file reference count `v_count` and free the inode with the function `ufs_ifree()` which is defined in `ufs_alloc.c`[16]

As shown above, that is how a file is deleted in Solaris 10. To see the full details, refer to the OpenSolaris source code. As we can see from the above trace, Solaris does not perform any destructive deletion to the actual content of the file that is residing on the disk. Recall from Figure 3-2 Inode structure that an inode contains a direct block pointer which holds the reference to the data block which contains the actual content of the file. When a file is deleted, the file reference count is decremented and the inode is freed. That means the relationship between the file and the data block is broken. Even though the file content is left untouched, it can be difficult to retrieve the full content of the original file because without the inode telling us where the data blocks are located, the data block could be scattered across the disk making them impossible to find.

3.4.1 Reconstructing Evidence

An important step in file system forensic analysis is to reconstruct evidence. In this section we will create some files on a newly created file system and demonstrate how to recover deleted files with TSK. The steps we have taken are outlined below:

- Create a small UFS file system on the disk
- Mount the new file system
- Create three new text files (file1, file2, file3) on the file system
- Append a string “this is fileX” to the file1, file2, and file3
- Delete file2
- Un-mount the new file system
- Create a disk image with a low level copy Unix command
- Examine the disk image with TSK

The TSK toolset provides a tool that displays general details of a file system. We use the `fls` tool from TSK to gather some details of the file system, as shown below:

```
$ fls -a -f ufs /var/tmp/c0t1d0s5.img
-/d 2:  .
-/d 2:  ..
-/d 3:  lost+found
-/r 4:  file1
```

[12] `usr/src/uts/common/syscall/unlink.c`
[13] `usr/src/uts/common/fs/vnode.c`
[14] `usr/src/uts/common/fs/ufs/ufs_vnops.c`
[15] `usr/src/uts/common/fs/ufs_thread.c`
[16] `usr/src/uts/common/fs/ufs/ufs_alloc.c`

```
-/- * 0:        file2
-/r 6:  file3
```

The first column of the `fls` output indicates what type of file it is, the second column is the inode number and the third column is the file name. Deleted file entries are marked by an asterisk in the fls output. As mentioned in Section 3.4, Solaris clears the inode when a file is deleted, that is why `fls` displayed the inode of file2 as 0 because it cannot find any details of the file.

The TSK utility `istat` provides information of a given inode. As we can see from the output of `fls`, the inode ranged from 2 to 6 with file2 being empty. It is a logical to presume that file2 may used to have inode 5. We will examine inode 5 with the `istat` tool.

```
$ istat -f ufs /var/tmp/c0t1d0s5.img 5
inode: 5
Not Allocated
Group: 0
uid / gid: 0 / 0
mode: ----------
size: 0
num of links: 0

Inode Times:
Accessed:       Tue Mar 24 08:13:38 2009
File Modified:  Tue Mar 24 08:16:32 2009
Inode Modified: Tue Mar 24 08:16:32 2009

Direct Blocks:
$ istat -f ufs /var/tmp/c0t1d0s5.img 6
inode: 6
Allocated
Group: 0
uid / gid: 0 / 0
mode: -rw-r--r--
size: 14
num of links: 1

Inode Times:
Accessed:       Tue Mar 24 08:13:23 2009
File Modified:  Tue Mar 24 08:13:23 2009
Inode Modified: Tue Mar 24 08:13:23 2009

Direct Blocks:
2363
```

From the above `istat` output, we can now see that file2 is at block 2362. Let's proceed with `dcat` and `dd` to see what is located at the block addressed at 2362. The `dcat` utility from TSK obtained its name from the standard Unix `cat` command. The Unix `cat` command concatenate and display files. The dcat command from TSK is similar but it takes an address as argument and will seek to that address on the disk to display the data that is located at the block address.

```
$ dcat -f ufs /var/tmp/c0t1d0s5.img 2362
this is file2
```

We can obtain the same result with the standard Unix command `dd` command. The `dd` command is a utility that allow users to seek to an address and output a specified amount of data. In the below example below, the `dd` utility is instructed to seek to the block at 2632 and display 1 kilobyte of data from 2362 onwards. The output of `dd` is piped to an `xxd` command which translate hex into ASCII. The `head` command indicates we only want the first two lines of the output.

```
$ dd if=/var/tmp/c0t1d0s5.img bs=1k skip=2362 | xxd | head -2
0000000: 7468 6973 2069 7320 6669 6c65 320a 0000  this is file2...
0000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
9+0 records in
```

```
8+0 records out
```

The above content above matches what we created for file2. We have found our file.

The idea here is to gather information with TSK which will provide us with the next clue and eventually we can follow the address stored in the direct block to arrive to the target file stored on disk. As we will see in Section 5, our new ZDB follows the same principle. It follows block address pointers until the final file content is reached.

3.5 ZFS

The main purpose of this section is to demonstrate that deleted data can be recovered on ZFS. In other words, we must ensure that ZFS does not delete the actual data that is stored on disk when a file is being deleted form the operating system. Otherwise, there is little point in trying to develop a file system forensic tool for ZFS.

File deletion on a Solaris system running ZFS is similar to UFS. The reason is because most of the work still goes through the virtual file system layer. This is also the beauty of the virtual file system layer, the underlying virtual file system layer is identical for all file system types. It does not matter what type of file system the operating system is running,

To ensure that ZFS does not clear the file data stored on disk during the file deletion process, we need to understand how ZFS implements file deletion. At a high level overview, the following occurs when a user requests for a file to be deleted from ZFS.

- User initiates a file deletion from the operating system
- The file deletion generates a system call to initiate the file removal
- The virtual file system layer receives the instruction from the operating system about the file removal
- The virtual file system layer now instruct the specific file system code to handle the deletion, in this case the file system being used is ZFS
- The ZFS code now removes the ZFS directory entry of the file from the file system
- The znode is now deleted from the ZFS file system. A znode is similar to an inode in UFS, it tells the file system how to access the data block of a file
- ZFS is a log file system, when a file is removed the file removal is recorded atomically in a log file. This prevents file system inconsistency by ensuring that each file system activity is reproducible even when the system crashes
- The file is now invisible from the operating system

There is frequent use of ZFS internal terminology in this section. A brief explanation will be given, but will not be discussed in detail. The ZFS internal design and architecture will be explained in Section 4. The main purpose of this section is to illustrate that ZFS does not perform destructive file deletion. The following is the chain of system calls that gets invoked when a file is deleted in ZFS. Readers may wish to follow the chain of functions by looking at the Open Solaris source code.

1. Any file removal is done via the `unlink()` system call. This system call is defined in `unlink.c`[17]
2. `unlink()` then invokes `vn_remove()` which is defined in `vnode.c`[18]
3. `vn_remove()` now calls `vm_removeat()` which is also defined in `vnode.c`

[17] `usr/src/uts/common/syscall/unlink.c`
[18] `usr/src/uts/common/fs/vnode.c`

4. `vn_removeat()` now call `VOP_REMOVE()`, which is a pointer to the `zfs_remove()` function defined in `zfs_vnops.c` [19]at this point, the file system is moving away from the virtual file system layer and moving into ZFS specific function calls
5. `zfs_remove()` calls `zfs_link_destroy()` to remove the directory entry, which is defined in `zfs_dir.c` [20]recall from 3.3 Historic Unix File System that in the Sixth Edition Unix, when a file is removed from the operating system, the reference count is decreased. The ZFS function `zfs_link_destroy()` is serving similar purpose here.
6. After the `zfs_link_destroy()` function return to `zfs_remove()` from completion, `zfs_remove()` calls upon `zfs_znode_delete()`; which in turn calls `zfs_znode_free()`. Both functions are defined in `zfs_znode.c`[21]. Their purpose is to delete and free the znode, similar to the UFS function `ufs_ifree()` from Section 3.4 which free up the inode
7. Last step before the `zfs_remove()` function exits is to handle the ZIL transaction by calling `zfs_log_remove()`, which is defined in `zfs_log.c`[22]. This function handles the ZFS transaction log in ZFS

As shown above, the file deletion process is very similar to UFS. Step 1 to 3 is identical because the Solaris file system framework is designed to be pluggable as mentioned in Section 3.4. Similar to UFS, when a file gets deleted in ZFS, the content of the file is still stored in the disk and there is nothing in the ZFS code that attempts to clear the content. This means that it should be possible for a tool to trace and gather all the data blocks of a file to recover deleted files on ZFS. In Section 5, we will demonstrate how our new feature of ZDB implements file recover on ZFS.

3.5.1 Reconstructing Evidence Attempt

In section 3.5, we have shown in theory that file deletion does not wipe the actual data. This is a crucial point in a ZFS forensic analysis tool because if the data blocks are cleared when a file is deleted, data recovery will need to involve much more expensive and advanced micro-scoping technique [Gutmann, P. 1996] and there is no point in developing a forensic tool. Below is a demonstration on retrieving deleted text file under ZFS with existing standard Unix commands.

1. Create a small ZFS file system with the `zpool` command

```
# zpool create -f testp /dev/dsk/c0t1d0s5
```

2. Mount the new ZFS and examine the mount point with the Unix `df` command. The `df` command displays the number of free disk blocks. We will see that the new file system is mounted on `/testp` and the file system size is around 28MB

```
# df -k /testp
Filesystem            kbytes    used   avail capacity  Mounted on
testp                  28672      20   28563     1%    /testp
```

3. Create and append a string to three new text files (file1, file2, file3) on the file system

```
# cd /testp
# echo this is file1 on zfs > file1
# echo this is file2 on zfs > file2
# echo this is file3 on zfs > file3
# ls -al
total 8
drwxr-xr-x   2 root     root           5 Mar 25 11:40 .
drwxr-xr-x  32 root     root        1024 Mar 25 11:22 ..
```

[19] usr/src/uts/common/fs/zfs/zfs_vnops.c
[20] usr/src/uts/common/fs/zfs/zfs_dir.c
[21] usr/src/uts/common/fs/zfs/zfs_znode.c
[22] usr/src/uts/common/fs/zfs/zfs_log.c

```
-rw-r--r--   1 root     root          21 Mar 25 11:40 file1
-rw-r--r--   1 root     root          21 Mar 25 11:40 file2
-rw-r--r--   1 root     root          21 Mar 25 13:53 file3
```

4. Delete file3 with the Unix `rm` command. This will invoke the series of ZFS functions in the operating system as shown in Section 3.5

```
# /bin/rm file3
# ls -l
total 4
-rw-r--r--   1 root     root          21 Mar 25 11:40 file1
-rw-r--r--   1 root     root          21 Mar 25 11:40 file2
```

5. Un-mount the new ZFS file system

```
# cd /
# umount /testp
```

6. Create a disk image with a low level copy Unix command – `dd`. The `if` argument specifies the input device, and the `of` argument is the output device. Therefore the following command is reading every data block sequentially from the disk (`c0t1d0s5`) and writing it to a file (`zfs.img`)

```
# dd if=/dev/dsk/c0t1d0s5 of=/var/tmp/zfs.img
132288+0 records in
132288+0 records out
```

7. Search for the required string on the disk image. We know that file3 contains the string "this is file3 on zfs", so we will look for the string "file3" in the disk image. The commands below uses `strings` to find printable strings in a binary file and piping it to `grep` which searches for the pattern required

```
# strings /var/tmp/zfs.img | grep file3
this is file3
file3
file3
file3
file3
this is file3 on zfs
this is file3 on zfs
file3
this is file3 on zfs
*is@ file32
*is@ file32
file3
file3
file3
*is@ file32
*is@ file32
*is@ file32
file3
```

As shown above, the content of file3 "this is file3 on zfs" is still stored on the disk even after the `rm` command has been executed on file3. This implies that if we can obtain the znode of the file from ZFS, it will provide us with a starting point for tracing the data back to the disk. Hence we should be able to mimic the behavior of TSK.

Note that the above approach with the standard Unix commands `dd`, `strings` and `grep` is impractical for a real life digital forensic scenario because using the standard Unix commands can only work for textual data. If the target file is binary then there will be nothing for the `strings` and `grep` command to work on. In Section 5, we will demonstrate how our new ZDB feature overcomes this problem.

3.6 Comparison

In terms of file deletion, UFS and ZFS are quite similar. There is the obvious implementation difference in terms of the code, but the basic idea is the same. When a user requests for a file deletion, the operating system passes the request to the file system layer and the file is "deleted" from the operating system so that the user can no longer see the file.

Readers may have already noticed from Section 3.4 and Section 3.5 that the main difference between UFS and ZFS is that ZFS is a log structured file system. ZFS saves transaction records of all changes to the file system in a log. In the event of a system crash, the operating system will replay the log to perform all uncommitted I/O transaction to prevent file system inconsistency. As we will see in the following section, ZFS is a more structural file system than UFS.

3.7 Summary

Having examined the current technology on file deletion on UFS and ZFS, we learnt that both of these file systems do not destroy the actual data block stored on the disk when a file gets deleted. The operating system basically hides the file from the operating system so that the end user does not see the file as being on the machine when they delete a file. To collect and reconstruct the evidence from the file system, the inode is an essential piece of information. Once the inode is resurrected, the information from the inode can be used to reconstruct the "deleted" file from the data blocks on the disk.

4 ZFS DESIGN AND ARCHITECTURE

4.1 Overview

This section presents the design and architecture of ZFS internals. It will provide sufficient ZFS information for readers to understand the extension that will be made to the ZFS file system debugger (ZDB) in Section 5. The layered design of the ZFS file system will be explained in detail. A short comparison will be presented at the end of the section to explain the difference between ZFS and traditional Unix file systems. These differences will indicate why current file system forensic tools do not work on ZFS.

