Forensic Timeline Analysis of the Zettabyte File System

Dylan Leigh

College of Engineering and Science, Victoria University, Melbourne, Australia

Submitted in partial fulfilment of the requirements of the degree of

Bachelor of Science (Honours) (Computer Science)

January 2015

ABSTRACT

During forensic analysis of computer systems, it is often necessary to construct a chronological account of events, including when files were created, modified, accessed and deleted. Timeline analysis is the process of collating and analysing this data, using timestamps from the filesystem and other sources such as log files and internal file metadata.

The Zettabyte File System (ZFS) uses a novel and complex structure to store file data and metadata across multiple devices. Due to the unusual structure and operation of ZFS, many existing forensic tools and techniques cannot be used to analyse ZFS filesystems.

In this project, it has been demonstrated that four of the internal structures of ZFS can be used as effective sources of timeline information. Methods to extract these structures and use them for timeline analysis are provided, including algorithms to detect falsified file timestamps and to determine when individual blocks of file data were last modified.

STUDENT DECLARATION

I, Dylan Leigh, declare that this BSc (Honours) thesis entitled Forensic Timeline Analysis of the Zettabyte File System is no more than 60,000 words in length including quotes and exclusive of tables, figures, appendices, bibliography, references and footnotes. This thesis contains no material that has been submitted previously, in whole or in part, for the award of any other academic degree or diploma. Except where otherwise indicated, this thesis is my own work.

Signature Date

ACKNOWLEDGEMENTS

First and foremost I would like to thank my supervisor, Associate Professor Hao Shi. Her assistance, patience, encouragement and guidance have been invaluable, and I am most grateful for the many hours she has devoted to this work.

I am also grateful to many other staff and students at Victoria University who have provided advice regarding academic writing and this project, especially Prof. Xun Yi, Dr Gitesh Raikundalia, Dr Russell Paulet, Robert Moonen, Dr Jakub Szajman and Prof. Nalin Sharda, as well as Dr Ron van Schyndel of RMIT University, Melbourne, Australia, who also inspired my interest in computer forensics.

I am indebted to my friends Simon Duff, Paul Miller, Dr Paul Bone, Dr Halil Ali and Dr Ade Ishs who provided me with invaluable advice and feedback on my writing.

I would also like to thank BSDCan for funding travel to and accommodation at the conference and providing me with an opportunity to speak to the BSD community. Conference organiser Dan Langille was most helpful in arranging my travel to Ottawa and ensuring the presentation went smoothly. Discussions at BSDCan also enhanced this project; in particular Matt Ahrens of Delphix provided me with vital corrections and advice regarding ZFS processes and terminology.

Last but not least, I am eternally grateful to my family and friends for their love, support and understanding. Without them I would not be able to devote so much to this project. I am especially grateful to my wife Zeti Maria binti Omar for her superhuman patience and support during my studies, and my father David Boyd Leigh for a lifetime of encouragement. It is to them that I dedicate this thesis.

LIST OF PUBLICATIONS

The following publication was developed from the research presented in this thesis:

(Leigh and Shi, 2014b) **Leigh D** and Shi H. "Forensic Timestamp Analysis of ZFS". BSDCan 2014; Ottawa, Canada; May 2014.

Available from: http://www.bsdcan.org/2014/schedule/ events/464.en.html.

An additional publication is to be submitted using material from Chapters 2 and 6.

CLARIFICATION: The following Digital Forensics Magazine article discussed software developed by the author for a separately assessed software project during Semester 1 2014, which made use of the research presented in this thesis. Although referenced in this thesis, the software and article itself does not form part of this thesis or research project.

(Leigh and Shi, 2014a) **Leigh, D**. and H. Shi. "Adding ZFS events to a super-timeline". Digital Forensics Magazine, Issue 20; August 2014.

Available from: http: //digitalforensicsmagazine.com/index.php?view=article&id=949.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1

INTRODUCTION

McKemmish (1999) defined forensic computing as "the process of identifying, preserving, analysing and presenting digital evidence in a manner that is legally acceptable". McKemmish expands on these four elements of a digital forensic investigation, reflecting that analysis is the most elaborate:

> "The analysis of digital evidence—the extraction, processing and interpretation of digital data—is generally regarded as the main element of forensic computing."

Timeline analysis is the term used for the forensic process of compiling a "timeline" or chronological account of events on a system being forensically examined. It involves the collation of event information from multiple sources such as filesystem timestamps, log files and web browser history into a detailed account of what occurred on the system.

The integrity and detail of the timeline may be critical to a legal case; for example, determining the exact time a crime occurred may be essential in identifying the culprit, or a witness statement may be excluded if it can be shown that a different sequence of events occurred. Spencer

(2014) describes a case involving an alleged Turkish military coup where timeline analysis was used to show that the evidence central to the prosecution case had been tampered with.

It is desirable to utilise as many sources of events as possible, for increased detail and corroboration of the same event from multiple sources. Some sources of events may be inaccurate or subject to deliberate tampering. Users can easily alter many sources of events - by changing the system clock or using a command to change timestamps manually - and it may be difficult to determine if any timestamps are genuine or have been falsified. Corroboration of timestamps from multiple sources of events can be used to detect forged timestamps.

The Zettabyte File System (ZFS) (Bonwick et al., 2003; Sun Microsystems, 2006), uses a novel pooled-disk structure for storing data and metadata. ZFS replaces the traditional volume management approach and allocates many disks to a pool, with filesystems, files and data stored within an object tree distributed amongst multiple disks.

Due to the unusual structure and operation of ZFS, many existing forensics tools and techniques are not applicable to forensic analysis of ZFS filesystems. There are few existing studies of ZFS or other unusual Unix filesystems (Beebe, 2009) and therefore few guidelines for forensic investigators in the field. The existing studies are based mostly on theoretical and source code analysis rather than empirical examination of real ZFS filesystems. As far as the author is aware this research (Leigh and Shi, 2014b) is the first to examine ZFS timeline analysis in detail.

Because ZFS operates in a different manner to most other filesystems, there is significant potential to discover new techniques for collecting additional timeline information. It has been noted by Beebe et al. (2009) that many metadata elements of ZFS are useful to a forensic investigator. In its internal structures, it contains many timestamps, incrementally increasing ID

numbers and previous copies of metadata which can be used to enhance timeline analysis and possibly provide verification of timestamps. However there are no tools, procedures or guidelines for forensic investigators to make use of these elements, apart from those developed from this research (Leigh and Shi, 2014a).

This project extends existing research by identifying six ZFS structures which may be useful for timeline analysis and six corresponding methods of detecting forged timestamps. It examines data collected from ZFS filesystems with forged timestamps, demonstrating that four structures (and methods) may be effectively used for forensic purposes. A technique to generate additional file modification events (beyond what can be obtained from the file timestamps) for timeline analysis is also demonstrated.

1.1 OBJECTIVES

This project aims to provide a research basis for development of forensic tools to support timeline analysis of ZFS, and recommendations for forensics investigators examining ZFS devices.

The objectives of this project are to identify internal ZFS metadata which can be used for timeline analysis, and provide methods to collect this metadata. It must also be determined how this metadata is related to the creation and modification time of files, and how long any useful metadata will be retained. Finally, this project should provide at least one effective method which uses ZFS metadata to detect falsified file timestamps.

The knowledge obtained from the study should be used to provide practical guidelines and procedures for forensic investigators, and shared amongst the community to provide the basis for future forensic research and development of ZFS forensic software. It is intended that at least one publication is produced from this research in the area of timeline forensics, plus one possible publication related to the study of the ZFS on-disk format.

1.2 APPLICATIONS

The observations and algorithms presented will be useful for forensic investigators who need to examine ZFS disks during an investigation, and developers of software used for forensic analysis of ZFS.

ZFS is a widely used filesystem for enterprise and local server storage, and it is also used in many other applications due to its unique features. Because it is Open Source software, it can be integrated into other hardware and software products free of charge, and has been ported to several operating systems including Illumos (illumos, 2014), FreeBSD (Dawidek, 2007; McKusick et al., 2014) and Linux (Powell, 2012). It is the default filesystem in Solaris(Oracle, 2014) and Illumos.

Software services companies utilising ZFS as part of their product include Helios, Joyent, Delphix, Nexenta(OpenZFS, 2014) and Oracle (Oracle, 2014). The volume dataset, snapshot and remote replication features of ZFS are used to provide features such as distributed storage and remote backups to clients. ZFS is also used within many off-the-shelf storage products; com-

panies producing ZFS-based storage devices include iXsystems, Losytec, Racktop (OpenZFS, 2014), Netgear (Netgear, 2014), and Oracle (Oracle, 2014).

As there are few existing studies of ZFS forensics, investigation of any of these systems requires new techniques and guidelines for forensic investigators. Section 6.2 contains practical recommendations for forensic investigators when analysing ZFS devices, including procedures for initial capture and later reconstruction of a ZFS pool.

Outcomes of this research may also be useful for investigators and researchers examining other filesystems, as variations on some techniques presented (such as using Transaction Group ID or Object ID as a source of timeline data) may be applicable to other systems.

1.3 THESIS ORGANISATION

The following two chapters introduce theory and review literature relating to this project. Chapter 2 **Background: ZFS and Timeline Analysis** contains necessary background theory regarding the internals of the Zettabyte File System and Timeline Analysis. Chapter 3 **Literature Review** reviews relevant literature in three areas: ZFS Internals, ZFS Forensics, and other Forensic literature not specific to ZFS.

Chapter 4 **Experiment Methodology** contains the method used to conduct the simulations of file system activity and collect internal ZFS metadata during and after the simulations, as well as describing the six algorithms to detect falsified timestamps which were tested on this collected metadata. Chapter 4 also contains the method used to conduct the “large file” experi-

ments and the algorithm they test, which detects the modification time of individual blocks of file data.

All results discussed in this thesis including the falsification detection algorithm results are presented in Chapter 5 **Results and Analysis**. This chapter also discusses the implications and findings arising from these results, including evaluations of all algorithms tested as well as some other factors relating to the use of ZFS metadata for timeline analysis.

Finally, Chapter 6 **Conclusion and Recommendations** summarises the key findings and research outcomes, lists practical recommendations for forensic investigators for analysis of ZFS disks, and discusses future study which may be beneficial.

The appendices following the final chapter contain extended versions of example output which was abbreviated for clarity in the background chapter, extended statistical data regarding the experiments, and the scripts used to run the simulations and collect data.

2

BACKGROUND: ZFS AND TIMELINE ANALYSIS

This chapter contains background theory on the Zettabyte File System (ZFS) and Timeline Analysis necessary for the remainder of this thesis. The former topic is described in greater detail as this research is dependent on the internal operations and data structures of ZFS.

This chapter consists of four sections. The first section introduces ZFS for readers who have not used it and are unaware of the differences between ZFS and other file systems.

For readers who have used ZFS as an end-user or administrator, but unfamiliar with the internal structure of ZFS, description of the internal operation of ZFS begins at section 2.2. Section 2.3 describes the ZFS data structures which are the focus of this research.

Finally, a brief introduction to timeline analysis is provided within section 2.4 for readers unfamiliar with digital forensics.

2.1 INTRODUCTION TO ZFS

The Zettabyte File System was designed (Bonwick et al., 2003) to meet the needs of enterprise server storage and surpass many limitations of existing filesystems. ZFS is a popular filesystem for many applications due to its scalability, high availability and guaranteed data integrity (Phromchana et al., 2011; Zhang et al., 2010).

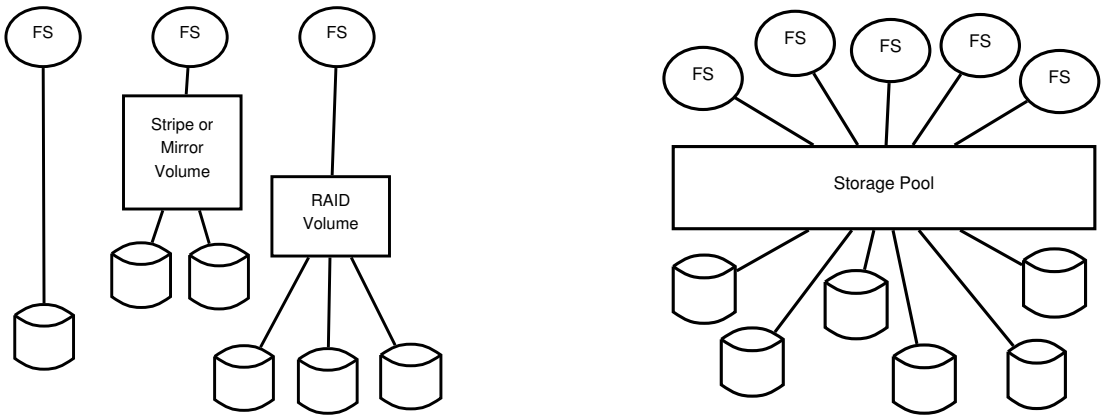
2.1.1 *Differences between ZFS and Traditional File Systems*

2.1.1.1 *Pooled Storage*

Traditional volume management involves mapping individual filesystems to volumes and disks; filesystems are fixed in location on the disk, and the redundancy of the volume (e.g. mirroring or RAID) is fixed upon creation. Volumes effectively present an abstracted virtual disk to the filesystem, which is unaware of the level of redundancy.

ZFS uses a different approach: all disks are allocated to a "pool"; and filesystems are created and mounted as required administratively. Data is stored within different devices by ZFS to optimise performance and maintain redundant copies to prevent loss of data from device failure. Figure 2.1 shows this pooled structure of the Zettabyte File System compared to "traditional" file systems.

Creating a new ZFS filesystem is a lightweight process, similar to creating a directory on other filesystems. ZFS Filesystems may make use of the entire space of the pool (although a quota



(a) Traditional FS/Volume (b) ZFS Pool

Figure 2.1: Structure of Traditional Filesystems and Volumes compared to ZFS Pools

limit may be set) and the level of redundancy and compression can be adjusted per filesystem, directory or file.

ZFS Pools may make use of up to 2^{64} devices, with the pooled structure allowing filesystems and files which are larger than a single device. ZFS Pools can be created with built-in fixed redundancy, by declaring a set of devices as a mirror or "RAID-Z" group. Devices may also be allocated as dedicated log or cache devices for improved performance or dedicated hot spares which will replace faulty devices in the pool.

2.1.1.2 *Guaranteed Data Integrity*

Data integrity and prevention of corruption was a key design consideration of ZFS (Bonwick et al., 2003), which provides automatic and continuous integrity checking and repair. ZFS checksums all data and metadata and transparently detects and recovers from corruption using the redundant copies. Even with a "pool" of a single device, the checksums detect corruption which would occur without warning on other filesystems (Zhang et al., 2010).

ZFS uses a tree of checksums to detect errors across the entire filesystem, including all file data and internal data. Unlike block-based checksums (which can only detect "bit rot" on disks), the checksum tree used by ZFS can detect other forms of corruption. This includes DMA errors, missing or incorrectly addressed disk writes and other errors caused by driver or drive firmware bugs. Checksums can also verify data reconstructed from a RAID-Z or Mirror virtual device (see section 2.2.5.1 for details of redundant virtual devices).

ZFS uses a copy-on-write transaction operation for all writes (see section 2.2.4 for details), which ensures that no data is ever overwritten in place and the disk is never left in an inconsistent state. All disk writes by ZFS will either complete atomically or not at all. This prevents corruption in the event of power failure or a hardware fault. ZFS can also recover faster from such faults as no file system check is required when the system is restarted.

2.1.1.3 *Other Features*

The storage mechanism and copy-on-write transactions of ZFS also provide other benefits besides data integrity. Due to the copy-on-write manner in which transactions are committed, snapshots can be created on ZFS with minimal storage or processing overhead compared to other file systems (Bonwick and Moore, 2008). Clones (writable virtual copies of snapshots) can also be created.

The storage method used by ZFS to manage objects across the pool also allows for granular compression and deduplication to be performed. Different algorithms for compression may be chosen by the administrator for different filesystems, directory trees or files to obtain the desired trade-off between performance and capacity.

Table 2.1: Zettabyte File System Limits

Block Address Size	128 Bits
Maximum Devices in a Pool	2^{64}
Maximum Pool Size	2^{78} Bytes (256 zebibytes)
Maximum Datasets in a Pool	2^{64}
Maximum Files in a Dataset	2^{48}
Maximum Size of a Single File	2^{64} Bytes (16 exbibytes)

Finally, ZFS supports remote replication and backup by sending its internal data as a stream. ZFS can send or receive entire filesystems or incremental streams between two snapshots. This facility is used by some organisations which use ZFS in their own products to provide distributed storage (OpenZFS, 2014).

2.1.2 *ZFS Capacity and Limits*

ZFS was designed to be a "future-proof" filesystem with no capacity limitations which would be reached "in the next few decades" (Bonwick et al., 2003). Many limits in ZFS have been designed so that they can not be theoretically reached, for example the 64 Bit Transaction Group ID allows for approximately 2.92 billion years of operation, and the 128 bit block address allows for more data to be addressed than the theoretical quantum limits of storage on earth (Bonwick and Moore, 2008). Table 2.1 shows some of the limits of the ZFS filesystem.

2.2 ZFS INTERNAL OPERATION

This section describes the internal operation of ZFS. Individual data structures which are the focus of this research are described in detail in section 2.3.

ZFS components which are not necessary background for later chapters of this thesis are not described here. In particular, Object Sets, the Meta Object Set, the Dnode structure, the in-memory log and cache, the on-disk cache and the ZFS Attribute Processor (ZAP) and its structures are not covered.

For more detailed information please refer to the articles discussed in section 3.1 of the literature review, in particular Bonwick, Ahrens et al's *The Zettabyte File System* (Bonwick et al., 2003), Sun Microsystems *ZFS On Disk Specification* (Sun Microsystems, 2006), Bruning's *ZFS On Disk Data Walk* (Bruning, 2008b), Ahrens' *OpenZFS: upcoming features and performance enhancements (Ahrens, 2014b)* and Chapter 10 of McKusick, Neville-Niel and Watson's *The design and implementation of the FreeBSD operating system*, 2nd Ed (McKusick et al., 2014).

2.2.1 *Overview*

ZFS stores all data and metadata within a tree of objects, stored within a tree of blocks. Each "Block Pointer" reference to another block can point to up to three identical copies of the target block, which ZFS spreads across multiple devices in the pool if possible to provide redundancy in case of disk failure.

A ZFS Pool is configured with one or more virtual devices (“Vdevs”), which may contain one or more virtual or physical devices. Virtual devices may combine physical devices in a mirror or “RAID-Z” for guaranteed redundancy.

Each physical device contains a “Device Label” with an array of 128 “Uberblocks”, the root of the block and object trees. Each transaction is committed to a new Uberblock (overwriting the oldest one in the array). The label (and Uberblock array) is duplicated 4 times in each Vdev for redundancy.

Each Uberblock contains a “Block Pointer” structure pointing to the root of the object tree, the “Meta Object Set”. Apart from the metadata contained within the device label, all data and metadata is stored within the object tree, including internal metadata such the free space maps and delete queues.

The “Meta Object Set” contains a directory of “Datasets”, collections of objects which make up a filesystem, snapshot or clone. These datasets are mounted at a specified directory like a traditional filesystem. The Meta Object Set stores datasets in a hierarchy - filesystems may have a parent filesystem, snapshot datasets depend upon the filesystem dataset they were created from, and clones depend on snapshots.

Volume datasets may also be created which act as a block device, which may be used to create another filesystem or exported as block storage.

2.2.2 *ZFS Pool and Filesystem Versions*

ZFS has undergone many revisions since first introduced. Both the ZFS Pool and each ZFS Filesystem contain a version number, and may generally be updated independently although some Filesystem versions require a minimum Pool version and vice versa. New versions of the pool have added features such as new compression algorithms, deduplication, and support for dedicated log and cache devices.

This project uses ZFS pool version 28 and filesystem version 5, which unless otherwise specified are the versions described in this chapter. These versions were current in the most recent stable release of FreeBSD when this project was started. ZFS Pool version 28 is also the last version which was used in the OpenSolaris system and is thus the ancestor of both the OpenZFS open source and Oracle closed source versions of ZFS (see section 3.1).

OpenZFS pools after version 28 use “Feature Flags” instead of a single version number. The pool version number is set to 5000, and the pool configuration maintains a list of feature strings which are used in the pool. The strings use a reverse-DNS naming scheme to allow new features to be developed and named by different organisations independently (Ahrens, 2014a). This allows development of OpenZFS to become more decentralised.

2.2.3 *Software Components*

ZFS internally stores objects of various types and their dependency relationships. The lower layers of the filesystem provide a mechanism for storing objects in blocks across the pool, replicating and compressing blocks, and managing object groups and dependency relationships. The upper layer interprets the object store and presents the data as a file system (Bonwick et al., 2003; Bonwick and Moore, 2008).

The lowest layer ("storage pool allocator" or SPA) manages the allocation (see section 2.2.5) and storage of blocks on devices within the pool. The SPA also handles error detection and correction by checking block checksums and replacing corrupt blocks with intact copies as they are read. It also handles duplicating or transferring blocks as required when devices are added or removed from the pool.

The middle layer ("Data Management Unit" or DMU) manages the allocation of objects into blocks and manages object sets. This layer also includes the "Dataset and Snapshot Layer" (DSL) which manages datasets and the dependencies between them, such as between a snapshot dataset and the filesystem dataset it was taken from.

The upper layer ("ZFS POSIX Layer" or ZPL) interprets these objects to present a standard file system to the OS and processes. The ZPL ensures system calls are translated into atomic and consistent transactions on ZFS objects. The ZPL is also responsible for creating new filesystems, and handling synchronous writes via the ZFS Intent Log (see section 2.2.4.2).

2.2.4 *Transactions and Transaction Groups (TXG)*

ZFS uses a transactional copy-on-write method for all writes. Blocks are never overwritten in place, instead a new tree of blocks is created from the bottom up while retaining existing pointers to unmodified blocks, with the new root written to the next Uberblock in the array.

Each transaction is the result of a system call, where each component of the transaction must be written atomically (either completely or not at all). For example, deleting a file would unlink the data blocks of the file, the file metadata object(s), and remove the file from the directory entry list, all in one transaction.

Transactions are collated in "Transaction groups" for committing to disk. Transaction groups are identified by a 64 bit transaction group ID (TXG), which is stored in all Block Pointers and some other structures written when the Transaction Group is flushed.

2.2.4.1 *Committing Transactions: Transaction States*

The transaction group commit process involves three states: open, "quiescing", and syncing (Leventhal, 2012). There may be up to three active TXG in memory, one in each state.

There is always one open TXG which is accepting new transactions. Transactions are written into in-memory structures and may continue while the TXG is in the open or quiescing state.

ZFS changes the TXG from the open to the quiescing state based on a time threshold (by default 5 seconds, although this is configurable) or at once if data needs to be flushed soon to free memory. ZFS will also advance the state before reaching the time or space threshold

in order to commit a significant operation which may delay other transactions (such as the creation or destruction of a dataset).

A quiescing TXG no longer accepts new transactions, but existing transactions are allowed to complete. A new open TXG is made active to accept any new transactions.

Once all transactions of a quiescing TXG are loaded into memory, it may advance to the syncing state where it is written to disk. The syncing state may add new internal data to be written to in-memory structures (e.g. to allocate blocks for data to be written, space must be allocated, which requires writing the free space metadata). ZFS writes data iteratively until all dirty structures from the TXG have been committed to disk, ultimately writing a new Uberblock in all device labels affected by the transaction.

Only one transaction group can be in each state at once; if a TXG is already in the syncing state a quiescing TXG may not start syncing. To ensure the process of writing metadata iteratively in a syncing TXG does not delay other TXG, ZFS may begin reusing blocks which were allocated and then freed earlier in the syncing process (earlier in the process, new blocks are used to optimise for large, continuous writes of file data; later iterations which write internal metadata will require smaller writes). ZFS will also stop compression and other optional operations and defer frees if a syncing TXG takes too many iterations to complete.

2.2.4.2 *Synchronous Writes and the ZFS Intent Log (ZIL)*

Synchronous writes and other transactions which must be committed to stable storage immediately are written to the ZFS Intent Log (ZIL). Transactions in the ZIL are then written to a transaction group as usual, the ZIL records can then be discarded or overwritten. When recover-

ing from a power failure, uncommitted transactions in the ZIL are replayed. ZIL blocks may be stored on a dedicated device or allocated dynamically within the pool. ZIL blocks may contain multiple records, each containing one system call transaction,

2.2.5 *Space Allocation and Free Space Management*

Allocating space in ZFS for a new object involves three steps.

(1) A virtual device is selected from the pool. The selection uses a modified round-robin algorithm which prioritises underutilised devices and attempts to write 512KB(Bonwick, 2006) to each device before moving on.

(2) A "metaslab" within the device is selected. Each device is divided up into equal sized regions called metaslabs, optimised for copy-on-write performance (Bonwick et al., 2003).

- The metaslab selection algorithm prioritises metaslabs with higher bandwidth (closer to the outer region of the disk) and those with more free space, but de-prioritises empty metaslabs.

(3) A region within the metaslab is allocated to the object using a modified first-fit algorithm.

The location of free space within the metaslab is recorded in a "Spacemap" object. Section 2.3.5 describes Spacemaps in detail.

2.2.5.1 *RAID-Z Devices*

ZFS Virtual devices may combine physical devices in a Mirror or "RAID-Z" configuration for guaranteed, fixed redundancy. Mirror devices duplicate all allocated blocks similar to a disk volume mirror (but operating at the ZFS block level). RAID-Z devices operate similar to RAID-5 or RAID-6, but also operate at the block level and take advantage of ZFS's storage mechanism to use a variable width stripe and eliminate the RAID-5 "Write hole" (Bonwick and Moore, 2008), which in a conventional RAID could cause the data and parity to become unsynchronised.

A virtual device constructed as a RAID-ZN group uses N blocks for parity (where N is 1, 2 or 3). For example, a five disk RAID-Z1 Vdev would use one parity block for every four data blocks (or part thereof); a five disk RAID-Z2 device would use two parity blocks for every three data blocks. The RAID-Z layout information is stored in the Block Pointer to each block. Parity blocks are distributed amongst the disks, and if a disk fails only the allocated blocks need to be rebuilt, reducing recovery time.

2.3 EXAMINING ZFS ON-DISK STRUCTURES

This section describes the ZFS structures analysed as part of this project. Some other ZFS structures which may be the subject of future research are discussed in section 6.3.2.

