Digital Forensic Implications of ZFS

By

Nicole Beebe, Sonia Mandes and Dane Stuckey

available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

Digital forensic implications of ZFS

Nicole Lang Beebe, Sonia D. Stacy, Dane Stuckey*

Dept. of Information Systems & Technology Management, The University of Texas at San Antonio, One UTSA Circle, San Antonio, TX 78249, USA

Keywords:
ZFS
File system
Forensics
Data recovery
Copy on write

abstract

ZFS is a relatively new, open source file system designed and developed by Sun Microsystems.[1] The stated intent was to develop "...a new kind of file system that provides simple administration, transactional semantics, end-to-end data integrity, and immense scalability" (OpenSolaris community). Its functionality, architecture, and disk layout take a relatively radical departure from many commonly used file systems (e.g. FAT, NTFS, EXT2/3, UFS, HFS+, etc.). Since file systems play a very important role in how and where data are stored, as well as the likelihood of their retrieval during digital forensic investigations, it is important that forensics researchers and practitioners understand ZFS and its forensic implications. That is the goal of this article. We first provide the reader with a primer of sorts about ZFS, which lays the foundation for our discussion of ZFS forensics. We then present the results of our analysis of ZFS functionality, architecture, and disk layout – identifying and discussing several digital forensic artifacts and challenges unique to ZFS.

1. Introduction

Large scale storage, information security, and ease of administration are key operational and business enablers in many organizations today. Organizations are ever increasing the amount of data they store, warehouse, retrieve, and mine. As is often the case in computing, IT solutions providers are hurriedly adapting current technologies to meet new storage, security, and administrative needs.

Perhaps one of the best examples of adapting current technologies to meet new needs is in the context of managing storage arrays and the advent of logical volume managers (LVMs). Storage arrays—their size, configuration, and management—quickly outgrew the capabilities of many existing file systems; hence, the advent of LVMs. The problem is that LVMs can be difficult to manage; the addition or removal of block devices in the array is not as simple as inserting or removing physical drives. So, while current technologies arguably do scale to meet current storage needs, they are not trivial to administer as storage needs and capacity frequently change. Furthermore, they have a finite capacity limit. Given the exponential growth in data stores in recent years, we contend that even 64 bit file systems will be a limiting factor someday.

Another growing concern is the issue of information security, particularly data integrity, of critical data stores. Organizations remain vulnerable to silent data corruption. We have relatively robust mechanisms in place to ensure data integrity of network-based data (e.g. message digests, message authentication codes in SSL, etc.), but we do not have robust, widespread means to scrub on-disk data. We have hashing, but that is certainly not self-healing. We have RAID

* *Corresponding author.*
E-mail address: nicole.beebe@utsa.edu (N.L. Beebe).

[1] As of the writing of this article, Sun Microsystems publicly acknowledged a plan for Oracle to acquire Sun. ZFS was released as open source software under a CDDL license. It is unknown what impact the not yet finalized acquisition will have on ZFS.

1742-2876/$ – see front matter © 2009 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2009.06.006

implementations that will rebuild a disk, should one fail, but typical RAID implementations do not prevent against silent data corruption and write-hole vulnerabilities. They do not calculate and verify checksums at an object (i.e. file) level, and self-heal when errors are detected. Such mechanisms are needed to validate the integrity of data stored on-disk, as well as when data is being read/written, which would detect and self-correct errors. Today, such errors remain largely undetected and uncorrected, except via user detection which is often too late to correct the errors.

For these reasons, and others, Sun Microsystems designed and developed ZFS (OpenSolaris community). ''ZFS is a new kind of file system that provides simple administration, transactional semantics, end-to-end data integrity, and immense scalability. ZFS is not an incremental improvement to existing technology; it is a fundamentally new approach to data management'' (OpenSolaris community).

Digital forensics researchers and practitioners know all too well that a different file system often necessitates fundamentally different approaches to search and retrieval – extraction and analysis. As Carrier points out, a thorough understanding of the file system is critical to one's knowledge of where digital artifacts will be found during an investigation (Carrier, 2005). If ZFS represents a paradigm shift in file system design from commonly used file systems (e.g. FAT, NTFS, EXT2/3, UFS, HFS+, etc), significant digital forensic implications follow.

ZFS is currently the native file system for OpenSolaris and Solaris 10. Kernel-level ports have been developed for FreeBSD, NetBSD, and Apple's OS X 10.5 Leopard (read-only). User-level ports have been developed for Linux and Apple's OS X 10.5 Leopard via FUSE. A kernel-level port is currently under development for Apple's OS X 10.6 Snow Leopard Server. These implementations and ports have been publicly announced, but there are also rumors of ports to Desktop Snow Leopard and Linux via a kernel-level port provided by Sun. Furthermore, we contend that even if ZFS's prominence in the file system market does not show marked increase in years to come, next-generation file systems will tackle modern storage array, security, and administration issues in a similar ways.

The purpose of this article is to introduce the digital forensics research and practitioner community to ZFS and the digital forensic implications thereof. The remainder of the article is structured as follows. First, we familiarize the reader with ZFS in general—its functionality, its architecture, and its disk layout. Second, and more importantly, we discuss significant digital forensic implications of this relatively new and different file system over commonly used file systems. Finally, we conclude with a discussion of limitations, future research needed, contributions, and a few concluding remarks.

