Pseudo-Random Access Compressed Archive for Security Log Data

Submitted in partial fulfilment
of the requirements of the degree of

Master of Science

of Rhodes University

Johannes Jurgens Radley

Grahamstown, South Africa
September 2015

Abstract

We are surrounded by an increasing number of devices and applications that produce a huge quantity of machine generated data. Almost all the machine data contains some element of security information that can be used to discover, monitor and investigate security events.

The work proposes a pseudo-random access compressed storage method for log data to be used with an information retrieval system that in turn provides the ability to search and correlate log data and the corresponding events. We explain the method for converting log files into distinct events and storing the events in a compressed file. This yields an entry identifier for each log entry that provides a pointer that can be used by indexing methods.

The research also evaluates the compression performance penalties encountered by using this storage system, including decreased compression ratio, as well as increased compression and decompression times.

Acknowledgements

I would like to express my deepest gratitude to my supervisors Dr. Karen Bradshaw and Prof. Barry Irwin for their guidance, infinite patience and support.

I would like to thank my wife Debbie and two sons, Chris and Abrie, for their support, understanding and love during the writing of this work.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background Information

We are surrounded by an increasing number of devices and applications that produce a huge quantity of machine generated data (Gantz and Reinsel, 2010, 2012; Donovan, 2012; Press, 2014). This consists in the greater part of application and system logs and metric information.

Almost all the machine data contains security data, from explicit security data like firewall and access logs to discrete data like network usage and web access logs which can be used for heuristic security analysis and forensic investigation.

It is impossible for a human to process the large volumes of machine generated data that are constantly generated. To address this problem, a number of solutions have been created to normalise the data and perform automated correlation and heuristic tests. The foremost among these are Security Information and Event Management (SIEM) systems.

While SIEM systems address a security monitoring need, they require a huge upfront normalisation of the log information and can be disrupted by even minor changes in the format of a single log event.

This limitation has led to the use of information retrieval (IR) techniques being applied to log data. A number of open-source solutions (ELSA[1], Kibana[2], Graylog2[3]) and pro-

[1]https://code.google.com/p/enterprise-log-search-and-archive/
[2]http://www.elasticsearch.org/overview/kibana/
[3]http://graylog2.org/

prietary products (Splunk[4], ArcSight Logger[5]) have appeared to manage and process machine data using IR methods.

1.2 Problem Statement

While there is a large body of literature devoted to IR and a number of products on the market, there is very little research available on the application of IR methods and techniques to machine generated data, especially log data.

The techniques and methods that store, index and search documents like web pages, books and articles, may not be optimal for large volumes of log data that need to be rapidly stored and indexed while being available for searching. Most IR systems work on a batch processing and optimisation cycle using techniques designed for natural languages to reduce the size of some structures. In most cases they do not put any emphasis on the temporal nature of the information being indexed, which is a major attribute of security-related machine generated data.

The large volumes of log data produced introduce their own challenges especially for storage. The main challenge is to create a storage with good compression while retaining the ability to have random access to the data.

1.3 Research Objectives

The aim of the research is to describe a read-only random access compressed archive that can be used as a storage for heterogeneous text security log data. The entry identifier produced for each entry can be used by an index as a pointer to the original data, making it suitable for use with IR indexing systems. This work will concentrate on evaluating the impact of compression methods and block sizes on the archiving system using selected sample logs.

1.4 Delineation and Limitations

The research is affected by the following delineations and limitations.

[4]http://www.splunk.com/
[5]http://www8.hp.com/us/en/software-solutions/arcsight-logger-log-management/

1.4.1 Log Sources and Compression Ratios

Compression ratios are subject to the raw data being compressed. A single log type is usually very repetitive in nature and will have a much smaller dictionary resulting in better compression.

Since only a limited number of log types are tested, the results may be different when applied to log types outside of those used in the experiments. The log samples are also heavily biased towards security type logs.

This study uses logs from a number of different sources including logs from: PFSense firewall, Dovecot/Postfix mail systems, Squid proxy logs, PCAP logs, Apache web access and Windows security events written to text files.

1.4.2 Compression Methods

For the purpose of this work only a small number of the possible compression methods and compression command line tools are tested. The methods chosen however are sufficient to perform the evaluation of the implementation discussed below.

1.5 Document Structure

The work consists of a number of chapters starting with the literature review and ending with the conclusion. Chapter 2 provides the reader with background information on methods and structures used in this work.

Chapter 3 explains the process of breaking log file entries into individual events and storing those events in an archive. The chapter also defines the structures used in the archive.

Chapter 4 evaluates a number of compression methods, using different block sizes, for selected security-related log samples. Chapter 5 tests the selected compression methods using the archive system described in Chapter 3.

Chapter 6 contains the contributions of this work and a discussion of suggested future research.

Chapter 2

Literature Review

This chapter discusses inverted files used by IR systems, compression methods, compressed random access files, compressed full text indexing structures and other topics used in the this work.

2.1 Inverted Index

Inverted index, sometimes called an inverted file, is one of the main data structures used in almost all IR systems (Büttcher, Clarke, and Cormack, 2010, p. 33). In simple terms, an inverted file provides a link between the individual terms in the text data and their location within the storage mechanism used to hold the text data (Frakes and Baeza-Yates, 1992, p. 28) (Zobel and Moffat, 2006) (Büttcher *et al.*, 2010, p. 33).

To explain the concept, a simple example is used. Table 2.1 represents seven phrases in a flat file with the start offset of each phrase. This is the storage or archive of the original text in the IR system.

Table 2.2 is a mapping created between a record number or entry identifier and the offset of the phrases in the file. This mapping creates an abstraction layer between the index and storage allowing for very large implementations and dynamic moving of the underlying data.

Each phrase is then split into keywords or terms and added to the inverted index with its record number and start position in the phrase. Table 2.3 shows the sorted final result after processing of all the phrases.

Table 2.1: Phrase flat file consisting of seven phrases and their offsets in bytes

Offset in File	Phrase
0	`aaa eee iii ggg bbb fff hhh ccc bbb`
35	`ggg ggg aaa hhh iii ccc aaa aaa`
66	`aaa iii ddd ggg fff jjj`
89	`iii hhh ggg iii ddd`
108	`bbb ggg aaa ddd ccc bbb iii`
135	`ggg iii jjj ggg eee`
154	`bbb eee hhh hhh ddd ccc`

Table 2.2: Mapping between record number and offset of phrases

Record Number	Offset
1	0
2	35
3	66
4	89
5	108
6	135
7	154

Table 2.3: Inverted list for terms in phrases

Term	Term Count	Inverted List
`aaa`	6	(1:0),(2:8),(2:24),(2:28),(3:0),(5:8)
`bbb`	5	(1:16),(1:32),(5:0),(5:20),(7:0)
`ccc`	4	(1:28),(2:20),(5:16),(7:20)
`ddd`	4	(3:8),(4:16),(5:12),(7:16)
`eee`	3	(1:4),(6:16),(7:4)
`fff`	2	(1:20),(3:16)
`ggg`	8	(1:12),(2:0),(2:4),(3:12),(4:8),(5:4),(6:0),(6:12)
`hhh`	5	(1:24),(2:12),(4:4),(7:8),(7:12)
`iii`	7	(1:8),(2:16),(3:4),(4:0),(4:12),(5:24),(6:4)
`jjj`	2	(3:20),(6:8)

The inverted index now provides an efficient method to search for terms and combination of terms in the original phrases. For example, if the user searches for `eee` the inverted index will return phrases 1, 6 and 7.

It is further possible to do Boolean searches (Moffat and Zobel, 1996, p. 369–370) like AND, OR and NOT. Consider the search "`eee AND fff`". The inverted index retrieves 1, 6, and 7 of `eee` and 1 and 3 for `fff` giving a final list of 1, 3, 6 and 7. For the search "`eee NOT fff`", the inverted index again returns 1, 6, and 7 for `eee` and 1 and 3 for `fff`. Since both contain 1 this will be eliminated from the entry identifier list of `eee` yielding a final result of 6 and 7.

Although there are many structures and methods that can be used to represent the sorted array of terms, they all use some way of pointing to the original data, the entry identifier, document identifier or record number.

2.2 Random Access in Compressed Files

Compression is essential when working with large volumes of data. Unfortunately, compressed data files do not usually support random access to the contained data. In most cases to find data at offset n it is necessary to start at offset 0 and uncompress all data to offset n–1 before uncompressing and reading the data requested.

York (2001) suggests the use of blocks with an accompanying index that maps the block offset in the file. This method, however, is not suitable for doing random access updates as the new data may not compress to the exact same size as the old data block. The solution though is well suited to cases where the data is only written once to the file.

The idea proposed by York (2001) is not unique as a similar concept was already employed in **dictzip** (Faith, 1997) which uses the extra field extension in the **gzip** (Deutsch, 1996) specification (see Section 2.3.1.1 of the specification).

Figure 2.1 depicts the file structure proposed by York (2001) consisting of a file header and a variable number of sections. The sections contain clusters of data written as individual blocks to the file. This file structure is very complicated as it makes provision for random write access. The file format would be simpler if the data were only written once to the file allowing only for random read access after that. This limitation is part of the design as specified in the research objectives given above (Section 1.3).

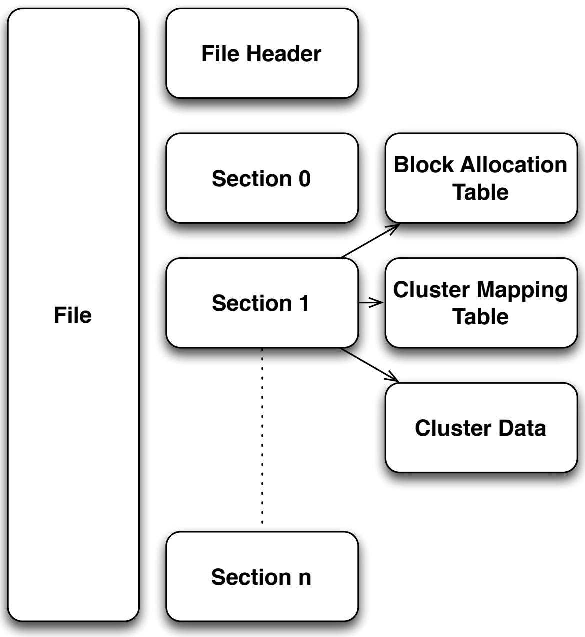


Figure 2.1: File structure of York's compressed random access file.

2.3 Compression

Compression aims to reduce the storage space required to store data by applying an encoding scheme of some kind. This work is only concerned with the group of algorithms that perform lossless data compression. Lossless means that the data after decompression is exactly the same as the original data before compression.

2.3.1 Huffman Coding

Huffman (1952) describes a method for constructing a binary tree based on the frequency of the occurrence of a symbol in a given corpus. The resulting binary tree yields shorter code lengths for more frequent symbols and longer code lengths for less frequent symbols. This results in a shorter encoding of the original corpus.

As an example, Table 2.4 shows six symbols each with their frequency of occurrence in the example corpus. The resulting Huffman binary tree is depicted in Figure 2.2. The code is then created by following the path from the top entry or root down to the symbol, with each left branch resulting in a `0` and right branch in a `1`. Table 2.4 clearly demonstrates that the higher the symbol frequency the shorter the code for the symbol.

Table 2.4: Huffman frequency example

Symbol	Frequency	Code
A	30	0
B	18	101
C	14	100
D	9	110
E	6	1111
F	4	1110

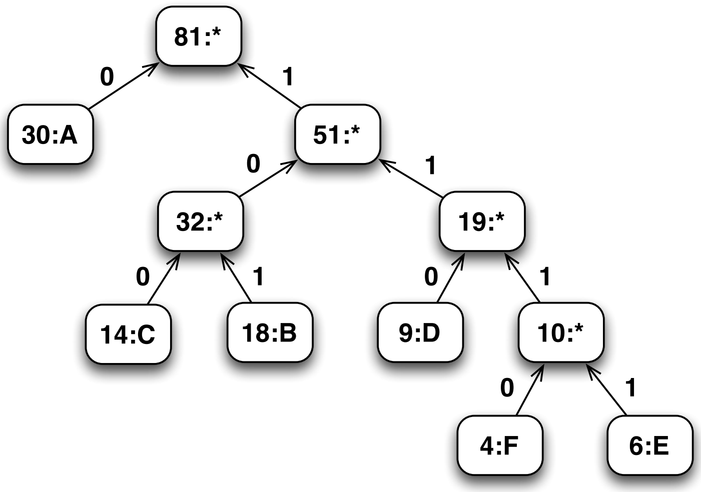


Figure 2.2: Example Huffman binary tree

Although this method dates back to 1952, both it and its variants are still used to complement other compression methods as described below.

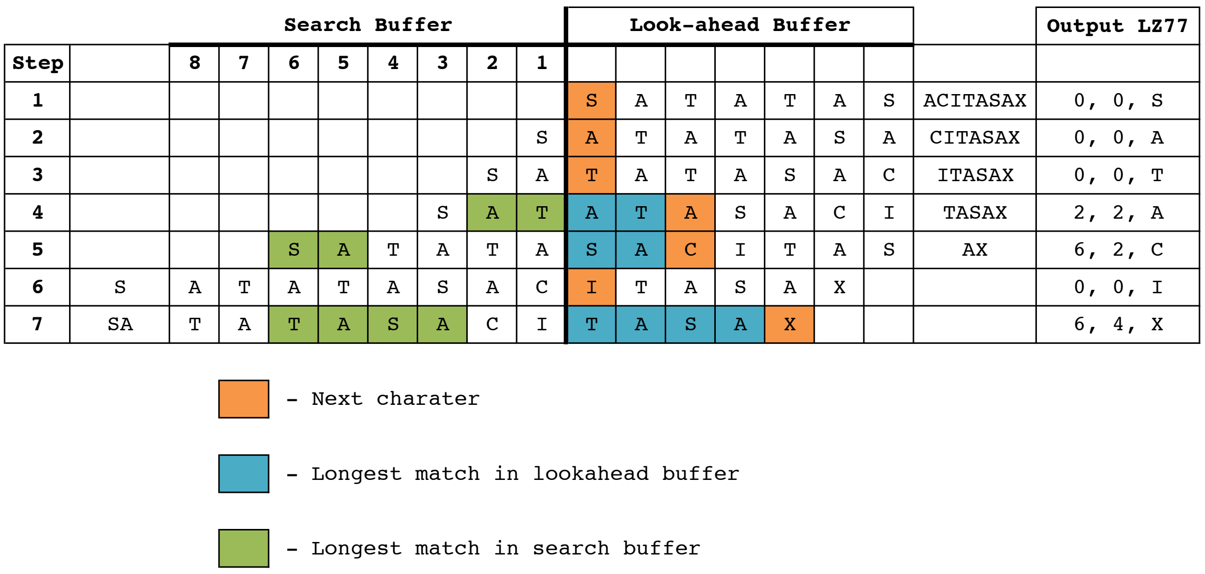
2.3.2 LZ77

While Huffman (1952) was concerned with compression of single symbols, Ziv and Lempel (1977) (LZ77) described a compression based on sequences of multiple symbols.