4.2 Concept

The ZFS file system is a new technology that provides dynamic storage which can grow and shrink without the need to re-partition the underlying storage. It does that by eliminating the concepts of partitions and volumes in traditional file systems. A ZFS file system consists of a common storage pool made up of writable storage media. The concept of files and directories are replaced by objects. A complete listing of all ZFS objects can be found in the *ZFS On-Disk Specification* [Sun Microsystems, 2006].

ZFS is a transactional file system, that keeps a record of all transactions to ensure the state of the file system is always consistent on disk. Traditional Unix file systems overwrite data in place, which means that if the machine loses power or crashes between the time a data block is allocated and when it is linked into a directory, the file system will be left in an inconsistent state because the file system now has an allocated block of data with no linkage to the file system. In ZFS, data is either entirely committed or entirely lost. While some data can be lost, the file system is never inconsistent, which is difficult or impossible to recover from in traditional Unix file systems.

All data and metadata is checksummed in ZFS. The checksum is processed at the file system layer and is transparent to applications. Since all data is checksummed, ZFS takes advantage of this and use the checksum to provide automatic self healing of corrupted data. ZFS supports storage pools with various RAID levels. When ZFS detects a bad block with an incorrect checksum, it fetches the correct data from another redundant copy to replace the bad data with the good copy.

ZFS is designed to be scalable. The file system is 128 bits, allowing 256 quadrillion zettabytes[23] of storage. Hence the name Zettabyte File System.

4.3 Architecture

ZFS is comprised of seven components: the SPA (Storage Pool Allocator), the DSL (Dataset and Snapshot Layer), the DMU (Data Management Layer), the ZAP (ZFS Attribute Processor), the ZPL (ZFS POSIX layer), the ZIL (ZFS Intent Log), and ZVOL (ZFS Volume). We will concentrate on SPA, DMU, DSL and ZAP as they are more relevant to our new feature of ZDB. For a complete description on all components, please see the *ZFS On-Disk Specification* [Sun Microsystems, 2006].

The **Storage Pool Allocator** (SPA) component of ZFS contains virtual devices (vdevs) which make up the ZFS storage pools. The virtual devices are described by a 256k virtual device label (vdev label). To provide redundancy, four copies of the vdev label are written to each vdev within a storage pool. The vdev label contains an array of uberblocks which provide the file system with information

[23] 1 zettabyte is equal to 1 billion terabytes

necessary to access the content of the storage pool. The uberblock is equivalent to the superblock in traditional Unix file systems, as it contains block pointers that describe blocks of data on disk. A vdev label of a block device of size N is shown in Figure 4-1.

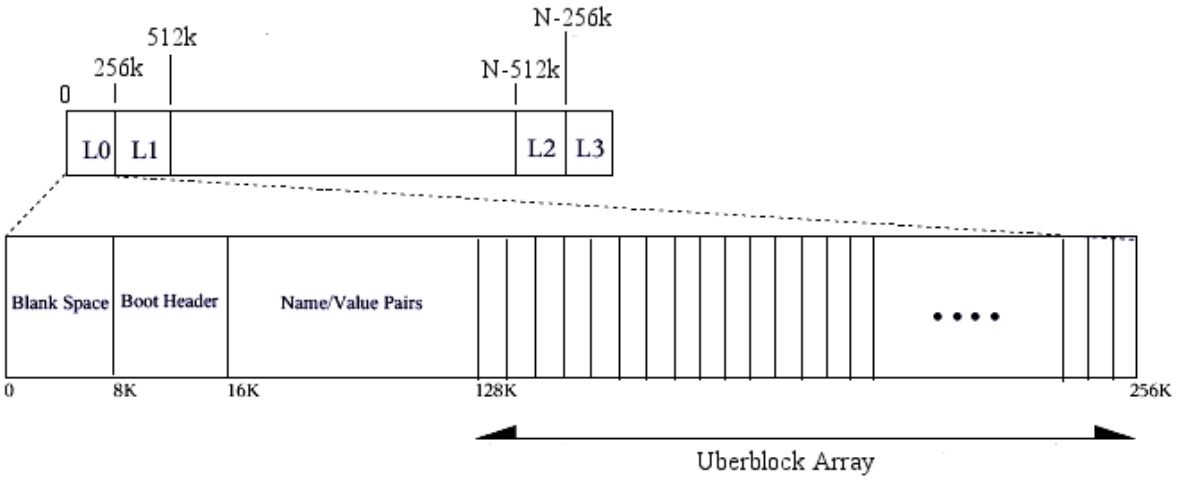

Figure 4-1 Vdev label for a block device of size N

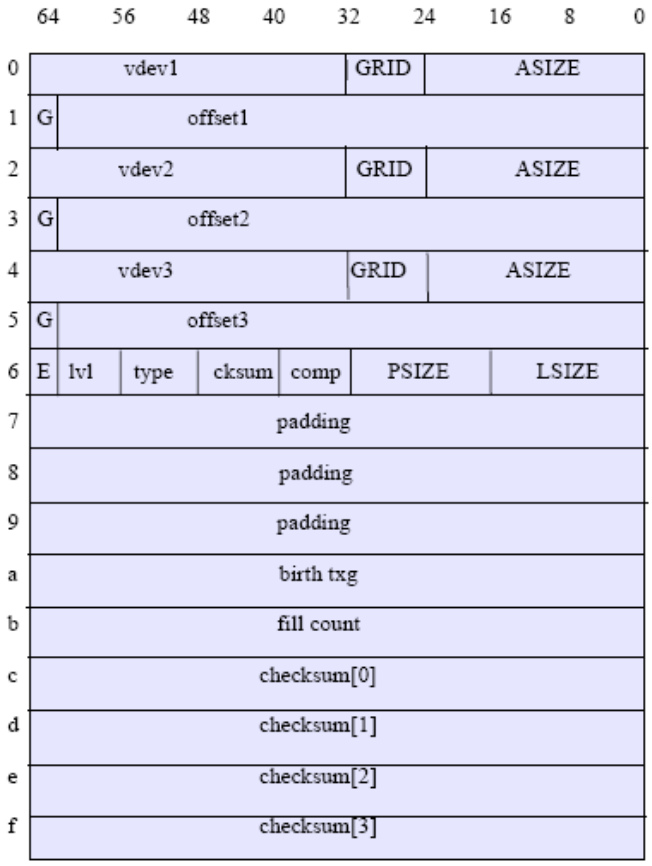

Figure 4-2 Block pointer structure layout

The information from the block pointer is heavily used by our ZDB extension. It provides Data Virtual Address (DVA) attribute which is used as a point of reference for tracing block pointers back to the data blocks of the target file on disk. We will explain and demonstrate how DVA is used in Section 5. The block pointer structure layout is shown below. DVA is made up of vdev and offset portion, for example vdef1 and offset1 make up a DVA.

The **Data Management Layer** (DMU) consumes blocks and groups them into objects. With the exception of low level infrastructure in SPA, everything in ZFS is an object. Objects are defined by structures called dnode. A dnode describes and organizes a collection of blocks making up an object. A file system is described by a group of objects called object sets. There are many object types defined in ZFS, refer to Chapter 3 of the *ZFS On-Disk Specification* [Sun Microsystems, 2006] for a complete listing. The table below shows a small subset which is used in this study:

Object Type	Description
DMU_OT_OBJECT_DIRECTORY	DSL object directory ZAP object
DMU_OT_DNODE	Dnode objects
DMU_OT_OBJSET	Collection of objects
DMU_OT_DSL_DIR	ZPL Directory ZAP Object
DMU_OT_DSL_DATASET	DSL dataset object used to organize snapshot and usage static information for objects of type DMU_OT_OBJSET.
DMU_OT_PLAIN_FILE	A plain file object
DMU_OT_MASTER_NODE	This is a ZAP object used to identify root directory, delete queue, and version for a filesystem.

The **Dataset and Snapshot Layer** (DSL) describe and manage the relationship between object sets. In DSL, object sets are grouped hierarchically into Dataset Directories. Each dataset object points to a DMU object set which contains the actual object data. There are four types of object sets in ZFS: file system, clone, snapshot, and volume.

The **ZFS Attribute Processor** (ZAP) is a module that operates the object used to store properties for a dataset, file system object and pool properties. A ZAP object is a DMU object used to store attributes such as properties for a dataset, navigate file system object and store pool properties. There are two basic types of ZAP objects, "microzap" and “fatzap”. Microzap objects are used when the attributes can fit in one block and fatzap objects are used when attributes require more than one block.

Below, is a brief description of the remaining components of ZFS which are not directly related to our study, but are included to show the complete architecture of ZFS.

The **ZFS POSIX Layer** (ZPL) makes the file system POSIX compliant. ZFS provides a set of POSIX services for the file system.

The **ZFS Intent Log** (ZIL) records all transactions of the file system. Its purpose is to replay the log records in the event of a machine panic or power failure. This prevents inconsistency in the file system.

ZFS Volumes (ZVOL) provides a mechanism for creating logical volumes in ZFS. A ZFS volume is exported as a block device and it can be used like any other block device in the operating system like a floppy disk or a UFS disk partition.

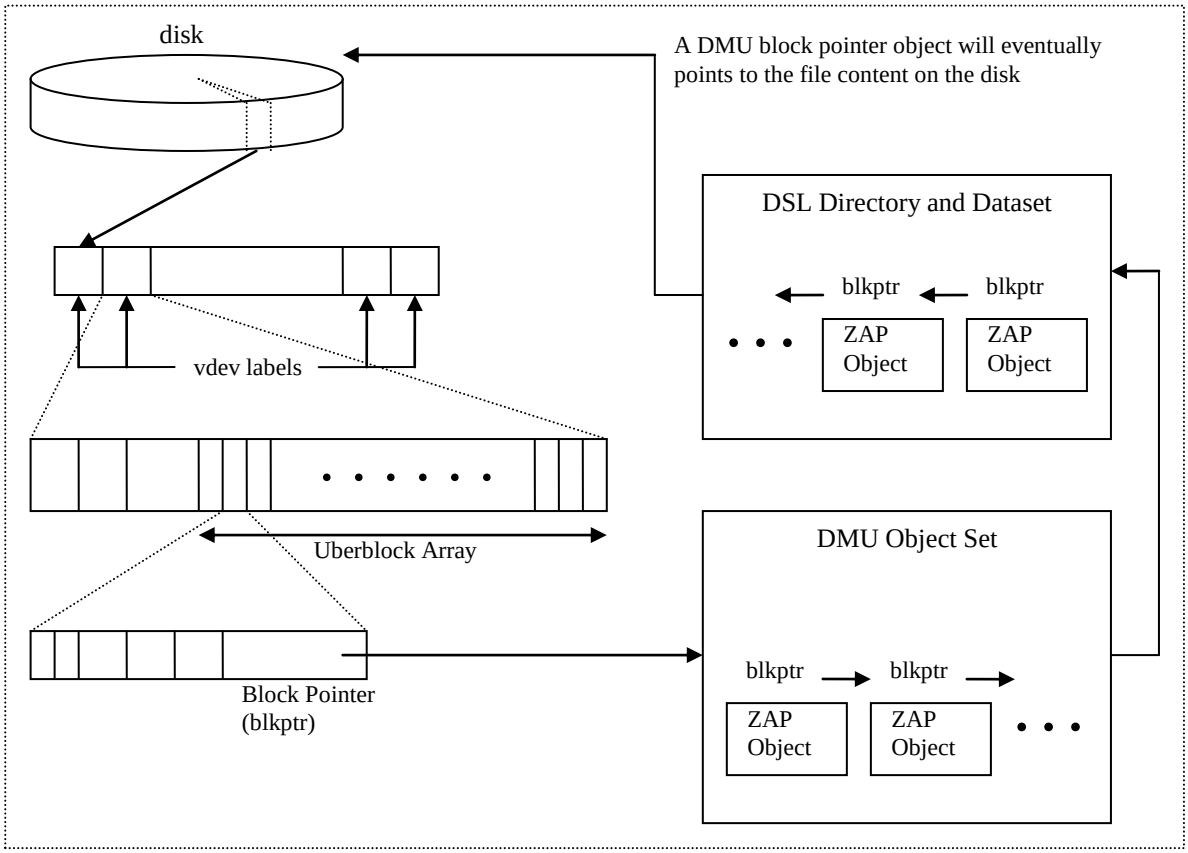

Figure 4-3 The relationship of SPA, DMU, DSL and ZAP components

4.4 Implementation

This section closely examines the ZFS internal structure that is used in our ZFS extension. It can be read in conjunction with Section 4.3 to assist readers in understanding our ZDB extension. Readers who are not interested in the detail implementation can skim through this section as it would not impact the overall understanding of the design of our new ZDB feature.

ZFS storage pools are made up of a virtual device which is referred to as vdev. Vdev are described by a vdev label which is a 256 kilobytes structure and it describes the hierarchy of the vdev tree. The vdev structure is defined in the OpenSoalris source file `vdev_impl.h`[24], a pictorial illustration of the vdev label is shown in Figure 4-1. There are four copies of the vdev label on a block device. As shown below, the C structure that defines the vdev label from `vdev_impl.h`.

```
typedef struct vdev_label {
char            vl_pad[VDEV_SKIP_SIZE];                 /*  16K */
vdev_phys_t     vl_vdev_phys;                           /* 112K */
char            vl_uberblock[VDEV_UBERBLOCK_RING];      /* 128K */
} vdev_label_t;                                         /* 256K total */
```

Figure 4-4 Structure of vdev label

[24] `usr/src/uts/common/fs/zfs/sys/vdev_impl.h`.