2. Background

This section will introduce the reader to ZFS and show how it fundamentally diverges from commonly used file systems. It is a primer of sorts, which is intended to serve the digital forensics community from an educational standpoint, but does not claim a knowledge contribution in the traditional research sense. Such background will help readers understand the subsequent digital forensic implication discussions—the primary contribution of this article. The background should also serve the community by lessening their burden in understanding the architecture of ZFS. We aim to coalesce what we have learned from ZFS documentation, conference presentations and proceedings, source code reviews, ZFS developer blog posts, direct observation, and other related sources into an instructive introduction of ZFS for the digital forensics community.

2.1. *ZFS functionality*

ZFS integrates traditional file system and logical volume manager functionality and **uses a pooled storage model**, facilitating a highly dynamic file system that supports flexible and large storage arrays. Block devices can be added to the storage array, into what ZFS calls *zpools*. ZFS facilitates the integration automatically and autonomously. The devices are immediately added to the pool, are immediately available for use/storage, and all of this occurs transparent to the user. ZFS eliminates the need for managing and resizing volumes.

Storage arrays, their constituent *zpools*, and their constituent datasets (i.e. file systems, snapshots, clones, and volumes) can be exceptionally large, as ZFS is the **first 128 bit file system**. Also, it is **endian adaptive**,[2] making it architecture independent. Data can be created by any architecture and read by any architecture, and the ordering can be intermixed within a zpool.

ZFS is designed to be **impervious to silent data corruption**, because of its extensive use of **checksumming**. A checksum[3] is calculated and stored for all ZFS objects (e.g. content and metadata). The checksum is stored in the data's metadata and verified incident to any and all data transactions. ZFS then uses the checksums to self-heal whenever errors are detected, thereby facilitating automatic, on-going live data scrubbing, as well as on-demand data scrubbing.

To ensure the validity of the on-disk state, ZFS **implements a copy-on-write (COW) transactional object model**. When block modifications are prescribed, the modified block(s) are written to newly allocated blocks. When the write transactions complete successfully, metadata are updated, also using the COW model. Following the transactional object model, synchronous operations are grouped and tracked in *ZFS intent logs* (ZILs). Should part of the transaction complete successfully and the other fail, all completed transactions are rolled back to ensure transaction synchronicity requirements are achieved. The model also improves I/O performance by batching write transactions and committing them every few (5–10) seconds, or earlier in the event of a forced sync. Both the COW model and transactional object model ensure that

[2] Endian ordering of data structures is a function of the architecture used to write it. A flag is set within objects' block pointers to indicate the endian ordering, which ZFS then uses to read the data in the appropriate order. The manipulation required is entirely transparent to the user.

[3] Supported checksums include SHA-256 (default), fletcher2, and fletcher4 algorithms.

unexpected events, such as system failure due to power failure, do not result in data corruption.

ZFS also facilitates **system resiliency and data redundancy through its native support of snapshots, clones, and ditto blocks** (multiple, automatic, on-disk copies of metadata and/or content category data). A snapshot is a read-only, point-in-time version of a file system. A clone is a read–writeable, fully operational file system, created from a snapshot. Ditto blocks are allocated copies of file metadata and potentially file content and will be discussed in greater detail later due to their digital forensic implications.

Compression (LZJB) is built-in and implemented by default in ZFS for metadata. Content category data is not compressed by default, but can be compressed. Its implementation gains return on the processing (CPU load) investment by significantly lessening disk I/O time.

2.2. ZFS architecture and disk layout[4]

Like most file systems, ZFS starts with a superblock—a block of data reserved for key file system category data, often statically located at the beginning of a physical disk. ZFS calls this the *uberblock* (big endian magic number: 0x00 ba b1 0c – note its phonetics). The uberblock is a 1 KB data structure (element) contained within an uberblock array (128 KB array). Even the uberblock follows the COW model in the sense that updates to the active uberblock are accomplished by writing to a different uberblock in the array than that which contains the active uberblock and then updating the transaction group (TXG) number. The uberblock with the highest TXG and valid checksum is deemed the active uberblock. The uberblock array starts at offset 128 KB within the *vdev label*.

A *vdev* is a virtual device; it may be a physical or logical device. Vdev types include: disk, file, mirror, clone, RAID-Z (similar to RAID-5), replacing, and root. (See Anonymous, 2006 for further explanation of these types.) A *vdev label* is a 256 KB data structure located in quadruplicate on each vdev (two consecutive copies at the beginning of the vdev and two consecutive copies at the end of the vdev). The vdev label contains information that facilitates access to zpool contents and verifies the zpool's integrity and availability. See Fig. 1 for illustrations of the above concepts.

This is the full extent of statically located data structures in ZFS. To put this point into perspective, we compare ZFS to EXT2/3. EXT2/3 divides the file system into sections, called block groups, and subsequently stores backup superblocks, group descriptor tables, block bitmaps, inode bitmaps, and inode tables at specific locations within each block group. File content is stored in equally sized logical allocation units within the block group. Finally, EXT2/3 intentionally co-locates file metadata and file content to minimize disk latency and seek time.

ZFS behaves oppositely by design. The analogous data listed above may be stored anywhere on a top-level vdev, which includes intentionally spreading it across multiple physical/leaf vdevs to reduce the risk of catastrophic (non-recoverable) data loss. While *metaslabs* subdivide vdevs in a similar manner as block groups, the corresponding data stored within the metaslabs is not stored in specific locations, as with EXT2/3 block groups.