In their work, Ziv and Lempel (1977) introduced a sliding window compression algorithm consisting of a *search buffer* and a *lookahead buffer*. The *search buffer* acts as a dictionary and is used for searching duplicate occurrences of a sequence of symbols in the *lookahead buffer*, replacing such matches with a pointer to the previous occurrence.

Figure 2.3 details the process of LZ77 encoding using the string `SATATASACITASAX`. The algorithm attempts to match the longest sequence of symbols in the *lookahead buffer* to the first occurrence of the sequence in the *search buffer*, outputting a tuple consisting of the start offset within the *search buffer*, the length of the match and the next non matching symbol in the *lookahead buffer*. In cases where no match is found, an offset and length of zero is written to the output. After each step, the offset of the sliding window is advanced by the length of the match plus one.

The output is usually encoded to reduce the storage space required. For example in gzip, Huffman encoding is used to compress the output tuples.



		Search Buffer								Look-ahead Buffer								Output LZ77
Step		8	7	6	5	4	3	2	1									
1										S	A	T	A	T	A	S	ACITASAX	0, 0, S
2									S	A	T	A	T	A	S	A	CITASAX	0, 0, A
3								S	A	T	A	T	A	S	A	C	ITASAX	0, 0, T
4							S	A	T	A	T	A	S	A	C	I	TASAX	2, 2, A
5				S	A	T	A	T	A	S	A	C	I	T	A	S	AX	6, 2, C
6	S	A	T	A	T	A	S	A	C	I	T	A	S	A	X			0, 0, I
7	SA	T	A	T	A	S	A	C	I	T	A	S	A	X				6, 4, X

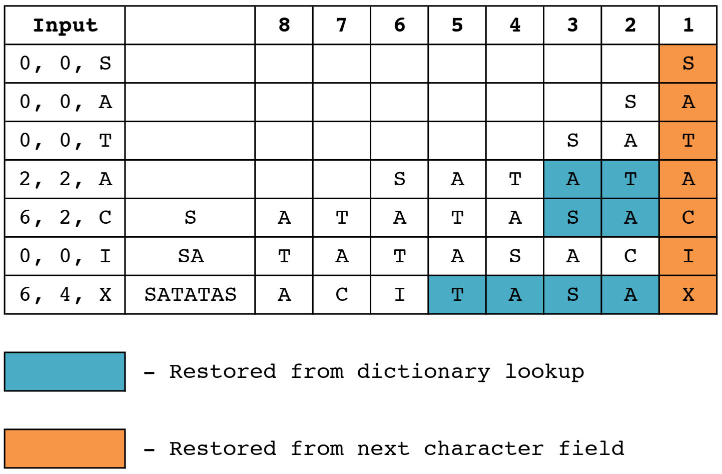
🟧 - Next charater

🟦 - Longest match in lookahead buffer

🟩 - Longest match in search buffer

Figure 2.3: LZ77 Encoding

Figure 2.4 illustrates the decoding process using the outputs from Figure 2.3 as its inputs. The decoding appends any matches defined in the input, as well as the next character symbol for that entry.



Input		8	7	6	5	4	3	2	1
0, 0, S									S
0, 0, A								S	A
0, 0, T							S	A	T
2, 2, A				S	A	T	A	T	A
6, 2, C	S	A	T	A	T	A	S	A	C
0, 0, I	SA	T	A	T	A	S	A	C	I
6, 4, X	SATATAS	A	C	I	T	A	S	A	X

🟦 - Restored from dictionary lookup

🟧 - Restored from next character field

Figure 2.4: LZ77 Decoding

GZIP, LZ4, LZO and LZMA are all variants of LZ77 with varying optimisation including the output formatting and sliding window size. The individual optimisations and their associated costs and benefits are beyond the scope of this work.

2.3.3 Burrows-Wheeler Transformation

Burrows and Wheeler (1994) introduced a reversible transformation algorithm that re-orders text so that it is easy to compress with other algorithms (Fenwick, 1996). This transformation algorithm forms the basis of the bzip2 compression method.

The algorithm is best explained with the use of an example. Taking the string `abracadabra`, an end-of-line marker (`$`) is added at the end resulting in the string `abracadabra$`. Then, each rotation of the string is sorted and the transformation output is created from the last character of each rotation resulting in `ard$rcaaaabb`. This process is illustrated in Table 2.5. Even this short sample string produces clumps of letters like the run of four `a`'s and two `b`'s at the end of the transformation output.

Table 2.5: Burrows-Wheeler Transformation example

Rotated List	Sorted List	Last Column
`abracadabra$`	`$abracadabra`	`a`
`bracadabra$a`	`a$abracadabr`	`r`
`racadabra$ab`	`abra$abracad`	`d`
`acadabra$abr`	`abracadabra$`	`$`
`cadabra$abra`	`acadabra$abr`	`r`
`adabra$abrac`	`adabra$abrac`	`c`
`dabra$abraca`	`bra$abracada`	`a`
`abra$abracad`	`bracadabra$a`	`a`
`bra$abracada`	`cadabra$abra`	`a`
`ra$abracadab`	`dabra$abraca`	`a`
`a$abracadabr`	`ra$abracadab`	`b`
`$abracadabra`	`racadabra$ab`	`b`

To reverse the transformation, a table is created with the output from the last column of the transformation step. In each iteration the table is sorted and the output column is prepended again. This continues until the process has been repeated for the number of characters in the string. The original string is the string that ends with the end-of-string marker. Table 2.6 illustrates the process.

Table 2.6: Inverse Burrows-Wheeler transformation example

Start	Step 1	Step 2	Step 3	Step 10	Step 11	Step 12
`a`	`a$`	`a$a`	`a$ab`	`a$abracadab`	`a$abracadabr`	`$abracadabra`
`r`	`ra`	`ra$`	`ra$a`	`ra$abracada`	`ra$abracadab`	`a$abracadabr`
`d`	`da`	`dab`	`dabr`	`dabra$abrac`	`dabra$abraca`	`abra$abracad`
`$`	`$a`	`$ab`	`$abr`	`$abracadabr`	`$abracadabra`	`abracadabra$`
`r`	`ra`	`rac`	`raca`	`racadabra$a`	`racadabra$ab`	`acadabra$abr`
`c`	`ca`	`cad`	`cada`	`cadabra$abr`	`cadabra$abra`	`adabra$abrac`
`a`	`ab`	`abr`	`abra`	`abra$abraca`	`abra$abracad`	`bra$abracada`
`a`	`ab`	`abr`	`abra`	`abracadabra`	`abracadabra$`	`bracadabra$a`
`a`	`ac`	`aca`	`acad`	`acadabra$ab`	`acadabra$abr`	`cadabra$abra`
`a`	`ad`	`ada`	`adab`	`adabra$abra`	`adabra$abrac`	`dabra$abraca`
`b`	`br`	`bra`	`bra$`	`bra$abracad`	`bra$abracada`	`ra$abracadab`
`b`	`br`	`bra`	`brac`	`bracadabra$`	`bracadabra$a`	`racadabra$ab`

2.4 Compressed Full-Text Indexes

Compressed Full-Text Indexes (Navarro and Makinen, 2007) or Self-Indexing Inverted Files (Moffat and Zobel, 1996) are indices constructed in such a way that the original document can be reconstructed from the index file itself, thereby eliminating the need to retain the original raw data.

Although these methods represent a saving on disk storage for data that is available for searching, they are not suitable for an archival phase where logs are no longer available for searching, but must be retained for compliance and/or possible later re-indexing. This is due to the fact that the data forms an integrated part of the term index.

2.5 Summary

This chapter describes inverted index structure and the need for an entry identifier. The entry identifier links the terms in the inverted index to the original data.

The random access compressed file describe by York (2001) introduces the concept of breaking up the data into smaller blocks, then compressing each individual block. Accessing the data is faster as only the block containing the required data, must to be decompressed. This approach offers both a saving on storage requirements and speeds up retrieval time of the original data.

The remainder of the chapter focused on the foundations of data compression. In particular LZ77 and the Burrows Wheeler Transformation algorithms are described, as the compression methods used in this work are variants of the two algorithms.

The next chapter explains the proposed pseudo-random access compressed archive and processing of the log files.

Chapter 3

Process Flow and Pseudo-Random Access Compressed Archive

This chapter describes the process of event breaking from reading the initial log file through to writing the processed data to storage. This includes a description of the data structures used to create the Pseudo-Random Access Compressed Archive.

The chapter starts with timestamp extraction (Section 3.1), followed by the event breaking process (Section 3.2), and discusses the event structure, line breaking, line merging and adding metadata. Section 3.3 introduces the archive file format covering variable serialisation, header structure, event blocks and the tailing record.

3.1 Timestamp Extraction

Time is critical when working with log data. An event requires a time to qualify as an event, otherwise it is just a data point. As an example knowing that the user `John` logged into the system at `2014-04-14T10:14:32+0200` is far more valuable than knowing that at some unknown time `John` logged into the system.

Timestamp extraction from a log entry consists of two phases: identify the start of the timestamp and then parse the timestamp into its constituent parts. The next sections discuss the methods used for timestamp extraction and problem use cases.

3.1.1 Timestamp Prefix

The first aspect of timestamp extraction is to determine where the timestamp is located within the event. The part preceding the timestamp in the event is called the timestamp prefix.

Regular expressions are used to find the start of the timestamp in a log line. The following two samples obtained from the OSSEC[1] web site are used as examples to further explain this:

```
Jul 10 16:07:14 cisco2621 636: .Jul 10 15:58:56.590 EDT: %SEC-6-IPACCESSLOGP:
list 102 denied tcp 10.0.6.56(3067) -> 172.36.4.7(139), 1 packet
```

Listing 3.1: Cisco IOS log entry starting with a timestamp

In Listing 3.1, the timestamp is at the very start of the line and the timestamp prefix can be matched with the regular expression `^`. As ^ matches the start of the text being matched, it will always match a non-empty line.

```
4872: Dec 11 08:02:53.887 pst: %SEC-6-IPACCESSLOGP: list 100 denied udp
200.174.153.126(1028) -> 66.81.85.65(137), 1 packet
```

Listing 3.2: Cisco IOS log entry not starting with a timestamp

In Listing 3.2, the entry starts with the message identifier. The timestamp prefix can be matched with the following regular expression `\d+:\s`. The regular expression matches one or more digits (numbers between 0 and 9), a colon and a white space. This means that for some log formats, as in the example above, the prefix will need to be manually constructed.

The timestamp at the start of a line is a very common format. It is the format used in all the sample log files in this research. Extracts of the log formats considered in this research are given in Appendix B.

3.1.2 Timestamp Parsing

Timestamp parsing is the process of converting a sequence of ASCII characters, as found in the input data, to a 64-bit integer representing the number of microseconds elapsed

[1] http://ossec-docs.readthedocs.org/en/latest/log_samples/cisco/cisco_ios.html

since the Unix Epoch, 1 January 1970. In the developed system, the parsing is performed using the Poco C++ Library[2] DateTimeParser class. The format syntax for the DateTimeParser is very similar to that of the `strptime()` function available in most standard C++ libraries.

Many different formats are used to represent timestamps: some are based on standards like ISO–8601 (ISO, 2005) and RFC 3339 (Klyne and Newman, 2002), while others are based on what the software developers thought would be most appropriate.

Both Listings 3.1 and 3.2 utilise the same date format, which can be parsed using the following DateTimeParser format `%b %d %H:%M:%S`, where `%b` represents the abbreviated month, `%d` the numeric day of the month, `%H` hours, `%M` minutes and `%S` seconds.

In some formats, certain values are omitted. In the two listings above the timestamp format excludes the year, the timezone offset with daylight savings time calculated and the fraction of a second. The year and timezone are critical for accurate timestamp extraction. The fraction of a second can improve the precision of later correlations between events.

Missing information forces the use of assumptions. The assumptions in the case of a missing century or year is dependent on correct time synchronisation across all logging devices and application. The following sections elaborate on the assumptions and risk factors associated with each one.

3.1.3 Missing Century

Some timestamps only contain a two digit value representing the year, omitting the leading two digits that indicate the century. This is the problem that created the Y2K issue where a two digit year value became ambiguous when the century changed from `1900` to `2000`. An example of this format is `22/03/12`, where the `12` represents the year.

The easiest assumption to make is that the logs being processed were generated in the current century and the year can be completed by prepending a `20` to the year making it `2012`. This will be valid for log entries generated between `2000` and `2099` and is extensible.

The second assumption that can be used is that if the year is greater than the current year the century is the current century minus one. So if the current year was `2011` in the example above, the year part of the event, twelve, would be greater than the current year,

[2]http://pocoproject.org/

eleven. The century would be calculated by taking the current century 20 and deducting one from the current century giving the value 19 to prepend to the date. This results in the value 1912. Similarly the current year in the example is 2013 and thus the century value of the current year, 20, will be prepended to the year. This results in the value 2011.

The second assumption has a risk window where the absence of correct time synchronisation between log generating devices/applications and the archiving server can lead to incorrect assumptions at the rollover of the year.

Take the following as an example where the log server time is ahead of the time on the archiving system: the current time on the archiving server is 2013-12-31T23:59:40 and the log entry is 2014-01-01T00:00:50 (01/01/14 00:00:50). Following the second assumption, the timestamp extraction routine sees that the log entry year, 14 is greater than the current time on the server and therefore the routine will take the current century minus one and assign a timestamp of 1914-01-01T00:00:50 to the entry.

For the purposes of this work, we use the assumption that all the logs were generated between 2000-01-01 and 2099-12-31.