Within the vdev label structure is a list of uberblocks. An uberblock is similar to a superblock in UFS. It defines how the storage pool should be accessed. The uberblock structure is defined in `uberblock_impl.h`[25]. The uberblock structure is shown below.

```
struct uberblock {
        uint64_t        ub_magic;       /* UBERBLOCK_MAGIC              */
        uint64_t        ub_version;     /* SPA_VERSION                  */
        uint64_t        ub_txg;         /* txg of last sync             */
        uint64_t        ub_guid_sum;    /* sum of all vdev guids        */
        uint64_t        ub_timestamp;   /* UTC time of last sync        */
        blkptr_t        ub_rootbp;      /* MOS objset_phys_t            */
};
```

Figure 4-5 Uberblock structure

The blkptr_t variable in the uberblock is a block pointer type. The block pointer structure is used heavily in the new feature of our ZDB. It contains the Data Virtual Address (DVA) type which describes the data blocks on disk. Below is the structure definition of the block pointer type and the DVA type from `spa.h`[26] as shown below.

```
/*
 * Each block is described by its DVAs, time of birth, checksum, etc.
 * The word-by-word, bit-by-bit layout of the blkptr is as follows:
 * vdev         virtual device ID
 * offset       offset into virtual device
 * LSIZE        logical size
 * PSIZE        physical size (after compression)
 * ASIZE        allocated size (including RAID-Z parity and gang block headers)
 * GRID         RAID-Z layout information (reserved for future use)
 * cksum        checksum function
 * comp         compression function
 * G            gang block indicator
 * E            endianness
 * type         DMU object type
 * lvl          level of indirection
 * birth txg    transaction group in which the block was born
 * fill count   number of non-zero blocks under this bp
 * checksum[4]  256-bit checksum of the data this bp describes
 */
typedef struct blkptr {
        dva_t           blk_dva[3];     /* 128-bit Data Virtual Address */
        uint64_t        blk_prop;       /* size, compression, type, etc */
        uint64_t        blk_pad[3];     /* Extra space for the future   */
        uint64_t        blk_birth;      /* transaction group at birth   */
        uint64_t        blk_fill;       /* fill count                   */
        zio_cksum_t     blk_cksum;      /* 256-bit checksum             */
} blkptr_t;
```

Figure 4-6 Block pointer structure

```
/*
 * All SPA data is represented by 128-bit data virtual addresses (DVAs).
 * The members of the dva_t should be considered opaque outside the SPA.
 */
typedef struct dva {
        uint64_t        dva_word[2];
} dva_t;
```

Figure 4-7 Data Virtual Address (DVA) type definition

[25] `usr/src/uts/common/fs/zfs/sys/uberblock_impl.h`
[26] `usr/src/uts/common/fs/zfs/sys/spa.h`

The DMU manages the transfer of data blocks and groups them into objects. These objects are defined by a 512 bytes structure called the dnode. The dnode structure is defined in `dnode.h`[27]. Its purpose is similar to an inode in UFS. The structure below defines the dnode type.

```
typedef struct dnode_phys {
        uint8_t dn_type;                /* dmu_object_type_t */
        uint8_t dn_indblkshift;         /* ln2(indirect block size) */
        uint8_t dn_nlevels;             /* 1=dn_blkptr->data blocks */
        uint8_t dn_nblkptr;             /* length of dn_blkptr */
        uint8_t dn_bonustype;           /* type of data in bonus buffer */
        uint8_t dn_checksum;            /* ZIO_CHECKSUM type */
        uint8_t dn_compress;            /* ZIO_COMPRESS type */
        uint8_t dn_flags;               /* DNODE_FLAG_* */
        uint16_t dn_datablkszsec;       /* data block size in 512b sectors */
        uint16_t dn_bonuslen;           /* length of dn_bonus */
        uint8_t dn_pad2[4];

        /* accounting is protected by dn_dirty_mtx */
        uint64_t dn_maxblkid;           /* largest allocated block ID */
        uint64_t dn_used;               /* bytes (or sectors) of disk space */

        uint64_t dn_pad3[4];

        blkptr_t dn_blkptr[1];
        uint8_t dn_bonus[DN_MAX_BONUSLEN];
} dnode_phys_t;
```

Figure 4-8 Dnode structure definition

The data type `objset_phys_t` describes a group of objects. There will be one of these for all object sets, and one for ZFS file system objects (files and directories). The `objset_phys_t` is defined in `dmu_objset.h`[28].

```
typedef struct objset_phys {
        dnode_phys_t os_meta_dnode;
        zil_header_t os_zil_header;
        uint64_t os_type;
        char os_pad[1024 - sizeof (dnode_phys_t) - sizeof (zil_header_t) -
            sizeof (uint64_t)];
} objset_phys_t;
```

Figure 4-9 Object Set structure

In the implementation of our new ZDB feature, the ZFS Attribute Processor (ZAP) object is frequently used. The attributes in ZAP object helps determine what objects our ZDB is looking for. The microzap objects are defined by `mzap_phys_t` in `zap_impl.h`[29]. The fatzap object is not used in our ZDB extension.

```
typedef struct mzap_phys {
        uint64_t mz_block_type; /* ZBT_MICRO */
        uint64_t mz_salt;
        uint64_t mz_normflags;
        uint64_t mz_pad[5];
        mzap_ent_phys_t mz_chunk[1];
        /* actually variable size depending on block size */
} mzap_phys_t;

typedef struct mzap_phys {
        uint64_t mz_block_type; /* ZBT_MICRO */
        uint64_t mz_salt;
        uint64_t mz_normflags;
```

[27] `usr/src/uts/common/fs/zfs/sys/dnode.h`
[28] `usr/src/uts/common/fs/zfs/sys/dmu_objset.h`
[29] `usr/src/uts/common/fs/zfs/sys/zap_impl.h`

```
        uint64_t mz_pad[5];
        mzap_ent_phys_t mz_chunk[1];
        /* actually variable size depending on block size */
} mzap_phys_t;
```

Figure 4-10 ZAP object definitions

The `dn_bonus` variable from the dnode structure holds a few different data type that will be used in our ZDB. The first one is `dsl_dir_phys_t`, this is a DMU_OT_DSL_DIR object defined in `dsl_dir.h`[30]. The `dd_head_dataset_obj` variable is a dnode object which our ZDB uses to get the DSL Dataset Object. This will be discussed in detail in Section 5.4.

```
typedef struct dsl_dir_phys {
        uint64_t dd_creation_time; /* not actually used */
        uint64_t dd_head_dataset_obj;
        uint64_t dd_parent_obj;
        uint64_t dd_origin_obj;
        uint64_t dd_child_dir_zapobj;
        /*
         * how much space our children are accounting for; for leaf
         * datasets, == physical space used by fs + snaps
         */
        uint64_t dd_used_bytes;
        uint64_t dd_compressed_bytes;
        uint64_t dd_uncompressed_bytes;
        /* Administrative quota setting */
        uint64_t dd_quota;
        /* Administrative reservation setting */
        uint64_t dd_reserved;
        uint64_t dd_props_zapobj;
        uint64_t dd_deleg_zapobj; /* dataset delegation permissions */
        uint64_t dd_flags;
        uint64_t dd_used_breakdown[DD_USED_NUM];
        uint64_t dd_pad[14]; /* pad out to 256 bytes for good measure */
} dsl_dir_phys_t;
```

Figure 4-11 DSL Directory Object structure

The second bonus buffer is the `dsl_dataset_phys_t`. This is a DMU_OT_DSL_DATASET object and it is related to the `dsl_dir_phys_t` above. The information that is of interest to our ZDB is the `ds_bp` variable which is a block pointer that gives ZDB further information for its tracing.

```
typedef struct dsl_dataset_phys {
        uint64_t ds_dir_obj;            /* DMU_OT_DSL_DIR */
        uint64_t ds_prev_snap_obj;      /* DMU_OT_DSL_DATASET */
        uint64_t ds_prev_snap_txg;
        uint64_t ds_next_snap_obj;      /* DMU_OT_DSL_DATASET */
        uint64_t ds_snapnames_zapobj;   /* DMU_OT_DSL_DS_SNAP_MAP for snaps */
        uint64_t ds_num_children;       /* clone/snap children for head */
        uint64_t ds_creation_time;      /* seconds since 1970 */
        uint64_t ds_creation_txg;
        uint64_t ds_deadlist_obj;       /* DMU_OT_BPLIST */
        uint64_t ds_used_bytes;
        uint64_t ds_compressed_bytes;
        uint64_t ds_uncompressed_bytes;
        uint64_t ds_unique_bytes;       /* only relevant to snapshots */
        /*
         * The ds_fsid_guid is a 56-bit ID that can change to avoid
         * collisions.  The ds_guid is a 64-bit ID that will never
         * change, so there is a small probability that it will collide.
         */
        uint64_t ds_fsid_guid;
        uint64_t ds_guid;
        uint64_t ds_flags;              /* DS_FLAG_* */
```

[30] `usr/src/uts/common/fs/zfs/sys/dsl_dir.h`

```
        blkptr_t ds_bp;
        uint64_t ds_next_clones_obj;    /* DMU_OT_DSL_CLONES */
        uint64_t ds_props_obj;          /* DMU_OT_DSL_PROPS for snaps */
        uint64_t ds_pad[6]; /* pad out to 320 bytes for good measure */
} dsl_dataset_phys_t;
```

Figure 4-12 DSL Dataset Object structure

The final data structure that is of importance to our ZDB is the znode. This is also stored in the bonus buffer of a dnode. It contains attributes for files and directories (time stamps, ownership, size, etc). This structure is the closest equivalent to the disk inode for UFS. Znode is defined in `znode.h`[31].

```
typedef struct znode_phys {
        uint64_t zp_atime[2];           /*  0 - last file access time */
        uint64_t zp_mtime[2];           /* 16 - last file modification time */
        uint64_t zp_ctime[2];           /* 32 - last file change time */
        uint64_t zp_crtime[2];          /* 48 - creation time */
        uint64_t zp_gen;                /* 64 - generation (txg of creation) */
        uint64_t zp_mode;               /* 72 - file mode bits */
        uint64_t zp_size;               /* 80 - size of file */
        uint64_t zp_parent;             /* 88 - directory parent (`..') */
        uint64_t zp_links;              /* 96 - number of links to file */
        uint64_t zp_xattr;              /* 104 - DMU object for xattrs */
        uint64_t zp_rdev;               /* 112 - dev_t for VBLK & VCHR files */
        uint64_t zp_flags;              /* 120 - persistent flags */
        uint64_t zp_uid;                /* 128 - file owner */
        uint64_t zp_gid;                /* 136 - owning group */
        uint64_t zp_zap;                /* 144 - extra attributes */
        uint64_t zp_pad[3];             /* 152 - future */
        zfs_acl_phys_t zp_acl;          /* 176 - 263 ACL */
        /*
         * Data may pad out any remaining bytes in the znode buffer, eg:
         *
         * |<---------------------- dnode_phys (512) ------------------------>|
         * |<-- dnode (192) --->|<----------- "bonus" buffer (320) ---------->|
         *                      |<---- znode (264) ---->|<---- data (56) ---->|
         *
         * At present, we use this space for the following:
         *  - symbolic links
         *  - 32-byte anti-virus scanstamp (regular files only)
         */
} znode_phys_t;
```

Figure 4-13 Znode structure definition

4.5 ZFS vs UFS

Historically, traditional Unix file systems like UFS have been constrained to one device so that the file systems themselves have been constrained to the size of the device. Creating and re-creating traditional file systems because of size constraints are time-consuming and sometimes difficult. The ZFS file systems are not constrained to specific devices, they can be created easily and quickly, similar to the way directories are created. ZFS file systems grow automatically within the space allocated to the storage pool.

ZFS is based on a concept of pooled storage. Unlike typical file systems, which are mapped to physical storage, all ZFS file systems in a pool share the available storage in the pool.

ZFS is a transactional file system. Most file system modifications are bundled into transaction groups and committed to disk asynchronously. Until these modifications are committed to disk, they are pending changes. This increases data integrity by maintaining a consistent file system at all time.

[31] usr/src/uts/common/fs/zfs/sys/zfs_znode.h

ZFS operates on raw devices. It eliminates the need for a separate volume manager. This allows ZFS to manage the physical device without extra complexity of any volume management software.

All of the above features of ZFS resulted in a very unique implementation of ZFS. As shown in Section 4.3 and 4.4, the ZFS internal architecture is different to those of traditional Unix file systems. This means that the existing file system forensic tools like TSK and TCT is not equipped to understand the ZFS internal structures. That makes existing file system forensic software incompatible with the ZFS file system.

4.6 Summary

This section has explained new concepts of ZFS and has identified that it is a new, dynamic and robust file system technology compared with the rigid traditional Unix file system design. ZFS is a well structured file system. It contains seven main components which forms a structural architecture for the ZFS file system. The implementation of selected layers have been explained where it relates to our new ZDB extension. Finally, a comparison between ZFS and UFS is presented, showing the difference between the designs of the two file systems prevents existing file system forensic tools to work with ZFS.

5 NEW ZDB FEATURE

5.1 Overview

In this section, we present the extension we made to the ZFS file system debugger (ZDB) which enables a user to traverse through the file system to reach the actual data stored on the disk without invoking the file system layer.

The remainder of this section is divided into the following structure: Section 5.2 specifies the requirement for building the new ZDB extension; Section 5.3 provides a high level overview of the new ZDB extension; Section 5.4 explains the extension in detail by referring to the OpenSolaris ZFS and ZDB source code. Section 5.5 will go through a demonstration of the new ZDB, showing how a deleted file is recovered. Readers may refer back to Figure 4-3 to help understand the behavior of the new ZDB feature to see how our ZDB travels through the ZFS file system layers.

5.2 Requirement For ZDB Extension

The source code of ZFS and ZDB are open sourced under the Common Development and Distribution License (CDDL[32]) Version 1.0. The header files of the ZFS structures mentioned in this section can be found in the directory `usr/src/uts/common/fs/zfs/sys` in the OpenSolaris source code[33] and the code for ZDB can be found at `usr/src/uts/cmd/zdb`. To compile and build any part of the OpenSolaris source tree, a copy of the Sun Studio 12 compiler is required. It can be downloaded from: http://www.opensolaris.org/os/community/tools/sun_studio_tools/sun_studio_12_tools/.