Unlike many commonly used file systems, including NTFS, EXT2/3, and HFS+, ZFS does not store file content in equally sized logical allocation units (what ZFS calls *file system blocks*, or *FSBs*). It uses neither block-based allocation, nor extent-based allocation in the traditional sense. Block-based allocation means files are allocated space at the block (i.e. sector) level, as is the case with UFS. When a file grows, more blocks are allocated to meet the need. Extent-based allocation usually means a file system's data storage space is broken up into equally sized, contiguous groups of blocks upon file system creation, and files are allocated space at an extent-level. (Note: Extents are also referred to as clusters, logical allocation units, and data units.) When a file grows, more extents (groups of blocks) are allocated to meet the need. While the extent size can be variably determined upon file system creation, all extents are usually equally sized and the extent size does not vary after install, nor between files.

In contrast and conceptually similar to JFS[5] (Eckstein, 2004), ZFS dynamically and variably sizes its extents according to the needs of individual files. ZFS 'right-sizes' extents up to the max FSB record size set for the file system (default = 128 KB). So, allocated extents are *not* equally sized in a single ZFS file system. Files smaller than FSB record size will be allocated to extents that are smaller than FSB record size. Files larger than FSB record size will be allocated to multiple extents equal in size to FSB record size.

This functionality means that ZFS maintains sector-level allocation awareness. This is accomplished via per metaslab allocation/free log files called *space maps*. Space maps are loaded into memory and converted to space efficient, offset sorted, AVL trees of free space.

Since space maps merely track free space and provide no other data location information, ZFS relies heavily on block pointers—direct and indirect—similar to EXT2/3. In fact, everything after the uberblock is located via block pointers. Unlike other file systems, ZFS does not use statically located inode tables, or file system category files to store data location information (e.g. NTFS's Master File Table – $MFT). ZFS supports six levels of indirection, and several instances of such indirection can exist as one traverses from the uberblock, to the *meta-object set*, to the *dnode* array for the file system objects (directories, files, etc.), and to the file content itself. This traversal is particularly important to digital forensic investigators, as it relates to how data is stored, located, and retrieved.

2.2.1. On-disk data walk

To provide the big picture of how content and metadata are stored, located, and retrieved, the following discussion walks through how to locate a target file starting from the uberblock. First, however, we must discuss a few key data structures.

All, or nearly all, objects in ZFS are described by dnodes. As "everything is a file" in NTFS, "everything is an object" in ZFS, thus the list of object types is long. See Anonymous (2006) for

[4] Primary resources for this section are (Anonymous, 2006; Bruning, 2008a,b,c).

[5] JFS and ZFS implement what Eckstein (2004) calls "variable-sized allocation units" in very different ways.

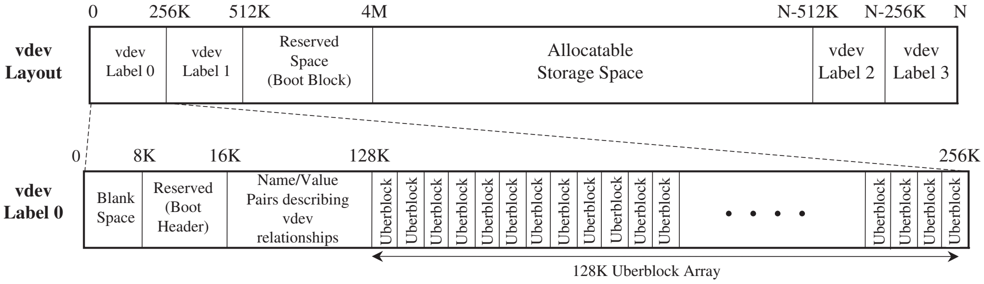

Fig. 1 – vdev, vdev label, and uberblock layouts (Figure adapted from ZFS On-Disk specification illustrations 2, 3 & 7).

a full listing, but a few example object types include: files, directory listings, space maps, intent logs, attribute lists, access control lists, and dnodes.

Dnodes (conceptually similar to inodes in UFS) are 512B data structures that contain, among other things, block pointers (data structure: ***blkptr_t***). Block pointers store information about what the object is, where it is, and how big it is. Dnodes are stored on-disk, usually in dnode arrays, but in-memory copies exist as objects are accessed. Dnode arrays are often fragmented (stored in non-contiguous space on-disk) (Bruning, 2008c).

Objects of similar type are grouped in ZFS and called object sets. Whereas individual objects are described by dnodes (***dnode_phys_t*** data structures), object sets are described by *metadnodes* (***objset_phys_t*** data structures; 1 KB in size). Objset_phys_t data structures contain: 1) a dnode that often points to a dnode array for that object set, 2) a ZIL header, and 3) the object set type. One object set is of particular importance: the meta-object set (MOS). The MOS is the superset of all objects in the zpool.

With an understanding of these key concepts and data structures, we can move forward with the on-disk data walk discussion. (See Fig. 2 for an illustrative view of the data walk).

At the zpool level, the active uberblock's block pointer points to the meta-object set's (MOS) metadnode. Its dnode's block pointer points to the MOS dnode array. The second element (index = 1) of the MOS dnode array points to the Object Directory ZAP object (ZFS Attribute Processor; an object that contains name–value pairs). One name–value pair within the Object Directory ZAP is the root_dataset pair, whose value is the object ID (index number) within the MOS dnode array for the DSL (Dataset and Snapshot Layer) directory. The DSL directory provides a mapping of object IDs (indices) of dnodes within the MOS dnode array for the various DSL datasets (e.g. a specific file system within the zpool). Respective DSL dataset dnodes then point to the metadnodes for the DSL datasets, which in-turn point to each dataset's dnode array.