2.3.1 *ZDB: The ZFS Debugger*

The ZFS Debugger (ZDB) can be used to examine ZFS structures, presenting the data in a human readable format. Depending on the options chosen, ZDB operates on physical devices, entire pools, individual datasets, or Data Virtual Address (see section 2.3.3.1).

The following subsections contain ZDB output for the specific structures examined in this project. Example output from ZDB is shown in Figure 2.2. This is the result of the "zdb -C <poolname>" command which displays the configuration of a pool. This is a simple pool with a single physical device.

2.3.2 *Labels and Uberblocks*

Each physical ZFS device contains a set of 4 identical "Vdev labels", 2 at either end of the device. The labels are written in a staggered manner to ensure that a valid label remains on disk at all times.(Sun Microsystems, 2006)

Each label contains information on the configuration of the pool and an array of 128 Uberblocks. The Uberblocks are the starting point of the block and object trees which contain the data and metadata stored within the ZFS pool. It is important to note for forensic analysis that Uberblocks do not point to the last 128 individual writes or transaction; due to batching of the writes into transaction groups, many transactions may be written in one group.

```
MOS Configuration:
        version: 5000
        name: 'zfs-sml-mirr'
        state: 0
        txg: 390188
        pool_guid: 2666171657549792388
        hostid: 966097692
        hostname: 'algernon'
        vdev_children: 1
        vdev_tree:
            type: 'root'
            id: 0
            guid: 2666171657549792388
            create_txg: 4
            children[0]:
                type: 'disk'
                id: 0
                guid: 4049685529925918728
                path: '/dev/ada0p3'
                phys_path: '/dev/ada0p3'
                whole_disk: 1
                metaslab_array: 30
                metaslab_shift: 33
                ashift: 9
                asize: 1466726612992
                is_log: 0
                DTL: 58
                create_txg: 4
        features_for_read:
```

Figure 2.2: Example Pool Configuration output from ZDB

```
  ...
Uberblock[72]
  ...
Uberblock[73]
        magic = 0000000000bab10c
        version = 28
        txg = 1737
        guid_sum = 4420604878723568201
        timestamp = 1382428544 UTC = Tue Oct 22 18:55:44 2013
        rootbp = DVA[0]=<0:3e0c000:200> DVA[1]=<1:3f57200:200> DVA[2]=<2:3593a00:200>
             [L0 DMU objset] fletcher4 lzjb LE contiguous unique triple size=800L/200
            P birth=1737L/1737P fill=42 cksum=15ffed59a7:7e9c9c594b0:17c4c7cb7a7eb
            :318e5de89d442a
Uberblock[74]
   ...
```

Figure 2.3: Example Uberblock dump from ZDB

The Vdev label (including Uberblocks) can be displayed using the ZDB command "`zdb -P -uuu -l /dev/<device>`". A partial example of the Uberblock list from the output of this command is shown in Figure 2.3. The complete version is provided in appendix section A.2.

Each Uberblock contains 6 components:

(1) Magic: A constant magic number which identifies an Uberblock.

(2) Version: The ZFS Pool version (see section 2.2.2 for details).

(3) TXG: The Transaction Group ID of the transaction group in which the Uberblock was written.

(4) GUID_Sum: This value is the sum of the GUID of all physical devices in the pool, used to verify that all pool devices are present when loading the pool.

(5) Timestamp: The UTC time that the Uberblock was written.

```
DVA[0]=<0:24001bc0600:c00> DVA[1]=<0:4a345f03600:c00>
[L1 ZFS plain file]
fletcher4 lzjb LE contiguous unique double
size=4000L/400P birth=625760L/625760P fill=3
cksum=63e2c4fa2d:40ade9c4737d:16af1a1451f90c:5aa133e0f1e06e8
```

Figure 2.4: Example Block Pointer output from ZDB

(6) RootBP: A Block Pointer which points to the block containing the Meta Object Set, the root of the ZFS object tree. Block Pointers are described further in the following subsection.

The TXG and timestamp may be particularly useful for forensic purposes. The potential for Uberblocks to be used as a source of timeline events is discussed in section 5.1.4.

2.3.3 *Block Pointers (BP)*

Block Pointers (BP) are a 128 byte structure which connects all blocks in the tree and are integral to the data integrity features of ZFS.

Block pointers are embedded in many objects including the Uberblock; many ZDB commands to dump data will include some embedded Block Pointers. Block Pointer output from ZDB used as an example in this section is shown in Figure 2.4; embedded Block Pointers in other structures can be seen in Figures 2.3 and 2.6.

ZDB output for a Block Pointer contains the following components:

(1) Data Virtual Address (DVA, see section 2.3.3.1 for details). Each Block Pointer contains space for three DVA entries which may point to identical, redundant copies of the block.

Less than three DVA entries will be present in the ZDB output if there are less than three copies of the block; in this case only two copies of the block are present. The number of DVAs used is referred to as the "wideness" of the Block Pointer: single, double or triple.

(2) The Level of the Block Pointer and the type of object it is pointing to - in this example this is a Level 1 pointer to ZFS file data. The "Level" refers to the levels of direct and indirect blocks used to point to the data - see section 2.3.3.2 for details.

(3) Checksum algorithm - ZFS supports several algorithms for the checksum. In this example the "fletcher4" algorithm is used.

(4) Compression algorithm - ZFS supports multiple algorithms for compression. In this example the "lzjb" algorithm is used.

(5) Flags - after the checksum and compression values ZDB shows additional flags or options which are enabled on the Block Pointer. In this case "contiguous" indicates that the data is stored in a contiguous region of the disk (not fragmented), "unique" indicates that this data is not shared (by a snapshot or clone), and "double" is the wideness of the Block Pointer (number of DVAs used).

(6) Size - the logical and physical size of the data the Block Pointer is pointing to. See 2.3.3.1 for an explanation of the different size terms used by ZFS. Because this is a level 1 pointer, this is the size of the array of level 0 pointers at the block this BP is pointing to - the actual file data is referenced by the level 0 BPs. Section 2.3.3.2 describes the operation of indirect blocks.

(7) Birth TXG - this is a 64 bit value containing the Transaction Group ID in which the Block Pointer was written. Section 2.2.4 describes TXG IDs and the ZFS Transaction process. Block Pointers have space for a physical (suffix "P") and logical (suffix "L") birth time to be stored; these are the same unless the pool uses deduplication. A deduplicated Block Pointer will contain the physical birth TXG of the previously allocated block.

(8) Fill count - This is the number of Block Pointers "underneath" this one in a tree of indirect blocks. For DMU Dnode objects (meta-objects containing other objects) the fill count is instead the number of free dnodes contained beneath the Block Pointer (Sun Microsystems, 2006).

(9) Checksum of the block. Note that this is a checksum of the block the Block Pointer points to, not the pointer itself. Storing the checksum in the "parent" structure provides greater integrity than storing it in the block itself. Simultaneous corruption of the data and checksum is less likely, it is not vulnerable to disk errors which fail to write data or write the correct data at an incorrect location, and it allows the tree to validate itself because the integrity of each checksum is itself guaranteed by the parent checksum. (Bonwick et al., 2003)

The potential for Block Pointers to be used to detect false timestamps and as a source for extra timeline events is discussed in sections 5.1.1 and 5.1.2.

2.3.3.1 *Data Virtual Address (DVA) and Block Size*

ZFS blocks in the pool are identified by a Data Virtual Address (DVA). The DVA contains three components: *"vdev"*, *"offset"* and *"asize"*.

The Vdev value is a 32 bit integer which specifies the virtual device where the block is stored. The offset is a 63 bit integer which specifies the block within the virtual device where the data is stored. The offset is the number of sectors (sectors are typically 512 or 4096 bytes) from the start of the device.

The asize is the "actual size" of the data at the block address, after compression, and including any RAID-Z parity data (Sun Microsystems, 2006). ZFS also uses the terms "psize" or "physical size" (to refer to the size of the data after compression but without parity overhead) and "lsize" or "logical size" (to refer to the size of the data without compression or parity).

2.3.3.2 *Block Pointer Levels and Indirect Blocks*

Where Block Pointers need to refer to multiple blocks (e.g. if not enough contiguous space is available for the object, or more than one block is required), a tree of "Indirect" blocks is used.

Indirect blocks are blocks which contain an array of Block Pointers. Indirect block pointers of levels one and higher point to blocks containing further Block Pointers; level zero Block Pointers are the leaves of the tree and point to the actual data. An example of a file object containing indirect blocks is shown in Figure 2.5; most fields of the file object have been removed for clarity. The complete version is provided in appendix section A.1.

When an object or file data is partially modified, level 0 pointers which point to unchanged data blocks remain unchanged. New blocks are written for any data changed, with corresponding new level 0 pointers; these are linked into new copies of the parent level 1 pointers and so forth until a new copy of the block tree is written (Sun Microsystems, 2006).

```
    Object  lvl   ...   dsize  ...  type
         7    2   ...   258K   ...  ZFS plain file ...
...
   gen   25
   size  145408
...
Indirect blocks:
    0 L1  DVA[0]=<0:16f200:400> DVA[1]=<0:3c36c00:400>
        [L1 ZFS plain file]
       ... size=4000L/400P birth=29L/29P fill=2 ...
    0  L0 DVA[0]=<0:dec00:20000> [L0 ZFS plain file]
        ... size=20000L/20000P birth=25L/25P fill=1 ...
    20000  L0 DVA[0]=<0:14f200:20000> [L0 ZFS plain file]
        ... size=20000L/20000P birth=29L/29P fill=1 ...
```

Figure 2.5: Simplified example File Object with Indirect Blocks

The example file in Figure 2.5 contains data in two segments written during different transactions. The file object points to one level 1 indirect block containing two level 0 leaf Block Pointers, which point to the file data. The data in the first level 0 BP was written during TXG 25, whereas the second level 0 BP was written in TXG 29 (requiring the parent BP to be also rewritten).

At most six levels of indirection are possible. With the default block size of 128KB, a block may contain 1024 Block Pointers; with the maximum level of indirection up to 2^{64} bytes of data may be allocated to an object.

2.3.4 *File Objects*

Each dataset contains a set of objects describing the dataset itself, as well as objects for each directory and file. This project did not examine the dataset or directory objects, although the possibility of examining them in future research is discussed in section 6.3.2.

All objects and Block Pointers associated with a dataset may be extracted using ZDB's `-b` and `-d` options, which may be repeated up to 6 times to provide increasingly verbose output. A sample file object is shown in Figure 2.6; this is from the output of the command "`zdb -P -dddddd -bbbbbb <poolname>/<dataset-name>`".

The ZDB file output can be divided into three parts: the object header, the "bonus buffer" key-value data, and the Block Pointer list. These are explained in the following subsections.

2.3.4.1 *Object Header*

The header lines in the output contain items which are common with other dataset objects, including directory objects, symlinks, and master dataset objects. This includes the "object number" ID, size information, object level and type.

All objects in a dataset have a 64 bit "object number", used as an internal identifier for ZFS. Object numbers are allocated incrementally but may be reused. If a previous block of 4096 objects contains 1024 objects or less (i.e. three quarters or more have been deleted) the deleted object numbers in that block are candidates for reuse.

The level and type in the header are the same as those used in Block Pointers (see section 2.3.3). "iblk" and "dblk" refer to the block size used to store the file metadata and data, respectively. "lsize" refers to the logical size of the data (before compression and without storage or parity overhead) and "dsize" refers to the actual size of the data on disk (referred to as "asize" in some other parts of ZFS).

```
    Object  lvl   iblk   dblk  dsize  lsize   %full  type
     15417    1  16384  67072  67072  67072  100.00  ZFS plain file (K=inherit) (Z=inherit)
                                        168   bonus  System attributes
        dnode flags: USED_BYTES USERUSED_ACCOUNTED
        dnode maxblkid: 0
        path    /1382428539-to-2147483647-reps-0
        uid     0
        gid     0
        atime   Tue Oct 22 17:55:40 2013
        mtime   Tue Oct 22 17:55:40 2013
        ctime   Tue Oct 22 17:55:40 2013
        crtime  Tue Oct 22 17:55:40 2013
        gen     1737
        mode    100644
        size    66566
        parent  4
        links   1
        pflags  40800000004
Indirect blocks:
               0 L0 DVA[0]=<2:353c200:10600> [L0 ZFS plain file]
fletcher4 uncompressed LE contiguous unique single
size=10600L/10600P birth=1737L/1737P fill=1
cksum=1e970f0f68f0:3f178f0ed3b3b88:1c0a9b8bd4c82800:eb83cbb696eca800

                segment [0000000000000000, 0000000000010600) size 67072
```

Figure 2.6: Example File Object dump from ZDB

2.3.4.2 *Bonus Buffer*

Many ZFS objects include a "bonus buffer" in the object structure which can be used for different purposes depending on the object type. In file objects this buffer is used to store the system attributes, These attributes include the Unix user and group IDs and access mode (permission) bits, file size and the number of hard links to the file.

The time of file creation (crtime), last modification (mtime), last access (atime) and last attribute change (ctime) are also stored here; these are stored in UTC but converted to local time by ZDB for display. This should be considered when performing timeline analysis, to ensure that the time of events is recorded correctly.

The "gen" value is of particular interest for forensic purposes. "gen" is the Generation TXG, the ID of the Transaction Group where this object was created. The use of Gen TXG for timeline analysis is discussed in section 4.2.4.

The "parent" is the parent directory object number, in this case object number 4 which is the root directory of this filesystem dataset. A "xattr" entry may also be present (not in this example) which stores the object number of a directory containing extended attributes of the file.

2.3.4.3 *Block Pointer List*

The final part of the File Object output is the list of Block Pointers which point to the data of the object. For non-file objects this list usually points to blocks containing ZFS object data. For file objects, this list contains the Block Pointers which point to the file data.

The example object in Figure 2.6 only has a single Block Pointer as the file data fits in a single block (by default, the maximum size of a single block is 128KB). Figure 2.5 contains a simplified example of a file object with more than one Block Pointer; the use of indirect blocks to store more than one block of data is explained in section 2.3.3.2.

2.3.4.4 *Obtaining the Last Modified TXG of a File*

The TXG ID in which the file was (apparently) last modified can be obtained from the birth TXG of the highest level Block Pointer of the file. In the ZDB output this will always be the first Block Pointer in the list of Indirect Blocks.

For example, in the example file object in Figure 2.5, the TXG in which the file was last modified is 29, the birth TXG of the level 1 Block Pointer.

2.3.5 *Space Maps*

Each metaslab (see section 2.2.5) has an associated space map object which records which regions within the metaslab are allocated or free.

Space maps are stored in a “log-structured” format - data is appended to them when disk space is allocated or released. Space maps are “condensed”(Bonwick, 2006) when the system detects that there are many allocations and deallocations which cancel out. The cancellations are performed in memory and a new space map object is written to replace the old one.

The -m option for ZDB can be used to display the spacemaps for all metaslabs in the pool, repeated up to 6 times for increasing verbosity. Example space map output from ZDB with some fields removed is shown in Figure 2.7, from the command "`zdb -P -mmmmmm <pool>`". The complete version is provided in appendix section A.3.

In this example, only the first metaslab has data allocated. There are many blocks allocated in two passes of a transaction, followed by a free in the second pass (see section 2.2.4 for an explanation of transactions and passes).

This Spacemap itself was written in Transaction Group 229; however the allocated blocks in this Spacemap may have been allocated and written in an earlier transaction group, then condensed in Transaction Group 229 or earlier.

2.4 TIMELINE ANALYSIS

This section provides a brief explanation of timeline analysis concepts for readers unfamiliar with digital forensics. For more information, please consult the articles from section 3.3.2 of the literature review.

Timeline analysis is the collation of events into an account of what occurred over time on a system in a digital forensics investigation. Timeline analysis is a vital component of many investigations. It is often essential to identify the sequence of events, validate disputed events, determine the the time at which a crime occurred, and/or provide corroboration of witness accounts or other evidence.

```
    vdev          0
    metaslabs   119   offset                spacemap          free
    ---------------   -------------------   ---------------   -------------
    metaslab      0   offset            0   spacemap     35   free    536655872
                      segments         15   maxsize  536123392   freepct   99%
        [     0] ALLOC: txg 229, pass 1
        [     1]    A  range: 0000000000-0000008000  size: 008000
        [     2]    A  range: 0000011000-0000016800  size: 005800
        [     3]    A  range: 000000f800-0000010800  size: 001000
        [     4]    A  range: 000001c800-000001d000  size: 000800
...
        [    14]    A  range: 00000ad000-00000b4800  size: 007800
        [    15]    A  range: 0000093000-000009a800  size: 007800
        [    16]    A  range: 0000043800-0000047800  size: 004000
        [    17] ALLOC: txg 229, pass 2
        [    18]    A  range: 0000008000-000000b000  size: 003000
        [    19] FREE: txg 229, pass 2
        [    20]    F  range: 00000c4000-00000c7000  size: 003000
    metaslab      1   offset     20000000   spacemap      0   free    536870912
                      segments          1   maxsize  536870912   freepct  100%
    metaslab      2   offset     40000000   spacemap      0   free    536870912
                      segments          1   maxsize  536870912   freepct  100%
...
```

Figure 2.7: Example Metaslab/Spacemap dump from ZDB

Events can be collected from many sources, including:

- Filesystem timestamps - typically file modification, access and creation time, sometimes referred to as MACtimes. Some filesystems also store a "change" time when the file's attributes last changed.
- Log files, including system logs, application logs, authentication logs and service or daemon logs.
- Internal file metadata, such as internal timestamps in document files indicating when content was changed.
- Web browser profile, including bookmarks, cache and cookies as well as the URL history.
- Lists of "Recently accessed" files or applications maintained by the user interface.
- Timestamps in email headers.

In the past forensic investigators would use a variety of software and/or self-developed scripts to collect event data from different sources. "Super-timeline" software (Gudjonsson, 2010) analyses an entire directory tree or disk image for events and collates them into a single timeline. Super-timelines also provide a mechanism to detect falsified timestamps and other tampering by detecting anomalies between the different sources of events. Falsification of timestamps may be performed in order to destroy or hide evidence (such as missing content or activity), or generate false evidence (such as planting an incriminating file on an innocent user's system)(Harris, 2006).

Some timeline analysis software may be used to perform higher-level analysis, such as grouping individual events into user activities(Inglot et al., 2012; Khan and Naeem, 2012). For exam-

ple, a user editing a document they received via email may involve many events: the browser history, cache and cookies from a web-mail access; saving the file to disk; temporary files created by a word processor and saving the modified file.

3

LITERATURE REVIEW

This literature review is divided into three sections which in turn examine literature related to ZFS in general (section 3.1), ZFS specific forensic literature (section 3.2), and finally related forensic literature not specific to ZFS (section 3.3 including timeline analysis literature in section 3.3.2).

Section 3.2 contains the most detail as it is the intersection of the two topics which is the focus of this project; Figure 3.1 shows the layout of the three sections and how the topics overlap.

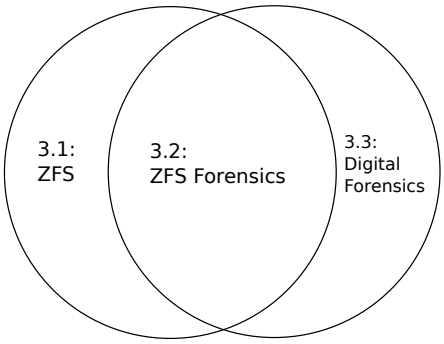


Figure 3.1: Sections and Topics of the Literature Review

3.1 ZETTABYTE FILE SYSTEM LITERATURE

This section discusses literature which describes the internal structure of ZFS, but do not have a forensic focus.

3.1.1 *Early ZFS*

The Zettabyte File System (Bonwick et al., 2003) is the seminal publication from the original developers at Sun Microsystems describing the new filesystem. It provides an overview of the motivation behind developing ZFS, the design principles, software model and a brief demonstration of using ZFS.

Although many elements and specifics of the ZFS filesystem were not finalised at this stage, the basic design and concepts of pooled storage, the block and object trees and the three layer software design were established. As an overview, the details of the ZFS on disk and memory structures and many details of implementation are not provided.

The "ZFS On-Disk Specification"(Sun Microsystems, 2006), published by the original developers at Sun Microsystems, was the definitive standard for the on-media structure of the ZFS file system. It describes ZFS Pool and Filesystem version 1 (see section 2.2.2 for more details of ZFS versions).

It provides an extremely detailed description of many on disk structures, including the Vdev labels, Block Pointers and ZFS intent log including the exact byte-by-byte layout of their structures.

However, some structures are not included in the description, notably many Data Management Unit structures including Spacemaps. The description of these structures is stated to be included in "future chapters" of the specification, but no later versions of this document were released by Sun Microsystems before it was acquired by Oracle Corporation (see "OpenZFS" below).

The official Sun Microsystem blog posts made by the ZFS developers contain a great deal of in-depth information. These often discuss technical matters; Bonwick (2006) in particular describes the metaslab and block allocation algorithm of ZFS.

Another source of information are the numerous conference presentations made by Sun Microsystems developers while promoting ZFS. "ZFS The last word in filesystems" (Bonwick and Moore, 2008) is oriented towards system administrators and end users but discusses many ZFS structures and procedures which are of interest. Another presentation by Ulrich Gräf, "ZFS Internal Structure" (Graf, 2009) provides more details on the internal structures of ZFS. However they do not provide substantial detail beyond the "ZFS On-Disk Specification"(Sun Microsystems, 2006).

Most importantly for timeline analysis is the description of the way Transaction Group ID numbers (see section 2.2.4) are stored in Block Pointers, with each update changing one path within the tree - this provides essential data on the previous iterations of the tree. This behaviour can be used for forensic purposes and is described further in section 5.1.2.

3.1.2 *OpenZFS and FreeBSD*

Ahrens (2014a) presents an overview of the OpenZFS project, its origins and future plans. After Sun Microsystems was acquired by Oracle Corporation in 2010, Oracle ceased contribution of source code and other support to the open source version of ZFS and OpenSolaris. By his time ZFS had already been incorporated into other open source operating systems which were developing their own variants. The OpenZFS project was announced on September 17th 2013 to support and manage continued open source development of ZFS.

Ahrens describes the ongoing activities of OpenZFS, particularly the efforts to facilitate communications between developers from different organisations via websites, mailing lists and conferences. New and planned features are described briefly, with particular focus on Feature Flags (see section 2.2.2) which are intended to facilitate distributed development by making it easier for developers from different organisations to add new features.

Some of these new features are described in another presentation by Ahrens (2014b), "*OpenZFS: upcoming features and performance enhancements*". Particular focus is on improvements to write latency and the new LZ4 compression algorithm. As well as code enhancements, the OpenZFS development model is discussed, particularly the opportunity to reduce the amount of platform-specific code in the Linux and FreeBSD ports.

ZFS was ported to the FreeBSD operating system by Pawel Jakub Dawidek. The presentation (Dawidek, 2007) of this work outlines the layout of the source code repository, the Solaris compatibility layer (including how atomic operations and other OS-specific operations were

ported) and the modifications required to fit ZFS into the FreeBSD GEOM Storage framework. Usage of the new kernel I/O routines and userland utilities are explained, as well as the test suite used to verify the porting process.

A broader examination of ZFS on FreeBSD is provided in chapter 10 of "The Design and Implementation of the FreeBSD Operating System" (McKusick et al., 2014). The design principles and organisation of ZFS are explained, as well the fundamental ZFS Block Pointer and Dnode (object) structures. The function of the object sets and Meta Object Set are discussed in detail, as well as ZFS operations including writing to disk, allocating and freeing space, snapshots, RAID-Z, deduplication and replication. Finally, the "design tradeoffs" of ZFS are discussed, particularly the performance effects from the copy-on-write operation of ZFS.

The procedures and operations of ZFS are discussed in great detail, and there is less focus on the data structures used. Byte-by-byte details of the on-disk structures are not included with the notable exception of Block Pointers. BPs are described in detail including changes made since version 1 of the ZFS Pool; the version described in the ZFS On-Disk Specification (Sun Microsystems, 2006).

3.2 ZFS FORENSICS LITERATURE

There are few existing studies of ZFS forensics, and these are based mostly on theoretical and source code analysis rather than examination of actual ZFS filesystems. Due to the unusual structure of ZFS many other filesystem studies and techniques are not applicable to forensic analysis of ZFS disks.

3.2.1 *Digital Forensic Implications of ZFS*

"Digital Forensic Implications of ZFS" (Beebe et al., 2009) is an overview of the forensic differences between ZFS and more "traditional" file systems; in particular it outlines many forensic advantages and challenges of ZFS. However, it is based on theoretical examination of the documentation and source code, noting that "we have yet to verify all statements through our own direct observation and reverse engineering of on-disk behavior".

Beebe identifies many properties of ZFS which may be useful to a forensic investigator, highlighting the opportunities for recovery of data and metadata from multiple allocated and unallocated copies. The copy-on-write transactions used in ZFS imply that many previous copies of data and metadata may remain on the disk. The increased "temporal state awareness" from previous copies of file content and metadata is also discussed; providing greater opportunity for an investigator to trace user activity or intrusions chronologically.

Challenges and disadvantages of investigating ZFS are listed, most resulting from the increased complexity of the filesystem causing increased complexity for the investigator. Compression is identified as a particular challenge. Whereas some filesystems implement compression at the volume or file level, each instance of a ZFS object can be configured to use different compression algorithms. Thus forensic tools which analyse ZFS devices require additional preprocessing on blocks before the data can be analysed.

The article also discusses how the use of dynamically sized extents in ZFS causes more complications for a investigator, as it affects the presence of old data in file "slack space" and makes it more difficult to locate the data from a specific file.

Although the article notes the limitations of the research, and areas which need further attention - particularly the ZIL (see section 2.2.4.2 for details) and snapshots - Beebe and the other researchers do not appear to have investigated ZFS further, as there are no further ZFS publications by these authors as of August 2014. This thesis provides some empirical analysis of on-disk filesystems which was noted in the article as worthy of further study.

Beebe mentions ZFS forensics another article - *"Digital forensic research - The good, the bad and the unaddressed"* (Beebe, 2009) - which examines the current state of forensic research (as of 2009) and discusses areas which have received less attention. Beebe notes that most disk forensics research focuses on filesystems used with Microsoft and Apple desktop operating systems, and there are few forensic studies of filesystems common in Linux and other Unix-like systems. ZFS, UFS and ReiserFS are mentioned as particular examples of "file systems that deserve to be the focus of more research".