To compile our new ZDB, first use the `bldenv` command from the SUNWonbld package to setup all the necessary environment variables. Next change directory to the ZDB subtree to compile the ZDB code. For example, if the OpenSolaris source tree is uncompressed to the /opensolaris_src directory:

```
$ cd /opensolaris_src
$ bldenv -d ./opensolaris.sh
$ cd usr/src/cmd/zdb
$ dmake all
```

On an x86 machine, the result of this build will generate the zdb binary which will be stored in the `usr/src/cmd/zdb/i386` directory.

Refer to the article Build/Install OpenSolaris[34] by Richard Teer, 2005 for a complete guide to building the whole OpenSolaris source tree.

5.3 Overview of New ZDB Extension

At a high level, the following steps are carried out by our new ZDB to retrieve the file content of a newly created file without using the file system layer of the operating system.

1. Retrieve the active ZFS uberblock and its block pointer
2. Retrieve the dnode for the metadata object set
3. Retrieve the Object Directory dnode and its ZAP object
4. Retrieve the DSL Directory object

[32] http://opensolaris.org/os/licensing/opensolaris_license/
[33] http://opensolaris.org/os/downloads/on/
[34] http://opensolaris.org/os/community/tools/building_opensolaris/

5. Retrieve the DSL Dataset object
6. Using the DSL Dataset dnode, retrieve the ZFS file system object set
7. Using the ZFS file system object, get the Master dnode and its ZAP object
8. From the ZAP object of the Master dnode, get the root directory dnode of the ZFS file system
9. From the block pointer of the root directory, find the object id of our target file
10. Using the address stored in the object id dnode, retrieve the block of data directly from the disk and output the raw data.

In summary, the above procedure allows ZDB to retrieves a chunk of data directly from the disk which contains the file content that we are searching for.

In a digital crime scene investigation, this new feature of ZDB will be useful because the investigator can use this tool to examine the disk media without the file system layer in the middle which can intervene with the examination. In a normal day to day operation, when a file is accessed via the operating system through the file system layer, the metadata of the file will be modified. The last access time, modification time, file owner, file size, and permission may change due to the nature of the file system. With the new ZDB feature, the file system is not invoked when the file is being accessed. Therefore, there is no record of the file being accessed, thus nothing on the file system will be updated and the file content and metadata remains untouched.

To achieve this, ZDB travels through the ZFS internal to retrieve clues from one layer leading to the next until it arrives at the final physical layer on the disk where the target data is stored. It is recommended that readers should understand the ZFS architecture described in Section 4 before continuing with the remaining parts of this section.

5.4 Detail Implementation of New ZDB Extension

This section provides a detailed explanation of our new version of ZDB. The new ZDB traverses through the various layers of ZFS using data structures from the ZFS source code. As it is a complex layout, readers may wish to refer back to Figure 4-3 and the high level overview in Section 5.3 when reading this section. For the technically inclined readers, you may wish to refer to the ZDB source code in Appendix A in parallel to fully understand the new ZDB feature we propose.

All of the changes for our ZDB are done in the `usr/src/cmd/zdb/zdb.c` file. It contains the main body of the ZDB code. Our new feature of ZDB introduced a new command line option for tracing the active uberblock back to the data. We have called this option –T, an example usage of our new ZDB on a ZFS pool named “example_zpool” is:

```
# zdb –T example_zpool
```

To enable the –T options, we modified the getopt loop in the main function body to accept an additional T option (Appendix A: Line 287-288) and the usage function to include a short description of the –T option (Appendix A: Line 23-24).

Step 1: Retrieving active uberblock and block pointer

The first step of the new ZDB extension is to retrieve an active uberblock from the uberblock array within the vdev label of the ZDB pool. Recall from Section 4.4 that each uberblock is stored in an `uberblock_t` structure defined in the header file `uberblock_impl.h`[35]. ZDB uses the `dump_config()` and `dump_uberblock()` function in `zdb.c` to display the details of the ZFS storage pool and the uberblock to screen.

[35] `uts/common/fs/zfs/sys/uberblock_impl.h`

The active uberbock contains a block pointer structure `blkptr_t` that is used to locate, describe and verify blocks on disk. Block pointers are defined in the header file `spa.h`[36]. The block pointer contains copies of Data Virtual Address (DVA) which describes the metadata in a ZFS file system. If we focus on the block pointer definition properly, we can see that there are actually three copies of DVA stored in the DVA variable type (See the `dva_t` variable in Figure 4-6 Block pointer structure in Section 4.4). The three DVAs actually hold the same data and they are called "ditto blocks". Ditto blocks are used for most of the metadata in a ZFS file system to provide a redundancy mechanism in case of failures.

The algorithm for this step of the ZDB is shown below. The complete C code is located in Appendix A: Line 334-345.

```
dump_config(argv[0]);
dump_uberblock(uberblock);
display_blkptr(block pointer from uberblock, poolname);
```

The `display_blkptr()` function displays all information about a block pointer. In this case, we pass in the block pointer from the uberblock. The most interesting piece of information is the DVA because it gives us the address location for our next piece of clue.

Step 2: Retrieving dnode for the metadata object set

The next task for the new ZDB is to make use of the DVA from the uberblock block pointer. This address points to a location on the disk that stores the metadata which describes the metadata object set. Our ZDB use the function `retrieve_blkptr()` to retrieve the metadata object set using the DVA from the uberblock. This metadata is described by the `dnode_phys_t` structure defined in `dnode.h`[37]. This interaction between the uberblock and the metadata object set demonstrates the relationship between the SPA layer and the DMU layer. As mentioned previously in Section 3, almost everything in ZFS is an object and all objects are described by a dnode. The `dnode_phys_t` contains another block pointer (See the `dn_blkptr` variable in Figure 4-8 Dnode structure definition in Section 4.4).

Similar to the block pointer from the uberblock, this block pointer also contains DVA addresses. This time the address points to a location on the disk containing an array of dnodes which makes up the metadata object set. The metadata object set is described by an `objset_phys_t` structure defined in `dmu_objset.h`[38] (See the metadata object set definition in Figure 4-9 Object Set structure). Our ZDB will now loop through the metadata object set array and display the block pointer from each dnode object with the display_blkptr() function.

The algorithm for this step of the ZDB is shown below. The complete C code is located in Appendix A: Line 347-395.

```
retrieve_blkptr(block pointer);
get metadata object set dnode;
display_blkptr(block pointer from metadata object set, poolname);
for (the size of the block pointer list) {
    display_blkptr(block pointer from dnode, poolname);
```

[36] `uts/common/fs/zfs/sys/spa.h`
[37] `uts/common/fs/zfs/sys/dnode.h`
[38] `uts/common/fs/zfs/sys/dmu_objset.h`

```
}
```

Step 3: Retrieving Object Directory dnode and its ZAP object

The new ZDB will now retrieve the Object Directory dnode within the metadata object set. An object directory is a ZAP object, it stores attributes for a ZFS object. This Object Directory dnode is always stored in the first element of the metadata object set array. The ZAP object used here is described by the structure `mzap_phys_t` and it is defined in `zap_impl.h`[39]. The ZAP object contains details of the root DSL directory for the storage pool. It describes all the top level dataset within the pool.

The algorithm for this step of the ZDB is shown below. The complete C code is located in Appendix A: Line 397-422.

```
retrieve_blkptr(first block pointer from metadata object set);
for (the number of ZAP objects) {
    if (object is a Micro-ZAP object) {
           get the position of DSL directory in the metadata object set array;
    }
}
```

Step 4: Retrieving DSL Directory object

The DSL Directory object is stored somewhere in the metadata object set that was retrieved initially from the uberblock in step 1. The ZAP object contains the location of the DSL Directory object inside the metadata object set. Recall that the metadata object set is an array of dnode and we have stored the position of the DSL directory object in the previous step. ZDB will now retrieve the dnode to obtain the DSL Directory object by copying the data from the metadata object set array to a dnode using the C `memcpy` function. This DSL Directory object is described by the `dsl_dir_phys_t` structure defined in `dsl_dir.h`[40], it gives us the next piece of information for retrieving the DSL Dataset object. This `dsl_dir_phys_t` is stored in the `dn_bonus` variable in our dnode (See Figure 4-11 DSL Directory Object structure in Section 4.4).

The algorithm for this step of the ZDB is shown below. The complete C code is located in Appendix A: Line 423-433.

```
memcpy(address of the DSL directory object into a dnode variable);
memcpy(address of dn_bonus into a dsl_dir_phys_t variable);
```

Step 5: Retrieving DSL Dataset object

The new ZDB now retrieves the DSL Dataset object using information from the DSL Directory object from the previous step. The `dd_head_dataset_obj` variable in the DSL Directory dnode stores the DSL Dataset object (See Figure 4-12 DSL Dataset Object structurein Section 4.4), ZDB will retrieve this variable by `memcpy`. The DSL Dataset object is stored in the structure `dsl_dataset_phys_t` which is also defined in `dsl_dir.h`. The `dsl_dataset_phys_t` is obtained

[39] `uts/common/fs/zfs/sys/zap_impl.h`
[40] `uts/common/fs/zfs/sys/dsl_dir.h`

from the `dn_bonus` buffer in the dnode of the `dd_head_dataset_obj` object, which contains a `blkptr_t`. Similar to step 4, ZDB uses `memcpy` to copy the `dn_bonus` buffer into a `dsl_dataset_phys_t` variable. Finally, the details of the block pointer will be displayed on screen using the `display_blkptr()` function. The algorithm for this step of the ZDB is shown below. The complete C code is located in Appendix A: Line 434-444.

```
memcpy(DSL Directory Object->dd_head_dataset_obj into a dnode variable);
memcpy(DSL Directory Object dnode->dn_bonus into a dsl_dataset_phys_t variable);
display_blkptr(buffer, block pointer from  dsl_dataset_phys_t, poolname);
```

Note: The DSL *Directory* Object in Step 4 and DSL *Dataset* Object in Step 5 refers to different structures in ZFS.

Step 6: Retrieving ZFS file system object set

The block pointer from the DSL Dataset Object contains the DVA of the root dataset of the file system. ZDB will now grab this chunk of data from the disk and use it in the next step. This step is simple, it is just a call to `retrieve_blkptr()` using the block pointer from step 5. See line 446-451 in Appendix A.

Step 7: Retrieving Master dnode and its ZAP object

Like everything else, the root dataset of the file system is another object dnode. This dnode contains a block pointer which will lead to the Master node. It maybe necessary to go through a few level of indirections to get to the Master node. The `blkptr_t` from the root dataset contains a variable `dn_nlevels` that specifies the level of indirection. If the `dn_nlevels` is one, it means that the `blkptr_t` points to another `blkptr_t` which points to the Master node object set array. Our ZDB will trace through the `blkptr_t` chain to arrive to the Master node object set and retrieve the ZAP object of the Master node. Recall from step 3 that the Object Directory is always the first element of the object set array, the same happens here. ZDB will retrieve the Master node ZAP object from the first element of the Master node object set. This ZAP object is stored in a `mzap_phys_t` structure which is defined in `zap_impl.h`[41].

The algorithm for this step of the ZDB is shown below, refer to Appendix A: Line 458-502 for the complete C code.

```
for (each level of indirections) {
     retrieve_blkptr(block pointer);
     display_blkptr(block pointer, poolname);
}
// We have arrived to the bottom of the block pointer chain
// Therefore the next block pointer points to the Master dnode object set
get Master node object set with retrieve_blkptr(block pointer);
for (each dnode in the object set) {
     memcpy(a dnode from the object set);
     display_blkptr(block pointer from the dnode, poolname);
```

[41] `uts/common/fs/zfs/sys/zap_impl.h`

```
}
get first block pointer with retrieve_blkptr();
memcpy(ZAP object into a mzap_phys_t variable);
```

Step 8: Retrieving root directory dnode

Once our ZDB arrives at the Master node, the Master node contains the DVA which points to another array of dnode. Note that this is the second array of dnode. The first array is the array of dnode that makes up the metadata object set obtained from the uberblock. The ZAP object of the Master node contains an object id which tells us where the root directory of the ZFS file system is located. Using the object id, we can locate the root directory from the Master node dnode array.

ZDB now search through the ZAP objects from step 7 and identify any ZAP object where the `mze_name` is "ROOT". Once found, the object id for that ZAP object will be recorded, and it will be used in the next step for retrieving the root directory znode. The znode structure is defined in `zfs_znode.h`[42] The algorithm is shown below, and the code is in Appendix A: Line 504-549.

```
for (each Master node ZAP object) {
     if (mze_name == "ROOT") {
            saves the object id of this object;
     }
}
memcpy(the ROOT directory dnode);
display_blkptr(block pointer of the dnode, poolname);
get the ZAP object array pointer with retrieve_blkptr(blkptr);
```

Step 9: Retrieving object id of our target file

The root directory dnode from the Master node dnode array contains a bonus buffer. This bonus buffer is a `znode_phys_t` structure that contains attributes like time stamps, ownership, and size of the file or directory. This `znode_phys_t` structure is defined in `znode.h`[43], its purpose is similar to an inode for a UFS file system. Using the object id of the ROOT ZAP object from step 8, we can identify the directory content znode within the directory dnode array. ZDB will use `memcpy` to copy the dnode from the object set array.