At the dataset level (e.g. a single file system), each dnode within the dataset's dnode array pertains to a specific file system object (e.g. files and directories). The block pointers within directory dnodes point to ZAP objects, which provide the object IDs for the directory's child objects (i.e. files and subdirectories in the case of a directory object). The block pointers within file dnodes point to file content (or block pointer arrays in the case of indirection).

2.2.2. *Files, directories and their metadata*

The discussion above centers on file and directory traversal by walking through the steps and data structures involved in ultimately locating file content. Some ambiguity may remain, however, with regard to data typically of forensic interest, such as filenames, directory structures, and file metadata.

Filenames and directory names are stored in ZAP objects (name–value pair data structures). The dnode within the file system's dataset dnode array that pertains to the file system's root directory points to the root directory ZAP object. Its constituent name–value pairs consist of the named directories and files within the root directory and their associated object IDs (dataset dnode array indices). Each file and directory has a dataset dnode, which either points to another directory ZAP object, or the file content, for directories and files respectively.

File and directory metadata is stored in its respective dnode within variably sized "bonus buffer" space (referred to as a znode when storing such metadata). This metadata includes MAC date/time stamps, creation transaction group number (relevant for chronologically ordering different versions of data), read–write–execute permissions (e.g. 755), file type (e.g. regular, symbolic link, socket, etc.), file size, object ID of the parent directory, number of hard links, owner ID, group ID, and access control lists. See Anonymous (2006) for specific data structure information.

3. Forensic implications

Now that readers are familiar with ZFS, its functions, its architecture, and its on-disk layout, we can shift our discussion to its digital forensics implications—the primary focus of this article. The discussion will outline several important artifacts left behind by normal ZFS operation that may be of investigative interest and recoverable during analysis. We also discuss several analytical challenges ZFS introduces.

3.1. *Artifact: copies of data*

Sun's design goals of protecting against data corruption, live data scrubbing, instantaneous snapshots and clones, fast native backup and restore, and file system survivability resulted in several features that create and leave behind copies of metadata and content. Such data is forensically

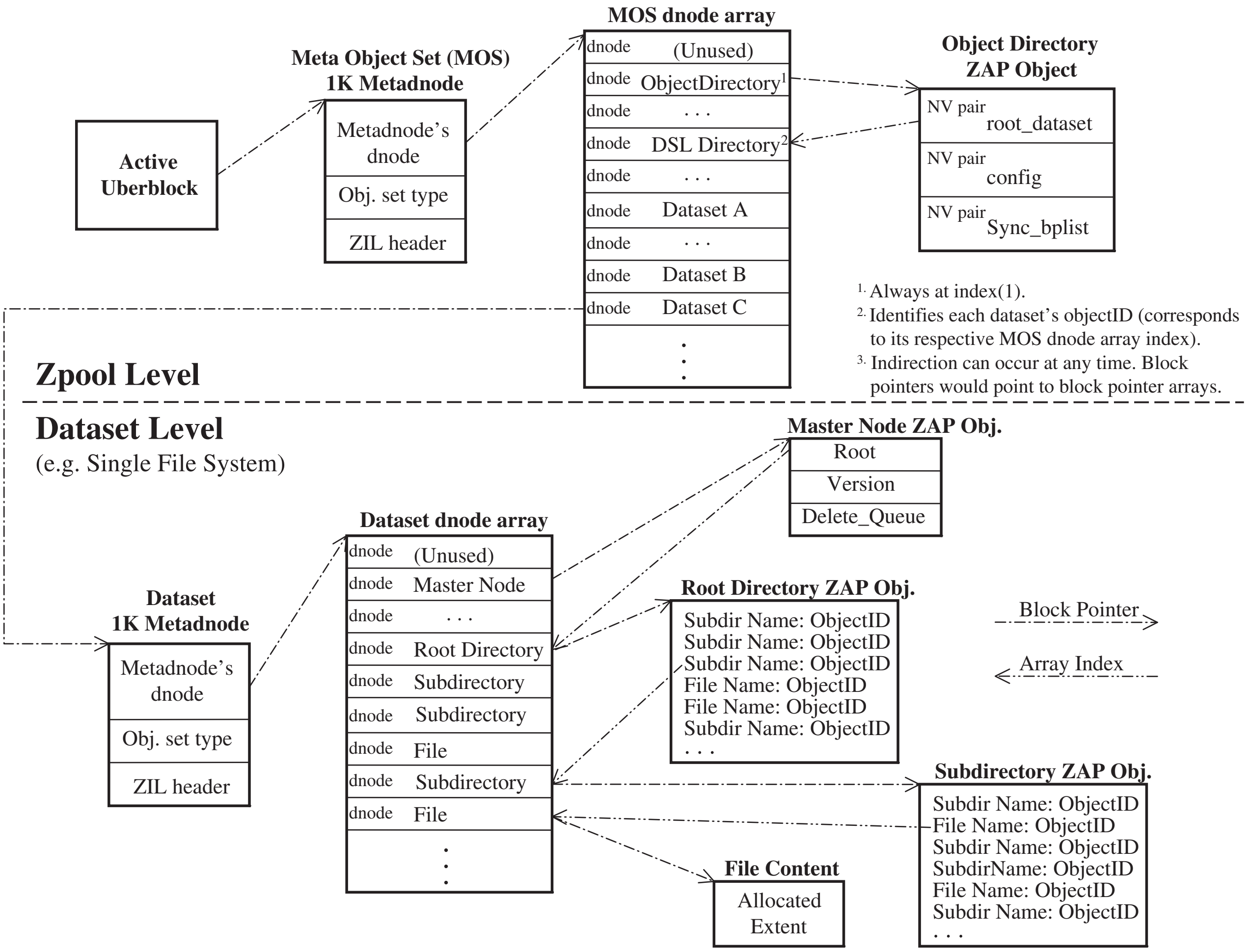

Fig. 2 – On-Disk Data Walk (Figure adapted from Bruning, 2008a).

recoverable. These features include the COW transactional object model, ditto blocks, snapshotting, and cloning.