3.2.2 *ZFS Timeline Analysis*

As far as the author is aware, no existing ZFS forensic studies examine timeline analysis in detail, apart from the research arising from this project. Section 3.3 examines timeline analysis literature not specific to ZFS.

The research in this thesis was presented at BSDCan 2014 (Leigh and Shi, 2014b). The presentation and proceedings paper briefly explained ZFS internals and the structures analysed in this research. Minimal results were presented, instead focusing on the the use of ZFS structures for forensic purposes and explaining which were most effective. Further forensic issues such as false positives and tampering with internal metadata were discussed as well as potential future research.

Another article (Leigh and Shi, 2014a) was published by the author as part of a separate project to incorporate this research into existing super-timeline forensic software. Two parsers were developed for the Plaso (Gudjonsson, 2013) timeline analysis software to parse ZFS Debugger (ZDB) output and add ZFS specific events to a Plaso timeline.

These parsers are the first timeline analysis tool to make use of ZFS internal metadata. The article describes the process of implementing the parsers, including the Plaso classes which were extended in the new code. The parsers were tested with the same simulated disk activity used in this research. It is noted that many timeline analysis procedures could be automated further in future revisions of the parsers, notably the automatic detection of anomalies.

3.2.3 *ZFS Data Recovery*

Recovery of deleted files on ZFS has been examined in more detail that other areas of ZFS forensics. Data recovery is indirectly related to timeline analysis as deleted files may provide a source of particularly important events for a super-timeline.

Max Bruning's *On Disk Data Walk* (Bruning, 2008b) demonstrates how to access ZFS data on disk manually. Using the ZFS Debugger (ZDB - see section 2.3.1 for details) and the Solaris Modular Debugger (MDB), Bruning begins with the Vdev label, extracting each object and parsing it to read the locations of further objects, continuing this process and briefly explaining each structure until the file data is reached. This process could be invaluable for forensics investigators seeking to locate file data on a damaged filesystem.

Bruning has also demonstrated techniques for recovering deleted files(Bruning, 2008a, 2013). Like the previous article, it uses ZDB and MDB to extract individual ZFS objects and parse them, iterating until the object for the deleted file can be located. The object is then parsed to locate the file data. These techniques only work if the blocks allocated to the file object and data have not been overwritten with new data, and the file object number has not been reused.

Li (2009) presents an enhancement to ZDB which allows it to trace target data within ZFS media without mounting the file system. The report also provides a useful and detailed description of the internal structure of ZFS, and a comparison to more traditional Unix filesystems.

The ZDB enhancement is described in great detail, including the process of parsing ZFS structures. The new feature is designed for locating data from deleted files or otherwise concealed within the filesystem, not for timeline analysis. However, the recovered data may provide an additional source of events.

3.2.4 *ZFS Anti-Forensics*

ZFS anti-forensics techniques are examined by Cifuentes and Cano (2012). This article was published in *IEEE Latin American Transactions* in Spanish, and the review below is based on an automatic translation into English which may be inaccurate.

Using the anti-forensics framework developed by Harris (2006), Cifuentes and Cano consider what anti-forensics techniques could be used on ZFS to destroy evidence, hide evidence, remove sources of evidence, and falsify evidence.

The potential for destruction of evidence is evaluated by attempting to remove all traces of a file from a ZFS pool, including the file object and references in other objects. It is found that once all references to a deleted object are removed, no abnormalities were reported by ZFS, although past copies of the data were present on disk due to the copy-on-write mechanism of ZFS. This confirms some of the theories presented by Beebe et al. (2009).

The potential to hide evidence in ZFS is evaluated by attempting to use extra space in objects, including space "reserved for future use", to store data. This data would not be visible to the filesystem itself. The block size was also increased to provide increased slack space. This was only partially successful, with the alterations to objects detected by ZFS checksums and some of the hidden data being destroyed when ZFS rewrote the object it was hidden in. However, as ZFS does not checksum unallocated space hiding data in unallocated blocks was successful.

Only the ZIL (see section 2.2.4.2 for details) is considered when evaluating the potential for sources of evidence to be removed. It was found that this can be done easily and prevents

the ZIL being used to recover file activity. The article does not consider any other sources of evidence which could be removed including the checksums (which can be disabled by the system administrator) or altering the time between transaction group writes to flush old data faster.

Falsification of evidence is considered by changing file attributes (including timestamps and the file owner) and internal metadata (including Uberblock timestamps). The experiments only tested changing Uberblock values. It was discovered that changing fields in the active Uberblock will be detected by ZFS using the checksum; however, previous Uberblocks did not have their checksums verified so they could be altered without detection.

This finding is not so significant as the reason the checksums are not normally verified is that under normal circumstances previous Uberblocks will never be read or used by ZFS, and will be quickly overwritten. Only the active Uberblock (with the highest Transaction Group ID) is used when reading the disk. If this Uberblock is corrupt, ZFS may discard it and read the next highest Uberblock (and so on), and will then verify all checksums. The experiments did not appear to test this by forcing ZFS to make use of any of the previous Uberblocks that were tampered with.

However, this finding may have some effect of the use of Uberblocks for detecting falsified timestamps. This is evaluated in this thesis in section 5.1.4. Section 6.3.5 also further discusses the effect of falsifying internal metadata including Uberblocks.

Finally, Cifuentes and Cano provide some practical recommendations for forensic investigators, to detect and counter the anti-forensic techniques which were examined.

The article does not appear to provide details of how file activity was simulated during the experiments, and some of the anti-forensic techniques may be more or less effective on a production system.

3.3 NON-ZFS SPECIFIC FORENSICS LITERATURE

This section discusses forensic literature which is not specific to forensic examination of ZFS disks, and contains two subsections. Section 3.3.1 discusses partially related filesystem forensic literature; Section 3.3.2 reviews literature relating to timeline analysis in general.

3.3.1 *Related Filesystem Forensics Literature*

This subsection discusses filesystem forensic literature which is related to this project but not specific to forensic examination of ZFS devices.

Hankins and Liu (2008) categorise various filesystems (including ZFS) comparing their "forensic aware" features. The article makes many generalisations and errors in categorisation of file systems, notably describing ZFS as an "exemplary" journaling filesystem. ZFS does not use a filesystem journal, although the ZIL performs logging operations similar to a journal (for synchronised writes only), and some previous versions of the object tree may be accessible via the Uberblock array.

The use of ZFS snapshots for forensic data retention is discussed, noting that an ideal forensic-aware file system would make it difficult for an intruder who gained administrative access to destroy such snapshots. Hankins and Liu raise several interesting points regarding assumptions made by filesystems developers and computer forensics investigators, in particular the common assumption that a suspect is unaware of anti-forensics techniques which they could use to alter or conceal digital evidence. They argue that intruders are increasingly likely to use anti-forensics methods, using the analogy of a criminal wearing gloves to thwart fingerprint detection.

Goja and Padha (2012) present an analysis of storage for digital forensics files, considering the performance of storage and retrieval. Because file data can be spread across multiple disks in the pool in ZFS, they find that ZFS is effective for storing large files such as forensic disk images. However, this article is not related to the forensic analysis of ZFS itself, only its use for storage of large files.

3.3.1.1 *Forensic Research Examining Other Filesystems*

Fairbanks (2012) studies the the Ext4 file system in a similar manner to this research into ZFS. Fairbanks' examination of Ext4 includes a detailed low-level analysis of its data structures, and briefly mentions how some of them may provide events for timeline forensics. Ext4 is a recent but traditional Unix filesystem, an updated version of the Ext2/Ext3 filesystem which is based on the Berkeley Fast File System (McKusick et al., 2014). Ext4 is thus similar to many other inode-based Unix filesystems and does not use the pooled devices, object tree or transactional copy-on-write approaches used by ZFS.

Martini and Choo (2014) use a detailed case study of XtreemFS as a case study for forensic analysis of distributed filesystems. ZFS is a local filesystem, although it supports remote replication and is widely used as a back-end for distributed filesystems (OpenZFS, 2014).

The internal structure of XtreemFS is examined in detail with a view to extracting forensically sound metadata, including timestamps. Notably, the effect of different timezones (from remote nodes of the distributed filesystem) is discussed; this complicates the collection of legally valid evidence from the filesystem. Complications of preserving evidence on a distributed system (where an examiner may not have access to physical media) are also discussed.

3.3.2 *Timeline Analysis*

This subsection discusses digital forensic timeline analysis literature which is not specific to ZFS.

"Zeitline" was one of the first special purpose forensic tools to manage a fully integrated timeline of events from many sources. Its design and implementation is described by Buchholz and Falk (2005).

Bucholz and Falk begin by reviewing existing forensic tools which manage timelines, but do not integrate events from different sources (e.g. file timestamps and browser history) into the one timeline. Zeitline was designed to manage events from any source, using an "Input Filter" to translate events from different sources into a generic format.

The software and user-interface design of Zeitline is described in detail, including its ability to timelines by categorising and grouping events. Forensic integrity is maintained by preventing the user from deleting events; events however may be hidden from view to allow visualisation of only the relevant events. Finally, potential functionality and user improvements are discussed.

A similar approach was taken by Gudjonsson (2010) in developing "log2timeline", which generates a "super-timeline" automatically from multiple file an event types. In greater detail than Buchholz and Falk (2005), Gudjonsson describes the flaws with "traditional" timeline analysis which considers events from different sources separately, and the effect this has on an investigation. The manual collection of different types of events, using different tools also causes added complication and requires more time on the part of the investigator.

Log2timeline attempts to resolve these issues by providing a single tool which analyses a set of files, detecting the file types and extracting any events contained within. File timestamps from the file system are also used. Details of the design and implementation of Log2timeline are provided, including the input modules which are matched against each file to determine if the file can be parsed by the module to generate events.

Gudjonsson later developed "Plaso" (Gudjonsson, 2013) to resolve architectural and usability deficiencies in the original Log2timeline. Plaso uses parallelization to allow faster collection and processing of events, and may be used on disk images as well as directory trees. The Plaso libraries support greater reuse of common features amongst input modules, making development of new input parsers and plugins simpler. As mentioned earlier, Leigh and Shi (2014a) extended Plaso to incorporate ZFS events into its super-timeline.

Plaso uses a special purpose library to access the forensic disk image to ensure as much metadata as possible is obtained and the integrity of the metadata is maintained. The effect of file retrieval from cloud services on the integrity of metadata (and timestamps in particular) is evaluated by Quick and Choo (2013).

Quick and Choo found that in many cases the timestamps of files downloaded or "synced" with a cloud storage service were not correct; for example, the creation time of files was usually set to the time the file was downloaded from the service, not the time the file on the cloud service was created.

Results varied between different cloud services and synchronisation methods. Some of the incorrect timestamps may have been caused by the software used to un-archive downloaded files; it was noted that other software may preserve timestamps when unpacking archives.

It is notable that the internal metadata of the client software (including the web browser, in the case of web-based cloud services) was found to be a useful source of evidence, providing additional log files and other sources of timeline events. This should be considered by investigators when such software is detected on a system, to ensure accurate timestamps are recorded.

Spencer (2014) presents a discussion of "relative time" events within super-timelines, where the time of the event is not known precisely, but is known to be within a range of times, possibly before and after other events. The article focuses on "anchors" which can be used to detect falsified timestamps using relative time events and determine the true order of events. Spencer discusses in detail a case study where this technique was used during an investigation of a high-profile criminal case.

The anchor-based methods described by Spencer are "high-level" timeline analysis, involving post-processing of collected events using relative times, rather than the "low-level" collection of events itself which is discussed in this thesis.

Some other high-level timeline analysis focuses on simplifying the workload of the investigator. Inglot et al. (2012) presents a framework which uses Zeitline (Buchholz and Falk, 2005) to assist the investigator by grouping related timeline events, highlighting unusual events and reducing irrelevant "noise" events. A similar study by Khan and Naeem (2012) uses Bayesian networks to classify and group filesystem events.

4

EXPERIMENT METHODOLOGY

This chapter describes the method used to conduct the experiments and analyse the data collected from them.

Section 4.1 describes the process used to conduct the main set of file activity simulations, including timestamp falsification, and how the internal ZFS metadata was collected. The metadata was then used to evaluate the effectiveness of the timestamp falsification algorithms described in section 4.2.

A smaller set of experiments were conducted after the primary set to confirm the behaviour of large ZFS objects observed during the primary simulations; the differences between these "large file" experiments and the primary set of experiments is discussed in the final section 4.3.

4.1 PRIMARY SIMULATION EXPERIMENTS

4.1.1 *Overview*

ZFS metadata was collected from test systems which used simulated disk activity. The systems used one to nine devices which were arranged in a total of 22 different pool configurations.

Each pool configuration was tested with two methods of falsifying timestamps, as well as a control where there was no falsification. The ZFS Debugger (ZDB; see section 2.3.1) was used to save metadata before, during and after each experiment, including the Uberblocks, Spacemaps and all Objects from the active filesystem.

4.1.2 *Simulated Filesystem Activity*

Disk activity on the test systems was simulated using data obtained from multiple surveys of activity on corporate network file storage (Leung et al., 2008; Agrawal et al., 2007; Roselli et al., 2000).

The statistical data from these surveys was used to implement a simple filesystem activity simulator which created, modified, deleted and changed attributes on files according to specified random distributions over time. Table 4.1 shows the distributions which were used by the simulator.

Table 4.1: File Activity Simulator Parameters

Activity	Parameters
File Write Rate	$Pois(2.7)$ per second
File Open Count	$\ln\mathcal{N}(0.33, 2.1)$ opens during lifetime
File Deletion Probability	55.2% of files never deleted
Deleted File Lifetime	47.5% deleted instantly Remainder deleted after $Exp(1.48 \times 10^{-6})$ seconds
Attribute Change Rate	$Pois(0.7)$ per second
Write Size	$\ln\mathcal{N}(8.32, 0.833)$ bytes

File writes - including both creation of new files and modification of existing files - use a Poisson distribution with $\lambda = 2.7$. The number of times a file is opened (i.e. the chance that it will be reopened for modification) is determined using a log-normal distribution when it is created; notably, 44% of files are never reopened(Leung et al., 2008).

The lifetime of the file and (therefore the deletion time) is also determined by the simulator when it is first created; 55.2% of files are never deleted, and 47.5% of the remainder are deleted instantly(Leung et al., 2008). The lifetime of the remaining files is determined by an exponential distribution.

Attribute changes occur at a rate determined by a Poisson distribution with $\lambda = 0.7$. Attribute changes (which would normally include permission bits as well as user/group identifiers) are simulated by toggling the execute bit of the file; this produces the same write behaviour on ZFS as any attribute change as the file object is rewritten with Block Pointers to the file data retained.

File size is determined by a log-normal distribution with mode of 2 KB and median of 4 KB (Leung et al., 2008; Agrawal et al., 2007).

The simulator was written in the Python (Python, 2014) language, and source code is provided in Appendix C.1. Existing simulator software such as FileBench (McDougall and Mauro, 2005) was examined but considered inappropriate for this project as it was incompatible with the FreeBSD or ZFS software, required a commercial licence, or was intended for benchmarking and "stress-testing" rather than simulating realistic behaviour for forensic experiments.

4.1.3 *Pool Configurations*

ZFS pools with 1, 2, 3, 4, 5, 7 and 9 devices were created on the test systems. The maximum of 9 devices was used as this is the maximum recommended number of devices which should be combined in a single virtual device(Watanabe, 2009); it is recommended that a hierarchy of devices be used for pools with more than 9 devices. Pools with 6 and 8 devices were not used due to time and hardware constraints.

These devices were arranged in all feasible pool configurations, including all levels of RAID-Z, mirrors and mirror pairs. In total 22 different pool configurations were used, as shown in Table 4.2 .

Note that some pool configurations which were technically possible were not used in experiments because they are not practical and provide no benefit compared to simpler configurations. For example, two devices could be configured in a RAIDZ-1; however this provides the same redundancy as a two device mirror but with slightly higher overhead, so this configuration would not be used in practice.

Table 4.2: Pool configurations used in Experiments

Devices	Pool Configuration					
	Flat	Mirror	Mirror Pair	RAIDZ1	RAIDZ2	RAIDZ3
1	Y	n/a	n/a	n/a	n/a	n/a
2	Y	Y	n/a	n/p	n/a	n/a
3	Y	Y	n/a	Y	n/p	n/a
4	Y	n/p	Y	Y	Y	n/p
5	Y	n/p	n/a	Y	Y	Y
7	Y	n/p	n/a	Y	Y	Y
9	Y	n/p	n/a	Y	Y	Y

Y indicates this configuration was used in experiments; n/a indicates that the configuration is not possible for the specified number of devices; n/p indicates configurations which were not used as they are technically possible but not practical.

4.1.4 *Timestamp Falsification*

Three experiments were conducted for each pool configuration: one control in which there was no falsification of timestamps, one where falsification was performed by changing the system clock and one where the Unix "`touch`" command was used to alter file timestamps. The falsifications were performed 5 hours into each experiment.

When the system clock method was used, the "`date`" command (IEEE, 2001) was used to set the system clock backward one hour. After ten seconds, the clock was then reverted to the correct time. All files written during this ten second period have an effectively falsified timestamp.

The touch method used the command "`touch -A-010000 <file>`" (IEEE, 2001) to alter the access and modification time of a file by one hour earlier than the real time. Only files which

Table 4.3: Test System Hardware and Software

CPU	Intel Pentium 4 3.00 GHz, 2 Cores
Memory	1 GB Real, 2GB Swap
Device Size	15 GB
Operating System	FreeBSD 9.1-RELEASE
ZFS Pool Version	28
ZFS Filesystem Version	5

would not be reopened later in the simulation were used to ensure that the false timestamp did not get changed to a true timestamp.

4.1.5 *Other Parameters*

Each experiment involved running the simulator for 24 hours. The 24 hour period was chosen primarily due to the hardware limits - it was discovered during pilot experiments that the systems would run out of memory after approximately one day of running the disk activity simulator.

The hardware and software configuration of the test system is shown in Table 4.3. Most significantly, the test systems used the FreeBSD 9.1 operating system(FreeBSD, 2014) with ZFS Pool version 28 and ZFS Filesystem version 5. These were the latest stable releases as of April 2013 when this project was started. ZFS Pool version 28 is also the last version which was used in the OpenSolaris system and is thus the ancestor of both the OpenZFS open source and Oracle closed source versions of ZFS.

Table 4.4: Data Collected during experiments

Metadata	Source	Command
All 128 Uberblocks	All devices in the pool	`zdb -P -uuu -l /dev/<device>`
All Spacemaps	All Metaslabs in each virtual device in the pool	`zdb -P -mmmmmm <pool>`
All objects and Block Pointers	The filesystem dataset with simulated activity	`zdb -P -dddddd -bbbbbb <poolname>/<dataset-name>`

All operating system files and saved data was stored on a separate device with a different file system (UFS2, (McKusick et al., 2014)).

4.1.6 *Data Collection*

Data was collected using a Bourne Shell (IEEE, 2001) script; the script is provided in Appendix section C.3.

Internal data structures were collected every 30 minutes as well as before and after all experiments, using the ZDB (ZFS Debugger) command - see section 2.3.1 for details of ZDB. Three types of metadata were collected, as shown in Table 4.4.

4.1.7 *Full Procedure for each Experiment*

The step by step procedure for undertaking each experiment was as follows:

4.1.7.1 *Naming Scheme*

Each experiment has an identifying name which is used for the name of the pool used to conduct the experiment and the directory used to hold the data.

The pool name is of the form "`poolvXrY`" where X is the number of virtual devices in the pool and Y is the pool mirror or RAID-Z configuration (0 for none, 1/2/3 for RAID-Z, or m for mirrored configurations).

The directory holding data for each experiment is of the form "<poolname>-<tampering>-<suffix>" where "<tampering>" is the type of timestamp tampering used - "sysclock", "touch", or "control" for the control experiments where no tampering was used. A numeric suffix is appended to any repeat experiments which use the same pool configuration and tampering method so the directory name is unique.

4.1.7.2 *Preparation*

(1) Create a new directory to contain data for the new experiment.

(2) Copy data-collection script and file activity simulator script to new directory.

(3) Edit the new copy of the data-collection script to include parameters of the new experiment.

- Parameters are the pool name, data directory, and the first and last devices in the pool.

(4) Prepare Cron (IEEE, 2001) to run the new data-collection script every 30 minutes by editing the "/etc/crontab" file.

- The crontab entry should remain commented out, to be enabled later when the experiment begins.

(5) Destroy any existing pool (with the command "`zpool destroy <poolname>`") and create a new pool with the correct name and configuration.

- For example, a RAID-Z1 pool using devices /dev/ada3, /dev/ada4 and /dev/ada5 would be created with the command "`zpool create poolv3r1 raidz1 /dev/ada0[3-5]`"

(6) Run "`zpool status`" to confirm that the pool configuration is correct.

(7) Create a new filesystem dataset called "filesim" within the pool

- For example, "`zfs create poolv3r1/filesim`".

(8) Run the command "`zdb -C -P <poolname> > zdb-C-start`" to save the pool configuration for later reference.

4.1.7.3 *Starting Simulation*

(1) Run the data-collection script manually to obtain the initial state.

(2) Uncomment the data-collection script in the crontab so it will run each 30 minutes.

(3) Run '`date >> starttime`' to record the time the experiment was started.

(4) Run '`python file-activity-sim.py /poolNAME/filesim | tee -a log`' to start the simulator and log status.

(5) A additional crontab entry should be added to terminate the simulator after 24 hours to prevent an out of memory situation which may corrupt data.

4.1.7.4 *Timestamp Falsification*

During non-control experiments, falsification of timestamps was performed five hours into the simulation.

SYSTEM CLOCK METHOD

(1) Record the current, true date and time.

(2) Use the "date <hour><minute>" command to set the system clock to one hour earlier than the true time.

(3) After ten seconds, use the date command to revert the system clock back to the correct time.

TOUCH METHOD

(1) Locate a file which will last until the end of the simulation and will not be later modified during the simulation (as this would overwrite any falsified timestamp with a later, genuine one).

(2) Record the file name and true modification time, and the current date and time.

(3) Use the touch command "`touch -A-010000 <file>`" to revert the file modification and access times by one hour.

(4) Record the new fake modification time and the touch command used to alter it.

4.1.7.5 *Concluding Simulation*

(1) Stop the file activity simulator script if still running.

(2) Run '`date >> endtime`' to save the time the experiment ended.

(3) Comment out the /etc/crontab data-gathering script so it is no longer executed automatically

(4) If the data-gathering script is still running, wait until it has completed before continuing.

(5) Run a final data-collection script manually to save the final state of the experiment.

4.1.8 *Repeats and Verification*

There are 66 possible combinations of pool configuration and timestamp falsification method (including control). Additional experiments were repeated at random as time permitted to verify the results of the initial set; the result of testing the algorithms on all repeat experiments was the same as the matching experiment in the initial set, affected only by the falsification method and Uberblock period (see section 5.1).

4.2 TIMESTAMP FALSIFICATION DETECTION METHODS

After each simulation, the algorithms described in this section were used on the metadata obtained from the simulations to attempt to detect the falsified timestamps. The algorithms were also used on the data from the control experiments to see if any false positives occurred.

Results and analysis of the effectiveness of the algorithms and the utility of the ZFS metadata elements for timeline analysis is in presented in section 5.1.

The timeline analysis was performed by manual examination of structures and using the Python (Python, 2014) and R (Ihaka and Gentleman, 1996) software. The script in Appendix C.4 was used to convert some ZDB data into XML in order to import it into R.

Each simulation was also examined manually (as for a case study) to observe any unexpected behaviour in the saved ZFS structures.

4.2.1 *Uberblock TXG and Timestamp*

As mentioned by Cifuentes and Cano (2012), Uberblocks (see section 2.3.2 for details) contain the time and transaction group ID (TXG) when they were written, and thus a way to link the two.

This can be used to verify file timestamps and detect false ones. Where a file is modified with the touch command, the timestamp on the file will not match the timestamp in the Uberblock with the same transaction group (TXG) as the file's highest level Block Pointer birth TXG (see

Algorithm 4.1 False last modification timestamp detection using Uberblocks

```
targetTXG = file.topLevelBP.birthTXG
maximumTime = file.modifyTime + 5 seconds
for each Uberblock:
    if (Uberblock.TXG = targetTXG):
        if (Uberblock.timestamp > maximumTime):
            file.modifyTime is inconsistent
```

Algorithm 4.2 System clock change detection using Uberblocks

```
lastTimestamp = 0
for each Uberblock sort by TXG:
    if (Uberblock.timestamp < lastTimestamp):
        Uberblock.timestamp is inconsistent
    lastTimestamp = Uberblock.timestamp
```

section 2.3.4.4). A tolerance of 5 seconds should be applied to the timestamp match as TXG are flushed every 5 seconds (see section 2.2.4 for details).

Algorithm 4.1 uses this technique to detect false modification times.

The Uberblock array may also be used to detect system clock changes by comparing adjacent timestamps. If Uberblocks are ordered by increasing TXG, the timestamps should also be in increasing order (typically every 5 seconds under a continuous write load). An anomaly in this order may be caused by a clock correction or deliberate tampering with the system clock to alter timestamps. Algorithm 4.2 shows how an out of order timestamp may be detected.

Section 5.1.4 discusses the effectiveness of both algorithms using Uberblocks to detect forged timestamps.

Algorithm 4.3 False last modification timestamp detection using Block Pointer Birth TXG

```
pairs = listOfPairs(TXG, modifyTime)
for each fileObject:
   pairs.append(fileObject.topLevelBP.birthTXG,
fileObject.modifyTime)
for each pair in pairs sort by TXG:
   if (pair.modifyTime < lastTime - 5 seconds):
      pair.modifyTime is inconsistent
   lastTime = pair.modifyTime
```

4.2.2 *Block Pointer Birth TXG ID*

The highest level Block Pointer Birth TXG obtained from a file object (see section 2.3.4.4) can be compared with the birth TXG of other files to detect potential false timestamps.

Assuming that each transaction group is flushed to disk after at most 5 seconds (see section 2.2.4), two files which have the same highest level TXG should have at most 5 seconds difference in modification timestamps. As TXG numbers increase over time, files with later TXG should also have later timestamps, and vice versa. Algorithm 4.3 shows how the birth TXG may be compared to detect false timestamps.