3.1.4 Missing Year

A missing year value from a timestamp presents a major obstacle in determining the correct timestamp of an event. Our first option is the use of some form of external metadata like the last modified date time of the file containing the logs or the date time encoded in a log file name (e.g., firewall_2014_03_12.log).

Using metadata in the file name is a very strong time anchor and should not complicate the parsing process unnecessarily as the file name only has to be parsed once and then passed to the parsing algorithm. This needs to be a configuration option for the specific source type. The last modified date as reported by the file system has several possible pitfalls, for example, some copy and archiving functions or options modify the date of the file to the date the action was executed.

The second option is to make an assumption that the logs are less than a year old. The algorithm for this subtracts the current year from the current timestamp. If the resulting time value is greater than the parsed time, the value one is deducted from the current century.

All the options fail in the case where the logs contain events spanning multiple years. For the purposes of this work, we assume that all the logs relate to the current year, where the year part is omitted from the timestamp.

3.1.5 Missing Timezone

A missing timezone can also affect the timestamp of an event. In Listing 3.3, for example, if the clock on the device was set to Coordinated Universal Time[3] (UTC) and the entry is parsed on an archiver with the clock set to South African Standard Time[4] (SAST), the parser will extract the time of the entry as `2004-05-18T02:11:03+02:00` instead of `2004-05-18T04:11:03+02:00`.

```
05/18/2004,02:11:03,Authen failed,bscorpio,punks,122.55.32.13,External DB user
invalid or bad password,,,15,10.27.3.1
```

Listing 3.3: Cisco Secure ACS log entry

When the search is initiated using the time range from `2004-05-18T04:00:00+02:00` to `2004-05-18T05:00:00+02:00`, the event will not be listed in the output as it is stored two hours in the past.

This also creates a time synchronisation problem that can affect the assumption stated as the solution in the previous two sections, Missing Century (Section 3.1.3) and Missing Year (Section 3.1.4).

3.2 Event Breaking

Event breaking is the process whereby a log file is read line-by-line and converted into a single entry. An entry can consist of more than one line, which is called a multi-line event. The entry is then enhanced by the addition of metadata to become a standalone event that can be stored in the archive. The following sections explain the event structure and each element of the event breaking process, followed by a summary of event breaking process flow (Section 3.2.6).

[3]http://www.timeanddate.com/time/aboututc.html
[4]http://www.timeanddate.com/library/abbreviations/timezones/africa/sast.html

3.2.1 Event Structure

Events are stored in an archive, in a key-value pair array, referred to as a field. The key is the field name and the value represents the field value.

This event data structure is used by the archiver to store the event data in memory during archiving and when retrieving events from the archive. Section 3.5 explains how the event structure is represented on disk.

Figure 3.1 depicts the event structure consisting of the `Event` class, which contains a vector of `EventField`'s. The `EventField` class represents the key-value pair of a field where the `name` property is the key and the `type` property indicates what type of data the field holds. The `numericValue` holds both `timestamp` and `numeric` value types, while the `stringValue` stores the `string` value type.

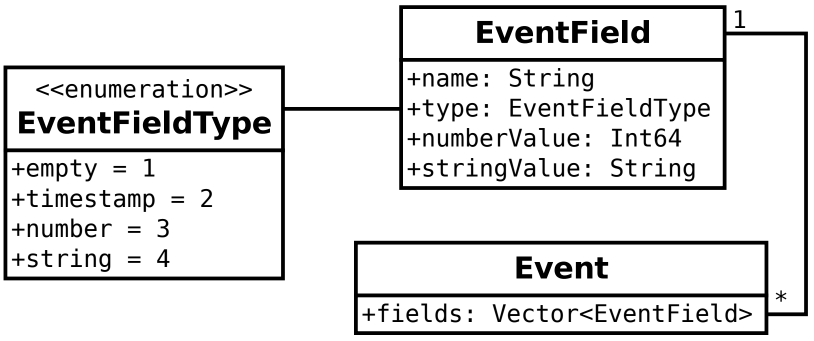


Figure 3.1: Structure of an event

3.2.2 Generic Fields

A number of generic fields are defined to store internal fields in the event data structure (Section 3.2.1). These fields, described in Table 3.2, are populated during the event breaking process.

These generic fields cover all the use cases for this implementation and no additional fields are created. Non-generic fields are only created when additional archive time extraction is done, which needs to be stored in the archive on disk.

To illustrate the generic fields, we use as an example the Squid proxy log entry (Listing 3.4) created on `2014-11-08T08:45:44+0200` in the log file `/var/log/access.log` on the server with the host name `proxy.example.conf` and archived one minute later.

Table 3.1: Generic fields for events

Field Name	Description
`_data`	Represents the original entry in the log file. A single or multiline event depending on the line merging settings (Section 3.2.4)
`_time`	Holds extracted timestamp of the event in Coordinated Universal Time (UTC) (Section 3.1).
`_tz`	Timezone offset as extracted from the entry (Section 3.1).
`_archivetime`	Coordinated Universal Time of when the event was added to the archive.
`_datatype`, `_source` and `_host`	Additional metadata added to an entry (See Section 3.7).

```
1380042813.978 29679 196.23.167.67 TCP_MISS/200 4629 CONNECT sites.google.com
:443 - DIRECT/155.232.240.59 -
```

Listing 3.4: Squid proxy log entry showing generic field values

Table 3.2: Example event with field values

Field Name	Field Value
`_data`	`1415429144 29679 196.23.162.64 TCP_MISS/200 4629 CONNECT sites.google.com:443 - DIRECT/155.232.240.59 -`
`_time`	`2014-11-08T06:45:44Z`
`_tz`	`120`
`_archivetime`	`2014-11-08T06:46:44Z`
`_datatype`	`proxy`
`_source`	`/var/log/access.log`
`_host`	`proxy.example.tld`

3.2.3 Line Breaking

The line breaking process starts by reading the log file and extracting single lines from the file. This implies that the work only uses text log files, as binary log files need specialised parsing to extract event information and is beyond the scoped objectives (Section 1.3).

Furthermore this work only focuses on text log files that end with either a carriage return (0x0D)(`\r`) and/or new line (0x0A)(`\n`) ASCII characters as this is the most common format for log files. This is largely due to them being human readable for troubleshooting at the command line, which is very common on Unix-based systems.

Line break detection is done using the regular expression `[\r\n]+`, which matches one or more carriage returns and/or new line characters. These lines serve as the input for the rest of the event breaking process, finally resulting in a complete event. Line breaking on its own does not define an event as with some log formats an event can consist of multiple lines, for example, Windows event logs (see Section B.2).

3.2.4 Merging Lines

Log entries can be contained in a single line or multiple lines. Aggregating the lines to construct an entry is called line merging.

Two user-configurable options can greatly assist with this process, but are not absolutely required. These options can simplify the basic line-merging decision-making algorithm.

The first option is specifying whether the log entries are single- or multi-line. If this option is set to single-line then each line is converted into an event with no further processing. Single-line events are predominant in the sample logs under consideration.

The second option is only used in the case of multi-line events, such as Windows event logs (Section B.2), and specifies whether a new event starts with a timestamp. The line merger collects lines until it encounters a line containing the timestamp as defined by the timestamp prefix and timestamp format. The previously collected lines will then form the whole entry and the new line containing the timestamp is the start of the next entry.

3.2.5 Metadata

Metadata is added to enhance the log event with additional information by the archiving system. The metadata adds both storage and processing overhead, but provides an increased search retrieval speed when performing searches on the data.

For example, assume that the firewall logs are stored in a file with the name `firewall.log`. The firewall logs can be stored in an archive with other types of logs, e.g. Windows Event, SSH and DHCP logs by the implementation. With metadata the search module only has to load the block containing entries with the metadata field *source* set to “firewall.log”.

Any blocks not loaded, because the *source* field is not set to “firewall.log”, will reduce the retrieval time as they do not have to be retrieved from disk or uncompressed. Table 3.2 shows the metadata for an example entry.

3.2.6 Event Breaking Process Flow

Having defined the process elements, we now describe the process flow of event breaking. Figure 3.2 depicts a flow chart describing the process flow.

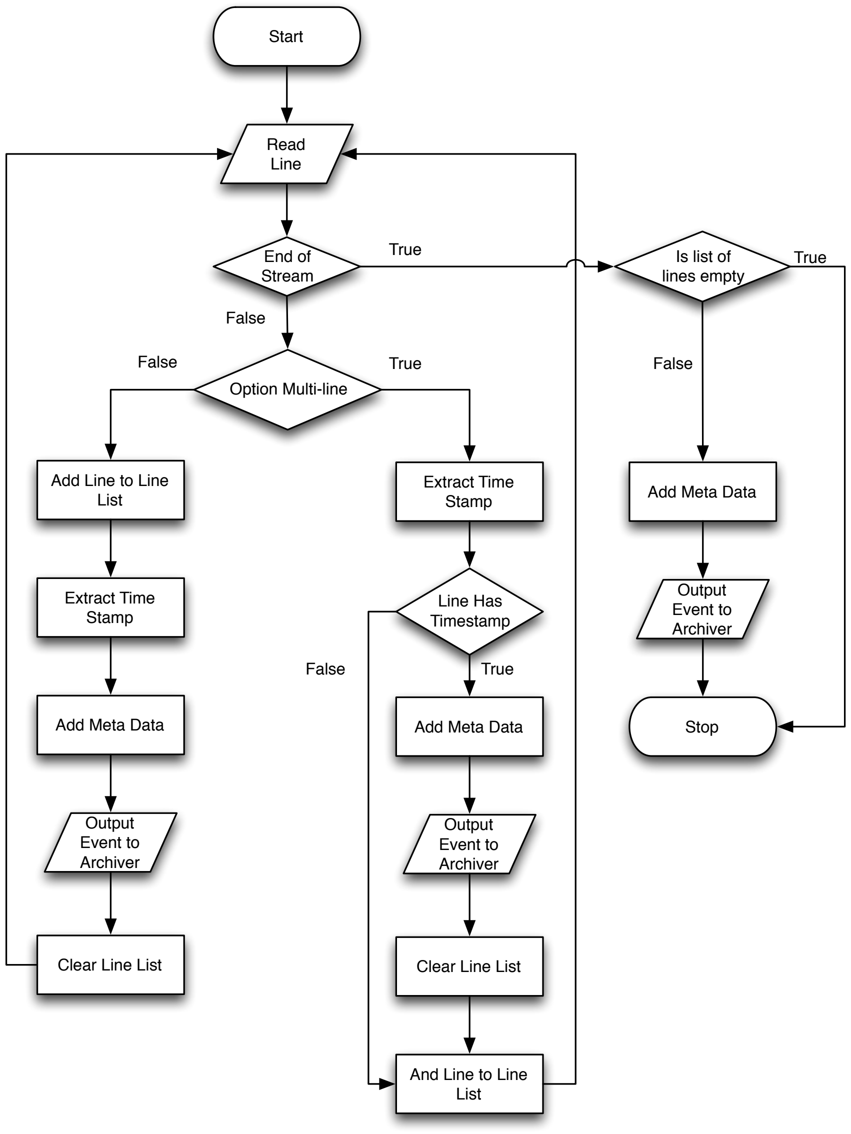


Figure 3.2: Event breaking process flow

The process starts by reading a line from the input stream based on the line breaking rules described in Section 3.2.3.

If the log-specific configuration specifies that the source has single line events, the line is added to the list of lines, and the timestamp is extracted. Metadata is added to the entry to create the event. The event is passed to the archiver, discussed in Section 3.3, after which the list of lines is cleared and the next line is read from the input stream.

If `break on timestamp` is enabled, the line is first parsed to see if it contains a timestamp. A line without a timestamp is added to the list of lines and the next line is read from the input stream. If the line contains a timestamp, the lines currently in the list forms the whole entry, and metadata is added to the entry to create the event that is then passed to the archiver. The list of lines are cleared and the current line, containing the timestamp, is added to it. The next line is then read from the input stream.

When event breaking has been completed the event is passed to the archiver for storing and indexing.

3.3 Event Archiver

As in the model described by York (2001), the archiver utilises a header structure at the start of the file followed by blocks of compressed data containing events. In addition, the block offset list, metadata and tailing record are also added to the end of the archive. Figure 3.3 depicts an overview of the structures composing the archive.

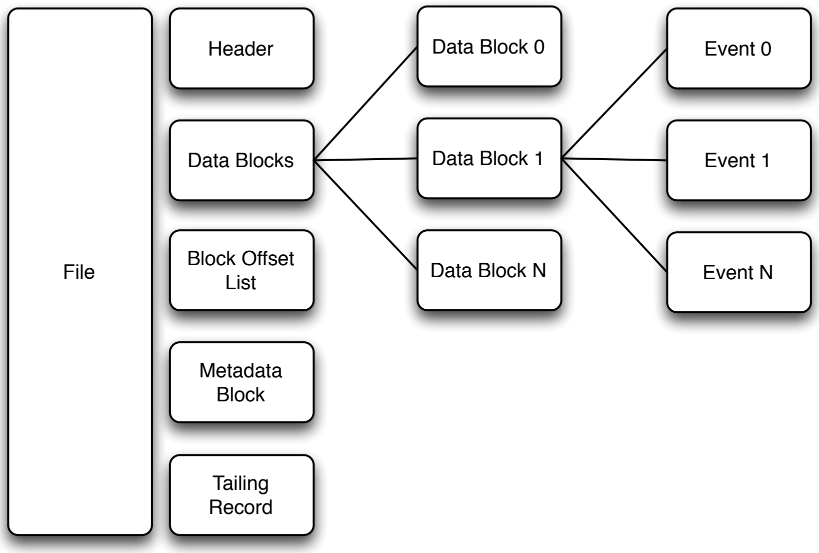


Figure 3.3: Pseudo-random access compressed file format

The process of persisting values to storage is called serialisation. The following sections describe the different serialisation methods used in the implementation.

3.3.1 Variable Width Integers

In many cases, the use of fixed width integers is space inefficient and a reasonable saving in storage space can be achieved by encoding fixed length integers into variable length integers.

The method is used in a number of solutions including ASN.1 DER/BER encoding[5], LLVM Bitcode File Format[6], DWARF Debugging Information Format[7], Dalvik Executable Format[8] (used by the Android operating system) amongst others.