3.1.1. *Copies in unallocated space*

Several unallocated copies of metadata and content will likely exist, because of the COW model. COW is designed to prevent data corruption by not overwriting in-place data. Instead, when data are to be modified, the modified blocks are written elsewhere on disk. When the modified blocks are written correctly, associated metadata are updated, also via the COW model. Because disk I/O is such a critical performance issue, ZFS (like most file systems) does not securely delete the outdated blocks and the data remains in unallocated space until overwritten. The blocks are, of course, discoverable and recoverable. The impact of this is that forensic examiners will likely find numerous copies of metadata and content throughout the zpool in unallocated space

COW is implemented at the FSB-level. If one sector within a file is modified, the respective FSB for that file will be copied on write. If the file consists of several FSBs, it appears that ZFS trades future increased seek time associated with possible file fragmentation for reduced disk IOPs associated with the COW transaction. The implication of FSB-level COW is that the duplicate metadata and content will likely be whole files (if smaller than FSB record size), or significant chunks of files (if larger than FSB record size).

Additional unallocated copies of metadata and content may exist due to the transaction object model ZFS uses and its resultant ZFS intent log (ZIL) objects. "The ZFS intent log (ZIL) saves transaction records of system calls that change the file system in memory with enough information to be able to replay them … There is one ZIL per file system". (Anonymous, 2006) (pg 51) (Note: Same text remains in the current *zil.c* source code (OpenSolaris 2008.11 release) located at: http://cvs.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/fs/zfs/zil.c).

Transactions are recorded in ZIL log records in memory. When the ZFS Data Management Unit commits a transaction group, the data are written to permanent storage (the writes are committed, e.g. new content is written to disk for a file and metadata are updated), and the respective ZIL records are discarded. When files and processes require synchronicity (e.g. fsync or O_DSYNC call), the ZIL records are flushed to the stable, on-disk ZIL, to facilitate replay and roll-back, and then the data are written to permanent storage. In either case, ZIL records must be committed in order of TXG number. If a commit/sync is forced for a higher TXG than the lowest one in the ZIL, then all previous records will be committed, in order, before the requested TXG is committed.

Research is needed to verify what happens to the data contained in log blocks and log records representing successful commits. It seems reasonable to assume, based on how ZFS uses TXG numbering to maintain state awareness, that committed records and blocks in the stable, on-disk log will remain until overwritten. An important question is whether ZFS allocates new ZIL blocks, or reuses old ones.

The issue of the ZIL, both the in-memory and on-disk versions, requires further research. They have forensic implications for live response, collection, and acquisition, as well as for 'dead'/static device analysis. With respect to live response, in-memory ZIL records might provide useful information regarding very recent processes and user actions. With respect to collection and acquisition, ''pulling the plug'' will represent a power failure, and the stable ZIL blocks/records may help investigators paint a more accurate picture of the current state of the system upon seizure. Finally, depending on how ZFS allocates new, or reuses old ZIL blocks, there may be numerous, unallocated ZIL blocks that can provide a much longer chronology of key file and process actions.

3.1.2. *Copies in allocated space*

In addition to the multiple copies of content and metadata category data that likely exists in unallocated space due to the COW transactional object model, multiple *allocated* copies of metadata and/or content will likely exist via ZFS's ''ditto block'' functionality. For redundancy purposes, ZFS replicates metadata – one, two, or three times, depending on the criticality of the metadata. ZFS also permits automatic copies of user data blocks upon request (administrative tuning of the file system).

The number of ditto blocks created for a given object is determinate based on the number of Data Virtual Addresses (DVAs) populated within the object's dnode's 128B block pointer. Three DVAs are permissible. The number used equates to the number of ditto blocks for the object, and corresponds to the block pointer's ''wideness''. By default:

- Three copies of global metadata (zpool level metadata) are maintained (one original and two ditto blocks); a ''triple-wide'' block pointer is stored.
- Two copies of dataset-level (e.g. file system objects, such as files and directories) metadata are maintained (one original and one ditto block); a ''double-wide'' block pointer is stored.
- One copy of user data (i.e. content) is maintained (Ditto blocks, 2006); a ''single-wide'' block pointer is stored.

When the zpool is tuned to increase the number of ditto blocks for a specific type of data (e.g. user content), this only affects future writes. Also, ZFS automatically increments the number of ditto blocks maintained for higher-level data (e.g. that content's metadata). For example, two copies of user data content would result in three copies of file system metadata pertaining to that content (Elling, 2007).

The location of ditto blocks is easily determined using the DVAs located in the object's dnode's block pointers. It is important to understand the structure of block pointers and their constituent DVAs, however. Each DVA consists of a 32b vdev ID and a 63b sector offset. The offset is relative to the first two copies of the vdev label (L0 and L1; 256 KB each) and the boot block (3.5 MB). Thus, the sector offsets listed are 4096B, or eight sectors, from absolute sector zero on the vdev (physical disk, slice, etc.) (see Fig. 1). The byte offset of the object within the vdev thus equals the DVA offset value times 512, plus 4096.