Once again, the block pointer of the dnode will be used to retrieve the next clue. The dnode we have just copied contains a bonus pointer. This bonus pointer is a znode that contains the metadata (e.g. access time, owner, change time, etc) for the directory entry for our file. C code is located at Appendix A: Line 551-590.

```
memcpy(dnode pointed to by object id in the root directory object set);
display_blkptr(block pointer from the dnode, poolname);
memcpy(bonus buffer of the block pointer to a znode);
retrieve_blkptr() to get the ZAP object directory block pointer;
for (each ZAP object) {
```

[42] `usr/src/uts/common/fs/zfs/sys/zfs_znode.h`
[43] `usr/src/uts/common/fs/zfs/sys/znode.h`

```
        search for the position of the ZAP object for the directory entry;
    }
```

Step 10: Retrieving final target file

We have now arrived at the final step to retrieve the data block on the disk. The root directory dnode from the Master node dnode array contains a block pointer that points to the target file. Our ZDB will use the DVA to retrieve a block of data from the disk. This data will be the content of the file that we are searching for. This completes our extension to ZDB.

Using the object id from step 9, we can locate the dnode from the directory entry dnode array. ZDB will again use memcpy to copy the address of the target dnode to store it into a dnode variable. As usual, the block pointer will contain three DVA addresses. This time the DVA will point to the actual data of the target file and the trace stops here.

Below is the algorithm of the steps involved, the C code is located at Appendix A: Line 592-620.

```
get the final dnode with memcpy(address of object id in object set array);
display_blkptr(final dnode, poolname);
retrieve_blkptr(block pointer of final dnode);
memcpy(DVA of the block pointer into a char buffer);
display the content of the final file pointed to by the DVA of the block pointer;
```

5.5 Demonstration

After examining our ZDB extension implementation, we will demonstrate how it works by creating a simple file recovery scenario. The output of our ZDB is quite verbose, it prints out details of each step as we have described in Section 5.4, by tracing through the example below thoroughly, the implementation of our ZDB extension will become clearer. The following steps are carried out:

Step 1: Create a new ZFS file system

The example below uses the loopback file driver to create a temporary dummy device. Normally the ZFS file system would be created using a disk slice or a whole disk, not a 100MB temporary file as we have done here.

```
# mkfile 100M /export/stage/foo
# ls -l /export/stage/
total 204577
-rw------T   1 root     root     104857600 Jun  4 12:11 foo
# lofiadm -a /export/stage/foo
/dev/lofi/1
# zpool create foo_pool /dev/lofi/1
# zfs create foo_pool/foo_fs
# df -k
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0d0s0      6050982 1703570 4286903    29%    /
... <output cropped> ...
/dev/dsk/c0d0s4      8068883 6842011 1146184    86%    /usr
/usr/lib/libc/libc_hwcap1.so.1
                     8068883 6842011 1146184    86%    /lib/libc.so.1
fd                         0       0       0     0%    /dev/fd
swap                  594516      48  594468     1%    /tmp
swap                  594520      52  594468     1%    /var/run
/dev/dsk/c0d0s3      7064379 4379869 2613867    63%    /opt
```

```
export                18450432      22 9595187     1%    /export
export/home           10485760 8749962 1735797    84%    /export/home
export/stage          18450432  102436 9595187     2%    /export/stage
foo_pool                 65024      20   64924     1%    /foo_pool
foo_pool/foo_fs          65024      19   64924     1%    /foo_pool/foo_fs
```

The command below lists all ZFS file systems on the file system. As expected, we see the ZFS storage pool `foo_pool` and the ZFS file system `foo_fs`.

```
# zfs list
NAME              USED  AVAIL  REFER  MOUNTPOINT
export           8.45G  9.15G    22K  /export
export/home      8.35G  1.65G  8.35G  /export/home
export/stage      100M  9.15G   100M  /export/stage
foo_pool          100K  63.4M  21.5K  /foo_pool
foo_pool/foo_fs    19K  63.4M    19K  /foo_pool/foo_fs
```

Step 2: Create a file with known content in the top directory of a mounted ZFS file system

```
# echo "foo foo foo" > /foo_pool/foo_fs/foo_file
# ls -l /foo_pool/foo_fs/
total 2
-rw-r--r--   1 root     root          12 Jun  4 12:59 foo_file
# cat /foo_pool/foo_fs/foo_file
This is foo_file
```

Step 3: Remove the file we have just created

```
# rm /foo_pool/foo_fs/foo_file
# ls -l /foo_pool/foo_fs/
total 0
```

Step 4: Use our new ZDB feature to recover the file we have just deleted

Using our new ZDB with the –T option, ZDB will trace the uberblock of our file back to the file content stored on the disk. Some non critical output from the ZDB command has been cropped to reduce space.

```
# cd ~anli/OpenSolarisSrc/usr/src/cmd/zdb/i386/
# ./zdb -T foo_pool
foo_pool
    version=14
    name='foo_pool'
    state=0
    txg=4
    pool_guid=3809538369000823799
    hostid=695107622
    hostname='elmo'
    vdev_tree
        type='root'
        id=0
        guid=3809538369000823799
        children[0]
                type='disk'
                id=0
                guid=11449085626578759141
                path='/dev/lofi/1'
                phys_path='/pseudo/lofi@0:1'
                whole_disk=0
                metaslab_array=23
                metaslab_shift=19
                ashift=9
                asize=100139008
                is_log=0

Uberblock
```

```
        magic = 0000000000bab10c
        version = 14
        txg = 9
        guid_sum = 15258623995579582940
        timestamp = 1244084355 UTC = Thu Jun  4 12:59:15 2009

Uberblock blkptr

        DVA[0]=<0:19a00:200>
        DVA[1]=<0:121a800:200>
        DVA[2]=<0:240ea00:200>
        LSIZE: 400          PSIZE: 200
        POOL: foo_pool              OBJECT: DMU objset (0xb)
        ENDIAN: LITTLE              GANG: FALSE
        BIRTH: 9            FILL: 33            LEVEL: 0
        CKFUNC: fletcher4   COMP: lzjb
        CKSUM: ecb8a0f3c:5a8da9d0554:11c05b5402ac2:2604356846bf28

Meta Object Set blkptr

        DVA[0]=<0:1a000:c00>
        DVA[1]=<0:1219c00:c00>
        DVA[2]=<0:240f000:c00>
        LSIZE: 4000         PSIZE: c00
        POOL: foo_pool              OBJECT: DMU dnode (0xa)
        ENDIAN: LITTLE              GANG: FALSE
        BIRTH: 9            FILL: 31            LEVEL: 0
        CKFUNC: fletcher4   COMP: lzjb
        CKSUM: a958cbbd86:1222dcfe65e95:12866883806be4c:e01260f97a335994

There are 32 dnode_phys_t in this MOS block array
Tracing the dnode_phys_t array...

        LSIZE: 0            PSIZE: 200
        POOL: foo_pool              OBJECT: unallocated (0x0)
        ENDIAN: BIG         GANG: FALSE
        BIRTH: 0            FILL: 0             LEVEL: 0
        CKFUNC: inherit     COMP: inherit
        CKSUM: 0:0:0:0
        BONUSTYPE: unallocated (0x0)

        DVA[0]=<0:2600:200>
        DVA[1]=<0:1202600:200>
        DVA[2]=<0:2400000:200>
        LSIZE: 200          PSIZE: 200
        POOL: foo_pool              OBJECT: object directory (0x1)
        ENDIAN: LITTLE              GANG: FALSE
        BIRTH: 4            FILL: 1             LEVEL: 0
        CKFUNC: fletcher4   COMP: uncompressed
        CKSUM: 5c1b6746c:1dc2e54e2a9:55cf093c75a8:b374c961df49e
        BONUSTYPE: unallocated (0x0)

... <output cropped> ...

Tracing ZAP object array mzap_phys_t  ...

        mz_block_type = 0x8000000000000003
        mze_value = 0x2
        mze_name = root_dataset

        mz_block_type = 0x14
        mze_value = 0x1
        mze_name = deflate

        mz_block_type = 0x15
        mze_value = 0x16
        mze_name = history

        mz_block_type = 0x0
        mze_value = 0x0
```

```
        mze_name =

Displaying DSL directory object from bonus buffer at &(MOS array)+2

        dd_creation_time = 0x4a273841
        dd_head_dataset_obj = 0x10

Displaying blkptr of the DSL Dataset object from bonus buffer at &(MOS array)+16

        DVA[0]=<0:15c00:200>
        DVA[1]=<0:1215c00:200>
        LSIZE: 400          PSIZE: 200
        POOL: foo_pool              OBJECT: DMU objset (0xb)
        ENDIAN: LITTLE              GANG: FALSE
        BIRTH: 8            FILL: 6             LEVEL: 0
        CKFUNC: fletcher4   COMP: lzjb
        CKSUM: 9a17229fe:40656a99dde:da0a862f8c39:1f277eaf4ab262

Get root dataset metadata from DSL Object Set blkptr

        DVA[0]=<0:15800:400>
        DVA[1]=<0:1215800:400>
        LSIZE: 4000         PSIZE: 400
        POOL: foo_pool              OBJECT: DMU dnode (0xa)
        ENDIAN: LITTLE              GANG: FALSE
        BIRTH: 8            FILL: 5             LEVEL: 6
        CKFUNC: fletcher4   COMP: lzjb
        CKSUM: 59483d2ad5:3df7ceb87fc7:16bd26c7c03c44:5d6c1d1dc20374e

        DVA[0]=<0:15400:400>
        DVA[1]=<0:1215400:400>
        LSIZE: 4000         PSIZE: 400
        POOL: foo_pool              OBJECT: DMU dnode (0xa)
        ENDIAN: LITTLE              GANG: FALSE
        BIRTH: 8            FILL: 5             LEVEL: 5
        CKFUNC: fletcher4   COMP: lzjb
        CKSUM: 59e8854f51:3e8e97f55c96:1704628551d0f6:5ed49ca86d0a6c5

... <output cropped> ...

There are 32 dnode_phys_t in this dnode_phys_t block array
Tracing the dnode_phys_t array...

        LSIZE: 0            PSIZE: 200
        POOL: foo_pool              OBJECT: unallocated (0x0)
        ENDIAN: BIG         GANG: FALSE
        BIRTH: 0            FILL: 0             LEVEL: 0
        CKFUNC: inherit     COMP: inherit
        CKSUM: 0:0:0:0
        BONUSTYPE: unallocated (0x0)

        DVA[0]=<0:0:200>
        DVA[1]=<0:1200000:200>
        LSIZE: 200          PSIZE: 200
        POOL: foo_pool              OBJECT: ZFS master node (0x15)
        ENDIAN: LITTLE              GANG: FALSE
        BIRTH: 4            FILL: 1             LEVEL: 0
        CKFUNC: fletcher4   COMP: uncompressed
        CKSUM: 2edc13e82:114b5e12053:36a82a27794c:79d23ef4f8592
        BONUSTYPE: unallocated (0x0)

        DVA[0]=<0:200:200>
        DVA[1]=<0:1200200:200>
        LSIZE: 200          PSIZE: 200
        POOL: foo_pool              OBJECT: ZFS delete queue (0x16)
        ENDIAN: LITTLE              GANG: FALSE
        BIRTH: 4            FILL: 1             LEVEL: 0
        CKFUNC: fletcher4   COMP: uncompressed
        CKSUM: 9e496988:4e6821f0f6:1392946b4685:348636736ac00
        BONUSTYPE: unallocated (0x0)
```

```
        DVA[0]=<0:12800:200>
        DVA[1]=<0:1212800:200>
        LSIZE: 200          PSIZE: 200
        POOL: foo_pool              OBJECT: ZFS directory (0x14)
        ENDIAN: LITTLE              GANG: FALSE
        BIRTH: 8            FILL: 1             LEVEL: 0
        CKFUNC: fletcher4   COMP: uncompressed
        CKSUM: 1c115ca3e:ca45c0d476:2e3299fb4c01:7221bc74f4dc4
        BONUSTYPE: ZFS znode (0x11)

... <output cropped> ...

Debug: retrieving Master Node ZAP object blkptr with foo_pool:0:0:200:r

Master Node ZAP object mzap_phys_t  ...

        mz_block_type = 0x8000000000000003
        mze_value = 0x3
        mze_name = VERSION

        mz_block_type = 0x2
        mze_value = 0x3
        mze_name = ROOT

        mz_block_type = 0x4
        mze_value = 0x0
        mze_name =

        mz_block_type = 0x0
        mze_value = 0x0
        mze_name =

Displaying ROOT dnode from &(dnode array)+3

        DVA[0]=<0:12800:200>
        DVA[1]=<0:1212800:200>
        LSIZE: 200          PSIZE: 200
        POOL: foo_pool              OBJECT: ZFS directory (0x14)
        ENDIAN: LITTLE              GANG: FALSE
        BIRTH: 8            FILL: 1             LEVEL: 0
        CKFUNC: fletcher4   COMP: uncompressed
        CKSUM: 1c115ca3e:ca45c0d476:2e3299fb4c01:7221bc74f4dc4
        BONUSTYPE: ZFS znode (0x11)

Debug: retrieving znode from bonus pointer
Znode details
        atime: Thu Jun  4 12:58:09 2009
        mtime: Thu Jun  4 12:58:09 2009
        ctime: Thu Jun  4 12:58:09 2009
        crtime: Thu Jun  4 12:58:09 2009
        size: 3

Debug: retrieving object directory blkptr

Root directory ZAP object mzap_phys_t from bonus pointer ...

        mz_block_type = 0x8000000000000003
        mze_value = 0x4000000000000005
        mze_name = foo_fs

        mz_block_type = 0x0
        mze_value = 0x0
        mze_name =

        mz_block_type = 0x0
        mze_value = 0x0
        mze_name =

        mz_block_type = 0x0
        mze_value = 0x0
        mze_name =
```

```
Displaying dnode from &(dnode array)+5

       DVA[0]=<0:11e00:200>
       DVA[1]=<0:1211e00:200>
       LSIZE: 200          PSIZE: 200
       POOL: foo_pool             OBJECT: ZFS directory (0x14)
       ENDIAN: LITTLE             GANG: FALSE
       BIRTH: 8            FILL: 1             LEVEL: 0
       CKFUNC: fletcher4   COMP: uncompressed
       CKSUM: 9e9ae87e:4e903e6e0a:139c877c4cfb:34a0bea0d1500
       BONUSTYPE: ZFS znode (0x11)

Debug: retrieving znode from bonus pointer
Znode details
       atime: Thu Jun  4 12:58:15 2009
       mtime: Thu Jun  4 12:58:15 2009
       ctime: Thu Jun  4 12:58:15 2009
       crtime: Thu Jun  4 12:58:15 2009

Debug: retrieving ZAP object directory blkptr with foo_pool:0:11e00:200:r

Root directory ZAP object mzap_phys_t from bonus pointer ...

       mz_block_type = 0x8000000000000003
       mze_value = 0x0
       mze_name =

       mz_block_type = 0x0
       mze_value = 0x0
       mze_name =

       mz_block_type = 0x0
       mze_value = 0x0
       mze_name =

       mz_block_type = 0x0
       mze_value = 0x0
       mze_name =

Displaying dnode from &(dnode array)+3

       DVA[0]=<0:12800:200>
       DVA[1]=<0:1212800:200>
       LSIZE: 200          PSIZE: 200
       POOL: foo_pool             OBJECT: ZFS directory (0x14)
       ENDIAN: LITTLE             GANG: FALSE
       BIRTH: 8            FILL: 1             LEVEL: 0
       CKFUNC: fletcher4   COMP: uncompressed
       CKSUM: 1c115ca3e:ca45c0d476:2e3299fb4c01:7221bc74f4dc4
       BONUSTYPE: ZFS znode (0x11)

Debug: retrieving znode from bonus pointer
Znode details
       atime: Thu Jun  4 12:58:09 2009
       mtime: Thu Jun  4 12:58:09 2009
       ctime: Thu Jun  4 12:58:09 2009
       crtime: Thu Jun  4 12:58:09 2009
       size: 12

Debug: retrieving ZAP object directory blkptr with foo_pool:0:12800:200:r

File contains:
foo foo foo
```