The effectiveness of this method as determined from the experiments is discussed in section 5.1.1.

Algorithm 4.4 False creation timestamp detection using Object Number

```
pairs = listOfPairs(objectNumber, creationTime)
for each fileObject:
   pairs.append(fileObject.objectNumber, fileobject.creationTime)
lastCRTime = 0
for each pair in pairs sort by objectNumber:
   if (pair.creationTime < lastCRTime):
      pair.creationTime is false
   lastCRTime = pair.creationTime
```

4.2.3 *Object Number*

Object numbers (see section 2.3.4 for background information) are allocated to new objects (including files and directories) in locally increasing order. Thus they are related to the order of creation of objects, and can possibly be used to detect objects with a falsified creation time.

As noted in section 2.3.4, object numbers may be reused when ZFS detects a large range of unused numbers (e.g. when many files are deleted). This may cause false positives where the object numbers change; this issue is discussed further in section 5.1.3.

Algorithm 4.4 shows how object numbers may be used to detect false creation timestamps. As two objects may be created at the same time, file objects should be sorted by the object number first and then checked for out of order creation times.

Section 5.1.3 discusses the effectiveness of using Object Numbers to detect forged timestamps.

Algorithm 4.5 False creation timestamp detection using Object Generation TXG ID

```
pairs = listOfPairs(genTXG, creationTime)
for each fileObject:
   pairs.append(fileObject.genTXG, fileObject.creationTime)
lastTXG = 0
for each pair in pairs sort by timestamp:
   if (pair.genTXG < lastTXG):
      timestamp is inconsistent
   lastTXG = pair.genTXG
```

4.2.4 *Object Generation TXG ID*

As mentioned in section 2.3.4, all file objects store the transaction group ID in which they were created, or "generated" (this is the "gen" attribute in Figure 2.6 on page 29).

Similar to Object numbers, the object Generation TXG increases with the order of creation of objects and can thus be used to detect objects with falsified creation time. Unlike Object numbers, they are never reused, and the order of Generation TXG should be consistent with the true order in which objects are created.

Algorithm 4.5 shows how the Generation TXG can be used to detect out of order creation timestamps. As a TXG may commit transactions from up to 5 seconds ago, the timestamp/gen TXG pairs are first sorted by timestamp, then processed to check that all TXG are in order.

The effectiveness of this method as determined from the experiments is discussed in section 5.1.3.

4.2.5 *Spacemap TXG ID*

Each Spacemap segment (see section 2.3.5) lists the TXG when the Spacemap itself was written. However, this is likely to be a more recent TXG than the one in which the space was allocated, because spacemaps are "condensed" and rewritten frequently. Therefore, presence of a Block Pointer's allocated space in a segment with a later TXG than the file's Block Pointer TXGs is not evidence of a forged timestamp.

To determine if a timestamp is false using Spacemaps, the Spacemap which references the block which is allocated to the file must be located. The transaction group ID of the Spacemap must be checked against the file's highest level Block Pointer birth TXG ID to see if the Spacemap has been condensed since the file modification; if so, it cannot be used for verification of the file timestamp.

The position of the Spacemap can then be compared to the objects which were (apparently) modified at the same time - if they are all allocated at the same time and in the same Spacemap then it can be assumed that they were written at the same time as well. As ZFS is a copy-on-write filesystem and data is never overwritten in place, the block must have been written when it was allocated. Section 5.1.5 discusses the effectiveness of this method.

4.2.6 *Vdev Number*

ZFS attempts to write 512KB to each device before moving to the next (see section 2.2.5 for details). Therefore there is a relationship between the device number in the DVA (see section 2.3.3.1) of a Block Pointer and the time it was allocated and written.

To use the Vdev number in DVA to determine if timestamps are out of order, objects must first be sorted by modification timestamp, and 512KB batches of blocks must be matched to Vdev numbers in order of modification time - as the DVA contains the size of the block, this operation can be performed using the DVA and modification time only.

This technique can only be used in pools where there is more than one virtual device in the pool, otherwise all Vdev numbers will be the same. The effectiveness of this technique is discussed in section 5.1.5.

4.3 LARGE FILE EXPERIMENT METHODOLOGY

This section describes the method used in the follow-up “large file” experiments. Results and analysis from these experiments is presented in section 5.1.2.

4.3.1 *Rationale*

During the primary simulation experiments, it was observed that the indirect Block Pointers (see section 2.3.3.2) holding the data for the dataset's root directory object contained different birth TXG values (as only some of the directory blocks were modified in later TXG).

As the simulator rewrote file data all at once, instead of modifying only some parts, this effect was not observed in the file objects.

A new experiment was designed to reproduce the partially modified behaviour on a file object, and determine if the birth TXGs in a multiple-block file could be used to determine the time that the previously modified blocks were written.

4.3.2 *Methodology*

A set of simulations was performed following the same procedures as for the primary experiments (described in previous sections in this chapter), with the following differences:

(1) The simulator is modified to create a "large file" at the start of the simulation. This file is appended to each pass with a random amount of data, using the same write size distribution as the other writes in the simulation (see Table 4.1). The code for the "large file" simulator is in Appendix section C.2.

(2) The simulation is run for ten minutes, with data collection only before and after (not during the experiment). The same data collection script was used. The ten minute limit

was chosen primarily for convenience; a long simulation is not necessary to demonstrate the viability of this method.

(3) No falsification of timestamps was used, and no falsification detection methods were tested. Instead, the timeline analysis method described below is used to attempt to determine past modification times of the blocks in the "large file".

One simulation was conducted for each pool configuration, for a total of 22 experiments.

4.3.2.1 *Timeline Analysis*

Rather than aiming to detect false timestamps, the analysis of the metadata from this experiment is intended to determine the time of previous modifications of the "large file".

This is performed by matching the Birth TXG of the level 0 Block Pointers in the large file to the birth TXG of the highest level Block Pointers in other files. If they can be matched, the block in the large file should have been written within 5 seconds of the listed modification time of the other file.

For example, Figure 2.5 on page 27 shows simplified ZDB output of a large file with two data blocks written during different transactions. The block pointed to by the first level 0 BP was written during TXG 25, whereas the second level 0 BP was written in TXG 29 (requiring the parent level 1 BP to be also rewritten).

If there is another file (or other object) in the pool which can link a timestamp to TXG 25, it can be used to determine the last time that the first block was written.

Algorithm 4.6 Past Modification time Detection using Block Pointer Birth TXG

```
largeFilePairs = listOfPairs(TXG, timestamp)
for each blockPointer in largeFile:
    if (blockPointer.level == 0):
        largeFilePairs.append(blockPointer.birthTXG, NULL)

otherFilePairs = listOfPairs(TXG, timestamp)
for each fileObject:
    otherFilePairs.append(fileObject.topLevelBP.birthTXG,
fileObject.modifyTime)

for each lfpair in largeFilePairs:
    for each ofpair in otherFilePairs:
        if (lfpair.TXG == ofpair.TXG):
            lfpair.timestamp = ofpair.timestamp ±5 seconds
```

Note that this technique can only discover the most recent write to each block; either block may have been rewritten multiple times. In this particular example the Generation TXG of 25 also indicates the first block was last modified in the same transaction group as the file was created, probably at the same time.

Algorithm 4.6 shows this procedure in detail. A script implementing this analysis is in Appendix section C.5.

5

RESULTS AND ANALYSIS

This chapter is divided into two sections. The first section contains the results of all experiments and evaluates the effectiveness of the new ZFS timeline analysis methods. In section 5.2, other factors which may affect the use of ZFS structures for timeline analysis are discussed.

In total, 85 primary simulation experiments were conducted (not including the 22 "large file" experiments) and over 110 gigabytes of ZFS metadata was collected. Additional data from the experiments which is not analysed in this chapter is provided in Appendix B.

5.1 TIMELINE ANALYSIS METHODS

This section discusses the effectiveness of the six timestamp falsification detection methods presented in section 4.2 and the past modification time detection method described in section 4.3.

The results of applying the Falsification Detection methods to the simulated file activity are shown in Table 5.1 (control experiments) and Table 5.2 (non-control experiments).

"True Positive" refers to a method detecting a genuine falsification, and "True Negative" refers to no falsifications incorrectly detected when there were none (i.e. on control experiments). "False Negative" refers to experiments where a falsification occurred but was not detected.

"False Positive" indicates detection of an out-of-order timestamp which was not actual tampering. This only occurred in two experiments and in both cases the experiment also involved a genuine falsification which was successfully detected by multiple algorithms. Section 5.2.1 discusses this in more detail.

5.1.1 *Comparing File Data Block Pointer Birth TXG*

Comparison of the Birth TXG from the highest level Block Pointers between files (the method described in section 4.2.2) was successful in detecting tampering in all non-control experiments, as shown in the third row of Table 5.2.

This includes both methods of tampering; Algorithm 4.3 is effective at detecting all false modification timestamps from the touch method as well as some false timestamps from each experiment from the system clock method. Although some files affected by the system clock method were later modified (causing the highest level Block Pointer to be rewritten with a new TXG) there was sufficient disruption to determine that tampering had occurred in all cases.

Table 5.1: Falsification Detection Results for Control Experiments

Detection Method	Control (no Tampering)	
	True Negative	False Positive*
Uberblock TXG (within 30 min.)	32	0
Uberblock TXG (after 24 hours)	32	0
Block Pointer TXG	32	0
Generation TXG	32	0
Object Number	32	0
Spacemaps	32	0
Vdev Number	32	0
(Number of Experiments)	32	

* Two false positives occurred during non-control experiments in which a genuine falsification was successfully detected. This is discussed further in section 5.2.1.

Table 5.2: Falsification Detection Results by Falsification Method

Detection Method	System Clock Tampering		Touch Command Tampering	
	True Positive	False Negative	True Positive	False Negative
Uberblock TXG (within 30 min.)*	15	11	11	16
Uberblock TXG (after 24 hours)	0	26	0	27
Block Pointer TXG	26	0	27	0
Generation TXG	26	0	0	27
Object Number	26	0	0	27
Spacemaps	0	26	0	27
Vdev Number	0	26	0	27
(Number of Experiments)	26		27	

* The Uberblock detection method was only successful when the data was collected before the relevant Uberblock was overwritten, typically within 10.6 minutes. See section 5.1.4 for further details.

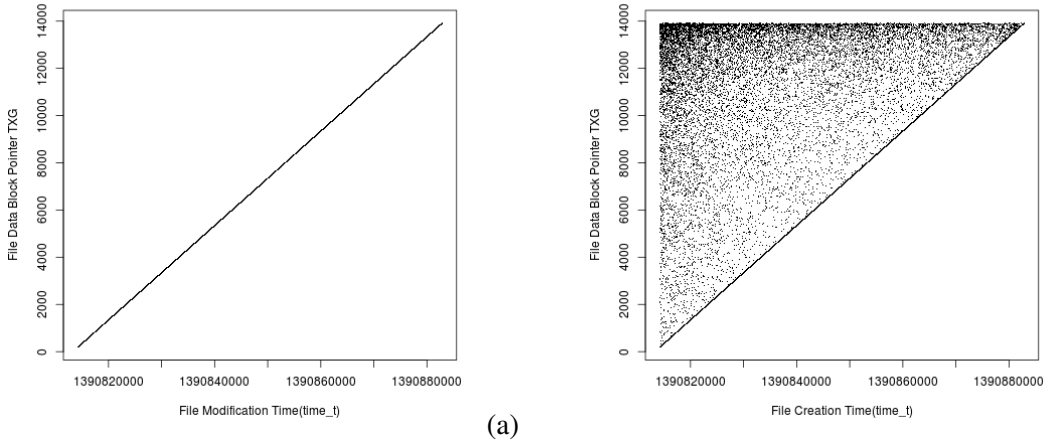


Figure 5.1: Modification Time (a) and Creation Time (b) of files vs the Birth TXG of their highest level Block Pointer. No tampering was involved in this experiment.

Figure 5.1 shows plots of file modification time (a) and creation time (b) vs the birth TXG of the highest level Block Pointer in the file, from an experiment with no tampering. It can be seen in (a) that there is a linear relationship between modification time and birth TXG, and a false modification time disrupts this. If the system clock is changed, it affects both timestamps, and as long as there is constant file activity the typical patterns of file activity visible in both (a) and (b) will be disrupted, which can be detected by Algorithm 4.6.

It is notable that the highest level Block Pointer birth TXG is also critical to the Uberblock method (as described in section 4.2.1), as this value is used to obtain the TXG in which the file was modified. Combined with its use for obtaining block modification times (discussed below), this makes Birth TXG the most useful element of ZFS metadata for timeline analysis.

5.1.1.1 *Example with Forged Modification Time*

Figure 5.2 shows an example of ZDB output where the Block Pointer Birth TXG may be used to detect a forged timestamp. Some fields have been removed for clarity.

The first object (number 15417) has a false modification time, forged by using the touch command to revert the time backwards one hour.

The second and third objects have the same birth TXG (1737) as the first object, but the true modification time. Thus it can be assumed that the first object was modified at the same time as the following two, and it has a false timestamp.

The last object (number 9301) is from the transaction group before the first object, but has a modification time one hour later. This further demonstrates that the first object has a false modification time.

5.1.2 *Block Modification Times from Birth TXG*

Table 5.3 shows how many of the large file Block Pointer Birth TXG IDs in the "large file" could be linked to another file with the same TXG. Across all of the "large file" experiments, over 99% of the Level 0 Block Pointer Birth TXG in the large file could be matched to the highest level Birth TXG of other files, allowing the modification times of those individual blocks in the large file to be determined using the algorithm described in section 4.3.2.1.

Even in the few cases where an exact match to the TXG ID cannot be obtained, it is possible to determine a range of time where the modification occurred by locating the nearest matches be-

```
    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     15417    1  16384  67072  67072  67072  100.00  ZFS plain file
                                      168   bonus  System attributes
        path    /1382428539-to-2147483647-reps-0
        mtime   Tue Oct 22 17:55:40 2013
        gen     1737
Indirect blocks:
               0 L0 DVA[0]=<2:353c200:10600> [L0 ZFS plain file]
               ... birth=1737L/1737P fill=1 cksum=...


    Object  lvl   iblk   dblk  dsize  lsize  %full  type
      8523    1  16384  15872  15872  15872  100.00  ZFS plain file
                                      168   bonus  System attributes
        path    /1382423926-to-1383825949-reps-10
        mtime   Tue Oct 22 18:55:43 2013
        gen     814
Indirect blocks:
               0 L0 DVA[0]=<0:3deba00:3e00> [L0 ZFS plain file]
               ... birth=1737L/1737P fill=1 cksum=...

    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     15418    1  16384   5120   5120   5120  100.00  ZFS plain file
                                      168   bonus  System attributes
        path    /1382428540-to-2147483647-reps-0
        mtime   Tue Oct 22 18:55:40 2013
        gen     1737
Indirect blocks:
               0 L0 DVA[0]=<2:3529c00:1400> [L0 ZFS plain file]
              ... birth=1737L/1737P fill=1 cksum=...


    Object  lvl   iblk   dblk  dsize  lsize  %full  type
      9301    1  16384  34304  34304  34304  100.00  ZFS plain file
                                      168   bonus  System attributes
        path    /1382424391-to-1382504521-reps-375
        mtime   Tue Oct 22 18:55:37 2013
        gen     907
Indirect blocks:
               0 L0 DVA[0]=<1:3f2d400:8600> [L0 ZFS plain file]
              ... birth=1736L/1736P fill=1 cksum=...
```

Figure 5.2: Detecting False Modification Time using Block Pointer TXG

Table 5.3: Results of Large File Experiments

Linkable TXG from Large File	Number of Experiments	Mean Percent of Linked TXG
None	0	n/a
Some	5	96.439%
All	17	100%
Total	22	99.191%

fore and after the TXG in the Block Pointer. Spencer (2014) discusses the use of such "relative time" events in timeline analysis and how they can be used.

In 77% of the experiments every single level 0 Block Pointer in the large file could be so linked. In the remaining cases, all but one (on average, 96.4%) of the TXG could be linked. This implies that although this technique only applies to large files with multiple blocks, if they are present they can be easily matched to other files, assuming files are written to the pool at least every 5 seconds as in the corporate network storage data used (Agrawal et al., 2007; Roselli et al., 2000; Leung et al., 2008). Leung et al. (2008) also implies that many files will remain indefinitely (55.2% of files are never deleted) therefore it is likely that a corporate ZFS pool in constant use will contain many other files which can be use to match past TXGs in a large file.

This technique may be less effective on ZFS pools in which fewer files are written and/or are written at a intermittent rate (e.g. from desktop or laptop computers), as there may not be as many files written in the same TXG.

This technique is most effective for large files which are modified only in small sections at a time (e.g. virtual machine images). In the primary experiments only 109 per million files required more than one Block Pointer for their data (see Table 5.4 for details). The median file size in the experiments was 4KB (Agrawal et al., 2007; Leung et al., 2008); with default settings ZFS uses blocks up to 128KB in size thus very few files required more than one block. On other systems with specialised workloads such as video capture, database storage or virtual machine storage this technique may be more useful.

Table 5.4: Average File and Object Counts for Simulation Experiments.

Count	Mean		Median		Standard Deviation	
	Raw	Norm.*	Raw	Norm.*	Raw	Norm.*
Total Files remaining on Disk	46691	48215	46026	48086	6899.9	5515.3
File Objects with >1 Data BP	5.1412	5.2649	5	5.1724	2.5502	2.4704
File Objects with >2 Data BP	0.63157	0.65944	0.5	0.43956	0.70884	0.74020

* Normalised per 24 hours.

The birth TXG from a multiple-block file could also be matched to the Generation TXG and file creation timestamp from other files, as the Generation TXG is the transaction in which the object was created, although this was not tested in the experiments.

5.1.3 *Object ID and Generation TXG*

In all experiments involving a falsified creation time (i.e. all experiments where tampering was performed by changing the system clock), both the Object Number and the Generation TXG were out of order and were successfully used to detect the falsification. However, in all experiments where the touch command was used to alter a file modification time, the Object Number and the Generation TXG were not effective in detecting the falsification. The 4th and 5th rows of Table 5.2 show the results of these methods.

Figure 5.3 shows plots of Object number (a) and Generation TXG (b) vs file creation time from an experiment where the system clock was adjusted one hour backwards to alter file timestamps; the outliers above the line are from files with false creation times. Although covering the same three hours the Generation TXG graph rises evenly over time as TXG are flushed

to disk at a constant rate, whereas file objects (and Object Numbers) are created at a variable rate.

Object Numbers and Generation TXG therefore provide a simple and effective method to detect false file creation times. However, as they are related to object creation, they are not affected by modifications to the file and cannot be used to detect falsified modification time, unless the modification time is altered to before the file was created. They cannot by themselves provide any extra event information to forensic investigation (beyond verifying the file creation time).

5.1.3.1 *Reuse of Object Numbers*

Object numbers are allocated incrementally but may be reused when ZFS detects a large range of unused numbers (e.g. when many files are deleted). This did not occur during the experiments and further study is required to determine how often numbers are reused during production conditions and how this would affect forensic use of object numbers.

As Object Numbers may be reused they should not be used for practical forensic purposes until they have been studied further. However, Generation TXG does not have this issue and is suitable for practical detection of the true creation order of ZFS file objects.

5.1.3.2 *Example of Forged Creation Time detection*

Figure 5.4 shows how Object Number and Generation TXG may be used to detect false file creation times. Many fields and objects have been omitted for clarity, with extra line breaks inserted to show where the false timestamps begin and end. This data is from an experiment

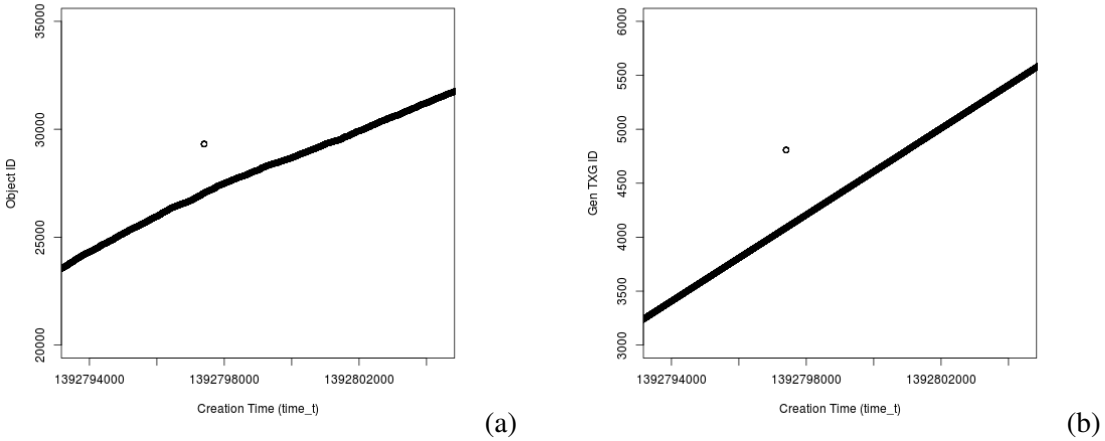


Figure 5.3: Object number (a) and generation TXG (b) vs. creation time, from an experiment with false creation times. Time values are “time_t” (seconds since January 1 1970).

where the system clock was used to falsify file timestamps; the same experiment which was used for the plots in Figure 5.3.

The first three objects (numbers 29302, 29304 and 29305) were written before the system clock was altered; the time was 20:09 and is accurately reflected in the creation timestamps of these files.

The system clock was then altered to one hour earlier to falsify the file timestamps. The next four objects (and some which were omitted for clarity, replaced with the ellipsis) have a false creation time of 19:10. The last three objects (numbers 29328, 29329 and 29330) were written after the clock had been reverted to the correct time of 20:10.

Although the listed creation time changes with the system clock, both the Generation TXG and the Object Numbers increase normally. This demonstrates that the file objects 29307, 29308, 29326 and 29327 were actually created between the first three and last three objects, although their creation timestamp is one hour earlier.

```
    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     29302    1  16384  10752  10752  10752  100.00  ZFS plain file
        crtime  Wed Feb 19 20:09:49 2014
        gen     4805

    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     29304    1  16384  10752  10752  10752  100.00  ZFS plain file
        crtime  Wed Feb 19 20:09:55 2014
        gen     4806

    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     29305    1  16384   3584   3584   3584  100.00  ZFS plain file
        crtime  Wed Feb 19 20:09:55 2014
        gen     4806



    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     29307    1  16384  30720  30720  30720  100.00  ZFS plain file
        crtime  Wed Feb 19 19:10:00 2014
        gen     4807

    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     29308    1  16384   2560   2560   2560  100.00  ZFS plain file
        crtime  Wed Feb 19 19:10:01 2014
        gen     4808

...

    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     29326    1  16384   8704   8704   8704  100.00  ZFS plain file
        crtime  Wed Feb 19 19:10:06 2014
        gen     4809

    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     29327    1  16384  11264  11264  11264  100.00  ZFS plain file
        crtime  Wed Feb 19 19:10:06 2014
        gen     4809



    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     29328    1  16384   4096   4096   4096  100.00  ZFS plain file
        crtime  Wed Feb 19 20:10:08 2014
        gen     4810

    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     29329    1  16384   4608   4608   4608  100.00  ZFS plain file
        crtime  Wed Feb 19 20:10:08 2014
        gen     4810

    Object  lvl   iblk   dblk  dsize  lsize  %full  type
     29330    1  16384   3584   3584   3584  100.00  ZFS plain file
        crtime  Wed Feb 19 20:10:12 2014
        gen     4811
```

Figure 5.4: Detecting False Creation Time using Object Number and Generation TXG
Many fields and objects have been omitted for clarity, with extra line breaks inserted to show where the false timestamps begin and end.

5.1.4 *Uberblocks*

As shown in the first two rows of Table 5.2, in 49% of all experiments involving tampering the Uberblocks were successfully used (method described in section 4.2.1) to detect a falsified file timestamp. However, this was only possible because data was collected every 30 minutes during the experiments; at the end of the simulation the relevant Uberblocks were already overwritten. Thus Uberblocks were never effective at detecting falsification 19 hours after the falsification occurred.

It was found that in most pool configurations, Uberblocks only last 10.6 minutes under a continuous writing load before they are overwritten. This is due to ZFS typically writing a new Uberblock every 5 seconds (there are 128 Uberblocks in the array, $5 \times 128 = 640$ seconds). In 73% of experiments, the oldest Uberblock in the array of 128 was between 634 and 636 seconds old (inclusive).

Due to the short lifespan of Uberblocks they are most useful in a "dawn raid" scenario, where a system is seized immediately after a suspect may have been deleting or modifying files, or when a server is shutdown within 10 minutes of an intrusion being detected. In this case they could also be used to access previous versions of the object tree and determine the most recent changes in the pool. This use of Uberblocks could be extremely useful in such scenarios and needs further investigation.

Another disadvantage of Uberblocks is that they are the simplest internal structure to tamper with as they are at the top of the ZFS checksum tree; if modified the forger only has to change

the Uberblock's internal checksum to match any tampering. Section 6.3.5 discusses this issue in more detail.

5.1.4.1 *Effect of Pool Configuration on Uberblock Period*

It was observed that there are three different patterns in the distribution of the period between most and least recent Uberblocks, corresponding to different groups of pool configurations: those with 3 or less top level virtual devices, those with 4 or more and no redundancy, and those with 4 or more using a mirror or RAID-Z configuration. The differences are shown in Figure 5.5 and Table 5.5.

This effect influences the detection rate of the Uberblock algorithms; Table 5.6 shows the number and rate of forged timestamps detected via Uberblocks, compared with the pool configuration. Using Uberblocks to detect false timestamps was more likely to be successful in the pools with 4 or more top level virtual devices, especially those with a flat configuration.

The overall median period from earliest to latest Uberblock at the end of each experiment was 636 seconds, with a mean of 1707.8 seconds and standard deviation of 2282.3 seconds.

Pools with 4 or more top level virtual devices (i.e. 4 or more disks and a "flat" pool with no redundancy) consistently had a significantly longer period between the oldest and newest Uberblock in the pool, with a median time of 4129 seconds (68.8 minutes). With more than three devices, some writes may not affect all devices, so a new Uberblock will not always be written to all of them for each TXG. Pools using RAID-Z or mirroring are less affected as writes are likely to affect all devices in a more redundant pool.