The method used in this work is based on a variant of Little Endian Base 128 encoding, originally defined as part of the DWARF 3.0 Debugging Format Standard (Workgroup, 2005, pp. 139–141), extended for 64-bit unsigned integers.

The algorithm uses the most significant bit of a byte as a marker to indicate whether additional bytes follow; if set to 1, another byte is indicated for the integer. This means that only 7 bits are available for the storage of data.

The encoding and decoding source code is shown in Listings 3.5 and 3.6, respectively. Error and boundary checking is removed to simplify the explanations.

```
void encode(unsigned long long value, unsigned char* buffer)
{
    do
    {
        unsigned char c = (unsigned char)(value & 0x7F);
        value >>= 7;
        if (value) c |= 0x80;
        *buffer = c;
        buffer++;
    } while (value != 0);
}
```

Listing 3.5: Variable width integer encoding source

[5]http://www.itu.int/ITU-T/studygroups/com17/languages/X.690–0207.pdf
[6]http://llvm.org/docs/BitCodeFormat.html#variable-width-value
[7]http://dwarfstd.org/doc/Dwarf3.pdf
[8]http://www.netmite.com/android/mydroid/dalvik/docs/dex-format.html

Encoding starts by masking the seven least significant bits of the `value` variable and assigning the result to variable `c`. The bits in `value` are then shifted to the left seven times effectively removing the bits assigned to `c`. If the `value` is non-zero the most significant bit is set on `c` to indicate that more bytes follow and `c` is assigned to the output. This process continues until `value` is zero.

```
unsigned long long decode(unsigned char* buffer)
{
    unsigned long long value = 0;
    int shift = 0;
    char c;
    do
    {
        c = *buffer;
        unsigned long long x = (c& 0x7F);
        x <<= shift;
        value += x;
        shift += 7;
        buffer++;
    } while(c &0x80);
    return value;
}
```

Listing 3.6: Variable width integer decoding source

Decoding starts by taking the first character from the input, masking out the least significant bits and assigning the result to variable `x`, which is shifted to the right based on the value of the `shift` variable, and finally assigned to the result value. `shift` is incremented by seven for the next right shift if more input is required. If the current character has the most significant bit set, the process continues with the next character from the input, else the result is returned. Table 3.3 provides examples of variable width integer encoded values.

The table shows that up to and including the number `562,949,953,421,311`, encoding reduces the storage space required to serialise an integer. From `562,949,953,421,311` up to and including the number `72,057,594,037,927,935`, encoding does not save any space, but does not require additional storage. Upwards of `72,057,594,037,927,935` encoding requires more storage than the size of a 64-bit unsigned integer, which is 8 bytes. The largest possible value of a 64-bit integer is `18,446,744,073,709,551,615` and this value requires two additional bytes for storage.

Table 3.3: Examples of variable width integers values

Value	Bytes Used	Binary Representation
20	1	`00010100`
400	2	`10010000 00000011`
10,000	2	`10010000 01001110`
16,384	3	`10000000 10000000 00000001`
2000,000	3	`10000000 10001001 01111010`
500,000,000	5	`10000000 11001010 10110101 11101110 00000001`
562,949,953,421,311	7	`11111111 11111111 11111111 11111111 11111111 11111111 01111111`
1388,527,213,000,000	8	`11000000 11101010 10101010 11111010 10111010 11011011 10111011 00000010`
72,057,594,037,927,935	8	`11111111 11111111 11111111 11111111 11111111 11111111 11111111 01111111`
9223,372,036,854,775,807	9	`11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 01111111`
18,446,744,073,709,551,615	10	`11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 00000001`

The value `1388,527,213,000,000` is the number of microseconds from 1 January 1970, the Unix epoch date, for the timestamp `2013-12-31T22:00:13.000000Z`. The encoding results in 8 bytes of storage, which is exactly the same as the storage required for a fixed width 64-bit unsigned integer.

Some overhead is introduced by this method as well as edge cases where the method can add up to two bytes of storage in addition to the original width of the integer, but the space saving for lower values and avoiding the need to convert between big endian and little endian byte orders makes this a desirable solution.

All integers in the archiving format are unsigned integers, with the exception of the trailing record and timestamps that are stored as variable width integers.

3.3.2 Fixed Width Integers

Fixed width integers, including 32- or 64-bit signed and unsigned integers, are serialised to storage in little endian (Blanc and Maaraoui, 2005) format. The serialisation process first checks if the system is using big endian or little endian integer registers. If big endian is used, the bits in the variable are flipped to be in little endian format. The entire variable is then written to disk as an array of characters. On reading, the process is reversed.

The choice between using fixed and variable width integers is done at design time for structures and at write time for field types. Timestamp field types and structure data members are stored as fixed-width integers while all other numeric field types and data members are stored as variable-width integers. The only exception to storing non timestamp data members as variable-width integers is in the tailing-record structure where the reason for the exception is explained in full in Section 3.8.

3.3.3 Strings

When serialising a string the length of the string is variable width encoded and written to output. This length value is followed by the 8-bit characters of the string omitting the null terminator (0x00) typically found in C and C++ strings. The null terminator is used to indicate the end of a string, but as the length of the string is already known, the null terminator is not required.

3.4 Archive Header

The header provides identification, version and other information about how the options are used to construct the archive. This enables an archive reader to extract the information contained in the archive correctly.

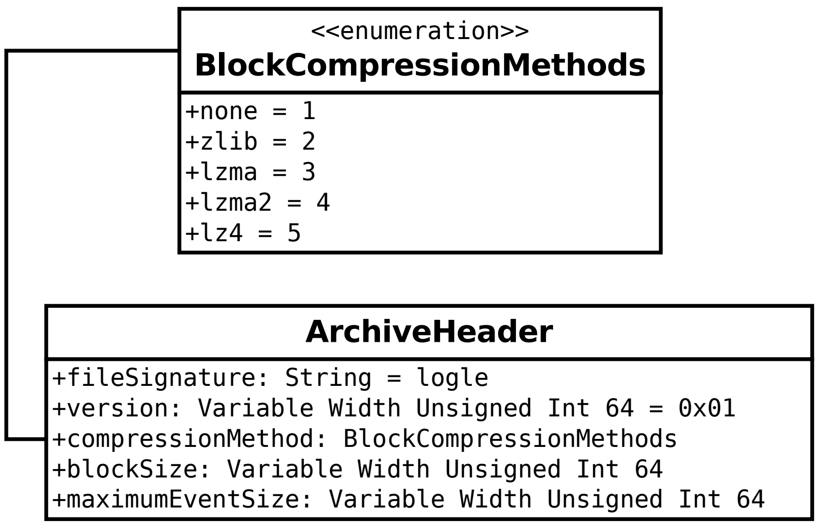
Figure 3.4 shows the fields of the header consisting of the file signature, format version, compression method, block size and maximum event size.

File Signature is the string “logle”. This is a “magic”[9] signature that identifies the file as a log archive storage.

Version is a variable width integer with the value 0x01. This is used by the parser to ensure that it can read the archive and for future expansion.

Compression Method is a variable width integer indicating the compression method used to compress the data blocks within a particular archive file. Currently defined values are listed in Table 3.4.

[9]http://www.magicdb.org



<<enumeration>>
BlockCompressionMethods
+none = 1
+zlib = 2
+lzma = 3
+lzma2 = 4
+lz4 = 5
ArchiveHeader
+fileSignature: String = logle
+version: Variable Width Unsigned Int 64 = 0x01
+compressionMethod: BlockCompressionMethods
+blockSize: Variable Width Unsigned Int 64
+maximumEventSize: Variable Width Unsigned Int 64

Figure 3.4: Archive header fields

Table 3.4: Compression method options in the archive header

Value	Method Name	Library Used
0x01	none	No compression is performed on the data blocks
0x02	zlib	Uses the GZIP compression functions contained in the zlib library
0x03	lzma	Uses the original LZMA library
0x04	lzma2/xz	Uses the new LZMA version 2 library, also known as XZ
0x05	lz4	Uses the LZ4 library

Block Size is a variable width integer denoting the default block size. This is needed as later in this research we investigate the properties of the different compression methods applied to different data block sizes. This value also enables the log parser to allocate a memory pool of blocks to minimise the overhead associated with repeated memory allocation.

Maximum Event Size is a variable width integer that stipulates the maximum size in bytes of a single event. The value is used to limit the size of the block chain that can be created before the data block is committed to disk. This value is used to calculate the pre-allocated memory used to store the maximum possible event size. The total size is calculated by taking the **Maximum Event Size** and adding one **Block Size** to provide for possible boundary over writes.

3.5 Event Data Blocks

Event data blocks contain the serialised event data. Each block starts with a block header followed by a variable number of events.

Figure 3.5 depicts the composition of a single event data block showing all the components. The next sections explain the block header structure, how events are serialised, event identifiers and block offset records.

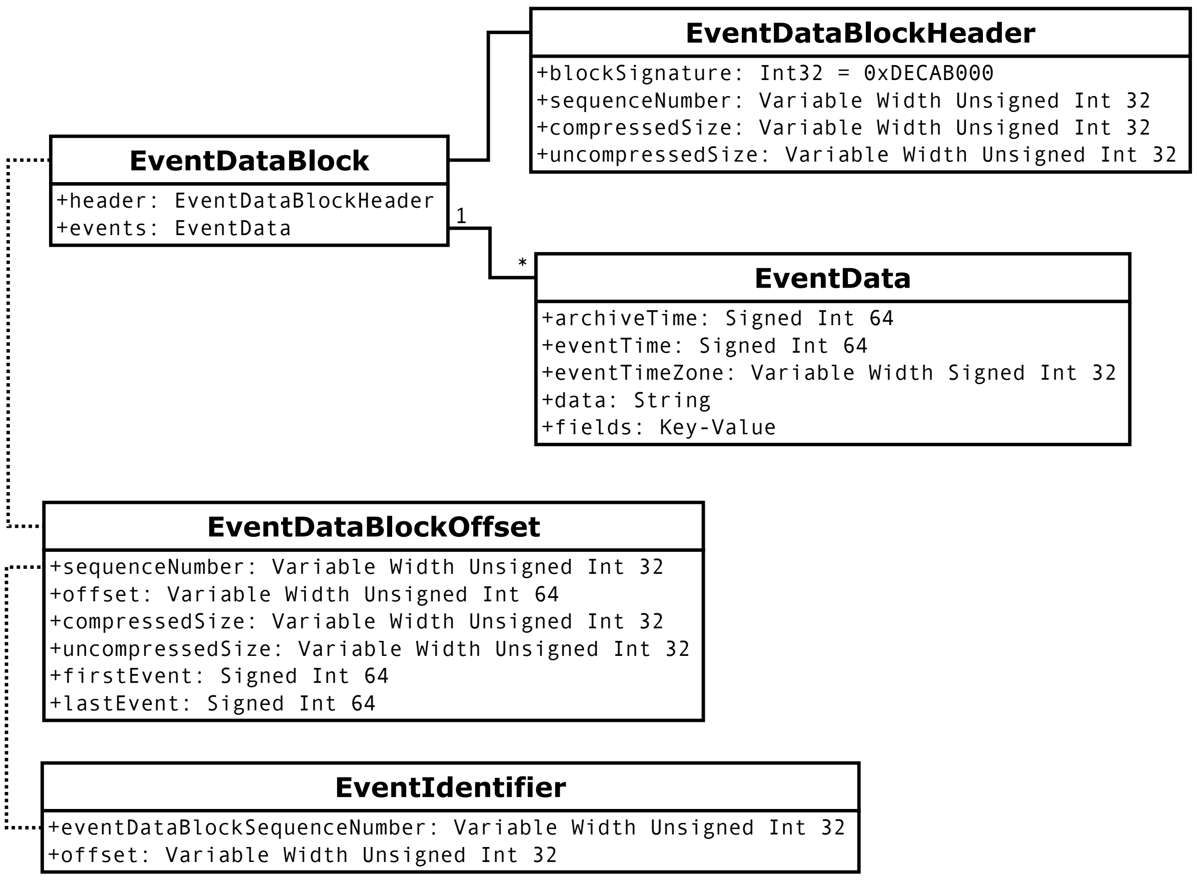

EventDataBlockHeader
+blockSignature: Int32 = 0xDECAB000
+sequenceNumber: Variable Width Unsigned Int 32
+compressedSize: Variable Width Unsigned Int 32
+uncompressedSize: Variable Width Unsigned Int 32
EventDataBlock
+header: EventDataBlockHeader
+events: EventData
1
*
EventData
+archiveTime: Signed Int 64
+eventTime: Signed Int 64
+eventTimeZone: Variable Width Signed Int 32
+data: String
+fields: Key-Value
EventDataBlockOffset
+sequenceNumber: Variable Width Unsigned Int 32
+offset: Variable Width Unsigned Int 64
+compressedSize: Variable Width Unsigned Int 32
+uncompressedSize: Variable Width Unsigned Int 32
+firstEvent: Signed Int 64
+lastEvent: Signed Int 64
EventIdentifier
+eventDataBlockSequenceNumber: Variable Width Unsigned Int 32
+offset: Variable Width Unsigned Int 32

Figure 3.5: Event data block structure

3.5.1 Event Data Block Header

The event data block header is depicted in Figure 3.5 under the name `EventDataBlockHeader`. Blocks starts with a block header providing a block signature identifying the start of the event block, the block sequence number, compressed size and uncompressed size of the event data.

Block Signature is a 32-bit integer constant set to `0xDECAB000`, which is used to identify the start of the block within the archive.

Sequence Number is a variable width integer denoting the sequence number of the block in the archive. As each block is created the sequence number is incremented. This sequence number is one of the numbers used to uniquely identify an event as described in Section 3.5.3; this is similar to a row number in a traditional database system. This sequence number is later combined with the block number to create an entry identifier that can be used to uniquely identify the event and used to retrieve the event from the archive.

Compressed Size is a variable width integer specifying the size in bytes of the compressed event data that follow. This provides the reading procedure with the number of bytes to read from the disk.

Uncompressed Size is a variable width integer specifying the size in bytes of the uncompressed data. This provides the archive reader with information to verify that sufficient memory is available as well as a test to verify that the uncompressed data is the same length as the original event data as calculated at the time of writing the event data to the storage.

3.5.2 Serialised Event Data

The event breaking process as described in Section 3.2 results in an array of fields consisting of key-value pairs. The format of the serialised event is shown as `EventData` in Figure 3.5.