This output matches the content of the file we created in Step 1.

5.6 Summary

In summary, the new ZDB makes frequent use of the Data Virtual Address (DVA) from the block pointer variable `blkptr_t` inside a dnode. This DVA address points to different layers of the ZFS file system and eventually leads us to the target file we are searching for. Since almost everything in ZFS is an object, just about every step of the trace involves dealing with a dnode, which is what ZFS uses to store any object.

6 FUTURE WORK

In this study, we have introduced extension in ZDB which takes only the active uberblock and traces it back to the data on disk. When investigating a disk taken from a real crime scene investigation, all files which have been stored inside the file system will need to be recovered. The code in our study was developed with this in mind to ease future enhancement. The majority of the code which performs the file system traversal has already been completed in this study. The future release of our new ZDB will incorporate this code into a loop which loops through the array of uberblock so that each uberblock can lead back to the actual data stored on disk, giving the investigator the file metadata and content of every file stored on the disk.

A ZFS snapshot is a read-only copy of a file system or volume. A snapshot initially consumes no additional disk space within the pool. However, as data within the active dataset changes, the snapshot consumes disk space by continuing to reference the old data and so prevents the space from being freed. Examination of ZFS snapshots will need to be included in future releases of our ZDB. The technique used on a ZFS snapshot will be similar to what has been done in this study for a normal ZFS file system.

Finally, the code from this study could be turned into a set of library function calls. This will enable other system utilities to perform direct file system access and will make the code in ZDB cleaner and easier to maintain, because the complexity has been transferred to the library functions. But the security implications of this will need to be further researched because this will allow user process to have raw access to the disk device. Fine grain access control like the Solaris Role Based Access Control[44] may be required.

In order to have our new ZDB feature included in future releases of OpenSolaris, it will need to go through a process like all other open source projects. All code will need to be posted to the OpenSolaris community for code review. Once the code is reviewed by the OpenSolaris community, the code will need to be submitted via an online application form[45]. After submitting the form, the code will go through another code review process by developers from Sun Microsystems. See the Improving OpenSolaris[46] webpage for a complete description of the code submission process.

[44] http://opensolaris.org/os/community/security/projects/rbac/
[45] http://bugs.opensolaris.org/
[46] http://opensolaris.org/os/communities/participation/

7 CONCLUSION

The work described in this paper presents a proof of concept that a digital forensic tool for ZFS is achievable, unlike the UFS file system where the relationship between the file and the data on disk is removed when a file is deleted, making it harder to trace the data back to the disk. File retrieval is performed by our new feature in ZDB, which travels through the various layers of the ZFS file system until it reaches the target file stored on disk. This means that the data on disk is being accessed directly without intervention from the file system layer of the operating system.

This new feature of ZDB is designed to help a digital crime scene investigator to retrieve evidence from an operating system with a Zettabyte File System. It enables investigators to retrieve data that has been deleted or hidden, which cannot be seen under normal operating system operations. Our new ZDB achieves this by tracing through virtual addresses stored in ZFS block pointers to dig into the ZFS file system layers until the target data is reached. By doing so, the file system layer of the operating system is not invoked and the data stored on the disk can be accessed directly. This enables investigators to gather reliable crime scene evidence.

8 REFERENCE

Bruning, Max. June 2008. *ZFS On-Disk Data Walk*. In OpenSolaris Developer Conference. June 25-27, 2008 Prague.

Card, R. Ts'o, T. & Tweedie, S. 1994. Design and Implementation of the Second Extended Filesystem. First Dutch International Symposium on Linux, Amsterdam, 1994.

Carrier, Brian. March 2005. *File System Forensic Analysis.* Addison Wesley Professional.

Farmer, Dan. & Venema, Wietse. 2005. *Forensic Discovery*. Addison-Wesley Professional.

Gutmann, P. 1996. Secure Deletion of Data from Magnetic and Solid-State Memory. In: Sixth USENIX Security Symposium Proceedings. San Jose, California July 22-25, 1996.

Lions, J., 1977. Unix Operating System Source Code Level Six. The University of New South Wales: Department of Computer Science

Lions, J., 1977. A commentary on the Sixth Edition UNIX Operating System. The University of New South Wales: Department of Computer Science

McDougall, R. & Mauro, J. 2006. The Sun Solaris UFS implementation. In: McDougall, R. & Mauro, J., ed. 2006. Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture. Prentice Hall PTR. Ch. 15.

McKusick, M. Joy, W. Leffler, S. & Fabry, R. 1984. A Fast File System for Unix. University of California, Berkeley: Computer Systems Research Group

Sun Microsystems, Inc. 2006. *ZFS On-Disk Specification*. Sun Microsystems, Inc

9 APPENDIX A – ZDB Source Code

This section shows the changes made to the `zdb.c`[47] source code. The output is generated from the Unix `diff` command. Lines beginning with a "+" means that the line is new and it did not exist in the original code. Lines beginning with a "-" means that this line in the original code. For example, in line 7, the `zap_impl.h` header file is a new include file that we have added for our ZDB extension. In line 45-46, the statement "`void *buf`" in the original `zdb.c` has been changed to "`void *buf, *tbuf;`" in our new `zdb.c`.

```
 1 --- zdb.c.orig  2009-03-05 08:52:54.000000000 +1100
 2 +++ zdb.c       2009-04-27 17:49:34.000000000 +1000
 3 @@ -32,6 +32,7 @@
 4  #include <sys/spa_impl.h>
 5  #include <sys/dmu.h>
 6  #include <sys/zap.h>
 7 +#include <sys/zap_impl.h>
 8  #include <sys/fs/zfs.h>
 9  #include <sys/zfs_znode.h>
10  #include <sys/vdev.h>
11 @@ -63,6 +64,7 @@
12  extern void dump_intent_log(zilog_t *);
13  uint64_t *zopt_object = NULL;
14  int zopt_objects = 0;
15 +int do_zdb_dump = 0;
16  libzfs_handle_t *g_zfs;
17  boolean_t zdb_sig_user_data = B_TRUE;
18  int zdb_sig_cksumalg = ZIO_CHECKSUM_SHA256;
19 @@ -119,6 +121,8 @@
20          (void) fprintf(stderr, "        -p <Path to vdev dir> (use with -e)\n");
21          (void) fprintf(stderr, "        -t <txg> highest txg to use when "
22              "searching for uberblocks\n");
23 +        (void) fprintf(stderr, "        -T trace active uberblock back to data"
24 +            " on disk\n");
25          (void) fprintf(stderr, "Specify an option more than once (e.g. -bb) "
26              "to make only that option verbose\n");
27          (void) fprintf(stderr, "Default is to dump everything non-verbosely\n");
28 @@ -1994,17 +1998,20 @@
29   *
30   *              * = not yet implemented
31   */
32 -static void
33 +extern int zio_decompress_data(int cpfunc, void *src, uint64_t srcsize,
34 +                               void *dest, uint64_t destsize);
35 +
36 +static void*
37  zdb_read_block(char *thing, spa_t **spap)
38  {
39          spa_t *spa = *spap;
40          int flags = 0;
41 -        uint64_t offset = 0, size = 0, blkptr_offset = 0;
42 +        uint64_t offset = 0, size = 0, blkptr_offset = 0, lsize = 0;
43          zio_t *zio;
44          vdev_t *vd;
45 -        void *buf;
46 +        void *buf, *tbuf;
47          char *s, *p, *dup, *pool, *vdev, *flagstr;
48 -        int i, error, zio_flags;
49 +        int i, j, error, zio_flags;
50
51          dup = strdup(thing);
52          s = strtok(dup, ":");
53 @@ -2028,7 +2035,7 @@
54          if (s) {
55                  (void) printf("Invalid block specifier: %s  - %s\n", thing, s);
```