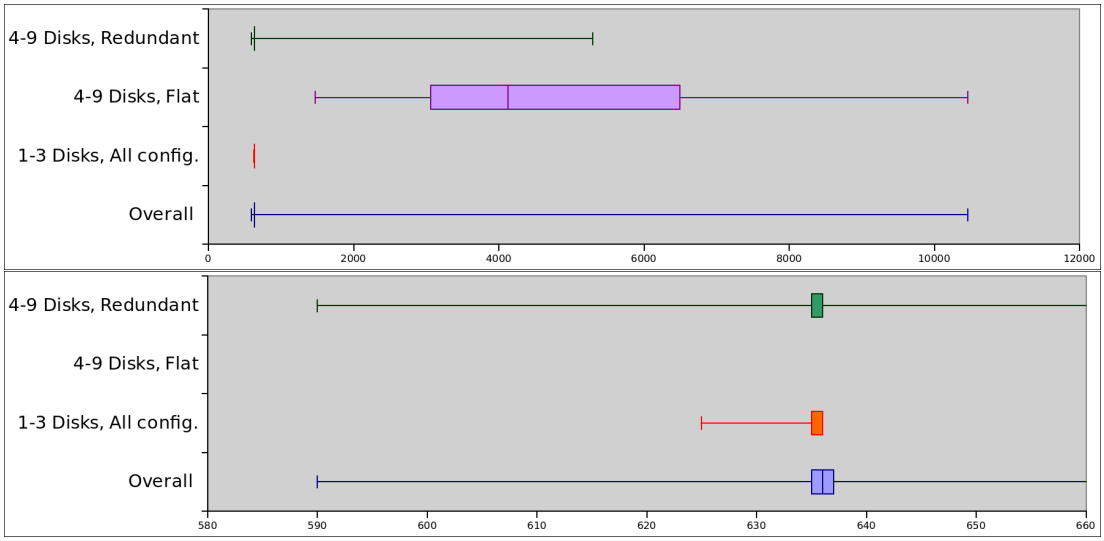


Figure 5.5: Period of Uberblocks for different pool configurations
All values are in seconds from earliest to latest Uberblock. Whiskers indicate the maximum and minimum values recorded. The lower graph is an expansion of the range 580 to 660 seconds for detail.

Table 5.5: Period of Uberblocks for different pool configurations

Pool Configuration	Experiments in Group*	Lowest	Lower Quartile	Median	Upper Quartile	Highest
4-9 Disks, Redundant	42	590	635	635	636	5296
4-9 Disks, Flat	18	1475	3062	4129	6492	10456
1-3 Disks, Flat or Redundant	25	625	635	635	636	636
All Pools	85	590	635	636	638	10456

All values are in seconds from earliest to latest Uberblock.
*These groups are distributed unevenly as the different behaviour of the groups was observed after the experiments, not prearranged.

Table 5.6: Uberblock Detection Rate by Pool Configuration (within 30 minutes of tampering)

Pool Configuration	Forgeries Detected via Uberblocks	Total Forgeries with Configuration*	Detection Rate
4-9 Disks, Redundant	16	28	57%
4-9 Disks, Flat	7	10	70%
1-3 Disks, Flat or Redundant	4	15	27%

*These groups are distributed unevenly as the different behaviour of the groups was observed after the experiments, not prearranged.

With three or less devices, the three redundant copies of pool metadata blocks will be spread across all devices in the pool, thus every transaction group committed will cause at least one write to all devices. This is reflected in the results, as all pools containing three or less disks had an Uberblock period 636 seconds or lower.

It should be emphasised that the longest period was still only 10456 seconds (2.90 hours); regardless of the pool configuration, detection via Uberblocks was only successful using the data which was collected every 30 minutes during the experiments. With the data obtained at the end of the experiments (19 hours after the tampering occurred), no false timestamps could be detected using Uberblocks.

5.1.4.2 *Example of System Clock Tampering Detection using Uberblocks*

Figure 5.6 shows a listing of Uberblocks from an experiment where the system clock was adjusted to alter file timestamps. The tampering occurred at 17:56 on December 18 2013; the Uberblocks were collected at 18:00. Many fields and Uberblocks which are not relevant have been removed for clarity.

The active Uberblock (with highest TXG) is in slot 11. It can be seen that the Uberblock array "wraps around" between 11 and 12; slot 12 has the lowest TXG and will be overwritten when the next Transaction Group is committed (see section 2.2.4).

The only other anomalies in the timestamps are between slots 86-91. In each sequential slot the TXG values are incrementing, but the timestamp goes back one hour between slots 86 and 87 then forward one hour between slots 90 and 91. This is where the clock has been manually set backwards one hour in order to falsify file timestamps.

```
Uberblock[7]
        txg = 3975
        timestamp = 1387349982 UTC = Wed Dec 18 17:59:42 2013
Uberblock[8]
        txg = 3976
        timestamp = 1387349987 UTC = Wed Dec 18 17:59:47 2013
Uberblock[9]
        txg = 3977
        timestamp = 1387349992 UTC = Wed Dec 18 17:59:52 2013
Uberblock[10]
        txg = 3978
        timestamp = 1387349996 UTC = Wed Dec 18 17:59:56 2013
Uberblock[11]
        txg = 3979
        timestamp = 1387350001 UTC = Wed Dec 18 18:00:01 2013
Uberblock[12]
        txg = 3852
        timestamp = 1387349395 UTC = Wed Dec 18 17:49:55 2013
Uberblock[13]
        txg = 3853
        timestamp = 1387349400 UTC = Wed Dec 18 17:50:00 2013
Uberblock[14]
        txg = 3854
        timestamp = 1387349405 UTC = Wed Dec 18 17:50:05 2013
Uberblock[15]
        txg = 3855
        timestamp = 1387349410 UTC = Wed Dec 18 17:50:10 2013
...
Uberblock[83]
        txg = 3923
        timestamp = 1387349750 UTC = Wed Dec 18 17:55:50 2013
Uberblock[84]
        txg = 3924
        timestamp = 1387349755 UTC = Wed Dec 18 17:55:55 2013
Uberblock[85]
        txg = 3925
        timestamp = 1387349760 UTC = Wed Dec 18 17:56:00 2013
Uberblock[86]
        txg = 3926
        timestamp = 1387349765 UTC = Wed Dec 18 17:56:05 2013
Uberblock[87]
        txg = 3927
        timestamp = 1387346160 UTC = Wed Dec 18 16:56:00 2013
Uberblock[88]
        txg = 3928
        timestamp = 1387346166 UTC = Wed Dec 18 16:56:06 2013
Uberblock[89]
        txg = 3929
        timestamp = 1387346170 UTC = Wed Dec 18 16:56:10 2013
Uberblock[90]
        txg = 3930
        timestamp = 1387346175 UTC = Wed Dec 18 16:56:15 2013
Uberblock[91]
        txg = 3931
        timestamp = 1387349762 UTC = Wed Dec 18 17:56:02 2013
Uberblock[92]
        txg = 3932
        timestamp = 1387349767 UTC = Wed Dec 18 17:56:07 2013
Uberblock[93]
        txg = 3933
        timestamp = 1387349772 UTC = Wed Dec 18 17:56:12 2013
Uberblock[94]
        txg = 3934
        timestamp = 1387349777 UTC = Wed Dec 18 17:56:17 2013
...
```

Figure 5.6: Detecting System Clock Tampering using Uberblocks

5.1.5 *Spacemaps and Vdev number*

Analysis of Spacemaps (method described in section 4.2.5) and the Vdev number (method described in section 4.2.6) were both completely ineffective at detecting falsified timestamps, with no positive results in any experiments.

The Spacemaps written when the relevant blocks were allocated had been "condensed" (see section 2.3.5), and could not be used to link the allocation to a particular time. Condensation is more frequent than might expected due to the frequency of temporary files created and deleted within seconds (Agrawal et al., 2007; Leung et al., 2008) and the ZFS metaslab allocation algorithm preferring to fill metaslabs towards full capacity to reduce fragmentation (increasing the chance of the Spacemaps of recently used metaslabs being condensed).

Figure 5.7 shows the median distribution of of the TXG in which spacemaps were written, compared to the distribution of File Data Block Pointer TXG (i.e. the TXG in which file data was written). To produce this plot, the TXG values were normalised between 0 at the start of the experiment and 1 for the TXG at the end of the experiment. For each experiment a normalised five number summary (lowest value, lower quartile, median, upper quartile, highest value) was calculated. The values used to generate the box-plot are the median of the summary values of each experiment.

It can be seen that the Block Pointer TXG (which represent the actual time of file writes) are evenly distributed but slightly skewed towards the end of the simulation (median of 0.55180). The minor skew in visible file writes is caused by a small number of files which are modified

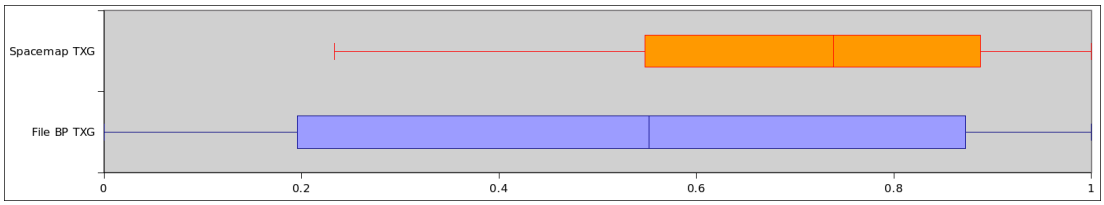


Figure 5.7: Normalised distribution of Spacemap TXG and Block Pointer TXG
Normalised over 0-1 between the first and last TXG of each experiment.

frequently (see section 4.1.2). However, Spacemap TXG are strongly skewed towards the end of the simulation (median of 0.73850), due to the frequent condensation - it can be seen that many Spacemaps were re-written in a later TXG than the corresponding file data.

Vdev number is also ineffective at detecting timestamps as the pattern of allocated data is broken up by the large quantity of file data and internal objects which were deleted or modified. Data and metadata is never overwritten in place on ZFS to ensure integrity (see section 2.2.4), so new copies of many objects are allocated and old versions deleted when each transaction group is flushed to disk.

Even if files were not modified or deleted (e.g. in an append-only or other special purpose system), transient internal objects (including spacemaps themselves) may have been modified or destroyed. Therefore the Spacemap and Vdev Number methods of forgery detection are unlikely to be effective even in special circumstances.

Spacemaps could be used to verify that a given block has not been modified since the Spacemap segment TXG (due to the copy-on-write system, blocks are never overwritten in place), although this is of limited forensic use.

5.2 FURTHER ANALYSIS

This section discusses some other factors observed during the experiments which affect timeline analysis of ZFS.

5.2.1 *Out-of-Order timestamps and False Positives*

Timestamps which occur out of order may be caused by normal, innocuous events as well as deliberate tampering and anti-forensics. In many cases a file may be *physically* written at a later date than its *logical* creation or modification timestamps. Care must be taken by forensic investigators to avoid incorrectly interpreting these events as deliberate forgery of timestamps, even if the timestamp is physically false.

The Object Number method is also vulnerable to physical false positives as object numbers may be reused (see section 5.2.1.3).

Future work is required to identify how frequently all forms of false positives occur in data from real ZFS systems. Surveys of ZDB data from production systems could be used to determine the false positive rate; areas of future research are discussed further in section 6.3.

5.2.1.1 *System Clock Corrections*

The system clock will usually be corrected on a regular basis (e.g. every night) which will affect all timestamps on the system. If the clock is running fast and is corrected backwards,

timestamps will appear out of order. This occurred in two of the 85 experiments - an overnight clock correction caused object number, Generation TXG and Block Pointer Birth TXGs to appear out of order with file timestamps.

Both false positives occurred during experiments where the tampering was performed by adjusting the system clock (details of the two experiments are shown in Table 5.7). The corrections occurred during a regularly scheduled 11PM system clock check, and adjusted the clock backwards, causing genuine timestamps to appear out of order. It is likely that the manual clock alteration triggered a larger than normal clock correction, increasing the anomalies in the time metadata.

It should be noted that this is not a *physical* false positive - these files truly were created in the order detected by the new ZFS timeline analysis methods and not the order indicated by their timestamps.

System clock corrections will generally be logged in a system log - such as `/var/log/ntpd` on FreeBSD (McKusick et al., 2014) - which can be used by the investigator to determine if any irregularity with timestamps was caused by a clock correction.

5.2.1.2 *Normal user Timestamp Alteration*

There are also valid reasons why timestamps may be changed by the user (or a userspace utility) to a previous value, such as unpacking archives or copying files with previous timestamps preserved - the newly created copies of the old files will appear to have “false” timestamps to forensic tools. Files which are downloaded when synchronising a distributed filesystem (Mar-

Table 5.7: Parameters of Experiments where False Positives Occurred

Parameter	Experiment 1	Experiment 2
Tampering Method	System Clock	System Clock
Devices in Pool	5	7
Pool Configuration	Flat	RAIDZ-2
System Clock Correction	-6.2 seconds	-12.1 seconds

tini and Choo, 2014) or cloud storage service (Quick and Choo, 2013) may also have their timestamps altered to reflect the creation and modification times of the remote file.

The on-disk effect of a preserved timestamp and a falsified timestamp are the same; the investigator must determine the user's intention.

5.2.1.3 *Object Number Reuse*

Physical false positives could be caused by reuse of Object Numbers by ZFS when it detects a previously allocated range of objects has been removed (see section 2.3.4.1). For this reason, Object Numbers should not be considered forensically sound.

No reuse of Object Numbers occurred during the experiments and further research is required to determine how frequent reuse is and how it would affect the practical use of Object Numbers for timeline analysis. It may be possible to detect and compensate for such reuse during forensic analysis as it occurs in specific ranges. The reuse could possibly be triggered deliberately by writing over 4096 files, then deleting them and creating over 4096 new files.

5.2.2 *Forensic Soundness*

"Forensic soundness" refers to "the application of a transparent digital forensic process that preserves the original meaning of the data for production in a court of law" (McKemmish, 2008). More simply, a process is "forensically sound" if it produces evidence which is admissible in court.

McKemmish (2008) identifies four criteria which must be addressed to consider any digital forensic process forensically sound:

(1) Meaning - Is the meaning or interpretation of the evidence unaffected by the process?

The process must not only preserve data but also the presentation or interpretation of the data (for example, timestamps may be stored in local time, even if the data is not modified the interpretation will be different in a different timezone).

(2) Errors - Have all errors been identified and fully explained?

This is important to avoid any doubt over the reliability of the evidence. Any errors occurring during the process must be logged, and the source of errors must be identified clearly.

For example, an "I/O error copying a file" is insufficient - the cause of the error (e.g. physical disk damage, filesystem corruption) should be identified, and it should be determined how much of the file is recoverable and which parts have been lost.

(3) Transparency - Is the process capable of being independently verified?

To be admissible in court the process must be fully documented including all tools and

software used, so that an independent expert can reconstruct the investigation and verify that the process is sound and the evidence is reliable.

(4) Experience - Has the process been performed by an expert with sufficient experience? Digital forensic investigation uses different processes to normal use of a system and should always be conducted by someone with specific training in digital forensics. The investigator must be familiar with forensic considerations such as soundness and preservation of evidence. Typically it is essential for an investigator to also be familiar with the specific technologies used by the system under investigation.

The forensic soundness of the processes presented in this thesis for ZFS timeline analysis are assured according to all criteria:

(1) Meaning - the meaning of the data is not affected by the processes involved, except as discussed in the relevant sections:

- As discussed in section 2.3, all timestamps are stored in UTC with only file object timestamps converted to local time when printed by ZDB; in practice these local timezone issues may be handled by the Plaso software (Leigh and Shi, 2014a; Gudjonsson, 2013).
- Sections 5.2.1.1 and 5.2.1.2 address the possibility of misinterpretation of mismatches between a timestamp and other metadata; the ZFS timeline analysis methods will detect when timestamps do not match the true creation or modification time of a file, but the investigator must determine if this is due to intentional tampering or innocent events such as a clock correction or copying old files.

- The potential for Object Numbers to be reused (see sections 4.2.4 and 5.2.1.3) means that Object Numbers cannot be reliably used to detect out of order timestamps and further research is required to determine if this method can be made forensically sound.

(2) Errors - any errors during disk imaging should be addressed by the investigator, following digital forensic best practice (McKemmish, 1999; Daltur and Hajdarevic, 2014). ZFS internal checksums may be used to determine if disk blocks have become corrupted.

(3) Transparency - all processes and analysed structures are fully documented in this thesis, and all software used is open source, allowing for verification by an independent expert.

(4) Experience - the process should be conducted by and individual with specialist training in digital forensics to address this criteria. Sections 2.1, 2.2 and 2.3 provide the background information required for the investigator to familiarise themselves with the specific ZFS knowledge required.

5.2.3 *Attribute Changes*

The file activity simulator simulated the effect of attribute changes (such as changes to file permissions, mode, user ID or group ID) by toggling the execute bit of files. This changes the "bonus buffer" in the file object itself (which contains the file attributes) but not the file data or pointers.

Although this writes a new copy of the file object, the attribute changes had no effect on the Object Number, Generation TXG or Block Pointers which point to file data, and appeared to have no effect on the successful falsification detection techniques; it may have contributed to the ineffectiveness of the Spacemap and Vdev number methods by increasing the number of modified objects.

6

CONCLUSION AND RECOMMENDATIONS

This chapter concludes this thesis by summarising the outcomes and key findings of this research, providing practical recommendations for forensic investigators (section 6.2), discussing potential future research in this area (section 6.3), and finally summarises and concludes the thesis in section 6.4.

6.1 RESEARCH OUTCOMES AND CONTRIBUTIONS

This project has exceeded its objectives and demonstrated four techniques for detecting falsified timestamps using ZFS metadata which are effective on real disks under simulated conditions. Additionally, a technique for determining the modification time of individual blocks of large files has been demonstrated. Key findings are listed in subsection 6.1.1 below.

In examining which ZFS structures are useful for forensic purposes, it has been determined that one of the six structures analysed (Uberblocks) has a short lifetime and in most cases will

only persist for 10.6 minutes after the initial examination, and that two other structures which were examined did not persist long enough to be useful, even if collected within 30 minutes of the events of interest .

This research has produced one publication (Leigh and Shi, 2014b) discussing timeline analysis of ZFS; it has also been used in a related project to extend existing super-timeline software to to process ZFS events which has produced another publication by the author (Leigh and Shi, 2014a). A third ZFS publication discussing the on-disk format of ZFS is planned for submission in late 2014 or early 2015 using the material from Chapters 2 and 3.

This project also contributes to forensic practice by providing methods for analysis of ZFS devices. Section 6.2 contains practical recommendations for forensic investigators when analysing ZFS systems, including general computer forensic procedures for ZFS as well as those specific to timeline analysis.

6.1.1 *Key Findings*

The most significant findings for ZFS timeline analysis are listed below:

(1) Comparing the birth TXG ID from the highest level file data Block Pointers is an effective and reliable method for detecting false timestamps. As birth TXG ID is closely related to the time and order of modification of blocks, they are useful for other timeline analysis techniques as well.

(2) The birth TXG from level 0 Block Pointers in files containing multiple blocks can be compared to the highest level birth TXG from other files to determine the last modification time of the individual blocks in a large file.

(3) The Generation TXG and Object Number of a file object can be used to determine the creation order of objects, and are effective at detecting files which have a false creation time. The Object Number method is not reliable as these numbers may be reused; the effect of this needs to be confirmed via further research. The use of Generation TXG is reliable and forensically sound.

(4) Uberblocks may be used to detect falsified timestamps but only if collected quickly, before the relevant Uberblock is overwritten. In a pool under a constant write load, Uberblocks are typically overwritten after 10.6 minutes, except for pools with more than 4 disks and no redundancy where Uberblocks may persist for several hours.

(5) Spacemaps and the Vdev number do not appear to be effective as a source of events for timeline analysis, due to the many transient objects caused by the copy-on-write mechanism of ZFS and the high rate at which Spacemaps are condensed.

6.2 RECOMMENDATIONS FOR FORENSIC INVESTIGATORS

This section contains practical recommendations for forensic investigators performing analysis of ZFS pools and disks.

Software, commands and procedures for ZFS Timeline Analysis are discussed in (Leigh and Shi, 2014a); A "Quick Reference" for ZFS Timeline Analysis commands is also provided in (Leigh, 2014).

6.2.1 *Initial Capture*

Before any procedures are carried out, ensure that relevant rules of evidence are observed to prevent any evidence collected from being tainted or ruled inadmissible in court. Considerations of forensic soundness are discussed in section 5.2.2.

Following forensic best practice (Daltur and Hajdarevic, 2014), a block-level image of all disks on the system should be taken, using "dd" (IEEE, 2001) or a specific forensic tool. ZFS devices are best saved in raw format as this allows them to be imported by the "`zpool`" command.

A digest should be taken of all images, and the original disks images and digests stored securely. Investigators should work only with copies of the original disk images, and copies of the original digests should be used to verify their integrity (Daltur and Hajdarevic, 2014).

If the data contained within the Uberblocks may be critical to an investigation, the system should be halted as soon as possible to ensure they are not overwritten. The opportunity to

perform live capture of system memory (including running processes and open files), as well as the ability to access any mounted encrypted filesystems (which may not be decryptable after halting the system), should be considered against the importance of recovering the Uberblocks.

6.2.2 *Importing the Pool for Analysis*

When importing a ZFS pool from the disk images, care must be taken to mount the pool read-only to prevent altering the evidence and to mount its datasets in a separate part of the directory hierarchy to prevent datasets from occupying existing mountpoints on the host system.

Read-only operation may be specified using the "readonly=on" property when importing the pool. Removing write permissions on the disk image files is not a sufficient precaution as the pool import command must be run as root which bypasses file permission checks.

ZFS datasets may contain a "mountpoint" property, which specifies their position in the filesystem hierarchy. These may be outside the mountpoint of the root dataset of the pool (e.g. a pool called "tank" and with root dataset mounted on "`/tank`" may contain other datasets with mountpoint set to "`/usr`" or "`/var`" - if imported directly these would override the "`/usr`" and "`/var`" system directories on the host computer).

The "altroot" property can be set when importing the pool which makes all mountpoints in the pool relative to the altroot; the "`-R`" option to "`zpool`" also sets this property. For example, using the alternate root "`/altroot`", a pool with "`/var`" and "`/usr`" datasets would mount those datasets on "`/altroot/usr`" and "`/altroot/var`".

Generally all disk images forming the pool should be imported at once to reconstruct the pool. Dedicated cache or spare devices are not required to import the pool but should still be investigated. The "`zpool`" command supports a "`-d`" option which will import pools from all devices from a specified directory (including disk images saved in raw format).

In conclusion, the command to import a set of images, specify read-only operation, and mount datasets relative to an alternate root is:

```
zpool import -R <alternate-root-dir> -o readonly=on -d <disk image directory>
```

6.2.3 *Initial Analysis of the Pool*

Once the pool has been imported, its status should be checked with the command "`zpool status -v <pool>`". This will report any missing or damaged devices.

ZFS Pools maintain a built in log of the creation of the pool, settings changes and the creation and deletion of datasets (including filesystems, snapshots and clones). This history log may contain invaluable information regarding deleted or hidden data and can be displayed using the command "`zpool history -il <pool>`".

All ZFS datasets in the pool may be listed with the command "`zfs list -t all`". Properties on the datasets may be listed with "`zfs get all`"; arbitrary properties may be set by the user and may contain or reference hidden data.

ZFS files and directories may contain extended attributes; commands to display these differ between different operating systems and distributions of ZFS. On FreeBSD, the "`lsextattr`" command may be used to list extended attributes.

Note that if compression of file data is enabled, many file carving tools will not function on ZFS as the headers used to identify files will have been compressed. Future ZFS-specific tools may be able to decompress blocks to counter this problem and should be used where possible.

6.2.4 *Using ZFS Structures for Timeline Analysis*

ZFS Structures may be examined as a source for extra modification events as well as for corroborating the time of file timestamp events.

The Plaso software (Gudjonsson, 2013) with ZDB parsers (Leigh and Shi, 2014a) may be used to add ZFS events to a super-timeline from ZDB output. These events can be used to corroborate timestamps and detect forged timestamps as described in section 5.1. These algorithms will flag any out-of-order timestamps, so it must be determined by the investigator if this was caused by deliberate tampering or an innocuous reason such as copying of old files or a regular system clock correction.

If performing manual analysis using ZFS structures, note that ZDB will display file timestamps converted to the local time of the system, although they are stored on disk in UTC time. Uberblock timestamps are both stored and displayed in UTC.

If the pool contains any large files (greater than 128KB) which have been modified in smaller parts over time, it may be possible to determine the modification time of individual blocks of the file. Algorithm 4.6 describes a method to do this; section 5.1.2 discusses the effectiveness of the method.

The following command may be used to list files with more than one Block Pointer to data, which may be examined with Algorithm 4.6:

```
zdb -P -bbbbbb -dddddd <poolname>/<dataset> | awk '($0 ~ /ZFS plain file/
&& int($2) > 1) {print}'
```

6.3 FUTURE WORK

Although this is the first time internal ZFS metadata has been examined for timeline forensics, the scope of this study is extremely limited and much more work needs to be done to validate the techniques presented here under varied conditions and explore other potential sources of evidence.

6.3.1 *Verification with Production Data*

Most importantly, data from production systems needs to be obtained to verify these results with real filesystem activity. This could involve a survey to obtain metadata from production systems, with volunteers asked to submit anonymised ZDB data (with file paths and names

removed). A script has already been developed to perform this anonymisation and ethical approval obtained for the survey.

The reuse of Object Numbers (see section 4.2.3) needs to be examined under realistic conditions to determine how often reuse occurs and how this affects forensic use of Object Numbers.

6.3.2 *Other ZFS Structures*

Only six ZFS structures have been examined at this stage. There are many other ZFS objects and structures which could be used for timeline analysis.

As identified by Beebe et al. (2009), the ZFS Intent Log (ZIL) contains metadata relating to filesystem events. As entries in the ZIL are discarded (but not destroyed) after they are committed to a TXG, information from ZIL entries may be recoverable long after they are written.

There are several per-dataset objects and attributes which are likely to be useful for timeline forensics, particularly the Delete Queue. Snapshot and clone datasets have not yet been examined and are likely to provide a great deal of time-related metadata to investigators. The Meta-Object-Set and its dataset directories may also contain useful timeline data.

Although Generation TXG were examined to detect falsified timestamps, other forensic uses which exploit the link between file creation time and the TXG have not been explored.