To explain event serialisation the following example of the PF firewall log entry (Listing 3.7) created on `2014-03-03T10:24:56+0200` in the log file `/var/log/firewall.log` on the server with the host name `firewall.example.conf` and archived 10 seconds later, is used. Table 3.5 represents the event of the example log entry with all the fields.

```
Mar  3 10:24:56 11.123.215.66 10: 20:25.746726 rule 0/0(match): block in on em1:
 11.130.45.170.3835 > 67.253.14.169.6874: UDP, length 44
```

Listing 3.7: PF log entry to show event serialisation

The current date and time are assigned to the `_archivetime` field; this value is then written to the archive as a fixed width 64-bit integer. The `_time`, `_tz` and `_data` fields

Table 3.5: Example event with field serialisation

Field Name	Field Value	Serialised Value
`_data`	`Mar 3 10:24:56 11.123.215.66 10:20:25.746726 rule 0/0(match): block in on em1: 11.130.45.170.3835 > 67.253.14.169.6874: UDP, length 44`	
`_time`	`2014-03-03T08:24:56Z`	
`_tz`	`120`	
`_archivetime`	`2014-03-03T08:25:06Z`	
`_datatype`	`firewall`	
`_source`	`/var/log/firewall.log`	
`_host`	`firewall.example.conf`	

(Section 3.2.1) are extracted from the field array. The `_time` is written as a fixed width 64-bit integer and `_tz` is written as a variable width integer followed by the `_data`, which is written as a string. The remaining fields in the array are written as field key-value pairs.

The field names are very repetitive as they are present in every event and can add a large storage overhead. For example, consider an archive that holds a million events, each with a `_datatype` field. Serialising the field name requires nine bytes, eight bytes for the name and one byte for the length of the string. This will add 9,000,000 uncompressed bytes to the raw archive.

To alleviate the problem, a lookup is created to associated field names with numbers during the serialisation of events in an event block. The first time a field name is encountered and it is not in the lookup, the name is written to the data block, added to the lookup, a sequence number is assigned to the field name and then the sequence number is also written to the store. If the field name is found in the lookup only the associated sequence number is written to the storage. In cases where there are less than 127 fields, the sequence number is only one byte in size, thereby eliminating the storage of the length and bytes needed to represent the full field name string. Table 3.6 shows a field name lookup.

Table 3.6: Serialisation field name lookup example

Field Name	Index
`_datatype`	1
`_host`	2
`_source`	3

Some metadata fields, like field names are also repetitive and use the same lookup mechanism as the field names to save storage space. Consider for example, an archive containing a million events, of which 25% are from the source file /var/log/messages. The filename is stored in the _source field. Each _source field adds 17 bytes for the characters and one byte for the string length adding 4,250,000 uncompressed bytes to the archive. The fields that are optimised are _host, _source and _datatype, which represent the metadata that is most repetitive. Table 3.7 shows an example field key-value list where the values specific to the log example used are highlighted.

Table 3.7: Field key-value lookup for both the PF firewall example as well as other events

Field Name	Field Value	Index
_datatype	dns	1
_datatype	firewall	2
_datatype	proxy	3
_host	firewall.example.conf	1
_host	ns.example.conf	2
_host	proxy.example.conf	3
_source	/var/log/access.log	1
_source	/var/log/firewall.log	2
_source	/var/log/named.log	3

For all other fields the raw field value is written to the storage: numeric fields are variable width encoded, timestamp fields are written as fixed width integers and strings are serialised as described above. Table 3.8 shows the values as they are written to the archive.

Both the field name and meta field value lookups introduce processing overhead, but the space saving achieved by these methods is significant as these are highly repetitive in nature.

3.5.3 Event Identifier

The event identifier serves as a unique identifier for a single event in a single archive as well as the parameters needed to retrieve an individual event. The event identifier is created after each event has been serialised. Figure 3.5 contains a depiction of the event identifier (EventIdentifier). As shown the identifier consists of an event data block sequence number and an offset of the event in the block.

Event Data Block Sequence Number is a variable width integer indicating the block that contains the event. This points to an entry in the Block Offset List (Section 3.5.4).

Table 3.8: Serialised PF log event

Data Written to Archive	Description
`0004F3AF865E5C80`	`_archivetime` is a fixed width 64-bit integer representing the microseconds elapse since 1 January 1970.
`0004F3AF85C5C600`	`_time` is a fixed width 64-bit integer representing the microseconds elapse since 1 January 1970.
`78`	`_tz` is a variable width integer.

`88014D61722020332031303A3234`
`3A35362031312E3132332E323135`
`2E36362031303A2032303A32352E`
`3734363732362072756C6520302F`
`30286D61746368293A20626C6F63`
`6B20696E206F6E20656D313A2031`
`312E3133302E34352E3137302E33`
`383335203E2036372E3235332E31`
`342E3136392E363837343A205544`
`502C206C656E677468203434`
	First, the length of the `_data` field is written as a variable width integer (`0x8801`); this is then followed by the contents of the string.

`0102`	`_datatype` field with the field name lookup value of `1` (Table 3.6) and the field value lookup value of `2` (Table 3.7).
`0201`	`_host` field with the field name lookup value of `2` (Table 3.6) and the field value lookup value of `1` (Table 3.7).
`0302`	`_source` field with the field name lookup value of `3` (Table 3.6) and the field value lookup value of `2` (Table 3.7).

Offset is a variable width integer denoting the offset of the event within the uncompressed event data block.

3.5.4 Event Data Block Offset Record

The block offset record holds the meta information for each event data block as it is written to the storage. Figure 3.5 shows how the event data block offset record fits into the event block design (`EventDataBlockOffset`).

The record contains the sequence number, start offset of the block within the archive file, the compressed and uncompressed sizes of the block and the timestamps of the earliest and latest events contained in the block. This information is used by the archive reader to retrieve the block from the archive file and to extract the events from the block when presenting results from searches.

Sequence Number is a variable width integer containing the sequence number of the block.

Offset is a variable width integer with the offset of the event data block within the archive file.

Compressed Size is a variable width integer denoting the compressed size of the block as it is found in the archive file.

Uncompressed Size is a variable width integer denoting the uncompressed size of the block in memory.

First Event is a signed 64-bit integer holding the UTC timestamp of the first event in the block.

Last Event is a signed 64-bit integer holding the UTC timestamp of the last event in the block.

3.6 Block Offset List

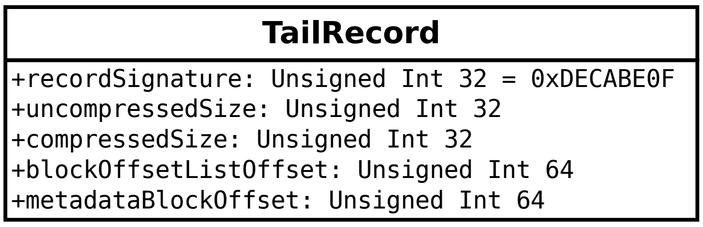
The block offset list serves as a directory of the location of each event data block in the storage. As each block is written to the archive, the block offset record (Section 3.5.4) for that block is added to the list. This list is written after the last event data block and the start offset of the list in the archive have been assigned in the tailing record (see Section 3.8).

3.7 Metadata

The lookup tables for fields and metadata are written as the last compressed block of the archive. Each lookup starts with a variable width encoded integer containing the number of entries in the lookup. Each lookup entry is then written starting with the variable width encoded sequence number, and followed by the string representing the value.

3.8 Tailing Record

The tailing record is used by the archive reader to read the block offset list and the metadata lookups.



TailRecord

```
+recordSignature: Unsigned Int 32 = 0xDECABE0F
+uncompressedSize: Unsigned Int 32
+compressedSize: Unsigned Int 32
+blockOffsetListOffset: Unsigned Int 64
+metadataBlockOffset: Unsigned Int 64
```

Figure 3.6: Tailing record fields

Data fields of this record are not variable width encoded. This provides a fixed sized record that the archive reader can seek at the end of the storage. Figure 3.6 shows the fields of the tailing record, which are explained below.

Record Signature is a 32-bit unsigned integer constant with the value 0xDECABEOF, which is used to identify the record for validation by the reader. If this value does not match, the storage is incomplete and must be repaired.

Uncompressed Size is a 32-bit unsigned integer specifying the uncompressed size metadata block.

Compressed Size is a 32-bit unsigned integer specifying the compressed size of the metadata block as written to the storage.

Offset of the Block Offset List is a 64-bit unsigned integer holding the offset of the block offset list within the storage.

Offset of Metadata Block is a 64-bit unsigned integer holding the offset of the block containing the metadata.

3.9 Summary

This chapter described a structure for a pseudo-random access compressed archive and the process flow involved in taking a log file, converting it into individual events and storing these in the archive.

The design incorporates the ability to retrieve single events from the archive without uncompressing the entire file. The entry identifier serves as a record number and makes it possible to create an inverted file (Zobel and Moffat, 2006), which serves as the basis of the IR engine.

The next chapter explores the compression methods available for compressing the individual event data blocks.

Chapter 4

Evaluation of Compression Methods

The objective of this chapter is to evaluate different compression methods to identify those suitable for inclusion in the pseudo-random access compressed archive performance testing in the next chapter. This also establishes a base line that can be used to measure the performance of the archiving process including compression ratio, compression time and decompression time. The following sections describe the experimental design, execution and results of the experiment, as well as the conclusions drawn.

4.1 Experimental Design and Execution

This experiment evaluates the performance of various command line compression tools available in the FreeBSD operating system as applied to small blocks of data. The block approach mimics the approach taken in a pseudo-random access compressed archive. A Python[1] (version 2.7) script is used to execute a number of compression methods on a number of predefined size blocks (64KB, 128KB, 256KB and 512KB) on a variety of log file types. A range of block sizes is used to investigate the impact of the block size on the different compression methods and compression options.

4.2 Compression Tools

With the exception of `lz4`, the compression tools are the default tools available in the FreeBSD 10 base operating system and FreeBSD ports. `lz4` was downloaded directly

[1]https://www.python.org/

from the developers' website[2] as the version available in the FreeBSD ports system was version r101, whereas the latest version at the time of the experiment was r116.

The command line tools and the versions used in this experiment are listed in Table 4.1.

Table 4.1: Compression tools

Application	Version
gzip	FreeBSD gzip 20111009
bzip2	Version 1.0.6, 6-Sept–2010
lzma	liblzma 5.0.4, xz (XZ Utils) 5.0.4
xz	liblzma 5.0.4, xz (XZ Utils) 5.0.4
lzop	lzop 1.03, LZO library 2.06
lz4	r116

When running the command-line tools with the `-version`, parameter both `xz` and `lzma` report using the same library (liblzma 5.0.4) where `xz` is the more modern version of `lzma` and is also known as `lzma2`.

With the exception of `lz4`, the tools selected represent utilities in regular use in Unix-type operating systems, especially for package compression. They also represent a variety of different compression methods. `lz4` was included because of its use in the ZFS file system and the reported compression and decompression speed benefits as discussed by Kiselkov (2013), among others.

4.3 Limitations

Various factors can distort the results obtained in this experiment. These include data bias, implementation of the command line tools, disk access caching of the files to be compressed and loading of the tools with the libraries needed to execute the command-line tools.

4.3.1 Data Bias

Compression ratios achieved by tools are greatly influenced by the entropy of the data being compressed. With a higher entropy of data, lower compression is to be expected (Shannon, 2001).

[2]https://code.google.com/p/lz4/

The entropy of log data can vary based on the applications or systems that created the log files. For example, the logs from the firewall systems have a lower entropy than those generated by the proxy system. The firewall log contains IP addresses with mostly the same length and have repetitive values, while proxy log entries contain uniform resource identifiers (URI) (Berners-Lee, 1998), which can be very expansive and have varying values.

Entropy can also vary within the different parts of the log files. It can happen that the entropy in the first 512KB of the file is different from that of a 512KB block in the middle of the same log file. This can be due to user behaviour, for example a proxy may show little activity between the hours of midnight and 04:00 mostly going to similar update web servers, whereas during office hours many users are likely to access a greater number of web servers, thereby creating greater entropy.

A variety of log samples from different applications were used to provide a better insight into the performance of the tools for different entropy values. Using only a small sample of each type of log type can also introduce data bias, but the samples should be sufficient for evaluation purposes.

All test runs cover exactly the same total size of log samples regardless of the block size and tool used to ensure that all the methods and blocks covered the same entropy. In the case of this testing, 160 64KB blocks and 20 512KB blocks of the same sample were used to cover the same 10MB of data, which comprises the whole test sample. The numbers of each block size are listed in Table 4.2.

Table 4.2: Test file sizes used in compression test

Block Size	Count	Total Size
64KB	160	10MB
128KB	80	10MB
256KB	40	10MB
512KB	20	10MB

4.3.2 Disk Access Caching

Operating systems and disk drive firmware implement caching of disk inputs and outputs to optimise disk access time. Caching uses random access memory (RAM) to speed up reading and writing from and to the disk by storing copies of the most recently accessed data in RAM. The data in memory is used to reduce input and output operations on the disk when the same data is reused by the operating system or software. This can mean that loading of the data sample files, command line tools and their associated libraries may differ during different stages of the execution of the experiment.

As the influence of caching cannot readily be quantified in the experiment we ran the same experiments a number of times. This should lessen the impact of caching. Additionally, the mean execution times were calculated for each run to mitigate caching distribution bias, while the standard deviation should highlight any excessive deviation within the data set.

4.3.3 Tool Implementation

Each tool has it own unique implementation for reading the input file, processing and writing the output file. They may have varying levels of optimisation that can distort the actual time performance of the compression or decompression algorithms.

Evaluating the optimisation of the source code implementation of each command line tool falls outside the scope of this experiment as this cannot be accurately quantified based solely on studying the source code. However, it is reasonable to expect that there are no serious flaws given the widespread use of the tools.

4.4 Implementation

The experiment was executed using four Python scripts each performing a specific function. The first script was used to calculate the Shannon entropy for each of the log samples as well as two additional base line files, which was then hard-coded into the third script, where the values are used for calculating part of the statistical information. The next script executed the command-line utilities on the samples and saved the results in a comma-separated values (CSV) file format (Shafranovich, 2005). The third script used the results from the second stage and performed statistical calculations and rankings of each tool. The final script loaded the results from the second stage and rendered the charts and tables included in the results (Section 4.5) and Appendix C. The details are discussed below.