[47] `usr/src/cmd/zdb/zdb.c`

```
 56                 free(dup);
 57 -               return;
 58 +               return NULL;
 59         }
 60
 61         for (s = strtok(flagstr, ":"); s; s = strtok(NULL, ":")) {
 62 @@ -2050,11 +2057,38 @@
 63                         p = &flagstr[i + 1];
 64                         if (bit == ZDB_FLAG_PRINT_BLKPTR)
 65                                 blkptr_offset = strtoull(p, &p, 16);
 66 +                       else if (bit == ZDB_FLAG_DECOMPRESS) {
 67 +                               p = strtok(p, ",");
 68 +                               if (!p) {
 69 +                                       (void) printf("***Must specify compression
algorithm\n");
 70 +                                       free(dup);
 71 +                                       return NULL;
 72 +                               }
 73 +                               for (j = 0; j < ZIO_COMPRESS_FUNCTIONS; j++) {
 74 +                                       if (strncmp(p, zio_compress_table[j].ci_name,
 75                                                                                 +
strlen(zio_compress_table[j].ci_name)) == 0)
 76 +                                               break;
 77 +                               }
 78 +                               if (j >= ZIO_COMPRESS_FUNCTIONS) {
 79 +                                       (void)printf("***Unknown compression type:
'%s'\n", p);
 80 +                                       free(dup);
 81 +                                       return NULL;
 82 +                               }
 83 +                               p = strtok(NULL, ",");
 84 +                               lsize = strtoull(p, &p, 16);
 85 +                               if (lsize <= 0) {
 86 +                                 (void) printf("***Must specify logical size
of decompressed data\n");
 87 +                                       free(dup);
 88 +                                       return NULL;
 89 +                               }
 90 +                       }
 91 +
 92                         if (*p != ':' && *p != '\0') {
 93                                 (void) printf("***Invalid flag arg: '%s'\n", s);
 94                                 free(dup);
 95 -                               return;
 96 -                       }
 97 +                               return NULL;
 98 +                       } else
 99 +                               break;  /* hmmm... */
100                 }
101         }
102
103 @@ -2072,13 +2106,15 @@
104         if (vd == NULL) {
105                 (void) printf("***Invalid vdev: %s\n", vdev);
106                 free(dup);
107 -               return;
108 +               return NULL;
109         } else {
110 -               if (vd->vdev_path)
111 -                       (void) printf("Found vdev: %s\n", vd->vdev_path);
112 -               else
113 -                       (void) printf("Found vdev type: %s\n",
114 -                           vd->vdev_ops->vdev_op_type);
115 +               if (do_zdb_dump) {
116 +                       if (vd->vdev_path)
117 +                               (void) printf("Found vdev: %s\n", vd->vdev_path);
118 +                       else
119 +                               (void) printf("Found vdev type: %s\n",
120 +                                   vd->vdev_ops->vdev_op_type);
121 +               }
122         }
123
```

```
124         buf = umem_alloc(size, UMEM_NOFAIL);
125 @@ -2099,12 +2135,34 @@
126                 goto out;
127         }
128
129 -       if (flags & ZDB_FLAG_PRINT_BLKPTR)
130 +       if (flags & ZDB_FLAG_DECOMPRESS) {
131 +               tbuf = umem_alloc(lsize, UMEM_NOFAIL);
132 +               zio_decompress_data(j, buf, size, tbuf, lsize);
133 +               if (do_zdb_dump)
134 +                       zdb_dump_block_raw(tbuf, lsize, flags);
135 +               buf = umem_alloc(lsize, UMEM_NOFAIL);
136 +               memcpy(buf, tbuf, lsize);
137 +#ifdef DEBUG
138 +               printf("Debug: copying %d bits from tbuf to buf\n\n", lsize);
139 +#endif
140 +               umem_free(tbuf, lsize);
141 +               free(dup);
142 +               return buf;
143 +       } else if (flags & ZDB_FLAG_PRINT_BLKPTR)
144                 zdb_print_blkptr((blkptr_t *)(void *)
145                     ((uintptr_t)buf + (uintptr_t)blkptr_offset), flags);
146 -       else if (flags & ZDB_FLAG_RAW)
147 -               zdb_dump_block_raw(buf, size, flags);
148 -       else if (flags & ZDB_FLAG_INDIRECT)
149 +       else if (flags & ZDB_FLAG_RAW) {
150 +               if (do_zdb_dump)
151 +                       zdb_dump_block_raw(buf, size, flags);
152 +               tbuf = umem_alloc(size, UMEM_NOFAIL);
153 +               memcpy(tbuf, buf, size);
154 +#ifdef DEBUG
155 +               printf("Debug: copying %d bits from buf to tbuf\n\n", size);
156 +#endif
157 +               umem_free(buf, size);
158 +               free(dup);
159 +               return tbuf;
160 +       } else if (flags & ZDB_FLAG_INDIRECT)
161                 zdb_dump_indirect((blkptr_t *)buf, size / sizeof (blkptr_t),
162                     flags);
163         else if (flags & ZDB_FLAG_GBH)
164 @@ -2115,6 +2173,7 @@
165  out:
166         umem_free(buf, size);
167         free(dup);
168 +       return NULL;
169  }
170
171  static boolean_t
172 @@ -2243,6 +2302,102 @@
173         return (error);
174  }
175
176 +void display_blkptr(char *blkbuf, blkptr_t *bp, spa_t *spa, const char* poolname) {
177 +       int i, bigendian, gang, nlevel;
178 +       u_longlong_t bplsize, bppsize;
179 +       u_longlong_t dva_vdev, dva_offset, dva_asize;
180 +       uint64_t asize, blkptr_offset;
181 +       dva_t *dva;
182 +
183 +       printf("\n");
184 +       for (i = 0; i < BP_GET_NDVAS(bp); i++) {
185 +           const dva_t *dva = &bp->blk_dva[i];
186 +
187 +           (void) printf("\tDVA[%d]=<%llu:%llx:%llx>\n", i,
188 +                       (u_longlong_t)DVA_GET_VDEV(dva),
189 +                       (u_longlong_t)DVA_GET_OFFSET(dva),
190 +                       (u_longlong_t)DVA_GET_ASIZE(dva));
191 +       }
192 +
193 +       /* Gather details of blkptr */
194 +       dva = &bp->blk_dva[0];
195 +       bplsize = (u_longlong_t)BP_GET_LSIZE(bp);
```

```
196 +       bppsize = (u_longlong_t)BP_GET_PSIZE(bp);
197 +       dva_vdev = (u_longlong_t)DVA_GET_VDEV(dva);
198 +       dva_offset = (u_longlong_t)DVA_GET_OFFSET(dva);
199 +       dva_asize = (u_longlong_t)DVA_GET_ASIZE(dva);
200 +       bigendian = !BP_GET_BYTEORDER(bp);
201 +       nlevel = BP_GET_LEVEL(bp);
202 +       gang = BP_IS_GANG(bp);
203 +       asize = bp_get_dasize(spa, bp);
204 +
205 +       /* Display details of the blkptr */
206 +       (void) printf("\tLSIZE: %llx\t\tPSIZE: %llx\n"
207 +                     "\tPOOL: %s\t\tOBJECT: %s (0x%x)\n"
208 +                     "\tENDIAN: %s\t\tGANG: %s\n"
209 +                     "\tBIRTH: %llu\t\tFILL: %llu\t\tLEVEL: %-2d\n"
210 +                     "\tCKFUNC: %s\tCOMP: %s\n"
211 +                     "\tCKSUM: %llx:%llx:%llx:%llx\n",
212 +                     (u_longlong_t)bplsize, (u_longlong_t)bppsize,
213 +                     poolname, dmu_ot[BP_GET_TYPE(bp)].ot_name,
214 +                     (int)BP_GET_TYPE(bp),
215 +                     BP_GET_BYTEORDER(bp) == 0 ? "BIG" : "LITTLE",
216 +                     BP_IS_GANG(bp) ? "TRUE" : "FALSE",
217 +                     (u_longlong_t)bp->blk_birth,
218 +                     (u_longlong_t)bp->blk_fill, nlevel,
219 +                     zio_checksum_table[BP_GET_CHECKSUM(bp)].ci_name,
220 +                     zio_compress_table[BP_GET_COMPRESS(bp)].ci_name,
221 +                     (u_longlong_t)bp->blk_cksum.zc_word[0],
222 +                     (u_longlong_t)bp->blk_cksum.zc_word[1],
223 +                     (u_longlong_t)bp->blk_cksum.zc_word[2],
224 +                     (u_longlong_t)bp->blk_cksum.zc_word[3]);
225 +
226 +       /*
227 +        * Read metadata from blkptr, this will be used in the function
228 +        * retrieve_blkptr()
229 +        */
230 +       if (BP_GET_COMPRESS(bp) == 2)
231 +               (void) sprintf(blkbuf, "%s:%llu:%llx:%llx:r",
232 +                       poolname,
233 +                       (u_longlong_t)dva_vdev,
234 +                       (u_longlong_t)dva_offset,
235 +                       (u_longlong_t)dva_asize);
236 +       else
237 +               (void) sprintf(blkbuf, "%s:%llu:%llx:%llx:d,%s,%x",
238 +                               poolname,
239 +                               (u_longlong_t)dva_vdev,
240 +                               (u_longlong_t)dva_offset,
241 +                               (u_longlong_t)dva_asize,
242 +                               zio_compress_table[BP_GET_COMPRESS(bp)].ci_name,
243 +                               (u_longlong_t)bplsize);
244 +}
245 +
246 +void *retrieve_blkptr(char* blkbuf) {
247 +       spa_t *spa;
248 +       void *object;
249 +
250 +       /*
251 +        * Read metadata from uberblock blkptr like zdb_read_block().
252 +        * This needs to simulate the -R option.
253 +        * Call zdb_read_block(), get the return pointer and
254 +        * assign it to a dnode_phys_t
255 +        */
256 +       flagbits['b'] = ZDB_FLAG_PRINT_BLKPTR;
257 +       flagbits['c'] = ZDB_FLAG_CHECKSUM;
258 +       flagbits['d'] = ZDB_FLAG_DECOMPRESS;
259 +       flagbits['e'] = ZDB_FLAG_BSWAP;
260 +       flagbits['g'] = ZDB_FLAG_GBH;
261 +       flagbits['i'] = ZDB_FLAG_INDIRECT;
262 +       flagbits['p'] = ZDB_FLAG_PHYS;
263 +       flagbits['r'] = ZDB_FLAG_RAW;
264 +
265 +       spa = NULL;
266 +       object = zdb_read_block(blkbuf, &spa);
267 +       if (spa)
```

```
268 +               spa_close(spa, (void *)zdb_read_block);
269 +       return object;
270 +}
271 +
272  int
273  main(int argc, char **argv)
274  {
275 @@ -2255,6 +2410,7 @@
276         int verbose = 0;
277         int error;
278         int exported = 0;
279 +       int trace = 0;
280         char *vdev_dir = NULL;
281
282         (void) setrlimit(RLIMIT_NOFILE, &rl);
283 @@ -2262,7 +2418,7 @@
284
285         dprintf_setup(&argc, argv);
286
287 -       while ((c = getopt(argc, argv, "udibcsvCLS:U:lRep:t:")) != -1) {
288 +       while ((c = getopt(argc, argv, "udibcsvCLS:U:lRep:t:T")) != -1) {
289                 switch (c) {
290                 case 'u':
291                 case 'd':
292 @@ -2317,6 +2473,11 @@
293                                         usage();
294                                 }
295                                 break;
296 +               case 'T':
297 +                       dump_opt[c]++;
298 +                       dump_all = 0;
299 +                       trace++;
300 +                       break;
301                 default:
302                         usage();
303                         break;
304 @@ -2364,6 +2525,7 @@
305                 flagbits['i'] = ZDB_FLAG_INDIRECT;
306                 flagbits['p'] = ZDB_FLAG_PHYS;
307                 flagbits['r'] = ZDB_FLAG_RAW;
308 +               do_zdb_dump = 1;
309
310                 spa = NULL;
311                 while (argv[0]) {
312 @@ -2447,6 +2609,318 @@
313                 spa_close(spa, FTAG);
314         }
315
316 +       if (dump_opt['T']) {
317 +               uberblock_t *ub;
318 +               blkptr_t *bp;
319 +               dva_t *dva;
320 +               int ndnode, nzap, i, dslpos, nlevel;
321 +                     char blkbuf[BP_SPRINTF_LEN], objdir_buf[BP_SPRINTF_LEN], *tmp,
*poolname;
322 +               uint64_t asize, blkptr_offset;
323 +               u_longlong_t bplsize, bppsize;
324 +               void *objset_array, *zap_obj;
325 +               dnode_phys_t *mos, *dnodephys, *objset;
326 +               mzap_phys_t *zapobj, *zapobjptr;
327 +               mzap_ent_phys_t *zapent, *zapentptr;
328 +               objset_phys_t *metadata;
329 +               dsl_dir_phys_t *ddobj;
330 +               dsl_dataset_phys_t *ddset;
331 +               znode_phys_t *zn;
332 +               timestruc_t now;
333 +
334 +               /* Display pool config and uberblock */
335 +               dump_config(argv[0]);
336 +               (void) printf("\n");
337 +               ub = &spa->spa_uberblock;
338 +               poolname = malloc(strlen(spa->spa_name)+1);
```

```
339 +                strncpy(poolname,spa->spa_name,strlen(poolname));
340 +                dump_uberblock(ub);
341 +
342 +                /* Display details of the active uberblock blkptr */
343 +                bp = &ub->ub_rootbp;
344 +                printf("Uberblock blkptr\n");
345 +                display_blkptr(blkbuf, bp, spa, poolname);
346 +
347 +                /* Get metadata from this blkptr */
348 +#ifdef DEBUG
349 +                printf("\nDebug: retrieving ditto block from %s\n", blkbuf);
350 +#endif
351 +                metadata = retrieve_blkptr(blkbuf);
352 +
353 +                /*
354 +                 * Display details of the MOS blkptr and track indirect
355 +                 * block back to level 0
356 +                 */
357 +                mos = &metadata->os_meta_dnode;
358 +                bp = mos->dn_blkptr;
359 +                printf("\nMeta Object Set blkptr\n");
360 +                display_blkptr(blkbuf, bp, spa, poolname);
361 +                nlevel = (int)BP_GET_LEVEL(bp);
362 +                for (i=0; i<nlevel; i++) {
363 +                        bp = (blkptr_t *)retrieve_blkptr(blkbuf);
364 +                        display_blkptr(blkbuf, bp, spa, poolname);
365 +                }
366 +
367 +                bplsize = (u_longlong_t)BP_GET_LSIZE(bp);
368 +                ndnode = (int)bplsize / sizeof(dnode_phys_t);
369 +                    printf("\nThere are %d dnode_phys_t in this MOS block array\n",
ndnode);
370 +                printf("Tracing the dnode_phys_t array...\n");
371 +
372 +                /* Read metadata from the MOS blkptr array */
373 +#if DEBUG
374 +                printf("\nDebug: retrieving ditto block from %s\n", blkbuf);
375 +#endif
376 +                objset_array = (void *)retrieve_blkptr(blkbuf);
377 +                objset = (dnode_phys_t *)objset_array;
378 +
379 +                /* Display details of the MOS blkptr array */
380 +                dnodephys = umem_alloc(sizeof(dnode_phys_t), UMEM_NOFAIL);
381 +                for (i=0; i<ndnode; i++) {
382 +#if DEBUG
383 +                         printf("\nDebug: copying %d bytes from objset+%lx (0x%lx) to
dnodephys\n",