6.3.3 *Deleted ZFS Structures*

Regions of the disk which are no longer referenced by any Block Pointer could be examined to see if they contain old ZFS objects or Block Pointers. Most ZFS structures contain magic numbers so that they can be easily identified by scanning unallocated blocks; many structures contain TXG IDs so the time they were written could be determined.

The copy-on-write nature of ZFS is likely to leave past copies of file objects, which will contain TXG that could be used to determine the time of some past modifications. It needs to be determined empirically if this analysis would provide any useful data to forensic investigators.

As mentioned in section 5.1.4, inactive Object Sets from previous Uberblocks could also be traced to determine recent changes. Although the entire Uberblock array has been used to detect TXG/timestamp inconsistencies, only the object tree connected to the active Uberblock has been examined so far.

6.3.4 *Other Pool Configurations and Workloads*

Both sets of experiments used simulated file activity based on a corporate/engineering network file storage workload (Agrawal et al., 2007; Roselli et al., 2000). Many factors in the analysis, especially the condensation of spacemaps and allocation of large files, are affected by the patterns of file activity. Other types of systems (e.g. desktop, home media storage, database server,

virtual machine storage, webserver) could be expected to have different patterns of file activity which may make some techniques more or less viable.

Due to hardware limitations, the experiments in this project only used a running time of 24 hours and disks of 15GB. It would be beneficial to repeat the experiments with larger disks more typical of modern storage devices, and longer simulations.

The experiments also used only a single filesystem dataset, whereas real systems would use many filesystems, with different patterns of file activity. Other dataset types (snapshots, clones, ZVOLs) were not examined. Many ZFS features such as compression and deduplication were not used. Pools with dedicated log or cache devices were not examined; these are used to improve performance in many production systems.

6.3.5 *Tampering with Internal Metadata*

A determined forger could manually alter internal ZFS structures to match any modified file timestamps, in order to make the modification more difficult to detect by forensic investigators. This project has not tested the effects of tampering with internal metadata, although the effects are theoretically considered here.

Uberblocks are the simplest structure to modify as they are at the head of the tree, and if modified only their own checksum needs to be corrected. Cifuentes and Cano (2012) have investigated this further. However, this may not be worthwhile for the forger as Uberblocks are quickly overwritten (see section 5.1.4).

ID numbers (including Object number, Gen TXG and Block Pointer TXGs) could be falsified easily if a suitable old ID can be inserted. On a typical filesystem with many transient files there would probably be many IDs which could be "reused" this way. Once this change is made, the forger must then correct the checksums in all parent objects and Block Pointers up the tree and the Uberblock, to prevent the tampering from appearing as disk corruption. ZFS would automatically detect this "corruption" and prevent use of the blocks, replacing them with redundant uncorrupted blocks if possible. Further research is required to determine how simple it would be to alter an object and all parent objects in this way.

If an old ID cannot be used, tampering would be possible but may require rewriting all files in the filesystem to fit the desired IDs. In the worst case (or best case for the forensics investigator), where no plausible TXGs could be inserted, the entire pool would have to be recreated. Apart from the time and effort involved, rewriting the disk may leave its own traces detectable to the forensic investigators (such as lack of fragmentation).

6.4 CONCLUSION

Although ZFS is widely used for enterprise storage, there is little forensic research into ZFS. Existing forensic software and techniques are not equipped to handle the unusual format of ZFS pools and filesystems. As far as the author is aware, this research is the first in depth study of ZFS timeline forensics.

This research has determined that four ZFS internal structures can be used as a source for timeline analysis, including for detection of forged timestamps. Block Pointers to file data are par-

ticularly useful - they can be used to detect false modification timestamps and can sometimes be used to determine the modification time of individual blocks of file data. Object Number and Generation TXG can be used to determine creation order of objects and verify the time a file was created. Uberblocks can be effectively used for forensic purposes only if collected soon after the tampering occurs.

Clock corrections, unarchiving of files and other normal behaviour could appear to be deliberate tampering with timestamps, and investigators should consider this possibility when investigating anomalies between different sources of events. Investigators should also be aware of the possibility that the ZFS metadata has itself been tampered with.

Although these techniques have been shown to work under laboratory conditions they are yet to be tested in the field and on a wide variety of systems and configurations. Future research in this area should include a survey of production systems using ZFS to obtain examples of real metadata for analysis.

REFERENCES

Agrawal, N., Bolosky, W. J., Douceur, J. R., Lorch, J. R., 2007. A five-year study of file-system metadata. ACM Transactions on Storage (TOS) 3 (3), 9.
URL: "http://dl.acm.org/citation.cfm?id=1288788" (Cited on pages 54, 55, 80, 90, and 108.)

Ahrens, M., 2014a. OpenZFS: a Community of Open Source ZFS Developers. AsiaBSDCon 2014, 27.
URL: http://www.open-zfs.org/w/images/0/0f/AsiaBSDcon_paper.pdf (Cited on pages 14 and 39.)

Ahrens, M., May 2014b. OpenZFS: upcoming features and performance enhancements. In: BSDCan 2014.
URL: http://www.bsdcan.org/2014/schedule/events/466.en.html (Cited on pages 12 and 39.)

Beebe, N., 2009. Digital forensic research - The good, the bad and the unaddressed. Advances in Digital Forensics V, 17–36.
URL:

http://link.springer.com/chapter/10.1007/978-3-642-04155-6_2

(Cited on pages 2 and 42.)

Beebe, N., Stacy, S., Stuckey, D., 2009. Digital forensic implications of ZFS. Digital Investigation 6, S99–S107.

URL: http://www.sciencedirect.com/science/article/pii/S1742287609000449

(Cited on pages 2, 41, 45, and 107.)

Bonwick, J., Nov 2006. ZFS Block Allocation. Sun Microsystems, [online].

URL: https://blogs.oracle.com/bonwick/en_US/entry/zfs_block_allocation

(Cited on pages 18, 31, and 38.)

Bonwick, J., Ahrens, M., Henson, V., Maybee, M., Shellenbaum, M., 2003. The zettabyte file system. In: Proc. of the 2nd Usenix Conference on File and Storage Technologies.

URL: http://users.soe.ucsc.edu/~scott/courses/Fall04/221/zfs_overview.pdf (Cited on pages 2, 8, 9, 11, 12, 15, 18, 25, and 37.)

Bonwick, J., Moore, B., 2008. ZFS: The last word in file systems. SNIA Software Developers Conference.

URL: http://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf

(Cited on pages 10, 11, 15, 19, and 38.)

Bruning, M., 2008a. Recovering Removed File on ZFS Disk. (website).

URL: http://mbruning.blogspot.com.au/2008/08/

recovering-removed-file-on-zfs-disk.html (Cited on page 44.)

Bruning, M., 2008b. ZFS On-Disk Data Walk. In: OpenSolaris Developer Conference.
URL: http://www.osdevcon.org/2008/files/osdevcon2008-max.pdf
(Cited on pages 12 and 44.)

Bruning, M., 2013. ZFS Forensics - Recovering Files From a Destroyed Zpool. (website).
URL: http://www.joyent.com/blog/
zfs-forensics-recovering-files-from-a-destroyed-zpool (Cited on page 44.)

Buchholz, F. P., Falk, C., 2005. Design and Implementation of Zeitline: a Forensic Timeline Editor. In: DFRWS.
URL: https:
//users.cs.jmu.edu/buchhofp/publications/zeitline_dfrws.pdf
(Cited on pages 49, 50, and 52.)

Cifuentes, J., Cano, J., 2012. Analysis and Implementation of Anti-Forensics Techniques on ZFS. Latin America Transactions, IEEE (Revista IEEE America Latina) 10 (3), 1757–1766.
URL: "http://ieeexplore.ieee.org/xpl/articleDetails.jsp?
arnumber=6222582" (Cited on pages 45, 64, and 109.)

Daltur, V., Hajdarevic, K., 2014. Digital Forensic Investigation, Collection and Preservation of Digital Evidence. In: International Symposium on Sustainable Development. International Burch University.
URL: http://eprints.ibu.edu.ba/2534/1/30DVahidin.pdf (Cited on pages 97 and 102.)

Dawidek, P. J., 2007. Porting the ZFS file system to the FreeBSD operating system. Proc. of AsiaBSDCon, 97–103.
URL: http://2007.asiabsdcon.org/papers/P16-paper.pdf (Cited on pages 4 and 39.)

Fairbanks, K. D., 2012. An analysis of Ext4 for digital forensics. Digital Investigation 9, S118–S130.
URL: http://www.dfrws.org/2012/proceedings/DFRWS2012-13.pdf (Cited on page 48.)

FreeBSD, 2014. (Operating System). The FreeBSD Foundation.
URL: http://www.freebsd.org/ (Cited on page 58.)

Goja, B., Padha, D., July 2012. RAID Dependent Performance on Storage and Retrieval of Digital Forensics Meta Data Files with Different File Systems. International Journal of Computer Science and Telecommunications 3.
URL: http://www.ijcst.org/Volume3/Issue7/p14_3_7.pdf (Cited on page 48.)

Graf, U., 2009. ZFS Internal Structure. In: OpenSolaris Developer Conference 2009.
URL: http://www.osdevcon.org/2009/slides/zfs_internals_uli_graef.pdf (Cited on page 38.)

Gudjonsson, K., 2010. Mastering the super timeline with log2timeline. SANS Institute.
URL: http://www.sans.org/reading_room/whitepapers/logging/

mastering-super-timeline-log2timeline_33438 (Cited on pages 34 and 50.)

Gudjonsson, K., 2013. Plaso: Reinventing the super timeline. In: DFIR Summit 2013. URL: http://www.sans.org/reading_room/whitepapers/logging/mastering-super-timeline-log2timeline_33438 (Cited on pages 43, 50, 96, and 105.)

Hankins, R., Liu, J., 2008. Towards a forensic-aware file system. In: Electro/Information Technology, 2008. EIT 2008. IEEE International Conference on. IEEE, pp. 84–89. URL: http://dx.doi.org/10.1109/EIT.2008.4554272 (Cited on page 47.)

Harris, R., 2006. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. Digital Investigation 3, 44–49. URL: http://www.dfrws.org/2006/proceedings/6-Harris.pdf (Cited on pages 34 and 45.)

IEEE, 2001. IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Shell and Utilities, Issue 6. IEEE, revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6. (Cited on pages 57, 59, 61, and 102.)

Ihaka, R., Gentleman, R., 1996. R: A language for data analysis and graphics. Journal of computational and graphical statistics 5 (3), 299–314. URL: http://www.amstat.org/publications/jcgs/ (Cited on page 64.)

illumos, 2014. (Operating System). Delphix.

URL: http://www.illumos.org/ (Cited on page 4.)

Inglot, B., Liu, L., Antonopoulos, N., 2012. A Framework for Enhanced Timeline Analysis in Digital Forensics. In: Green Computing and Communications (GreenCom), 2012 IEEE International Conference on. IEEE, pp. 253–256.

URL:

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6468322

(Cited on pages 34 and 52.)

Khan, A., Naeem, M., 2012. Performance analysis of Bayesian networks and neural networks in classification of file system activities. Computers & Security.

URL: http:

//www.sciencedirect.com/science/article/pii/S0167404812000533

(Cited on pages 34 and 52.)

Leigh, D., May 2014. ZFS Timeline Forensics Quick Reference.

URL: http://research.dylanleigh.net/zfs-bsdcan-2014/

zfs-timeline-quickref.pdf (Cited on page 102.)

Leigh, D., Shi, H., August 2014a. Adding ZFS Events to a Super-Timeline. Digital Forensics Magazine (20).

URL: http://digitalforensicsmagazine.com/index.php?option=com_

content&view=article&id=949 (Cited on pages vi, 3, 43, 50, 96, 100, 102, 105, and C1.)

Leigh, D., Shi, H., May 2014b. Forensic Timestamp Analysis of ZFS. In: BSDCan 2014.

URL: http://www.bsdcan.org/2014/schedule/events/464.en.html

(Cited on pages vi, 2, 43, and 100.)

Leung, A. W., Pasupathy, S., Goodson, G. R., Miller, E. L., 2008. Measurement and Analysis of Large-Scale Network File System Workloads. In: USENIX Annual Technical Conference. Vol. 1. pp. 5–2.

URL: http://www.ssrc.ucsc.edu/Papers/leung-usenix08.pdf (Cited on pages 54, 55, 80, and 90.)

Leventhal, A., Dec 2012. ZFS Fundamentals: transaction groups. Delphix.

URL: http://blog.delphix.com/ahl/2012/zfs-fundamentals-transaction-groups/ (Cited on page 16.)

Li, A., June 2009. Digital Crime Scene Investigation for the Zettabyte File System. Tech. rep., Macquarie University.

URL: http://web.science.mq.edu.au/~rdale/teaching/itec810/2009H1/FinalReports/Li_Andrew_FinalReport.pdf (Cited on page 44.)

Martini, B., Choo, K.-K. R., 2014. Distributed filesystem forensics: XtreemFS as a case study. Digital Investigation 11 (4), 295 – 313.

URL: http://www.sciencedirect.com/science/article/pii/S1742287614000942

(Cited on pages 48 and 93.)

McDougall, R., Mauro, J., 2005. FileBench.

URL:

http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf (Cited on page 56.)

McKemmish, R., 1999. "What is forensic computing?". Australian Institute of Criminology. URL: http://isis.poly.edu/kulesh/forensics/ti118.pdf (Cited on pages 1 and 97.)

McKemmish, R., 2008. "When is digital evidence forensically sound?". Vol. 285. Springer. URL: http://dx.doi.org/10.1007/978-0-387-84927-0_1 (Cited on page 95.)

McKusick, M. K., Neville-Neil, G. V., Watson, R. N. M., 2014. The design and implementation of the FreeBSD operating system, 2nd Ed. Addison-Wesley Professional. (Cited on pages 4, 12, 40, 48, 59, and 93.)

Netgear, 2014. Netgear ReadyDATA Storage Products. URL: http://www.netgear.com/business/products/storage/readydata/ (Cited on page 5.)

OpenZFS, September 2014. OpenZFS Wiki: Companies. URL: http://open-zfs.org/wiki/Companies (Cited on pages 4, 5, 11, and 49.)

Oracle, 2014. Oracle ZFS Storage Software. URL: https://www.oracle.com/storage/nas/zfs-appliance-software/index.html (Cited on pages 4 and 5.)

Phromchana, V., Nupairoj, N., Piromsopa, K., 2011. Performance evaluation of ZFS and LVM (with ext4) for scalable storage system. In: Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on. IEEE, pp. 250–253.
URL: http://dx.doi.org/10.1109/JCSSE.2011.5930130 (Cited on page 8.)

Powell, H., 2012. ZFS and Btrfs: a quick introduction to modern filesystems. Linux Journal 2012 (218), 5.
URL:
http://www.linuxjournaldigital.com/linuxjournal/201206?pg=104
(Cited on page 4.)

Python, 2014. (Programming Language). Python Software Foundation.
URL: http://www.python.org/ (Cited on pages 56 and 64.)

Quick, D., Choo, K.-K. R., 2013. Forensic collection of cloud storage data: Does the act of collection result in changes to the data or its metadata? Digital Investigation 10 (3), 266 – 277.
URL: http:
//www.sciencedirect.com/science/article/pii/S1742287613000741
(Cited on pages 51 and 94.)

Roselli, D. S., Lorch, J. R., Anderson, T. E., et al., 2000. A Comparison of File System Workloads. In: USENIX Annual Technical Conference, General Track. pp. 41–54.
URL: https://www.usenix.org/legacy/event/usenix2000/general/full_papers/roselli/roselli.pdf (Cited on pages 54, 80, and 108.)

Spencer, M., February 2014. Beyond Timelines - Anchors in Relative Time. Digital Forensics Magazine.
URL: http://www.digitalforensicsmagazine.com/index.php?option=com_content&view=article&id=914&Itemid=99 (Cited on pages 1, 51, and 80.)

Sun Microsystems, 2006. ZFS On Disk Specification. Tech. rep., Sun Microsystems.
URL: https://maczfs.googlecode.com/files/ZFSOnDiskFormat.pdf (Cited on pages 2, 12, 20, 25, 26, 37, 38, and 40.)

Watanabe, S., 2009. Solaris 10 ZFS essentials. Pearson Education. (Cited on page 56.)

Zhang, Y., Rajimwale, A., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., 2010. End-to-end data integrity for file systems: a ZFS case study. In: Proceedings of the 8th USENIX conference on File and storage technologies. USENIX Association, pp. 3–3.
URL: http://research.cs.wisc.edu/adsl/Publications/zfs-corruption-fast10.pdf (Cited on pages 8 and 9.)

APPENDICES

A

ZDB EXAMPLE OUTPUT

Several ZDB output listings were abbreviated for space and clarity in the ZFS background material (chapter 2); more detailed versions are provided in this Appendix.

A.1 COMPLETE FILE OBJECT WITH MULTIPLE BLOCK POINTERS

```
    Object  lvl   iblk   dblk  dsize  lsize   %full  type
         7    2    16K   128K   258K   256K  100.00  ZFS plain file (K=inherit) (Z=
             inherit)
                                      264   bonus  ZFS znode
        dnode flags: USED_BYTES USERUSED_ACCOUNTED
        dnode maxblkid: 1
        path    /testfile
        uid     0
        gid     0
        atime   Tue Oct  7 18:44:06 2014
        mtime   Tue Oct  7 18:46:03 2014
        ctime   Tue Oct  7 18:46:03 2014
        crtime  Tue Oct  7 18:44:06 2014
        gen     25
        mode    100644
        size    145408
        parent  3
        links   1
        xattr   0
        rdev    0x0000000000000000
Indirect blocks:
               0 L1  DVA[0]=<0:16f200:400> DVA[1]=<0:3c36c00:400> [L1 ZFS plain file]
                     fletcher4 lzjb LE contiguous unique double size=4000L/400P birth
                    =29L/29P fill=2 cksum=5d9a1273ac:3e248eb0dbe0:160c4014973910:5846f
                    11b6631c3e
               0  L0 DVA[0]=<0:dec00:20000> [L0 ZFS plain file] fletcher4
                    uncompressed LE contiguous unique single size=20000L/20000P birth
                    =25L/25P fill=1 cksum=53c853c8000:14f23ed629e4000:dc31a8634c
                    698000:1a31a7d7a94f2000
           20000  L0 DVA[0]=<0:14f200:20000> [L0 ZFS plain file] fletcher4
                uncompressed LE contiguous unique single size=20000L/20000P birth=29L
                /29P fill=1 cksum=929e929e00:454d3c9bf74f00:65a2311b55898a00:887e8b65c
                1442780
```

A.2 PARTIAL VDEV LABEL AND UBERBLOCK ARRAYS

Each ZFS device has 4 identical labels for redundancy, only the first one has been reproduced here. The complete output of a single label is 922 lines; most Uberblocks have been removed to reduce the output to 83 lines.

Uberblock 124 is the active Uberblock, with the highest TXG and most recent timestamp.

```
--------------------------------------------
LABEL 0
--------------------------------------------
    version: 28
    name: 'poolv3r0'
    state: 0
    txg: 4
    pool_guid: 3719156361862507906
    hostid: 681876320
    hostname: 'dleigh-test'
    top_guid: 15256763276442738464
    guid: 15256763276442738464
    vdev_children: 3
    vdev_tree:
        type: 'disk'
        id: 2
        guid: 15256763276442738464
        path: '/dev/ada0p5'
        phys_path: '/dev/ada0p5'
        whole_disk: 1
        metaslab_array: 30
        metaslab_shift: 27
        ashift: 9
        asize: 21470117888
        is_log: 0
        create_txg: 4
Uberblock[0]
        magic = 0000000000bab10c
        version = 28
        txg = 1664
        guid_sum = 4420604878723568201
        timestamp = 1382428179 UTC = Tue Oct 22 18:49:39 2013
        rootbp = DVA[0]=<0:32c7c00:200> DVA[1]=<1:3416200:200> DVA[2]=<2:2a93c00:200>
             [L0 DMU objset] fletcher4 lzjb LE contiguous unique triple size=800L/200
            P birth=1664L/1664P fill=42 cksum=155e460153:7a5e17959ed:16ce954826746:2f
            2ae0e9bee664
Uberblock[1]
        magic = 0000000000bab10c
        version = 28
        txg = 1665
        guid_sum = 4420604878723568201
        timestamp = 1382428184 UTC = Tue Oct 22 18:49:44 2013
```

```
        rootbp = DVA[0]=<1:3447400:200> DVA[1]=<2:2ab4c00:200> DVA[2]=<0:32fa600:200>
             [L0 DMU objset] fletcher4 lzjb LE contiguous unique triple size=800L/200
            P birth=1665L/1665P fill=42 cksum=16b09d1752:7f5e36bc85d:173ba6e3f4af9:2
            ef487c8330d41
Uberblock[2]
        magic = 0000000000bab10c
        version = 28
        txg = 1666
        guid_sum = 4420604878723568201
        timestamp = 1382428189 UTC = Tue Oct 22 18:49:49 2013
        rootbp = DVA[0]=<2:2af9a00:200> DVA[1]=<0:332ca00:200> DVA[2]=<1:3461200:200>
             [L0 DMU objset] fletcher4 lzjb LE contiguous unique triple size=800L/200
            P birth=1666L/1666P fill=42 cksum=1527de5cd8:795cf0fd543:169e8911cd46b:2
            eacc0e4a01dd2
...
Uberblock[123]
        magic = 0000000000bab10c
        version = 28
        txg = 1787
        guid_sum = 4420604878723568201
        timestamp = 1382428794 UTC = Tue Oct 22 18:59:54 2013
        rootbp = DVA[0]=<0:4582400:200> DVA[1]=<1:471fa00:200> DVA[2]=<2:3f48600:200>
             [L0 DMU objset] fletcher4 lzjb LE contiguous unique triple size=800L/200
            P birth=1787L/1787P fill=42 cksum=13d8a1ba4e:6f4c153e489:145af4701a
            760:2963d35a7446e7
Uberblock[124]
        magic = 0000000000bab10c
        version = 28
        txg = 1788
        guid_sum = 4420604878723568201
        timestamp = 1382428799 UTC = Tue Oct 22 18:59:59 2013
        rootbp = DVA[0]=<1:474da00:200> DVA[1]=<2:3f58000:200> DVA[2]=<0:4598200:200>
             [L0 DMU objset] fletcher4 lzjb LE contiguous unique triple size=800L/200
            P birth=1788L/1788P fill=42 cksum=15b7c6c784:79822032122:161abf08c9f79:2c
            9543a87c8a80
Uberblock[125]
        magic = 0000000000bab10c
        version = 28
        txg = 1661
        guid_sum = 4420604878723568201
        timestamp = 1382428164 UTC = Tue Oct 22 18:49:24 2013
        rootbp = DVA[0]=<0:3239e00:200> DVA[1]=<1:336d800:200> DVA[2]=<2:2a43200:200>
             [L0 DMU objset] fletcher4 lzjb LE contiguous unique triple size=800L/200
            P birth=1661L/1661P fill=42 cksum=15fb0480f5:7cd9ad868b2:17332850156e
            9:30090ee897e6e2
Uberblock[126]
        magic = 0000000000bab10c
        version = 28
        txg = 1662
        guid_sum = 4420604878723568201
        timestamp = 1382428169 UTC = Tue Oct 22 18:49:29 2013
        rootbp = DVA[0]=<1:3395000:200> DVA[1]=<2:2a4ee00:200> DVA[2]=<0:3278e00:200>
             [L0 DMU objset] fletcher4 lzjb LE contiguous unique triple size=800L/200
            P birth=1662L/1662P fill=42 cksum=152528eacc:79212a8c862:1689d95c056c2:2e
            6ba7a6d4c250
Uberblock[127]
        magic = 0000000000bab10c
        version = 28
        txg = 1663
        guid_sum = 4420604878723568201
        timestamp = 1382428174 UTC = Tue Oct 22 18:49:34 2013
```

```
rootbp = DVA[0]=<2:2a79a00:200> DVA[1]=<0:3294800:200> DVA[2]=<1:3400200:200>
     [L0 DMU objset] fletcher4 lzjb LE contiguous unique triple size=800L/200
    P birth=1663L/1663P fill=42 cksum=15fc53ee1f:7f6a308f8bb:17ee74591fc52:31
    a270192d5991
```