4.4.1 Sample Log Files

Sample log files were used to generate the compression performance data. Seven log files were selected from six diverse applications to address some of the data bias issues discussed above. Table 4.3 shows the list of applications that generated the log files which were used by the scripts. The logs are samples from PF Firewall[3], Windows Security Events[4], BIND[5], Squid Proxy[6], Postfix MTA[7] and PCAP[8]. This table gives the application name, short name used in the rest of the thesis and a description of the function performed by the application. Log extracts for each sample application can be found in Appendix B.

Table 4.3: List of applications that generated the sample log files

Application	Short Name	Description
PF Firewalls	`firewall`	PF is the OpenBSD and FreeBSD firewall.
Windows Security Event Logs	`windows-event`	Microsoft Windows Security Event Logs. The events were extracted from the Windows Management Instrumentation (WMI) and written to a continuous text file.
BIND	`dns`	BIND is a widely used Domain Name Service found in Unix type operating systems.
Squid Proxy Logs	`proxy`	Squid is a widely used open source web proxy.
Postfix Mail Logs	`mail`	Postfix is a popular open source mail transport system.
PCAP Files	`pcap-text`, `pcap-binary`	The pcap capture file format is commonly used by network diagnostic tools. The files are binary and can be converted to text. The binary sample file is the abbreviated format containing packet header information. The text file is the binary file converted to text, containing the same abbreviated information.

[3]http://www.openbsd.org/faq/pf/
[4]http://msdn.microsoft.com/en-us/library/aa394582(v=vs.85).aspx
[5]http://www.isc.org/downloads/bind/
[6]http://www.squid-cache.org/
[7]http://www.postfix.org/
[8]http://www.tcpdump.org/manpages/pcap.3pcap.html

4.4.2 Shannon Entropy

Shannon Entropy (Shannon, 2001) is calculated for each 10MB sample log type and two additional baseline files, one containing 10MB of zero value bytes and one containing 10MB of random data.

The commands used to generate the zero and random filled baseline files are shown in Listing 4.1.

```
dd bs=1024 count=10240 if=/dev/zero of=baseline_zero.dat
dd bs=1024 count=10240 if=/dev/urandom of=baseline_urandom.dat
```

Listing 4.1: Shannon baseline file generation

The entropy code is generated based on the Python source code, given in Listing 4.2, from the Rosetta Code[9] website.

```
import math
from collections import Counter

def entropy(s):
    p, lns = Counter(s), float(len(s))
    return -sum( count/lns * math.log(count/lns, 2) for count in p.values())
```

Listing 4.2: Python implementation of Shannon entropy calculation

4.4.3 Executing Command Line Tools

Generating the compression performance data starts by extracting small blocks from the sample files. Then each of the blocks are compressed twice using the fastest and the best compression option, where the best option provides the best compression ratio at the expense of compression time and the fastest option provides the fastest compression time at the expense of compression ratio. After each compression operation, the decompression tool of the compression tool is used to decompress the block.

The test cycle begins by extracting 160 64KB blocks, 80 128KB blocks, 40 256KB blocks and 20 512KB blocks from the first 10MB of the sample files. For each tool and option the following metrics were collected; compression time, decompression time, original file size

[9]http://rosettacode.org/wiki/Entropy#Python:_More_succinct_version

and compressed file size. The cycle was repeated 20 times and the results were written to a CSV file for further processing.

4.4.4 Compression Ratios

Compression ratios represent a measurement of the space saving achieved by using a specific compression method. This is calculated for each compression method by option, sample log type and block size. Compression ratio is calculated by dividing the uncompressed file size by the compressed file size. This is done by totalling the uncompressed and compressed file sizes for each result in a given population, and then dividing the uncompressed file size total by the compressed file size total. For the example discussed in Section 4.4.2, the 3200 uncompressed and compressed file sizes for the 64KB block extracted from the first 64KB of the firewall sample file, compressed with `gzip -1` are totalled and the compression ratio is then calculated, giving the 64KB block firewall compression ratios for `gzip -1`.

4.4.5 Mean

The calculation stage reads the results from the data generation stage and calculates the means of the compression ratio, and compression and decompression times. Additionally the standard deviation is calculated for the compression and decompression times. No standard deviation is calculated for the compression ratio as this is the same for each block.

The mean is calculated by separately totalling the compression ratio, compression times and decompression times for each compression method by option, sample log type and block size. These totals are then divided by the number of blocks in the sample population. As an example, the 3200 compression time results for the 64KB block extracted from the first 64KB of the firewall sample file, compressed with `gzip -1` are totalled and divided by 3200 providing the mean compression time for the 64KB block firewall compression using `gzip -1`.

The second part then allocates weights to each block by compression method and sample file type based on the ranking of each method per metric. This process is described in the next section (Section 4.4.6).

4.4.6 Weighting

The results from the first statistical calculations (Sections 4.4.4 and 4.4.5) are used to assign weights to each compression method by option, block size and sample log file type. This is performed for the compression ratio, mean compression time and mean decompression time. For each compression method per block size per sample type, the data is sorted in descending order of performance. The highest value is assigned a weighting of 13 and the lowest 1.

4.4.7 Standard Deviation Calculation

Standard deviation is calculated by totalling the square of the difference between the mean and the value of each metric used to calculate the mean above. The square root of the total provides the standard deviation for that metric. Continuing with the example above the square of the difference between the compression time and mean compression time for each of the 3200 results is totalled. The square root of the total gives the standard deviation for the 64KB block firewall compression using `gzip -1`.

The data points within one, two, three and four standard deviations are calculated by calculating the difference between the mean and the value of each metric divided by the standard deviation. These results are saved to disk for further analysis.

4.5 Evaluation of Results

In this section we discuss the results obtained from the experiment starting with the Shannon entropy and standard deviation and finally evaluating the compression performance results.

4.5.1 Shannon Entropy Results

The Shannon Entropy is the entropy values calculated for each sample and baseline file. The entropy values are between 0 and 8 denoting the entropy in bits.

Table 4.4 shows the values obtained from the calculation of the whole 10MB of each sample file. The results show that the zero baseline file has an entropy value of zero and is therefore highly compressible, while the urandom file has a 100% entropy which makes it unsuitable for compression. Compressing the 10MB zero file using `gzip -9` yields a compressed file size of 49 bytes giving a compression ratio of 1024.10. Compressing the 10MB random file using `gzip -9` yields a compressed file size of 10,488,999 bytes giving a compression ratio of 1.

Table 4.4: Shannon entropy values of sample and baseline files

File Type	Shannon Entropy	gzip -9 Compression Ratio	bzip2 -9 Compression Ratio
zero	0	1,024.10	213,995.10
urandom	7.9999	1.00	1.00
firewall	4.6423	9.87	14.18
proxy	5.5952	4.67	6.78
dns	5.0203	7.40	10.24
pcap-text	4.8580	7.80	**8.97**
mail	5.4721	7.69	16.19
pcap-binary	6.0081	3.40	3.48
windows-event	5.1906	40.49	**73.9**

Table 4.4 shows that there is very little correlation between the entropy reported by Shannon and the compression ratio achieved by the two command line tools. The `pcap-text` and `windows-event` results highlight this point perfectly. The `pcap-text` has a Shannon value of 4.8580 and the `windows-event` has a Shannon of 5.1906; this would mean that the `pcap-text` has a lower entropy and should therefore have a better compression ratio. The `bzip2` compression ratio results show that `windows-event` (73.9) far exceeds the compression of `pcap-text` (8.97).

The Shannon entropy for the sample data does not provide any guidance to a possible data bias when calculating the compression ratio. Consequently this data is not used in any further analysis and weighting.

4.5.2 Standard Deviation for Compression Times

Three metrics were determined for each test iteration; compression ratio, compression time and decompression time. Compression ratio is only dependent on the data and not influenced by disk or operating system caching as discussed previously in the limitations section. Standard deviations are calculated to evaluate the influence of caching on compression and decompression times. This is done by reviewing the distribution of the results based on its standard deviation.

Table 4.5 lists the mean compression times in seconds for each compression method for all block sizes and repeated runs. Table 4.6 shows the distribution of compression times for the experimental results.

Table 4.5: Mean compression time and standard deviation

Compression Method		Mean Time	Standard Deviation
bzip2	–1	0.1927	0.0036
bzip2	–9	0.2679	0.0060
gzip	–1	**0.0752**	0.0028
gzip	–9	0.0801	0.0028
lz4	–1	**0.0734**	0.0028
lz4	–9	0.0883	0.0028
lzma	–1	0.0898	0.0028
lzma	–9	0.2587	0.0032
lzop	–1	**0.0716**	0.0028
lzop	–9	0.0967	0.0028
xz	–1	0.0908	0.0028
xz	–9	0.2599	0.0032
xz	-e	0.6462	0.0111

The results summarised in Table 4.5 show that the mean times for faster compression options (`-1`) are lower than those of the better compression options (`-9` and `-e`). There is also clear variance in the compression times based on the actual compression method used. The values themselves do not show the best method to use, but are later combined with the compression ratio to do a better evaluation of the compression methods.

Table 4.6: Standard deviation of compression time for all block sizes

Compression Method		1σ	2σ	3σ	$4\sigma+$
bzip2	–1	26,860 (63.95%)	14,175 (33.75%)	862 (2.05%)	103 (0.25%)
bzip2	–9	27,013 (64.32%)	13,946 (33.20%)	922 (2.20%)	119 (0.28%)
gzip	–1	25,880 (61.62%)	15,343 (36.53%)	769 (1.83%)	8 (0.02%)
gzip	–9	27,203 (64.77%)	13,469 (32.07%)	1,200 (2.86%)	128 (0.30%)
lz4	–1	25,921 (61.72%)	15,317 (36.47%)	756 (1.80%)	6 (0.01%)
lz4	–9	26,772 (63.74%)	14,119 (33.62%)	1,046 (2.49%)	63 (0.15%)
lzma	–1	26,808 (63.83%)	14,085 (33.54%)	1,017 (2.42%)	90 (0.21%)
lzma	–9	30,161 (71.81%)	9,937 (23.66%)	1,505 (3.58%)	397 (0.95%)
lzop	–1	25,763 (61.34%)	15,527 (36.97%)	706 (1.68%)	4 (0.01%)
lzop	–9	28,001 (66.67%)	12,592 (29.98%)	1,234 (2.94%)	173 (0.41%)
xz	–1	26,772 (63.74%)	14,111 (33.60%)	1,032 (2.46%)	85 (0.20%)
xz	–9	30,010 (71.45%)	10,052 (23.93%)	1,496 (3.56%)	442 (1.05%)
xz	-e	30,424 (72.44%)	9,768 (23.26%)	1,348 (3.21%)	460 (1.10%)

Table 4.6 shows some uneven distribution that indicates a small bias on data caching.

Faster options have a greater deviation, which shows that they are more affected by the disk access than other options. The deviation, however, is still within acceptable limits for the evaluation of compression times.

4.5.3 Standard Deviation for Decompression Times

In this section we continue with the dissection of the distribution of experimental results focusing on decompression time. Table 4.7 list the mean decompression times, in seconds for each compression method, for all block sizes and repeated runs. Table 4.8 shows the distribution of results of the decompression times. The standard deviation has a very even distribution showing that no method or option is biased by reading from the disk.

Table 4.7: Mean decompression time and standard deviation

Compression Method		Mean Time	Standard Deviation
bzip2	–1	0.0836	0.0030
bzip2	–9	0.0888	0.0029
gzip	–1	0.0720	0.0029
gzip	–9	0.0718	0.0028
lz4	–1	0.0729	0.0028
lz4	–9	0.0730	0.0044
lzma	–1	0.0727	0.0028
lzma	–9	0.0727	0.0029
lzop	–1	0.0715	0.0028
lzop	–9	0.0713	0.0028
xz	–1	0.0738	0.0028
xz	–9	0.0738	0.0028
xz	-e	0.0736	0.0028

4.5.4 Compression Performance

Compression ratio is the most important metric when evaluating compression methods, because the influence of both the compression and decompression times is reduced due to the processing overhead in reading and writing event data to and from the archive. Figure 4.1 depicts the total weighted compression ratios. The weighted values are listed in Table C.1, calculated as described in Section 4.4.6.

The results show that `xz -e` is the best compression method with 353 points, `lzma -9` second with 335 points and `xz -9` third with 304 points. From this it is clear that the `lzma/xz` family provides the best compression ratio of all the methods under review.