384 +                                        sizeof(dnode_phys_t), i*sizeof(dnode_phys_t),
objset+i);
385 +#endif
386 +                        memcpy(dnodephys, objset+i, sizeof(dnode_phys_t));
387 +                        /* Object directory is always id 1 */
388 +                        if (i == 1)
389 +                                display_blkptr(objdir_buf, dnodephys->dn_blkptr, spa,
poolname);
390 +                        else
391 +                                    display_blkptr(blkbuf, dnodephys->dn_blkptr, spa,
poolname);
392 +                        printf("\tBONUSTYPE: %s (0x%x)\n",
393 +                                  dmu_ot[dnodephys->dn_bonustype].ot_name, dnodephys-
>dn_bonustype);
394 +                }
395 +                umem_free(dnodephys, sizeof(dnode_phys_t));
396 +
397 +                /* Retrieve object directory */
398 +#if DEBUG
399 +                printf("\nDebug: retrieving object directory blkptr with %s\n",
400 +                        objdir_buf);
401 +                printf("Debug: sizeof mzap_phys_t is %d\n"
402 +                        "Debug: sizeof mzap_ent_phys_t is %d\n",
403 +                        sizeof(mzap_phys_t),sizeof(mzap_ent_phys_t));
404 +#endif
```

```
405 +                nzap = 512 / sizeof(mzap_phys_t);
406 +                zap_obj = (void *)retrieve_blkptr(objdir_buf);
407 +                zapobjptr = (mzap_phys_t *)zap_obj;
408 +
409 +                /* Display ZAP object array */
410 +                zapobj = umem_alloc(sizeof(mzap_phys_t), UMEM_NOFAIL);
411 +                printf("\nTracing ZAP object array mzap_phys_t  ...\n");
412 +                for (i=0; i<nzap; i++) {
413 +                        memcpy(zapobj, zapobjptr+i, sizeof(mzap_phys_t));
414 +                        printf("\n\tmz_block_type = 0x%llx\n"
415 +                        "\tmze_value = 0x%llx\n\tmze_name = %s\n",
416 +                        zapobj->mz_block_type, zapobj->mz_chunk->mze_value,
417 +                        zapobj->mz_chunk->mze_name);
418 +                        if (zapobj->mz_block_type == ZBT_MICRO)
419 +                                dslpos = (int)zapobj->mz_chunk->mze_value;
420 +                }
421 +                umem_free(zapobj, sizeof(mzap_phys_t));
422 +
423 +                /* Tracing back to DSL objects */
424 +                printf ("\nDisplaying DSL directory object from bonus buffer "
425 +                        "at &(MOS array)+%d\n", dslpos);
426 +                dnodephys = umem_alloc(sizeof(dnode_phys_t), UMEM_NOFAIL);
427 +                memcpy(dnodephys, objset+dslpos, sizeof(dnode_phys_t));
428 +                ddobj = umem_alloc(sizeof(dsl_dir_phys_t), UMEM_NOFAIL);
429 +                memcpy(ddobj, dnodephys->dn_bonus, dnodephys->dn_bonuslen);
430 +                printf ("\n\tdd_creation_time = 0x%llx\n"
431 +                        "\tdd_head_dataset_obj = 0x%x\n",
432 +                        ddobj->dd_creation_time, ddobj->dd_head_dataset_obj);
433 +
434 +                /* Tracking back to the DSL Dataset object */
435 +                printf ("\nDisplaying blkptr of the DSL Dataset object "
436 +                        "from bonus buffer at &(MOS array)+%d\n",
437 +                        (int)ddobj->dd_head_dataset_obj);
438 +                memcpy(dnodephys, objset+(int)ddobj->dd_head_dataset_obj,
439 +                        sizeof(dnode_phys_t));
440 +                ddset = umem_alloc(sizeof(dsl_dataset_phys_t), UMEM_NOFAIL);
441 +                memcpy(ddset, dnodephys->dn_bonus, dnodephys->dn_bonuslen);
442 +                display_blkptr(blkbuf, &(ddset->ds_bp), spa, poolname);
443 +                umem_free(ddobj, sizeof(dsl_dir_phys_t));
444 +                umem_free(dnodephys, sizeof(dnode_phys_t));
445 +
446 +                /* Get the object set from the DSL Dataset object blkptr */
447 +                metadata = retrieve_blkptr(blkbuf);
448 +                mos = &metadata->os_meta_dnode;
449 +                bp = mos->dn_blkptr;
450 +                printf("\nGet root dataset metadata from DSL Object Set blkptr\n");
451 +                display_blkptr(blkbuf, bp, spa, poolname);
452 +                nlevel = (int)BP_GET_LEVEL(bp);
453 +                for (i=0; i<nlevel; i++) {
454 +                        bp = (blkptr_t *)retrieve_blkptr(blkbuf);
455 +                        display_blkptr(blkbuf, bp, spa, poolname);
456 +                }
457 +
458 +                /* Trying to find Master node */
459 +                objset_array = (void *)retrieve_blkptr(blkbuf);
460 +                objset = (dnode_phys_t *)objset_array;
461 +                bplsize = (u_longlong_t)BP_GET_LSIZE(bp);
462 +                ndnode = (int)bplsize / sizeof(dnode_phys_t);
463 +                     printf("\nThere are %d dnode_phys_t in this dnode_phys_t block
array\n", ndnode);
464 +                printf("Tracing the dnode_phys_t array...\n");
465 +                dnodephys = umem_alloc(sizeof(dnode_phys_t), UMEM_NOFAIL);
466 +                for (i=0; i<ndnode; i++) {
467 +#if DEBUG
468 +                         printf("\nDebug: copying %d bytes from objset+%lx (0x%lx) to
dnodephys\n",
469 +                                 sizeof(dnode_phys_t), i*sizeof(dnode_phys_t),
objset+i);
470 +#endif
471 +                        memcpy(dnodephys, objset+i, sizeof(dnode_phys_t));
472 +                        /* Object directory is always id 1 */
473 +                        if (i == 1)
```

```
474 +                                   display_blkptr(objdir_buf, dnodephys->dn_blkptr, spa,
poolname);
475 +                           else
476 +                                   display_blkptr(blkbuf, dnodephys->dn_blkptr, spa,
poolname);
477 +                           printf("\tBONUSTYPE: %s (0x%x)\n",
478 +                                   dmu_ot[dnodephys->dn_bonustype].ot_name, dnodephys-
>dn_bonustype);
479 +                   }
480 +                   umem_free(dnodephys, sizeof(dnode_phys_t));
481 +
482 +                   /* Retrieve Master Node ZAP object */
483 +#if 1
484 +                   printf("\nDebug: retrieving Master Node ZAP object blkptr with %s\n",
485 +                           objdir_buf);
486 +#endif
487 +                   zap_obj = (void *)retrieve_blkptr(objdir_buf);
488 +                   zapobjptr = (mzap_phys_t *)zap_obj;
489 +                   zapobj = umem_alloc(sizeof(mzap_phys_t), UMEM_NOFAIL);
490 +                   nzap = 512 / sizeof(mzap_phys_t);
491 +                   printf("\nMaster Node ZAP object mzap_phys_t  ...\n");
492 +                   memcpy(zapobj, zapobjptr+i, sizeof(mzap_phys_t));
493 +                   for (i=0; i<nzap; i++) {
494 +                           memcpy(zapobj, zapobjptr+i, sizeof(mzap_phys_t));
495 +                           printf("\n\tmz_block_type = 0x%llx\n"
496 +                           "\tmze_value = 0x%llx\n\tmze_name = %s\n",
497 +                           zapobj->mz_block_type, zapobj->mz_chunk->mze_value,
498 +                           zapobj->mz_chunk->mze_name);
499 +                           if (strcmp(zapobj->mz_chunk->mze_name,"ROOT") == 0 )
500 +                                   dslpos = (int)zapobj->mz_chunk->mze_value;
501 +                   }
502 +                   umem_free(zapobj, sizeof(mzap_phys_t));
503 +
504 +                   /* Trying to get directory contents znode */
505 +                   printf ("\nDisplaying ROOT dnode from &(dnode array)+%d\n",
506 +                           dslpos);
507 +                   dnodephys = umem_alloc(sizeof(dnode_phys_t), UMEM_NOFAIL);
508 +                   memcpy(dnodephys, objset+dslpos, sizeof(dnode_phys_t));
509 +                   zn = umem_alloc(sizeof(znode_phys_t), UMEM_NOFAIL);
510 +                   display_blkptr(blkbuf, dnodephys->dn_blkptr, spa, poolname);
511 +                   printf("\tBONUSTYPE: %s (0x%x)\n",
512 +                           dmu_ot[dnodephys->dn_bonustype].ot_name,
513 +                           dnodephys->dn_bonustype);
514 +#if 1
515 +                   printf("\nDebug: retrieving znode from bonus pointer\n");
516 +#endif
517 +                   memcpy(zn, dnodephys->dn_bonus, dnodephys->dn_bonuslen);
518 +                   printf("Znode details\n\tatime: %s"
519 +                           "\tmtime: %s\tctime: %s\tcrtime: %s\tsize: %lld\n",
520 +                           ctime((clock_t *)zn->zp_atime),
521 +                           ctime((clock_t *)zn->zp_mtime),
522 +                           ctime((clock_t *)zn->zp_ctime),
523 +                           ctime((clock_t *)zn->zp_crtime),
524 +                           zn->zp_size);
525 +                   umem_free(zn, sizeof(znode_phys_t));
526 +                   umem_free(dnodephys, sizeof(dnode_phys_t));
527 +
528 +                   /* Get the root directory ZAP object */
529 +#if 1
530 +                   printf("\nDebug: retrieving object directory blkptr\n",
531 +                           blkbuf);
532 +#endif
533 +                   zap_obj = (void *)retrieve_blkptr(blkbuf);
534 +                   zapobjptr = (mzap_phys_t *)zap_obj;
535 +                   zapobj = umem_alloc(sizeof(mzap_phys_t), UMEM_NOFAIL);
536 +                   nzap = 512 / sizeof(mzap_phys_t);
537 +                   printf("\nRoot directory ZAP object mzap_phys_t from "
538 +                           "bonus pointer ...\n");
539 +                   memcpy(zapobj, zapobjptr+i, sizeof(mzap_phys_t));
540 +                   for (i=0; i<nzap; i++) {
541 +                           memcpy(zapobj, zapobjptr+i, sizeof(mzap_phys_t));
542 +                           printf("\n\tmz_block_type = 0x%llx\n"
```

```
543 +                       "\tmze_value = 0x%llx\n\tmze_name = %s\n",
544 +                       zapobj->mz_block_type, zapobj->mz_chunk->mze_value,
545 +                       zapobj->mz_chunk->mze_name);
546 +                       if (zapobj->mz_block_type == ZBT_MICRO)
547 +                               dslpos = (int)zapobj->mz_chunk->mze_value;
548 +               }
549 +               umem_free(zapobj, sizeof(mzap_phys_t));
550 +
551 +               /* Getting directory entry from dnode array */
552 +               printf ("\nDisplaying dnode from &(dnode array)+%d\n",
553 +                       dslpos);
554 +               dnodephys = umem_alloc(sizeof(dnode_phys_t), UMEM_NOFAIL);
555 +               memcpy(dnodephys, objset+dslpos, sizeof(dnode_phys_t));
556 +               display_blkptr(blkbuf, dnodephys->dn_blkptr, spa, poolname);
557 +               printf("\tBONUSTYPE: %s (0x%x)\n",
558 +                       dmu_ot[dnodephys->dn_bonustype].ot_name,
559 +                       dnodephys->dn_bonustype);
560 +               zn = umem_alloc(sizeof(znode_phys_t), UMEM_NOFAIL);
561 +#if 1
562 +               printf("\nDebug: retrieving znode from bonus pointer\n");
563 +#endif
564 +               memcpy(zn, dnodephys->dn_bonus, dnodephys->dn_bonuslen);
565 +               printf("Znode details\n\tatime: %s"
566 +                       "\tmtime: %s\tctime: %s\tcrtime: %s",
567 +                       ctime((clock_t *)zn->zp_atime),
568 +                       ctime((clock_t *)zn->zp_mtime),
569 +                       ctime((clock_t *)zn->zp_ctime),
570 +                       ctime((clock_t *)zn->zp_crtime));
571 +#if 1
572 +               printf("\nDebug: retrieving ZAP object directory blkptr with %s\n",
573 +                       blkbuf);
574 +#endif
575 +               zap_obj = (void *)retrieve_blkptr(blkbuf);
576 +               zapobjptr = (mzap_phys_t *)zap_obj;
577 +               zapobj = umem_alloc(sizeof(mzap_phys_t), UMEM_NOFAIL);
578 +               nzap = 512 / sizeof(mzap_phys_t);
579 +               printf("\nRoot directory ZAP object mzap_phys_t from "
580 +                       "bonus pointer ...\n");
581 +               memcpy(zapobj, zapobjptr+i, sizeof(mzap_phys_t));
582 +               for (i=0; i<nzap; i++) {
583 +                       memcpy(zapobj, zapobjptr+i, sizeof(mzap_phys_t));
584 +                       printf("\n\tmz_block_type = 0x%llx\n"
585 +                       "\tmze_value = 0x%llx\n\tmze_name = %s\n",
586 +                       zapobj->mz_block_type, zapobj->mz_chunk->mze_value,
587 +                       zapobj->mz_chunk->mze_name);
588 +                       if (zapobj->mz_block_type == ZBT_MICRO)
589 +                               dslpos = (int)zapobj->mz_chunk->mze_value;
590 +               }
591 +
592 +               /* Displaying ZFS plain file dnode object */
593 +               printf ("\nDisplaying dnode from &(dnode array)+%d\n",
594 +                       6);
595 +               dnodephys = umem_alloc(sizeof(dnode_phys_t), UMEM_NOFAIL);
596 +               memcpy(dnodephys, objset+dslpos, sizeof(dnode_phys_t));
597 +               display_blkptr(blkbuf, dnodephys->dn_blkptr, spa, poolname);
598 +               printf("\tBONUSTYPE: %s (0x%x)\n",
599 +                       dmu_ot[dnodephys->dn_bonustype].ot_name,
600 +                       dnodephys->dn_bonustype);
601 +               zn = umem_alloc(sizeof(znode_phys_t), UMEM_NOFAIL);
602 +#if 1
603 +               printf("\nDebug: retrieving znode from bonus pointer\n");
604 +#endif
605 +               memcpy(zn, dnodephys->dn_bonus, dnodephys->dn_bonuslen);
606 +               printf("Znode details\n\tatime: %s"
607 +                       "\tmtime: %s\tctime: %s\tcrtime: %s\tsize: %lld\n",
608 +                       ctime((clock_t *)zn->zp_atime),
609 +                       ctime((clock_t *)zn->zp_mtime),
610 +                       ctime((clock_t *)zn->zp_ctime),
611 +                       ctime((clock_t *)zn->zp_crtime), zn->zp_size);
612 +#if 1
613 +               printf("\nDebug: retrieving ZAP object directory blkptr with %s\n",
614 +                       blkbuf);
```

```
615 +#endif
616 +               zap_obj = (void *)retrieve_blkptr(blkbuf);
617 +               /* This time we're getting the file content */
618 +               memcpy(blkbuf, zap_obj, (size_t)zn->zp_size);
619 +               blkbuf[zn->zp_size+1] = '\0';
620 +               printf("\nFile contains:\n%s\n", blkbuf);
621 +
622 +               /* Finish up */
623 +               umem_free(zapobj, sizeof(mzap_phys_t));
624 +               umem_free(zn, sizeof(znode_phys_t));
625 +               return (0);
626 +       }
627 +
628         fuid_table_destroy();
629
630         libzfs_fini(g_zfs);
```