A.3 PARTIAL SPACEMAP OUTPUT

This output is from a pool with a single file to provide a simpler example. The listing provided here is still abbreviated as the complete output is 538 lines, most of which refer to unused metaslabs and are not relevant.

```
Metaslabs:
   vdev          0
   metaslabs   119   offset                spacemap          free
   ---------------   -------------------   ---------------   -------------
   metaslab      0   offset            0   spacemap     35   free    536655872
   segments         15   maxsize  536123392   freepct   99%
       [     0] ALLOC: txg 229, pass 1
       [     1]    A  range: 0000000000-0000008000  size: 008000
       [     2]    A  range: 0000011000-0000016800  size: 005800
       [     3]    A  range: 000000f800-0000010800  size: 001000
       [     4]    A  range: 000001c800-000001d000  size: 000800
       [     5]    A  range: 0000041800-0000043000  size: 001800
       [     6]    A  range: 0000036800-000003a000  size: 003800
       [     7]    A  range: 0000017800-000001a000  size: 002800
       [     8]    A  range: 0000052000-0000053000  size: 001000
       [     9]    A  range: 0000065800-0000066000  size: 000800
       [    10]    A  range: 000005c800-0000060000  size: 003800
       [    11]    A  range: 000009b000-000009c800  size: 001800
       [    12]    A  range: 00000c4000-00000c7000  size: 003000
       [    13]    A  range: 00000b5000-00000b6800  size: 001800
       [    14]    A  range: 00000ad000-00000b4800  size: 007800
       [    15]    A  range: 0000093000-000009a800  size: 007800
       [    16]    A  range: 0000043800-0000047800  size: 004000
       [    17] ALLOC: txg 229, pass 2
       [    18]    A  range: 0000008000-000000b000  size: 003000
       [    19] FREE: txg 229, pass 2
       [    20]    F  range: 00000c4000-00000c7000  size: 003000
   metaslab      1   offset     20000000   spacemap      0   free    536870912
   segments          1   maxsize  536870912   freepct  100%
   metaslab      2   offset     40000000   spacemap      0   free    536870912
   segments          1   maxsize  536870912   freepct  100%
   metaslab      3   offset     60000000   spacemap      0   free    536870912
   segments          1   maxsize  536870912   freepct  100%
...
   metaslab     20   offset    280000000   spacemap      0   free    536870912
   segments          1   maxsize  536870912   freepct  100%
   metaslab     21   offset    2a0000000   spacemap      0   free    536870912
   segments          1   maxsize  536870912   freepct  100%
   metaslab     22   offset    2c0000000   spacemap      0   free    536870912
   segments          1   maxsize  536870912   freepct  100%
   metaslab     23   offset    2e0000000   spacemap     34   free    536655872
   segments         15   maxsize  536123392   freepct   99%
       [     0] ALLOC: txg 229, pass 1
       [     1]    A  range: 02e0000000-02e0008000  size: 008000
       [     2]    A  range: 02e0011000-02e0016800  size: 005800
```

```
        [     3]    A  range: 02e000f800-02e0010800  size: 001000
        [     4]    A  range: 02e001c800-02e001d000  size: 000800
        [     5]    A  range: 02e0041800-02e0043000  size: 001800
        [     6]    A  range: 02e0036800-02e003a000  size: 003800
        [     7]    A  range: 02e0017800-02e001a000  size: 002800
        [     8]    A  range: 02e0052000-02e0053000  size: 001000
        [     9]    A  range: 02e0065800-02e0066000  size: 000800
        [    10]    A  range: 02e005c800-02e0060000  size: 003800
        [    11]    A  range: 02e009b000-02e009c800  size: 001800
        [    12]    A  range: 02e00c4000-02e00c7000  size: 003000
        [    13]    A  range: 02e00b5000-02e00b6800  size: 001800
        [    14]    A  range: 02e00ad000-02e00b4800  size: 007800
        [    15]    A  range: 02e0093000-02e009a800  size: 007800
        [    16]    A  range: 02e0043800-02e0047800  size: 004000
        [    17] ALLOC: txg 229, pass 2
        [    18]    A  range: 02e0008000-02e000b000  size: 003000
        [    19] FREE: txg 229, pass 2
        [    20]    F  range: 02e00c4000-02e00c7000  size: 003000
    metaslab     24   offset    300000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%
    metaslab     25   offset    320000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%
    metaslab     26   offset    340000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%
...
    metaslab     44   offset    580000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%
    metaslab     45   offset    5a0000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%
    metaslab     46   offset    5c0000000   spacemap     33   free    536780800
    segments           3   maxsize  536686592   freepct   99%
        [     0] ALLOC: txg 229, pass 1
        [     1]    A  range: 05c0000000-05c0007000  size: 007000
        [     2]    A  range: 05c000d000-05c000d800  size: 000800
        [     3]    A  range: 05c0008000-05c000c800  size: 004800
        [     4]    A  range: 05c0061000-05c0064000  size: 003000
        [     5]    A  range: 05c0029800-05c002d000  size: 003800
        [     6]    A  range: 05c001d000-05c0020800  size: 003800
        [     7] ALLOC: txg 229, pass 2
        [     8]    A  range: 05c0007000-05c0008000  size: 001000
        [     9]    A  range: 05c000d800-05c000f000  size: 001800
        [    10]    A  range: 05c000c800-05c000d000  size: 000800
        [    11] FREE: txg 229, pass 2
        [    12]    F  range: 05c0061000-05c0064000  size: 003000
    metaslab     47   offset    5e0000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%
    metaslab     48   offset    600000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%
    metaslab     49   offset    620000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%
...
    metaslab    115   offset    e60000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%
    metaslab    116   offset    e80000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%
    metaslab    117   offset    ea0000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%
    metaslab    118   offset    ec0000000   spacemap      0   free    536870912
    segments           1   maxsize  536870912   freepct  100%

    vdev          1
    metaslabs   140   offset                spacemap          free
```

```
   ---------------   --------------------   ---------------   -------------
   metaslab      0   offset            0   spacemap      0   free    67108864
   segments          1   maxsize  67108864   freepct  100%
   metaslab      1   offset      4000000   spacemap      0   free    67108864
   segments          1   maxsize  67108864   freepct  100%
   metaslab      2   offset      8000000   spacemap      0   free    67108864
   segments          1   maxsize  67108864   freepct  100%
...
   metaslab    137   offset    224000000   spacemap      0   free    67108864
   segments          1   maxsize  67108864   freepct  100%
   metaslab    138   offset    228000000   spacemap      0   free    67108864
   segments          1   maxsize  67108864   freepct  100%
   metaslab    139   offset    22c000000   spacemap      0   free    67108864
   segments          1   maxsize  67108864   freepct  100%
```

B

SIMULATION EXPERIMENT STATISTICS

This appendix contains general statistics from the primary simulations which was not significant to the key findings of this thesis and was not included in Chapter 5 with the main results. This data is provided in Table B.1 for completeness.

Clarification of terms:

- "Active files" refers to the number of files which were active in the simulator at the time the experiment was terminated and would have been modified and/or deleted later in the simulation.
- "Total Metaslabs per Vdev" refers to to the number of metaslabs in each virtual device. In flat pools there is one for each disk in the pool; in redundant pools there was one set of metaslabs across all devices as all are combined in a mirror or RAID-Z Vdev.
- "Spacemap TXG Distribution" shows the 5 number summary distribution of the TXG spacemaps were written (including collations) at the end of the experiment. The highest value is also the TXG when the file activity simulator was terminated.

Parameter	Lower Quart.	Median	Upper Quart.	Mean	Std. Dev.
File Object Statistics:					
Active files at end of simulation	24462.5	25110	25985	25317	2021.5
Files on disk at end of simulation	44099	46026	47734	46691	6899.9
Highest Object ID	57346	59864	62087	60849	9367.1
Files Deleted During Simulation	13125	13810	14262	13432	6375.9
File Objects with >1 Block Pointer	3	5	6	5.1412	2.5502
File Object Statistics, Normalised per 24 hours:					
Active files at end of simulation	25084	26106	27231	26136	2376.8
Files on disk at end of simulation	46042	48086	49954	48215	5515.3
Highest Object ID	59958	62501	64842	61987	10031
Files Deleted During Simulation	13831	14364	14877	13762	6933.3
File Objects with >1 Block Pointer	3.3333	5.1724	6.6932	5.2649	2.4704
Metaslab Statistics:					
Used Metaslabs	8	15	31	20.618	16.870
Total Metaslabs per Vdev	127	127	159	136.37	17.546
Percentage of Used Metaslabs	6.305%	11.417%	21.260%	15.203%	12.763%
Spacemap TXG distribution:					
Earliest Spacemap TXG	2350	3845	4593	3895.6	2342.0
25th Percentile	7895.5	8626	9568.5	9566.7	3805.4
Median Spacemap TXG	11192	12002	13104	12837	4034.9
75th Percentile	13534	14656	15873	15373	4310.7
Final Pool TXG	15363	16580	17308	17314	4699.4

Table B.1: General Simulation Experiment Statistics

C

SCRIPTS USED FOR EXPERIMENTS AND ANALYSIS

This appendix contains the source code of the scripts used to run the experiments, including the simulators and data collection script, as well as some of the scripts used for analysis. Some analysis scripts developed during this project have been superseded by the Plaso ZDB parsers (Leigh and Shi, 2014a).

C.1 FILE ACTIVITY SIMULATOR

This script was used to simulate file activity for the primary set of experiments; see section 4.1 for details.

Listing 1: File Activity Simulator

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

import os,sys,time,random,math,stat

# File names have the following syntax:
# <creation_time_t>-to-<deletion_time_t>-reopens-<x>
# In case of a collision (which because of the statistics involved is
# actually likely), a random letter is appended to the end.

# NOTE: An EXPONENTIAL distribution is used for the time
# between events in a Poisson process (continuously and independently
# at constant average rate).

# FIXME TODO : Directories, Partial reopen writes, Reopen timing

# TODO: Random speedup thoughts:
#  - Seperate list for reopened files
#  - Sort by deletion time rather than searching all each pass

################################################################
#######################################   Constants
################################################################

# Time of each pass in seconds. If a pass takes shorter than this
# we sleep for the remainder.
PASS_TIME = 1.0

# Abort if too many passes take longer that this time.
PASS_LIMIT = 2.2
LONG_PASS_STREAK_LIMIT = 3

# Used to increase the rate to simulate more activity over time for
# testing the script
RATE_MULTIPLIER = 1
# Base rate of writes and re-writes to do each pass
BASE_WRITE_RATE = 2.7 * PASS_TIME * RATE_MULTIPLIER
# TODO: Consider modifying by % space free ?
# This is derived from base write rate - ratio of 11.6 to 3
BASE_METADATA_RATE = 0.70 * PASS_TIME * RATE_MULTIPLIER
# NOTE: Scrap idea to do differing BH and AH rates as this also
# requires remodelling the rewrite and delete behaviour.

# Lognorm equivalent to 2KB Mode and 4KB Median
FILESIZE_MU = 8.31777
FILESIZE_SIGMA = 0.832555
```

```
# TODO FIXME Size of rewrite? For now this is a new size... ZFS is COW
# so file data will be changed anyway?

# Lognorm equivalent to 43.73 <= 1 and 72.73% <= 5; this is modified
# from Leung's figures to ignore the 21.27% of files which are deleted
# instantly (and will thus never be reopened).
REOPEN_MU = 0.33
REOPEN_SIGMA = 2.1
REOPEN_MAXTIME= 172800      # 81 days / 2 = 3499200 seconds
                # 2 days    # The maximum time for a file reopen,
                # FIXME     # equal to half the length of Leung's study
                            # because we are using his figures

# These are applied before the distribution below
# 44.78% Deleted thus 55.22% not deleted
LIFETIME_NO_DEL = 0.5522  # Prob. deleted never (applied first!)
LIFETIME_DEL_NOW = 0.475   # Prob. deleted instantly (applied second!)

# Approximate by exponential dist. such that 12.8% left after 24 hours
# or 86400 seconds. Value returned for lifetime is in seconds.
LIFETIME_LAMBDA = 1.48E-06

TIME_FOREVER = 2147483647 # about 2037

WRITE_FULL_SEQ = 0.1115  # Probability of "fully sequential" write;
                         # otherwise the file is partly written

################################################################
########################################     Misc Functions
################################################################

def randomPoisson(lam):
   '''Uses Knuth's pseudo-random number sampling algorithm to generate
      a poisson-distributed random variable with given lambda lam.
      Note that this algorithm is inefficient for larger lambda values.'''

   L = math.exp(-lam)
   k = 0
   p = 1

   while (p > L):
      k += 1
      p *= random.random() # float between 0 and 1

   return k-1

def writeSimFile(fname,size):
   '''Writes size bytes of 'x' to a file.'''
   fhandle = open(fname, 'w')
   while (size > 0): # TODO FIXME very inefficient
      fhandle.write('x')
      size -= 1
   fhandle.close()

################################################################
################################ SimFile Generator Functions
################################################################

def genReopens(creation, destruction):
   '''Returns a List of times this file will be reopened for writing
      between its creation and destruction. This may be an empty list.'''
   # if about to be deleted don't do reopens
```

```
    if (creation+1 >= destruction):
        return [] # empty list

    # else generate count according to stats
    count = random.lognormvariate(REOPEN_MU,REOPEN_SIGMA)
    count = count - 1 # Fix result of lognorm to the dicrete pattern
    count = int(count)
    if (count <= 0):
        return [] # empty list

    if (destruction > creation+REOPEN_MAXTIME): # clip reopen times, see
        destruction = creation+REOPEN_MAXTIME # constant description for why

    l = []
    for i in range(count):
        # Distribution of reopens is skewed towards write time TODO better
        l.append(random.triangular(creation+1,destruction,creation+1))

    l.sort() # earliest to latest
    return l

def genFileSize():
    '''Returns a randomly-generated size in bytes for the file,
        according to the FILESIZE_* constants'''
    return random.lognormvariate(FILESIZE_MU,FILESIZE_SIGMA)

def genDeletionTime(now):
    '''Determines a random time between now and infinity when a file
        will be deleted, following the statistics from the LIFETIME_*
        constants. now and the return value are time_t.'''

    # Will the file be deleted at all?
    if (random.random() < LIFETIME_NO_DEL):
        return TIME_FOREVER

    # Will the file be deleted instantly?
    if (random.random() < LIFETIME_DEL_NOW):
        return now

    # Determine time using an exponential distribution:
    return now + random.expovariate(LIFETIME_LAMBDA)

############################################################
#######################################   Classes
############################################################

# Class to hold all information about a single file
# Data to hold (all generated by the constructor except creation time):
#  - Filename (unique)
#  - File size
#     - TODO: Randomly redone on rewrite, this should be more
#       realistic, although ZFS is COW anyway.
#  - Last write time (note: sort by this value?)
#  - Creation time
#  - Deletion time (may be infinity; may be same as creation)
#  - List of reopens (may be empty)
#     - Each is a time_t between creation and deletion
class SimFile (object):

    def __init__(self, fdir, now):
        '''Creates a new SimFile, generating all attributies including
        lifetime info, filename, creating the filename and handle. Takes
```

```
        in the directory to create the file in, and the current time_t'''
        self.creTime = now

        self.delTime = genDeletionTime(now)
        self.size = genFileSize()
        self.reopens = genReopens(self.creTime,self.delTime)
        self.execP = False # created without execute perm

        # generate name for the file, containing a lot of write time info
        # <creation_time_t>-to-<deletion_time_t>-reopens-<x>-...
        self.fname = "%s/%d-to-%d-reps-%d"% \
                 (fdir,self.creTime,self.delTime,len(self.reopens))

        # Cancel due to file name limits
        #for i in self.reopens:     # append reopen times to filename
        #   self.fname = "%s-%d"%(self.fname,i)

        # check for name collisions
        while (os.path.exists(self.fname)):
            self.fname = "%s_%s"%\
                (self.fname,random.choice('abcdefghijklmnopqrstuvwxyz'))

        # Throw the exceptions if IO fails so the caller can delete this
        # object and/or stop/pause the simulation
        writeSimFile(self.fname,self.size)
        self.lastWrite = now

    def checkNoEvents(self):
        '''Returns True if this file has no future events and will stay
        in its current state forever.'''

        if (self.delTime == TIME_FOREVER and len(self.reopens) == 0):
            return True
        else:
            return False

    def checkReopen(self, now):
        '''Checks to see if a re-open and write should happen to this file
        in this pass. Returns true if this file WAS written by this func'''
        if (len(self.reopens) == 0):
            return False
        else:
            if (self.reopens[0] <= now):
                newSize = genFileSize() # TODO: more realistic
                writeSimFile(self.fname,newSize) # rewrite file
                self.size = newSize
                self.reopens.pop(0) # remove this reopen
                return True
        return False

    def checkDelete(self, now):
        '''Checks to see if a re-open or delete should happen to this file
        in this pass. Returns true if this file WAS deleted by this func'''
        if (self.delTime <= now):
            os.remove(self.fname)
            return True
        return False

    def toggleMeta(self):
        '''Toggles a metadata component of the file (e.g. execute bit)
        to simulate a metadata / set attribute operation'''
        if (self.execP):
```

```
            os.chmod(self.fname,stat.S_IRUSR|stat.S_IWUSR)
            self.execP = False
         else:
            os.chmod(self.fname,stat.S_IRUSR|stat.S_IWUSR|stat.S_IXUSR)
            self.execP = True

##############################################################
#######################################   Globals / Init
##############################################################

# Initialisation
fileList = list() # List of all file objects

# Directory to put the simulated file activity
try:
   workDir = sys.argv[1]
except:
   sys.exit("You must specify a working directory on the command line")
if (not os.path.isdir(workDir)):
   sys.exit("You must specify an existing working directory")
os.chdir(workDir)

##############################################################
#######################################   Main Loop
##############################################################

# Main loop begins here
longPassStreak = 0
while True:
   passStart = time.time() # so we can use this for time since epoch
   now = passStart
   passWrites = 0

   # Write to old files which have a re-open <= now
   for i in fileList:
      if (i.checkReopen(now)):
         passWrites += 1
   passReopens = passWrites # more writes to come

   # Generate new files to make up to write rate
   targetWriteRate = randomPoisson(BASE_WRITE_RATE)

   while (targetWriteRate > passWrites):
      fileList.append(SimFile('./',now)) # create new simfile
      passWrites += 1

   # Generate metadata (set attrib) writes
      # Set GID To random value
   targetMetaRate = randomPoisson(BASE_METADATA_RATE)
   passMetas = 0
   if (len(fileList) > 0):
      while (passMetas < targetMetaRate):
         random.choice(fileList).toggleMeta()
         # Note: can't use rand.sample in case metarate > len(filelist)
         passMetas += 1

   # Delete files for deletion time <= now
   # re-fetch the value of now in case we have taken a long time this
   # pass
   now = time.time()
   rmlist = list()
   delPass = 0
```

```
    for i in fileList:
       if (i.checkDelete(now)):
          rmlist.append(i)  # remove this items from the list later
          delPass += 1
       else:                # See if we ever need to check again
          if (i.checkNoEvents()):
             rmlist.append(i)  # we can ignore these files forever
    for i in rmlist:
       fileList.remove(i)

    passEnd = time.time()
    passTime = passEnd-passStart
    # Note writes includes reopens and new files.
    print 'File %d, Write %d, Reopen %d, Attrib %d, Del %d, Ign %d, Time %f'%\
             (len(fileList),targetWriteRate,passReopens,passMetas,\
              delPass,len(rmlist)-delPass,passTime)

    if (passTime > 0 and passTime < PASS_TIME): # If pass took less than
       time.sleep(PASS_TIME-passTime)         # allocated, sleep for remainder
    elif (passTime > PASS_LIMIT):
       print('Pass at %d (%s) too long!'%(passEnd,time.ctime(passEnd)))
       longPassStreak += 1
       if (longPassStreak > LONG_PASS_STREAK_LIMIT):
             sys.exit('Aborting at %d, too many passes over limit!'%passEnd)
    else:
        longPassStreak = 0

# No cleanup - leave disk/files in final state.
```

C.2 LARGE FILE SIMULATOR

This script was used to simulate file activity for the follow-up “large file” experiments; see section 4.3 for details.

Listing 2: Large File Activity Simulator

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

import os,sys,time,random,math,stat

# The large file is named "large-append-file-<creationtime>"
# Other file names have the following syntax:
# <creation_time_t>-to-<deletion_time_t>-reopens-<x>
# In case of a collision (which because of the statistics involved is
# actually likely), a random letter is appended to the end.

# NOTE: An EXPONENTIAL distribution is used for the time
# between events in a Poisson process (continuously and independently
# at constant average rate).

# TODO: Random speedup thoughts:
#  - Seperate list for reopened files
#  - Sort by deletion time rather than searching all each pass

################################################################
#######################################   Constants
################################################################

# Time of each pass in seconds. If a pass takes shorter than this
# we sleep for the remainder.
PASS_TIME = 1.0

# Abort if too many passes take longer that this time.
PASS_LIMIT = 2.2
LONG_PASS_STREAK_LIMIT = 3

# Used to increase the rate to simulate more activity over time for
# testing the script
RATE_MULTIPLIER = 1
# Base rate of writes and re-writes to do each pass
BASE_WRITE_RATE = 2.7 * PASS_TIME * RATE_MULTIPLIER
# Consider modifying by % space free ?
# This is derived from base write rate - ratio of 11.6 to 3
BASE_METADATA_RATE = 0.70 * PASS_TIME * RATE_MULTIPLIER

# Lognorm equivalent to 2KB Mode and 4KB Median
FILESIZE_MU = 8.31777
FILESIZE_SIGMA = 0.832555

# Consider size of rewrite? For now this is a new size...

# Lognorm equivalent to 43.73 <= 1 and 72.73% <= 5; this is modified
```

```
# from Leung's figures to ignore the 21.27% of files which are deleted
# instantly (and will thus never be reopened).
REOPEN_MU = 0.33
REOPEN_SIGMA = 2.1
REOPEN_MAXTIME= 172800     # 81 days / 2 = 3499200 seconds
                # 2 days   # The maximum time for a file reopen,
                # NOTE     # equal to half the length of Leung's study
                           # because we are using his figures

# These are applied before the distribution below
# 44.78% Deleted thus 55.22% not deleted
LIFETIME_NO_DEL = 0.5522  # Prob. deleted never (applied first!)
LIFETIME_DEL_NOW = 0.475   # Prob. deleted instantly (applied second!)

# Approximate by exponential dist. such that 12.8% left after 24 hours
# or 86400 seconds. Value returned for lifetime is in seconds.
LIFETIME_LAMBDA = 1.48E-06

TIME_FOREVER = 2147483647 # about 2037

WRITE_FULL_SEQ = 0.1115  # Probability of "fully sequential" write;
                         # otherwise the file is partly written

################################################################
#######################################     Misc Functions
################################################################

def randomPoisson(lam):
   '''Uses Knuth's pseudo-random number sampling algorithm to generate
      a poisson-distributed random variable with given lambda lam.
      Note that this algorithm is inefficient for larger lambda values.'''

   L = math.exp(-lam)
   k = 0
   p = 1

   while (p > L):
      k += 1
      p *= random.random() # float between 0 and 1

   return k-1

def appendSimFile(fname,size):
   '''Appends size bytes of 'x' to a file.'''
   fhandle = open(fname, 'a')
   while (size > 0): # TODO FIXME very inefficient
      fhandle.write('x')
      size -= 1
   fhandle.close()

def writeSimFile(fname,size):
   '''Writes size bytes of 'x' to a file.'''
   fhandle = open(fname, 'w')
   while (size > 0): # TODO FIXME very inefficient
      fhandle.write('x')
      size -= 1
   fhandle.close()

################################################################
################################ SimFile Generator Functions
################################################################
```

```
def genReopens(creation, destruction):
   '''Returns a List of times this file will be reopened for writing
      between its creation and destruction. This may be an empty list.'''
   # if about to be deleted don't do reopens
   if (creation+1 >= destruction):
      return [] # empty list

   # else generate count according to stats
   count = random.lognormvariate(REOPEN_MU,REOPEN_SIGMA)
   count = count - 1 # Fix result of lognorm to the dicrete pattern
   count = int(count)
   if (count <= 0):
      return [] # empty list

   if (destruction > creation+REOPEN_MAXTIME): # clip reopen times, see
      destruction = creation+REOPEN_MAXTIME # constant description for why

   l = []
   for i in range(count):
      # Distribution of reopens is skewed towards write time
      l.append(random.triangular(creation+1,destruction,creation+1))

   l.sort() # earliest to latest
   return l

def genFileSize():
   '''Returns a randomly-generated size in bytes for the file,
      according to the FILESIZE_* constants'''
   return random.lognormvariate(FILESIZE_MU,FILESIZE_SIGMA)

def genDeletionTime(now):
   '''Determines a random time between now and infinity when a file
      will be deleted, following the statistics from the LIFETIME_*
      constants. now and the return value are time_t.'''

   # Will the file be deleted at all?
   if (random.random() < LIFETIME_NO_DEL):
      return TIME_FOREVER

   # Will the file be deleted instantly?
   if (random.random() < LIFETIME_DEL_NOW):
      return now

   # Determine time using an exponential distribution:
   return now + random.expovariate(LIFETIME_LAMBDA)

############################################################
#######################################   Classes
############################################################

# Class to hold all information about a single file
# Data to hold (all generated by the constructor except creation time):
#  - Filename (unique)
#  - File size
#  - Last write time (note: sort by this value?)
#  - Creation time
#  - Deletion time (may be infinity; may be same as creation)
#  - List of reopens (may be empty)
#     - Each is a time_t between creation and deletion
class SimFile (object):

   def __init__(self, fdir, now):
```

```
        '''Creates a new SimFile, generating all attributies including
        lifetime info, filename, creating the filename and handle. Takes
        in the directory to create the file in, and the current time_t'''
        self.creTime = now

        self.delTime = genDeletionTime(now)
        self.size = genFileSize()
        self.reopens = genReopens(self.creTime,self.delTime)
        self.execP = False # created without execute perm

        # generate name for the file, containing a lot of write time info
        # <creation_time_t>-to-<deletion_time_t>-reopens-<x>-...
        self.fname = "%s/%d-to-%d-reps-%d"% \
                 (fdir,self.creTime,self.delTime,len(self.reopens))

        # Cancel due to file name limits
        #for i in self.reopens:     # append reopen times to filename
        #   self.fname = "%s-%d"%(self.fname,i)

        # check for name collisions
        while (os.path.exists(self.fname)):
            self.fname = "%s_%s"%\
                (self.fname,random.choice('abcdefghijklmnopqrstuvwxyz'))

        # Throw the exceptions if IO fails so the caller can delete this
        # object and/or stop/pause the simulation
        writeSimFile(self.fname,self.size)
        self.lastWrite = now

    def checkNoEvents(self):
        '''Returns True if this file has no future events and will stay
        in its current state forever.'''

        if (self.delTime == TIME_FOREVER and len(self.reopens) == 0):
            return True
        else:
            return False

    def checkReopen(self, now):
        '''Checks to see if a re-open and write should happen to this file
        in this pass. Returns true if this file WAS written by this func'''
        if (len(self.reopens) == 0):
            return False
        else:
            if (self.reopens[0] <= now):
                newSize = genFileSize()
                writeSimFile(self.fname,newSize) # rewrite file
                self.size = newSize
                self.reopens.pop(0) # remove this reopen
                return True
        return False

    def checkDelete(self, now):
        '''Checks to see if a re-open or delete should happen to this file
        in this pass. Returns true if this file WAS deleted by this func'''
        if (self.delTime <= now):
            os.remove(self.fname)
            return True
        return False

    def toggleMeta(self):
        '''Toggles a metadata component of the file (e.g. execute bit)
```

```
      to simulate a metadata / set attribute operation'''
      if (self.execP):
         os.chmod(self.fname,stat.S_IRUSR|stat.S_IWUSR)
         self.execP = False
      else:
         os.chmod(self.fname,stat.S_IRUSR|stat.S_IWUSR|stat.S_IXUSR)
         self.execP = True

##############################################################
#######################################   Globals / Init
##############################################################

# Initialisation
fileList = list() # List of all file objects

# Directory to put the simulated file activity
try:
   workDir = sys.argv[1]
except:
   sys.exit("You must specify a working directory on the command line")
if (not os.path.isdir(workDir)):
   sys.exit("You must specify an existing working directory")
os.chdir(workDir)

# create large file name
lfname = "large-append-file-%s"%(time.time())

##############################################################
#######################################   Main Loop
##############################################################

# Main loop begins here
longPassStreak = 0
while True:
   passStart = time.time() # so we can use this for time since epoch
   now = passStart
   passWrites = 0

   # Write to large file on each pass.
   appendSimFile(lfname,genFileSize())
   passWrites += 1

   # Write to old files which have a re-open <= now
   for i in fileList:
      if (i.checkReopen(now)):
         passWrites += 1
   passReopens = passWrites # more writes to come

   # Generate new files to make up to write rate
   targetWriteRate = randomPoisson(BASE_WRITE_RATE)

   while (targetWriteRate > passWrites):
      fileList.append(SimFile('./',now)) # create new simfile
      passWrites += 1

   # Generate metadata (set attrib) writes
      # Set GID To random value
   targetMetaRate = randomPoisson(BASE_METADATA_RATE)
   passMetas = 0
   if (len(fileList) > 0):
      while (passMetas < targetMetaRate):
         random.choice(fileList).toggleMeta()
```

```
            # Note: can't use rand.sample in case metarate > len(filelist)
            passMetas += 1

        # Delete files for deletion time <= now
        # re-fetch the value of now in case we have taken a long time this
        # pass
        now = time.time()
        rmlist = list()
        delPass = 0
        for i in fileList:
            if (i.checkDelete(now)):
                rmlist.append(i)  # remove this items from the list later
                delPass += 1
            else:                 # See if we ever need to check again
                if (i.checkNoEvents()):
                    rmlist.append(i)  # we can ignore these files forever
        for i in rmlist:
            fileList.remove(i)

        passEnd = time.time()
        passTime = passEnd-passStart
        # Note writes includes reopens and new files.
        print 'File %d, Write %d, Reopen %d, Attrib %d, Del %d, Ign %d, Time %f'%\
                  (len(fileList),targetWriteRate,passReopens,passMetas,\
                   delPass,len(rmlist)-delPass,passTime)

        if (passTime > 0 and passTime < PASS_TIME): # If pass took less than
            time.sleep(PASS_TIME-passTime)          # allocated, sleep for remainder
        elif (passTime > PASS_LIMIT):
            print('Pass at %d (%s) too long!'%(passEnd,time.ctime(passEnd)))
            longPassStreak += 1
            if (longPassStreak > LONG_PASS_STREAK_LIMIT):
                    sys.exit('Aborting at %d, too many passes over limit!'%passEnd)
        else:
            longPassStreak = 0

# No cleanup - leave disk/files in final state.
```

C.3 DATA COLLECTION SCRIPT

This script was used to collect ZFS metadata via ZDB for all experiments; see section 4.1.6 for details.