Each block pointer also includes *asize* (allocated size in blocks; *asize* minus one), indicating the length of the block to which the DVA points, and a ''G'' bit value. When set, the block pointer's ''G'' bit value indicates that the pointer points to a gang block—a block of block pointers. This occurs when files become fragmented. It is important to note, though, that fragmentation is arguably less frequent in ZFS than in commonly used file systems, which frequently use a 'next available' allocation strategy (e.g. NTFS). This is because ZFS utilizes a 'first-fit block allocation' strategy (per **metaslab.c** source code). This is a bit of a hybrid approach between 'next available' and 'best fit', and results in less object (i.e. file) fragmentation. This may make traditional carving techniques even more successful.

The digital forensic implication of the ditto block functionality is clear: multiple *allocated* copies of metadata and/or content will likely exist. Proper understanding of block pointer and DVA data structures is important when examining DVAs in both the ditto block context, as well as in the more basic 'on-disk data walk' context.

Native support for instantaneous snapshots and clones is also important in this context, but warrants little discussion. It is presumed that readers understand that instances of snapshots and clones will intuitively result in additional allocated copies of metadata and content. In fact, ZFS's implementation of snapshots and clones mean that the previously discussed implications of the COW model—leaving copies of metadata and content in unallocated space—will result in the same data now being allocated, and thus persistent.

3.2. *Artifact: increased state awareness*

Multiple copies of data in allocated and unallocated space results in increased state awareness. Typically, when forensic investigators seize computers, they only gain insight into the current state of data. Investigators are sometimes able to cull limited temporal, or state, information from application created temp files, crash files, file system journals, and log files, but such information is limited at best. As a result, investigators eagerly seek logical level file system backups (e.g. tape backups, restore points, etc.). Such backups provide 'snapshots' of the file system at various points in time. Analyzing those backups chronologically provides investigators insight into the progression of intrusions and changes to user-level data over time. This type of temporal information and knowledge is often critical to investigations.

Digital forensics investigators will benefit from the increasingly frequent inclusion of snapshotting capabilities now built-in to operating and file systems, such as ZFS. While snapshotting is certainly not new and dates back ten or more years (e.g. Veritas's VxFS, AIX's JFS/JFS2, and UFS as far back as Solaris 8), its implementation and use are becoming increasingly mainstream. For example, Windows Vista™ now stores "point-in-time copies" of user-level data by default in its Ultimate, Business, and Enterprise editions via its "Shadow Copy" service. This service has been available since 2005 for Windows XP™ users via the "Volume Shadow Copy" service in SDK 7.2, but it was not installed or running by default—now it is. Similar functionality exists by default in the current Windows 7™ release candidate. Apple's "Time Machine", implemented in MAC OS X 10.5, creates automated, on-disk snapshots on via user configured incremental backup scheduling. ZFS implements snapshotting via the *zfs snapshot* command.

The benefit of such file systems' snapshotting capabilities is that copies of data are now stored on-disk in allocated space. The snapshots include both metadata and content. They are often created automatically and transparent to the user. They represent a treasure trove of temporal, state information for the investigator (as do clones, of course).

Now, combine ZFS's native support for snapshotting and cloning, with its COW transactional object model, de-referenced ditto blocks, and the potential for ZIL blocks to be scattered all over the pool. The digital forensic investigator will have unprecedented temporal, state information available. This data is both allocated and unallocated (or de-referenced, if you will). We contend that the digital forensic investigator will be able to trace system and user activity chronologically much more successfully than in the past.

3.3. *Challenge: compression*

Compression has always been a challenge for traditional digital forensic indexing and searching techniques, resulting in additional preprocessing burden due to decompression activity. As such, it is important to note that compression plays a major role in ZFS's static, on-disk data storage. Analysis of ZFS systems will not be as simple as decompressing known compressed file types, or decompressing entire volumes during the preprocessing stage. ZFS implements compression on a massive scale, but does so at the object, and sometimes even data structure level. Prevailing digital forensic techniques are currently unable to deal with such pervasive metadata compression and datasets with such a significant mix of data in various compression states—not compressed, compressed, and compressed with a variety of algorithms.

Metadata compression is enabled by default. It can be disabled at any time, although it only appears to disable compression on indirect block pointers; direct pointers remain compressed (ZFS evil tuning guide). Additionally, ZAP objects do not appear to be compressed, even when metadata compression is turned on. Content category data is not compressed by default, but this setting can easily be tuned (PrincetonUniversity, 2007). These settings are globally set on a zpool basis, but tuning only affects new data storage. The compression state of currently saved metadata and content remains as is upon tuning. Thus, a forensic analyst will not be able to treat all objects of the same type, or even different instances of the same object the same analytically with respect to compression. Object compression states will have to be determined at the individual object level, as each object's metadata indicates the object's compression state in its respective block pointer. Digital forensic tools developed to analyze ZFS evidence will have to analyze data structures for compression settings and subsequently decompress the objects accordingly, prior to search, extraction, and analysis processes.