Table 4.8: Standard deviation of decompression time for all block sizes

Compression Method		1σ	2σ	3σ	4σ+
bzip2	–1	26,069 (62.07%)	15,074 (35.89%)	834 (1.99%)	23 (0.05%)
bzip2	–9	26,199 (62.38%)	14,924 (35.53%)	847 (2.02%)	30 (0.07%)
gzip	–1	25,819 (61.47%)	15,456 (36.80%)	718 (1.71%)	7 (0.02%)
gzip	–9	25,812 (61.46%)	15,488 (36.88%)	696 (1.66%)	4 (0.01%)
lz4	–1	25,937 (61.75%)	15,371 (36.60%)	678 (1.61%)	14 (0.03%)
lz4	–9	26,019 (61.95%)	15,286 (36.40%)	687 (1.64%)	8 (0.02%)
lzma	–1	25,948 (61.78%)	15,219 (36.24%)	819 (1.95%)	14 (0.03%)
lzma	–9	25,958 (61.80%)	15,195 (36.18%)	830 (1.98%)	17 (0.04%)
lzop	–1	25,843 (61.53%)	15,450 (36.79%)	699 (1.66%)	8 (0.02%)
lzop	–9	25,833 (61.51%)	15,464 (36.82%)	694 (1.65%)	9 (0.02%)
xz	–1	25,954 (61.8%)	15,224 (36.25%)	805 (1.92%)	17 (0.04%)
xz	–9	25,923 (61.72%)	15,255 (36.32%)	812 (1.93%)	10 (0.02%)
xz	-e	25,977 (61.85%)	15,178 (36.14%)	831 (1.98%)	14 (0.03%)

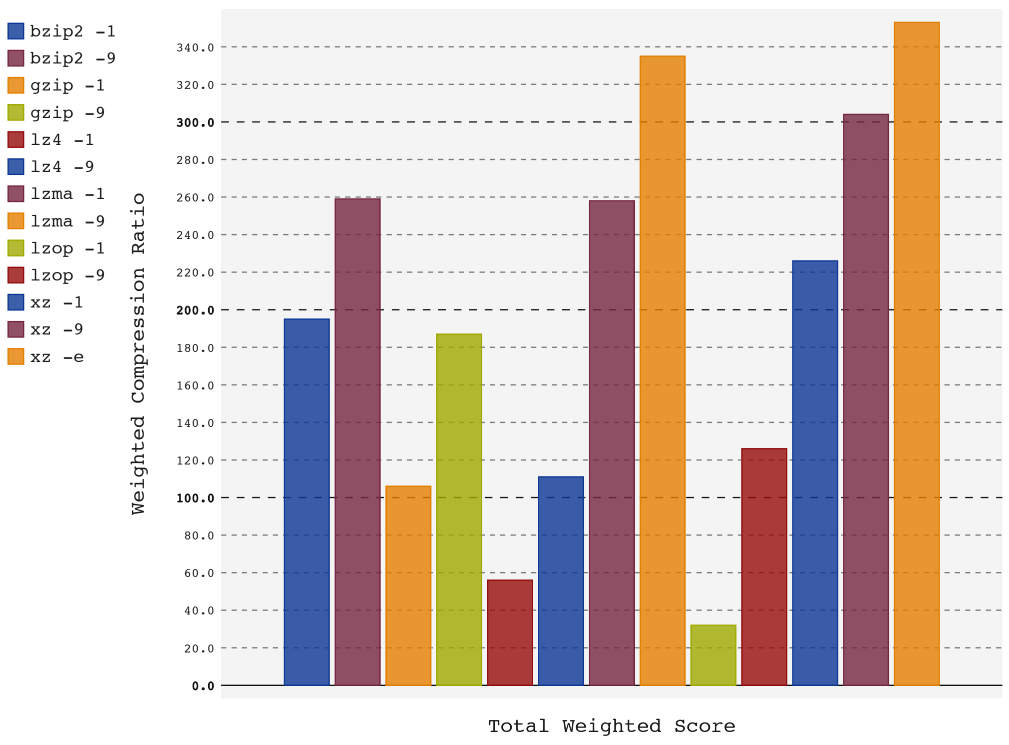


Figure 4.1: Total weighted compression ratios

Although the compression ratio achieved by a tool is by far the biggest factor, this research also considers the compression and decompression times.

4.5.5 Compression Time

Figure 4.2 depicts the total weighted compression times per compression method. The weighted values are listed in Table C.2, calculated as described in Section 4.4.6. It is interesting to note that the three methods with the worst compression ratios (`lzop -1`, `lz4 -1`, `gzip -1`) have the fastest compression times. On the other side of the spectrum, the three methods with the best compression ratios (`xz -e`, `lzma -9`, `xz -1`) have the worst compression times. The overhead introduced by the event breaking and serialisation of the event data should hide the bad compression times of the high compression ratio methods.

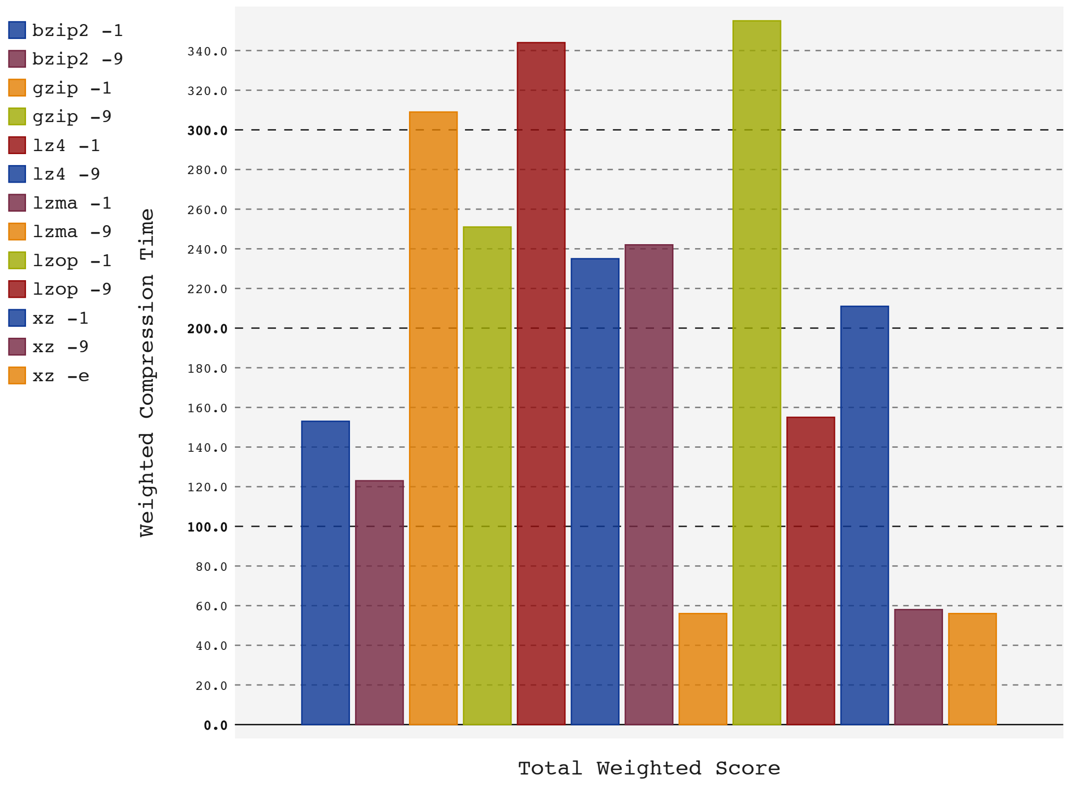


Figure 4.2: Total weighted compression times

4.5.6 Decompression Time

Figure 4.3 shows the total weighted decompression times per compression method, with the weighted values listed in Table C.3, and calculated as described in Section 4.4.6.

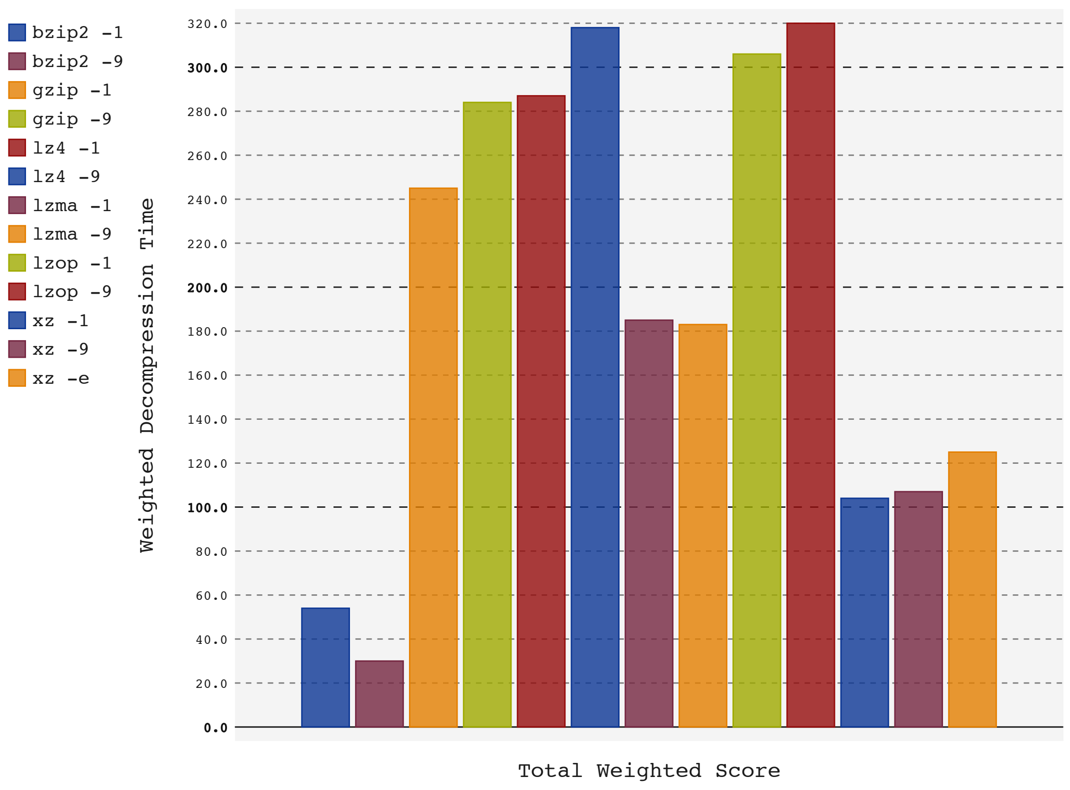


Figure 4.3: Total weighted decompression times

As with the compression times the `lzo`, `lz4` and `gzip` methods are the fastest. Again, `lzma` and `xz` methods have significantly slower decompression times. This drawback is also negated by the additional processing in reading and extracting events.

4.6 Summary

This part of the research has provided insights into the different compression methods when working with smaller data blocks. The next phase of the work uses `xz` and `lzma` methods due to their excellent compression ratios and `lz4` and `gzip` as a baseline insight into the compression and decompression time penalties encountered in the archiving solution. These methods yielded enough objective data to provide insights into the performance of the pseudo-random access compressed archive, and therefore the `bzip2` and `lzo` methods was not implemented.

The following chapter reports on the use of the selected compression methods to test the compression performance of the archive implementation.

Chapter 5

Implementation Performance Testing

In this chapter the cost in terms of time and compression ratio associated with the archiving process are evaluated. This provides insight into the disadvantages of the archiving system. The following sections describe the archive system implementation, experimental design, execution and results of the experiment as well as the conclusions drawn.

5.1 Archive Implementation

To conduct the performance experiment the archiving system described in Chapter 3 was implemented. The sections below describe the tool chain, several key concepts and important data structures used in the implementation.

5.1.1 Archive Implementation Tool Chain

The pseudo-random access compressed archive was implemented in C++, using Visual Studio 2013 on Windows 7 and later ported to Clang version 3.4, first on MacOSX Maverick and then FreeBSD 10. Table 5.1 shows the libraries used in the implementation, excluding the C++ Standard Library.

Poco C++[1] consists of libraries and frameworks that provide the ability to create cross platform applications with a single source code base. **lz4**[2] provides the implementation of the `lz4` compression method, while **zlib**[3] implements the `gzip` compression method.

[1]http://pocoproject.org/
[2]https://code.google.com/p/lz4/
[3]http://www.zlib.net/

Table 5.1: C++ libraries used

Library	Version
Poco C++	1.5.3
lz4	r116
zlib	1.2.8
liblzma	9.22 beta

Finally, **lzma**[4] caters for both the `lzma` and `xz`(`lzma2`) compression methods.

The experiment source code uses the same tool chain and is encoded in the last unit testing module for the archive implementation test suite. Before running the experiment, the entire test suite was executed on the platform where the experiment was carried out to ensure that all the needed functionality of the implementation was working correctly on the test platform.

5.1.2 Block Array Class

The serialised event data, as detailed in Section 3.5.2, is stored in memory in an array of memory blocks. The block sizes are determined by configuration options and are one of the following 64KB, 128KB, 256KB or 512KB.

An array of blocks was used for two reasons. Firstly, it is possible that a single event can exceed the given block size and could therefore require multiple blocks to store in memory. Secondly, the last event written in the block could exceed the allocated block size and additional memory would be required. For example, in a 64KB block where the current write pointer in a block is 65,305, only 231 bytes are available in the block. If the last event is 300 bytes the block will overflow and an additional block in memory will be needed. To prevent the second condition from being triggered with every block, the archiving system checks the available space left in the block after each write and if the available space is below 128 bytes, the block is written to disk and a new block is started.

Figure 3.5 shows the block array structures used for storing the events before writing and after reading the serialised event data from the archive on disk.

[4]http://7-zip.org/sdk.html

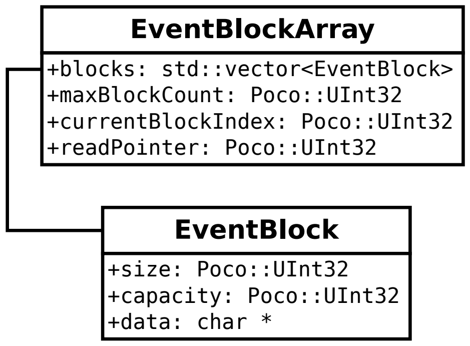

EventBlockArray
+blocks: std::vector<EventBlock>
+maxBlockCount: Poco::UInt32
+currentBlockIndex: Poco::UInt32
+readPointer: Poco::UInt32
EventBlock
+size: Poco::UInt32
+capacity: Poco::UInt32
+data: char *

Figure 5.1: Event block array structures

5.1.3 Block Array Caching

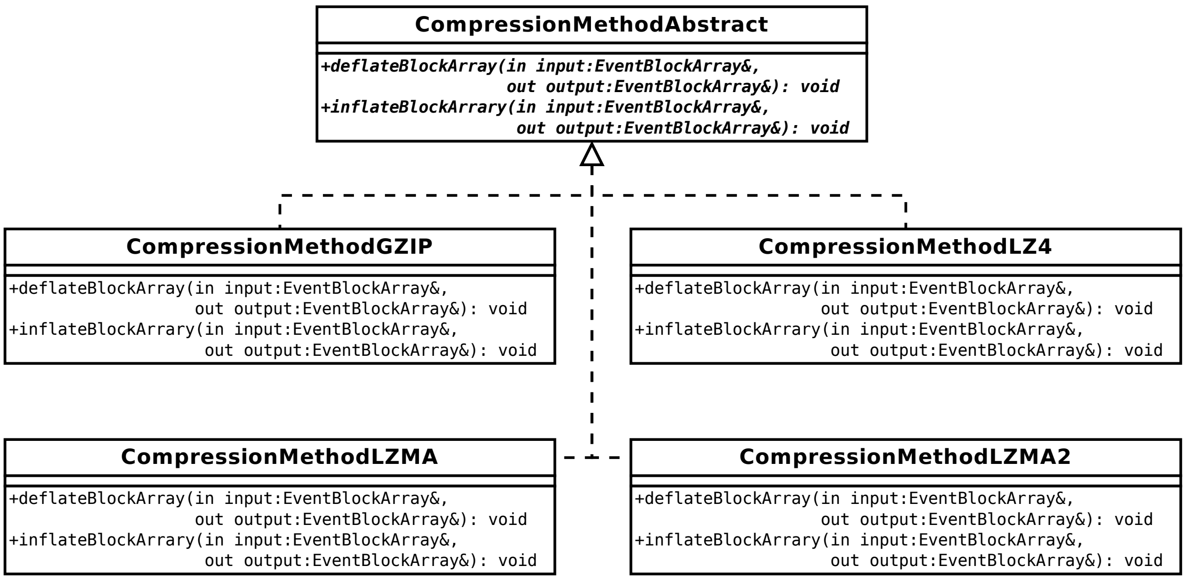
Optimising batch retrieval from the archive is an essential requirement of the implementation. Before the process starts the system must sort the entry identifiers; this groups entry identifiers by data blocks. Once this has been done retrieval can be optimised by caching data blocks in memory.