Listing 3: Data Collection Shell Script

```
#!/bin/bash

# Script to automatically collect ZDB data and timestamp it appropriately
# Configuration is NOT saved as it doesn't change once pool created

# Pool devices are from ada0pX to ada0pY
STARTDEV=4
ENDDEV=12

POOLNAME=poolv9r0
DATASET=filesim

SAVEDIR=/data-dltest2/$POOLNAME-control

cd $SAVEDIR
# wait after start of minute for disk writes to settle
sleep 3

for i in `seq $STARTDEV $ENDDEV`
do
zdb -P -uuuuuu -l /dev/ada0p$i > `date +%F_%T`-zdb-6u-$POOLNAME-ada0p$i.txt
done

zdb -P -mmmmmm $POOLNAME > `date +%F_%T`-zdb-6m-$POOLNAME.txt

zdb -P -bbbbbb -dddddd $POOLNAME/$DATASET > `date +%F_%T`-zdb-6b6d-$POOLNAME_$DATASET
    .txt

# For finish time logging
touch `date +%F_%T`-datasaved-$POOLNAME_$DATASET.txt

bzip2 `date +%F*`.txt
```

C.4 BP AND MACTIME EXTRACTOR

This script was used to convert file objects in ZDB output into XML, which could then be easily imported into Python, R and other software for analysis. The Block Pointers and the Modification/Access/Creation times were the primary elements of interest however other components of the file objects are also imported. See section 4.2 for details.

Listing 4: BP and MACtime extractor

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

'''
Dylan Leigh 2013

Script to scan for ZFS Objects with MACtimes (Modify/Access/Create
times) and block pointers in the input files and put all their data
into a big XML file.

Block pointers are subordinate to the Object they are in. Each MACTime
line is also its own element.

'''

import sys,os,tempfile,re,fileinput,time
import xml.etree.ElementTree as ET


################################################## CONSTANTS

# XXX: Important to STOP parsing block pointers after this line - this section
# of the output relists all the BPs in the pool (looking for leaked allocations)
ZDB6B6D_ENDOFOBJECTS = "Traversing all blocks to verify nothing leaked ..."

################################################## REGEXES

# For matching 2 column "name value" data without a colon (:)
# e.g.:
#       uid     0
#       gid     0
#       atime   Tue Apr 23 19:45:28 2013
#       mtime   Tue Dec  4 13:34:19 2012
# This tends to appear in 6b6d and ZAP objects; other structures use nvlists
# which have colons to seperate field: value
re2Column = re.compile(r'[\s]*([\w]+)[\s]+([\S]+.+)$')
# NOTE: Value in right column here can have nonalphanumeric characters, particularly
    /.*

# For matching block pointers:
# TODO: make the .* more specific? Gets all BPs on test pools...
```

```
reBlockPtr = re.compile(r'(DVA\[0\]=<.*) \[(.*)\] (.*) size=([abcdef\d]*)L/([abcdef\d
    ]*)P birth=([\d]*)L/([\d]*)P fill=([\d]*) cksum=(.*)')
# (all the DVAs)  (objype)   (all the flags) size (L)(P) birth(L)(P) fill() cksum*()
# DVA0 [DVA1] [DVA2] \[objtype\] assorted flags size=xxL/xxP birth=xxL/xxP fill=xx
    cksum=xx:xx:xx:
# NOTE: cksum not saved by the script

# For matching an individual DVA:
# DVA[(slot)]=<(vdev):(offset):(size)>
reSingleDVA = re.compile(r'DVA\[([\d])\]=<([abcdef\d]+):([abcdef\d]+):([abcdef\d]+)>'
    )
# Note slot is 0-2, a single character


################################################## 6b6d REGEXES

# TODO: BNF Descriptions of all to thesis.

# for first line of 'zdb -bbbbbb -dddddd'
# args: datasetname, idnumber, creationtransaction, size, numobjects,
# ( rootbps ), settype, (<singleargs> <valueargs>)
reDatasetLine = re.compile(r'^Dataset (\w+) \[ZPL\], ID (\d+), cr_txg (\d+), (\d+),
    (\d+) objects, rootbp (.*) [L0 DMU objset] (.*)$')
# FIXME: need [] for character classes or remove from regex below?
# XXX NOTE: Within each Dataset (including MOS) the object IDs are unique, but only
    within the dataset
# XXX: Last arg on line must be further expanded into properties
# XXX: rootbps can be expanded into up to three block pointers

# for header line of 6b6d output (headings only)
reObjectHeadings = re.compile(r'Object[\s]+lvl[\s]+iblk[\s]+dblk[\s]+dsize[\s]+lsize
    [\s]+%full[\s]+type')
# for actual header info following above regex
# XXX: must use -P so it outputs 16384 instead of 16K etc
# XXX: note %full uses (\d).(\d) - must be recombined
# XXX: Note "type" above is followed by inherited props - must be further seperated
#                                    obj        lvl         iblk        dblk
    dsize       lsize       full   full
reObjectHeaders = re.compile(r'[\s]?([\d]+)[\s]+([\d]+)[\s]+([\d]+)[\s]+([\d]+)[\s
    ]+([\d]+)[\s]+([\d]+)[\s]+([\d]+\.[\d]+)[\s]+([\S+].*)$')    #([\(\)=\w\s])$')
# (type [(inherit)] [(inherit)] ...)

################################################## FUNCTIONS

def saveXMLFile(etree):
   '''Saves the etree to an XML file.'''
   # proposed alternate function name: ETreePhoneHome()

   tmpfile = tempfile.NamedTemporaryFile(delete=False, # deleted by bz2
         prefix='zfs-data-', suffix='.xml')
   tfname = tmpfile.name

   # TODO: Use python's builtin BZfile to save compressed directly?

   print 'Saving data to %s ...'%tfname
   etree.write(tmpfile, method='xml')
   tmpfile.close(); # make sure its flushed before compressing
   print 'done.'

   print 'If you would like to view the data, use:'
   print 'xmllint -format %s | less'%(tfname)
```

```
def addDVAList(parentelem, strng, note=None):
   '''Parses a string for DVA[0]=<vd:offset:size> and adds the DVA
      pointers as a subelement to the passed element. If note is passed,
      add it as an attribute (e.g. 'rootbp' etc)'''

   # split string on spaces
   # each entry: DVA[(slot)]=<(vdev):(offset):(size)>

   for line in strng.split():
      mat = reSingleDVA.match(line)
      if (mat):
         dvaelem = ET.SubElement(parentelem, 'DVA')
         dvaelem.set('slot',mat.group(1)) # why does this not need
                                          # string conversion when the
                                          # UB one does?

         elem = ET.SubElement(parentelem, 'vdev')
         elem.text = mat.group(2)
         elem = ET.SubElement(parentelem, 'offset')
         elem.text = mat.group(3)
         elem = ET.SubElement(parentelem, 'size')
         elem.text = mat.group(4)
      # else not a DVA, ignore and continue

def addBlockPointer(parentelem,rematch, note=None):
   '''Parses a regex-match for a block pointer and adds all the XML as a
   subelement to the passed element. If note is passed, add it as an
   attribute (e.g. 'rootbp' etc)'''

   bpelem = ET.SubElement(parentelem, 'blockpointer')
   if (note):
      bpelem.set('note', note)

   # TODO FIXME: must check that BPs in all forms of RAIDZ parse
   # RAIDZ MAY HAVE AN ASIZE ENTRY AS WELL AS L AND P !!

   # Several sub-parts:
   #  -  DVAs (one to three)           - all in group 1
   #  -  object type (not always ??)   - group 2
   #  -  set of flags (one word each)  - all in group 3
   #  -  size=xL/xP birth=xL/xP        - groups 4,5,6,7
   #  -  fill=x cksum=x                - groups 8,9

   # Parse DVA - may be 1, 2 or 3
   # e.g. DVA[0]=<vdev:offset:size>
   elem = ET.SubElement(bpelem, 'dvas')
   addDVAList(elem, rematch.group(1), note)

   # Save objtype, flags (as one big string in <flags>)
   elem = ET.SubElement(bpelem, 'objecttype')
   elem.text = rematch.group(2)

   # NOTE: Some of these are not "flags" as such but representations of
   # an enumerated integer type (e.g. checksum type, compression type)
   # TODO: Seperate out some of the flags?
   flagElem = ET.SubElement(bpelem, 'flags')
   flagElem.text = rematch.group(3)

   # Save L and P size, L and P birth (VERY important)
   elem = ET.SubElement(bpelem, 'size')
   elem.set('sizetype', 'l')
   elem.text = rematch.group(4)
```

```
    elem = ET.SubElement(bpelem, 'size')
    elem.set('sizetype', 'r')
    elem.text = rematch.group(5)

    elem = ET.SubElement(bpelem, 'birth')
    elem.set('birthtype', 'l')
    elem.text = rematch.group(6)
    elem = ET.SubElement(bpelem, 'birth')
    elem.set('birthtype', 'r')
    elem.text = rematch.group(7)

    # Save fill and ignore cksum for now
    fillElem = ET.SubElement(bpelem, 'fill')
    fillElem.text = rematch.group(8)

def parse6b6d(text, baseElem):
    '''Parses 6b6d data and saves object header, MACtime, path, size, gen and
       block pointer data in a <object> subelement of baseElem'''

    # TODO: Save segment data for space map analysis?
    # TODO: Save bonus buffer header
    # TODO: Save flags?
    # break up inherits TODO

    ### Basic pseudocode:
    # if (lastlinewasheadings):
    #    if mat(headers)
    #    else WARN
    # If match (newobjectheadings): start new object XML element
    #     NEXT line - match objectheaders
    #   TODO: what to do about bonus buffer header? new regex? FIXME
    # If match (2 col):
    #     If (mat (atime|mtime|ctime|crtime) : add a timestamp element with a raw
        time_t
    #     If (gen|path|size): add as single element
    # IF match blocpointer: add blockpointer

    currObjElem = None
    lastWasHeadings = False # flag for object headings which mark a new object

    for line in text:
        if (lastWasHeadings):
            lastWasHeadings = False
            mat_headers = reObjectHeaders.search(line)
            if (mat_headers):
                currObjElem = ET.SubElement(baseElem, 'object')
                currObjElem.set('objectid', mat_headers.group(1))
                currObjElem.set('lvl', mat_headers.group(2))
                currObjElem.set('iblk', mat_headers.group(3))
                currObjElem.set('dblk', mat_headers.group(4))
                currObjElem.set('dsize', mat_headers.group(5))
                currObjElem.set('lsize', mat_headers.group(6))
                currObjElem.set('percentfull', mat_headers.group(7))
                # break up inherits TODO
                currObjElem.set('typeandinherit', mat_headers.group(8))
                continue
            else:
                print 'WARN: Expected new Object headers, got %s'%line
                continue
        elif (ZDB6B6D_ENDOFOBJECTS in line):
            break; # No more objects in output
        else: # do regex matches
```

```
            if (currObjElem is not None): # skip if not within an object
               mat_bp = reBlockPtr.search(line)  # match value to BP regex
               if (mat_bp):
                  addBlockPointer(currObjElem,mat_bp) # add to given element
                  continue
            # headings which mark a new object
            mat_headings = reObjectHeadings.search(line)
            if (mat_headings):
               currObjElem = None
               lastWasHeadings = True
               continue

            # 2 colum last as it is a weak match
            if (currObjElem is not None): # skip if not within an object
               mat_2c = re2Column.match(line)
               if (mat_2c):
                  #print mat_2c.group(1)
                  if (mat_2c.group(1) in ('atime','mtime','ctime','crtime')):
                     # XXX: If first character is = - this is probably an
                     # entry for a file which is NAMED "ctime".
                     if ('=' != mat_2c.group(2)[0]):
                        # its a timestap - generate time_t
                        # convert to int then str for XML serialisation
                        timet = str(int(time.mktime(time.strptime(mat_2c.group(2)))))
                        # save the name, value and time_t
                        newElem = ET.SubElement(currObjElem, mat_2c.group(1))
                        newElem.text=mat_2c.group(2)
                        newElem.set('rawtimet', timet)
                  elif (mat_2c.group(1) in ('path','size','gen')):
                     # its something else we are after - save the name and value
                     newElem = ET.SubElement(currObjElem, mat_2c.group(1))
                     newElem.text=mat_2c.group(2)
                  # else, some other field we arent interested in right now
                  continue

      return False # ignore whatever it is

   def parseZDBFields(zdbout):
      '''scans ZDB output for "<whitespace><field>:<value>" and returns a
      list of dicts {indent:, field:, value:} where indent is the amount
      (integer) of whitespace'''

      # NOTE: All the non-config items have special case lines not in the
      # above form. Provide a way to handle those here?

      ret=list()

      for line in zdbout:
         # skip empties
         if (0 == len(line)):
            continue
         if (line.isspace()):
            continue

         # count and trim whitespace at start
         trimmed= line.lstrip()
         ws= len(line) - len(trimmed)

         # split field:value and add tuple to return list
         splt = trimmed.partition(":")
         if (splt[1] == ':'):
            # split succeeded
```

```
            ret.append({'indent':ws, 'field':splt[0], 'value':splt[2]})
         else:
            print 'WARN: Failed to parse %s'%line

   return ret

################################################# MAIN STARTS HERE

if (__name__ == '__main__'):

   # Create initial Element tree
   etroot = ET.Element('zfs6b6ddata')
   etree = ET.ElementTree(etroot)

   print 'Scanning files for ZFS Block pointers and saving to XML.'

   # parse all input
   parse6b6d(fileinput.input(sys.argv[1:]), etroot)

   # save data
   saveXMLFile(etree)

   # DEBUG: print data: ET.dump(etree)
   # end.
```

C.5 LARGE FILE TXG ANALYSIS SCRIPT

This script was used in the "large file" experiments (see section 4.3) to match birth TXG from level 0 Block Pointers in the large file to birth TXG from highest level Block Pointers in other files, allowing the modification time of individual blocks in the large file to be determined.

Listing 5: Large File TXG Analysis Script

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

'''
Dylan Leigh 2014

Script to match the birth TXG from '/large-append-file' to birth TXG of other
files so the time of modification of the matching blocks in the large file can
be determined.

Outputs only the number of matches for results/analysis (number of entries in
the lists), throws away the lists with the data.

'''

import sys,os,tempfile,re,fileinput,time


################################################## GLOBALS

largefileTXG = list()
smallfileTXG = list()

################################################## CONSTANTS

# XXX: Important to STOP parsing block pointers after this line - this section
# of the output relists all the BPs in the pool (looking for leaked allocations)
ZDB6B6D_ENDOFOBJECTS = "Traversing all blocks to verify nothing leaked ..."

################################################## REGEXES

# For matching 2 column "name value" data without a colon (:)
# e.g.:
#       uid     0
#       gid     0
#       atime   Tue Apr 23 19:45:28 2013
#       mtime   Tue Dec  4 13:34:19 2012
# This tends to appear in 6b6d and ZAP objects; other structures use nvlists
# which have colons to seperate field: value
re2Column = re.compile(r'[\s]*([\w]+)[\s]+([\S]+.+)$')
# NOTE: Value in right column here can have nonalphanumeric characters, particularly
    /.*

# For matching block pointers:
```

```
# TODO: make the .* more specific? Gets all BPs on test pools...
reBlockPtr = re.compile(r'(DVA\[0\]=<.*) \[(.*)\] (.*) size=([abcdef\d]*)L/([abcdef\d
    ]*)P birth=([\d]*)L/([\d]*)P fill=([\d]*) cksum=(.*)')
# (all the DVAs)  (objype)   (all the flags) size (L)(P) birth(L)(P) fill() cksum*()
# DVA0 [DVA1] [DVA2] \[objtype\] assorted flags size=xxL/xxP birth=xxL/xxP fill=xx
    cksum=xx:xx:xx:
# NOTE: cksum not saved by the script

# For matching an individual DVA:
# DVA[(slot)]=<(vdev):(offset):(size)>
reSingleDVA = re.compile(r'DVA\[([\d])\]=<([abcdef\d]+):([abcdef\d]+):([abcdef\d]+)>'
    )
# Note slot is 0-2, a single character


################################################## 6b6d REGEXES

# TODO: BNF Descriptions of all to thesis.

# for first line of 'zdb -bbbbbb -dddddd'
# args: datasetname, idnumber, creationtransaction, size, numobjects,
# ( rootbps ), settype, (<singleargs> <valueargs>)
reDatasetLine = re.compile(r'^Dataset (\w+) \[ZPL\], ID (\d+), cr_txg (\d+), (\d+),
    (\d+) objects, rootbp (.*) [L0 DMU objset] (.*)$')
# FIXME: need [] for character classes or remove from regex below?
# XXX NOTE: Within each Dataset (including MOS) the object IDs are unique, but only
    within the dataset
# XXX: Last arg on line must be further expanded into properties
# XXX: rootbps can be expanded into up to three block pointers

# for header line of 6b6d output (headings only)
reObjectHeadings = re.compile(r'Object[\s]+lvl[\s]+iblk[\s]+dblk[\s]+dsize[\s]+lsize
    [\s]+%full[\s]+type')
# for actual header info following above regex
# XXX: must use -P so it outputs 16384 instead of 16K etc
# XXX: note %full uses (\d).(\d) - must be recombined
# XXX: Note "type" above is followed by inherited props - must be further seperated
#                                    obj        lvl         iblk        dblk
    dsize       lsize       full   full
reObjectHeaders = re.compile(r'[\s]?([\d]+)[\s]+([\d]+)[\s]+([\d]+)[\s]+([\d]+)[\s
    ]+([\d]+)[\s]+([\d]+)[\s]+([\d]+\.[\d]+)[\s]+([\S+].*)$')    #([\(\)=\w\s])$')
# (type [(inherit)] [(inherit)] ...)

################################################## FUNCTIONS

def birthTXGofBP(rematch):
   '''Parses a regex-match for a block pointer and returns the birth TXG'''

   # Several sub-parts:
   #  -  DVAs (one to three)           - all in group 1
   #  -  object type (not always ??)   - group 2
   #  -  set of flags (one word each)  - all in group 3
   #  -  size=xL/xP birth=xL/xP        - groups 4,5,6,7
   #  -  fill=x cksum=x                - groups 8,9

   # XXX: may need this later?
   #objtype = rematch.group(2)

   birthl = rematch.group(6)
   #birthp = rematch.group(7)

   return birthl;
```

```
def parse6b6d(text):
   '''Parses 6b6d data and saves lists of large-file and other-file
TXGs in the global lists.'''

   ### Basic pseudocode:
   # if (lastlinewasheadings):
   #    if mat(headers)
   #    else WARN
   # If match (newobjectheadings): start new object XML element
   #     NEXT line - match objectheaders
   # If match (2 col):
   #     If (mat (atime|mtime|ctime|crtime) : add a timestamp element with a raw
       time_t
   #     If (gen|path|size): add as single element
   # IF match blocpointer: add blockpointer

   lastWasHeadings = False # flag for object headings which mark a new object
   objectid = None
   objTypeInherit = None

   for line in text:
      if (lastWasHeadings):
         # New object headers
         lastWasHeadings = False
         mat_headers = reObjectHeaders.search(line)
         if (mat_headers):
            objectid = int(mat_headers.group(1))
            #objLevel = mat_headers.group(2) # XXX: Not used at this stage
            objTypeInherit = mat_headers.group(8)
            continue
         else:
            print 'WARN: Expected new Object headers, got %s'%line
            continue
      elif (ZDB6B6D_ENDOFOBJECTS in line):
         break; # No more objects in output
      else:
         if (objectid is not None): # skip if not within an object
            if (objTypeInherit == 'ZFS plain file (K=inherit) (Z=inherit)'): # not
                the directory
               mat_bp = reBlockPtr.search(line)  # match value to BP regex
               if (mat_bp):
                  if (objpath.startswith('/large-append-file')): # for large file
                      only
                     largefileTXG.append(int(birthTXGofBP(mat_bp)))
                  else:
                     smallfileTXG.append(int(birthTXGofBP(mat_bp)))
                  continue
         # headings which mark a new object
         mat_headings = reObjectHeadings.search(line)
         if (mat_headings):
            objectid = None
            objLevel = None
            objTypeInherit = None
            lastWasHeadings = True
            continue

         # 2 colum last as it is a weak match
         if (objectid is not None): # skip if not within an object
            mat_2c = re2Column.match(line)
            if (mat_2c):
```

```
                # XXX: Keep this in case we want to get the timestamp
                # itself later. For now we are just matching TXG.
                #
                #print mat_2c.group(1)
                #if (mat_2c.group(1) in ('atime','mtime','ctime','crtime')):
                #   # XXX: If first character is = - this is probably an
                #   # entry for a file which is NAMED "ctime".
                #   if ('=' != mat_2c.group(2)[0]):
                #      # its a timestap - generate time_t
                #      # convert to int then str for XML serialisation
                #      timet = str(int(time.mktime(time.strptime(mat_2c.group(2)))))
                #      # save the name, value and time_t
                #      newElem = ET.SubElement(currObjElem, mat_2c.group(1))
                #      newElem.text=mat_2c.group(2)
                #      newElem.set('rawtimet', timet)
                #elif (mat_2c.group(1) in ('path','size','gen')):

                if (mat_2c.group(1) in ('path',)):
                   objpath = mat_2c.group(2)
                # else, some other field we arent interested in right now
                continue

   return False # ignore whatever it is

def parseZDBFields(zdbout):
   '''scans ZDB output for "<whitespace><field>:<value>" and returns a
   list of dicts {indent:, field:, value:} where indent is the amount
   (integer) of whitespace'''

   # NOTE: All the non-config items have special case lines not in the
   # above form. Provide a way to handle those here?

   ret=list()

   for line in zdbout:
      # skip empties
      if (0 == len(line)):
         continue
      if (line.isspace()):
         continue

      # count and trim whitespace at start
      trimmed= line.lstrip()
      ws= len(line) - len(trimmed)

      # split field:value and add tuple to return list
      splt = trimmed.partition(":")
      if (splt[1] == ':'):
         # split succeeded
         ret.append({'indent':ws, 'field':splt[0], 'value':splt[2]})
      else:
         print 'WARN: Failed to parse %s'%line

   return ret

################################################## MAIN STARTS HERE

if (__name__ == '__main__'):

   print 'Scanning files for ZFS BP TXG from large and small files.'

   # parse all input
```

```
parse6b6d(fileinput.input(sys.argv[1:]))

# sets for unique listings and matches
largeTXGset = set(largefileTXG)
smallTXGset = set(smallfileTXG)
matches = largeTXGset.intersection(smallTXGset) # returns a new set

# printouts
print 'Large File TXG (%d total, %d unique)'%(len(largefileTXG),len(largeTXGset))
#print largefileTXG
print 'Small File TXG (%d total, %d unique)'%(len(smallfileTXG),len(smallTXGset))
#print smallfileTXG
print 'Large file TXG with a small TXG match: %d'%(len(matches),)
#print matches
```

COLOPHON

This document was written using the LyX[1] document processor, using the `classicthesis`[2] document class by André Miede. The typesetting was performed using the TeX Live[3] system. The JabRef[4] and BibTeX[5] bibliographic database software was used to store reference information and typeset citations.

Page size is A4 with a 4cm left margin and 2.5cm top, bottom and right margins . Body text uses the freely licensed Nimbus Roman No9 L[6] variant of the proprietary Times New Roman typeface, with line spread of 1.9.

Diagrams were produced using the Dia[7] software and the Gnumeric[8] spreadsheet software was used to tabulate data and produce charts.

1 http://www,lyx.org
2 http://code.google.com/p/classicthesis/
3 http://tug.org/texlive/
4 http://jabref.sourceforge.net/
5 http://www.ctan.org/pkg/bibtex
6 http://www.urwpp.de/english/home.html
7 https://wiki.gnome.org/Apps/Dia/
8 http://www.gnumeric.org/