3.4. *Challenge: dynamically sized extents*

As stated earlier, ZFS uses neither block-based allocation, nor extent-based allocation in the traditional sense. While block-based allocation results in maximum space utilization, it suffers from expensive disk I/O (unless augmented with supplemental read/write and grouping algorithms, as UFS did). Extent-based allocation improves disk I/O, but can result in wasted disk space. The solution to this problem has typically been 'variably sized block' functionality, where the extent (AKA cluster, logical allocation unit, data unit) size is defined upon file system creation and is optimally set based on intended use. In other words, the file system creator selects a small extent size if she knows the file system will be used to store many small files. Regardless of what extent size is selected during file system install, this extent size remains fixed for the entire file system and all subsequent files will be allocated to these equally sized extents.

ZFS uses a bit of a hybrid between block-based and extent-based allocation. It is more aptly described as extent-based, but the extents will be dynamically sized according to individual file requirements.

The *record size* of an FSB has a consistent maximum size (default = 128 KB), but upon file allocation, ZFS dynamically sizes the FSB to just fit the data allocated. In other words, a 700B file would be allocated to a two-sector FSB (presuming 512B sector), or 1024B. A 4000B file would be allocated to an eight-sector FSB, or 4096B. A 4097B file would be allocated to a nine-sector FSB, or 4608B. Files larger than the FSB record size (i.e. greater than 128 KB if record size remains set to the default 128 KB) will be allocated to multiple FSBs.

On the surface, this might appear to be block-based allocation, but it is not. The metadata stored is still stored at the extent-level similar to data runs in NTFS or extent records in HFS+ (e.g. first fragment starts at cluster 140 for a length of 10 clusters, second fragment starts at cluster 800 for a length of 5 clusters), rather than lists of allocated blocks (e.g. listing all 15 cluster numbers). Two key differences exist between ZFS and traditional extent-level allocation implementations, however. First, ZFS extents vary in size between files, within a file system. Second, the metadata that specify the location and length of the file fragments refer to the fragments' sector offsets within vdev labels and fragment length (in sectors).

Additionally, the FSB record size can be repeatedly tuned after install (though, not advised). Thus, new files and copies of old files may exhibit very different FSB sizes from each other and from old files. (Bourbonnais, 2006) The FSB size for a specific object (e.g. file) is stored in the object's dnode, within the 16b ***dn_datablkszsec*** data structure (size in sectors).

Dynamically sized extents negatively impact the ease with which analytical inferences can be made. With file systems that used traditional extent-based allocation (i.e. fixed size extents, variably determined during file system install), forensic examiners can map extents from the start of volume, data area, etc. as appropriate, and know that all data in a specific block consists of that file's content and its file slack. Since the file system will be full of differently sized FSBs, the investigator cannot use simple arithmetic to identify extent boundaries. The investigator will have to consult ZFS metadata pertaining to individual files, as well as the space maps for each metaslab.

3.5. *Artifact: dynamically sized extents*

In addition to the analytical challenge that dynamically sized extents presents, such variability also greatly impacts a critical digital forensics concept – file slack. Many investigators have benefited from finding old file content in file slack. Since extents are variably and dynamically sized to best fit the file content, less file slack should exist. Files smaller than the FSB record size should exhibit no useful file slack, presuming the sector slack is zeroed out as with many other file systems.

Files larger than the FSB record size, on the other hand, will exhibit a significant amount of file slack in many cases. This amount will be unusually large relative to many commonly used file systems, since the default maximum FSB record size is 128 KB. In NTFS, HFS+, and EXT2/3, the extent size is typically set to 512, 1024, 2048, 4096, or 8192 bytes, with 4096B being the norm. So, when files are not exact increments of the fixed extent size (e.g. 4 KB) additional extents are allocated, and the last extent may contain useful file slack. If the last extent is larger than 4 KB, as is the case with ZFS and its default maximum extent size of 128 KB, much more file slack will likely exist. JFS is one file system that is capable of even larger extent sizes (up to 16 GB), but it is designed to right-size the last extent to minimize file slack (Ray, 2004). ZFS is not currently designed to right-size the last extent, although conversations with ZFS developers suggest this may change this in future releases.

4. Concluding remarks

4.1. *Contribution*

To the best of our knowledge, this is the first article that examines and discusses ZFS in the digital forensic context. We identified digital forensic artifacts that exist, due to ZFS functionality and design – specifically, the increased advent of copies of metadata and content in both allocated and unallocated space, due to ditto blocks, COW, the ZIL, etc. We further discussed the implication of harvesting and analyzing such data to provide a greater chronological sight picture than has been previously possible. We outlined key data structures and how to locate them, which is important for forensic tool development and manual data-walking by forensic investigators. Finally, we alerted the community to the unique challenges that compression and dynamically sized extents represent for the forensic investigator. Though not discussed explicitly, the challenge of finding data through massive amounts of indirection and in the absence of statically located metadata and/or file system category files should also be noted.

Although the forensic implication discussions summarized above represents the primary research contribution of this article, we also hope readers find the background informative and instructive. So, to a lesser extent, the 'ZFS primer' is also a contribution of this article.

4.2. *Limitations and future research*

The primary limitation of this article is that we have yet to verify all statements through our own direct observation and reverse engineering of on-disk behavior. We worked very hard to leverage authoritative sources of information and corroborate statements regarding ZFS functionality, architecture, and disk layout. We analyzed ZFS source code to some extent, although not exhaustively. We acknowledge that our implications and challenges discussions are 'academic' and need empirical verification.

Aside from the overarching need for empirical verification stated above, there are a few specific areas where future research is needed in particular. One such area is the ZIL. Documentation for the ZIL is limited. Experimentation and more thorough source code analyses are needed to fully understand its in-memory and on-disk formats, its similarities and differences with existing journaling file systems, and the digital forensic implications thereof.

Another area ripe for future research regards the increased state awareness of system and user level activity provided by the greater incidence of both allocated and unallocated snapshot data. Without a doubt, more state data will be available on-disk. Several unknowns remain, however. We do not fully understand what the scope of such data will be in practice. Research is needed respecting the recovery and analysis of such data, particularly as it relates to relating the data, transforming it into chronological information, and deriving investigative knowledge thereof.

5. Conclusion

ZFS is different. Its data structures are unique. Its on-disk behavior and artifacts are not what digital forensic investigators are used to seeing. Very little is statically located in devices managed by ZFS. A great deal of data is compressed, and soon (rumor has it) much will be encrypted. File slack is arguably non-existent in small files and unusually abundant in large files. Numerous, extra copies of metadata and content will be literally scattered about devices. Such data can be used to increase temporal, state information in support of investigations. The list goes on.

Mac forensics used to be just a niche area of forensics, reflective of the user market. It is no more. We predict that ZFS will follow a similar adoption curve amongst consumers, and perhaps more importantly, a much greater share in enterprise systems where end-to-end data integrity and large scale storage is necessary. We predict that next-generation file systems will adopt and integrate many of the solutions offered by ZFS to combat today's IT challenges. As such, it is important for researchers and practitioners to understand ZFS and its forensic implications. We hope this article has made strides in both areas.

Acknowledgements

The authors wish to thank HEI Systems & Solutions of San Antonio, Texas. It was discussions with them regarding their innovative use of ZFS that sparked our interest in this subject. They also proved to be very knowledgeable and helpful during the course of writing this article. We also wish to thank the DFRWS reviewers for their insightful comments and suggestions that improved this paper greatly.

REFERENCES

Anonymous. ZFS On-disk specification (DRAFT); 2006. pp. 1–55.

Anonymous (Unknown). OpenSolaris community: ZFS, http://opensolaris.org/os/community/zfs [Date accessed: March 2009].

Anonymous (Unknown). ZFS evil tuning guide, http://www.solarisinternals.com/wiki/index.php/ZFS_Evil_Tuning_Guide [Date accessed: March 2009].

Bourbonnais R. The dynamics of ZFS, http://blogs.sun.com/roch/entry/the_dynamics_of_zfs; 2006 [Post Date: 21.06.06] [Date accessed: March 2009].

Bruning M. ZFS on-disk data walk (Or: where's my data), slides from presentation at the OpenSolaris developer conference. Prague, Czech Republic, http://www.osdevcon.org/2008/files/osdevcon2008-max.pdf; June 25–27, 2008a.

Bruning M. ZFS on-disk data walk (Or: where's my data), video of presentation at the OpenSolaris developer conference. Prague, Czech Republic, http://video.google.com/videoplay?docid=2325724487196148104; June 25–27, 2008b.

Bruning M, ZFS On-disk data walk (or: Where's my data?) (DRAFT). In: Proceedings of the OpenSolaris developer Conference. Prague, Czech Republic, June 25–27; 2008c. pp. 36–57.

Carrier B. File system forensic analysis. Upper Saddle River, NJ: Addison-Wesley; 2005. pp. 1–569.

Eckstein K, Forensics for advanced UNIX file systems. In: Proceedings of the fifth annual IEEE SMC information assurance workshop. West Point, NY, 10–11 June 2004, pp. 377–85.

Elling R. ZFS, copies, and data protection. URL: http://blogs.sun.com/relling/entry/zfs_copies_and_data_protection; 2007 [Post Date: 04.05.07] [Date accessed: 10.03.2009].

Ray M. JFS tuning and performance. Hewlett Packard Company; 2004. pp. 1–23.

Unknown. Ditto blocks – the amazing tape repellent. URL: http://blogs.sun.com/bill/entry/ditto_blocks_the_amazing_tape; 2006 [Post Date: 12.05.06] [Date accessed: 03.03.09].

ZFS management. PrincetonUniversity. URL: http://www.princeton.edu/~unix/Solaris/troubleshoot/zfs.html; 2007 [Date accessed: March 2009].

Nicole Lang Beebe, Ph.D. is an Assistant Professor in the Department of Information Systems & Technology Management, at the University of Texas at San Antonio (UTSA). UTSA a National Center of Academic Excellence in Information Assurance for both education (CAEIAE) and research (CAE-R). Dr. Beebe received her Ph.D. in Information Technology from UTSA in 2007. She has over ten years experience in information security and digital forensics, in both the commercial and government sectors. She was a computer crime investigator for the Air Force Office of Special Investigations from 1998 to 2007. She has been a Certified Information Systems Security Professional (CISSP) since 2001. She has published several journal articles related to information security in digital forensics in The DATABASE for Advances in Information Systems, Digital Investigation, and Journal of Information System Security (JISSEC). Her research interests include digital forensics, information security, and data mining.

Sonia D. Stacy recently graduated (May 2009) from The University of Texas at San Antonio with a master of science degree in Information Technology and a concentration in Infrastructure Assurance. She is now working for the U.S. government in the computer security field.

Dane Stuckey is an undergraduate student at The University of Texas at San Antonio, pursuing a Bachelor of Business Administration degree in both Infrastructure Assurance and Information Systems. Dane currently serves as the lead laboratory administrator for UTSA's Advances Laboratories for Infrastructure Assurance and Security (ALIAS). He also holds an internship with the Air Force Office of Special Investigations where he focuses on computer crime and digital forensic investigations.