The caching mechanism keeps a decompressed copy of the last blocks read from the archive. This speeds up the retrieval of events in the same block as the data block does not need to be read from the disk and decompressed again. For this experiment, the cache is limited to 10% of the total blocks in the archive file.

5.1.4 Compression Method Abstraction

To simplify the coding of the experiment an abstract class is used to provide an interface for the compression and decompression functionality. Figure 5.2 depicts the abstract compression method class and all four of the implemented classes, one for each of the compression methods, i.e., `gzip`, `lz4`, `lzma` and `lzma2`.

When the archive writer or reader is created an appropriate compression method class is created to provide the necessary functionality by implementing the pure virtual functions `deflateBlockArray` and `inflateBlockArray`, which perform compression and decompression of an event block array respectively. Both functions take as input an event block array as described in Section 5.1.2, and also output an event block array.


CompressionMethodAbstract
+deflateBlockArray(in input:EventBlockArray&, out output:EventBlockArray&): void
+inflateBlockArrary(in input:EventBlockArray&, out output:EventBlockArray&): void
CompressionMethodGZIP
CompressionMethodLZ4
CompressionMethodLZMA
CompressionMethodLZMA2
Figure 5.2: Abstract compression method design

5.2 Experimental Design and Execution

The experiment creates an archive for each log type, using different block sizes, compression methods and compression levels. The next section (Section 5.2.1) describes the log samples used, while Section 5.2.3 explains the work flow of the experiment.

5.2.1 Log Samples

For this experiment, we used the same 10MB text log files created for the evaluation of the compression methods (Chapter 4). The files were modified to remove any truncated or partial events on the 10MB boundary.

As an example, Listing 5.1 shows the last four events at the very end of the 10MB firewall sample file. The first three events are complete, but the last entry is incomplete. Listing 5.2 shows the end of the log file after the incomplete entry has been removed using a text editor.

```

Mar  3 10:37:17 11.123.215.66 10: 32:47.488678 rule 0/0(match): block in on em2:
 192.16.0.145.1383 > 148.81.111.111.80: tcp 20 [bad hdr length 8 - too short, <
20]
Mar  3 10:37:17 11.123.215.66 10: 32:47.539650 rule 0/0(match): block in on em2:
 192.16.1.41.1932 > 192.16.160.192.445: tcp 20 [bad hdr length 8 - too short, <
20]
Mar  3 10:37:17 11.123.215.66 10: 32:47.545895 rule 0/0(match): block in on em2:
 192.16.1.209.4461 > 11.206.142.159.445: tcp 20 [bad hdr length 8 - too short, <
 20]
Mar  3 10:37:17 11.123.215.66 10: 32:47.555138 rule 0/0(match): block in on em2:
 192.16.1.41.1899 > 192.16.211.190.445: tcp 20 [bad hdr length 8 - t
```

Listing 5.1: Untruncated PF firewall sample log

```

Mar  3 10:37:17 11.123.215.66 10: 32:47.488678 rule 0/0(match): block in on em2:
 192.16.0.145.1383 > 148.81.111.111.80: tcp 20 [bad hdr length 8 - too short, <
20]
Mar  3 10:37:17 11.123.215.66 10: 32:47.539650 rule 0/0(match): block in on em2:
 192.16.1.41.1932 > 192.16.160.192.445: tcp 20 [bad hdr length 8 - too short, <
20]
Mar  3 10:37:17 11.123.215.66 10: 32:47.545895 rule 0/0(match): block in on em2:
 192.16.1.209.4461 > 11.206.142.159.445: tcp 20 [bad hdr length 8 - too short, <
 20]
```

Listing 5.2: Truncated PF firewall sample log

This truncation reduces the file size below the original 10MB, but as shown in Table 5.2 the reduction is small enough that it does not have a significant impact when evaluating the results of this experiment, against the baseline created in Chapter 4.

5.2.2 Event Count Calculation

The event count represents the number of unique events in each log file. Each event in the log file should be in the archive solution after processing. To do the comparison it is necessary to calculate the number of events in each log file before they are ingested into the archive.

Table 5.2: Truncated sample log file sizes

Application	Original Size (Bytes)	Truncated Size(Bytes)	Difference
`firewall`	10,485,760	10,485,611	149 (0.0014%)
`windows-event`	10,485,760	10,485,548	212 (0.0020%)
`dns`	10,485,760	10,485,724	36 (0.0003%)
`proxy`	10,485,760	10,485,087	673 (0.0064%)
`mail`	10,485,760	10,485,725	35 (0.0003%)
`pcap-text`	10,485,760	10,485,673	87 (0.0008%)

In the sample logs there are both single-line and multi-line events. For the single-line event log samples the `wc` utility is used to count the number of lines that should each be represented by a single event in the archive.

The `wc` utility is used to calculate words, lines, characters or bytes in a file. Listing 5.3 shows how the `wc` utility is used to calculate the number of lines in the `firewall` sample log file name `event_pflog_small.log`.

```
wc -l event_pflog_small.log
```

Listing 5.3: Example of calculating event count in single-line event log files

The `windows-event` is the only sample with multi line events. To count the number of events the `grep` text search utility is used in conjunction with the `wc` utility. The pattern `EventCode=` only appears once in every event. The `grep` utility is used to do a line-by-line search for the pattern which is then output to the console where it is counted by `wc`. Listing 5.4 shows how `grep` and `wc` utilities are used to calculate the number of lines in the `windows-event` sample log file name `event_pflog_small.log`.

```
grep "EventCode=" event_wineventlog_small.log | wc -l
```

Listing 5.4: Example of event counting in Windows event logs

5.2.3 Experiment Workflow

A single run in the experiment takes the provided sample file, specified compression method, compression level and block size and archives the logs into the pseudo-random access compressed archive. After the sample logs have been processed, the experiment does a number of read operations. First, it reads each event data block (Section 3.5) into memory to establish the basic decompression times. Next it reads back each individual event using both sorted and random list of event entry identifiers as described in Section 5.2.4.

5.2.4 SHA1 Validation of Events

During the archiving process a list is created of records containing the entry identity and SHA1 hash of the `_data` field. This list is later used to verify each entry read from the archive.

After archiving has been completed the experiment reads each event from the archive, using the random access read method, first based on the list above sorted in sequential order and then in three random sequences. The read from the sorted list is to obtain the best speed metrics for the most optimised reading conditions. The three random reads provide the metrics for adverse reading conditions testing the cost of the random access functionality. While the sorted read benefits the most from the block array caching (Section 5.1.3) the random read sequences do not benefit from caching and force far more reads from the disk.

To create the random list the Mersenne Twister pseudorandom number generator (PRNG) as proposed by Matsumoto and Nishimura (1998) was used. This algorithm provides a uniform random number generator that can reproduce the same sequence of numbers across platforms given the same initial start value of salt. The C++11 standard implementation called MT19937[5] was used.

The three values to initialise the PRNG for each run were generated on www.random.org. They used atmospheric noise as the source to generate random numbers. The values were generated with a minimum setting of 1 and maximum setting of 100,000. The resulting values are 35298, 4479 and 46636.

5.3 Validation of Archiving Process

To evaluate the performance of the archive it is necessary to confirm that the sample log file was correctly processed and stored in the archive. This was done in two parts: first comparing the event count in the archive with a manual calculation (Section 5.2.2) of the

[5]http://www.cplusplus.com/reference/random/mt19937/

number of events in each log file, and then in a separate run of the experiment, doing an event by event comparison of the SHA1 hash of the original event calculated at archive time with the SHA1 hash of the events read from the archive.

5.3.1 Event Count

The first result that needs to be considered is the event count to ensure that the archiving system processed the correct number of events when the sample logs were processed. Table 5.3 shows the number of events contained in the log files. This includes the manual event count calculation described in Section 5.2.2 as well as the event count reported by the archiving system after each log file was ingested.

Table 5.3: Event count in sample files

Application Type	Sample File Name	Manual Event Count	Archive Event Count
`dns`	event_bind_small.log	101,562	101,562
`mail`	event_maillog_small.log	74,178	74,178
`pcap-text`	event_pcap_text_small.log	76,864	76,864
`firewall`	event_pflog_small.log	76,107	76,107
`proxy`	event_proxy_small.log	47,056	47,056
`windows-event`	event_wineventlog_-small.log	7,758	7,758

Table 5.3 shows that both the manual calculation and the event count reported agree. It can therefore be deduced that event breaking was done correctly.

5.3.2 SHA1 Validation

After each sample log was processed, the experimental run validated each event using the method described in Section 5.2.4. The validation was successful for each iteration of the experimental run. This means that the event data read from the input were correctly archived to and retrieved from the implementation.

5.4 Evaluation of Results

The results from this pseudo-random access compressed archive performance testing were compared with those from the previous chapter (Chapter 4) to determine whether the compression methods show the same result profiles. Next, the performance of the archiver was evaluated for each of the metrics namely compression ratio, archive time and reading time. Finally, the cost related to compression ratio and random access for using the archiving system was determined.

5.4.1 Weighted Compression Ratios

As in the previous experiment in Chapter 4, the compression ratio was calculated for each run of the experiment by dividing the original log sample uncompressed file size by the archive file size.

Figure 5.3 shows the weight results of the compression ratio for each compression method and block size. Again the `lzma` and `lzma2` (`xz`) families provide the best compression ratios over all sample log types and block sizes used. The best compression is again obtained from `lzma2 -9` followed by `lzma -9`.

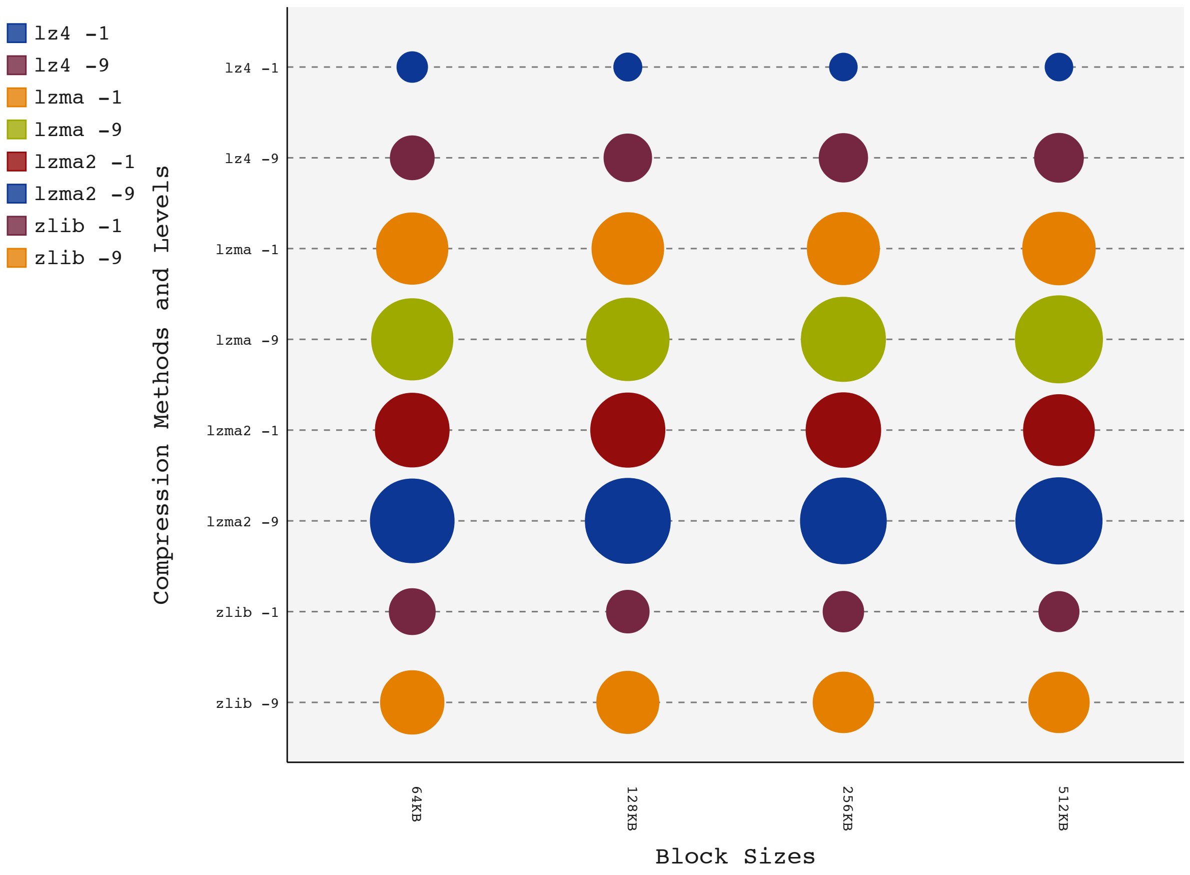


Figure 5.3: Weighted compression ratio in archive

From these results it is clear that the additional metadata introduced by the archiving process has the same compression profile as shown in the first experiment. Furthermore, these results are consistent across all of the block sizes.

5.4.2 Average Compression Ratios

The compression ratios provide further insights into the performance of the compression methods used in the archive. The mean of each compression method was calculated over each log sample and block size.

Figure 5.4 depicts the mean compression ratios. From this we can see that `lzma -9` and `lzma2 -9` yield significantly higher compression ratios than the other options. This also shows that there is a marked increase in compression ratios obtained with an increase in block size used.

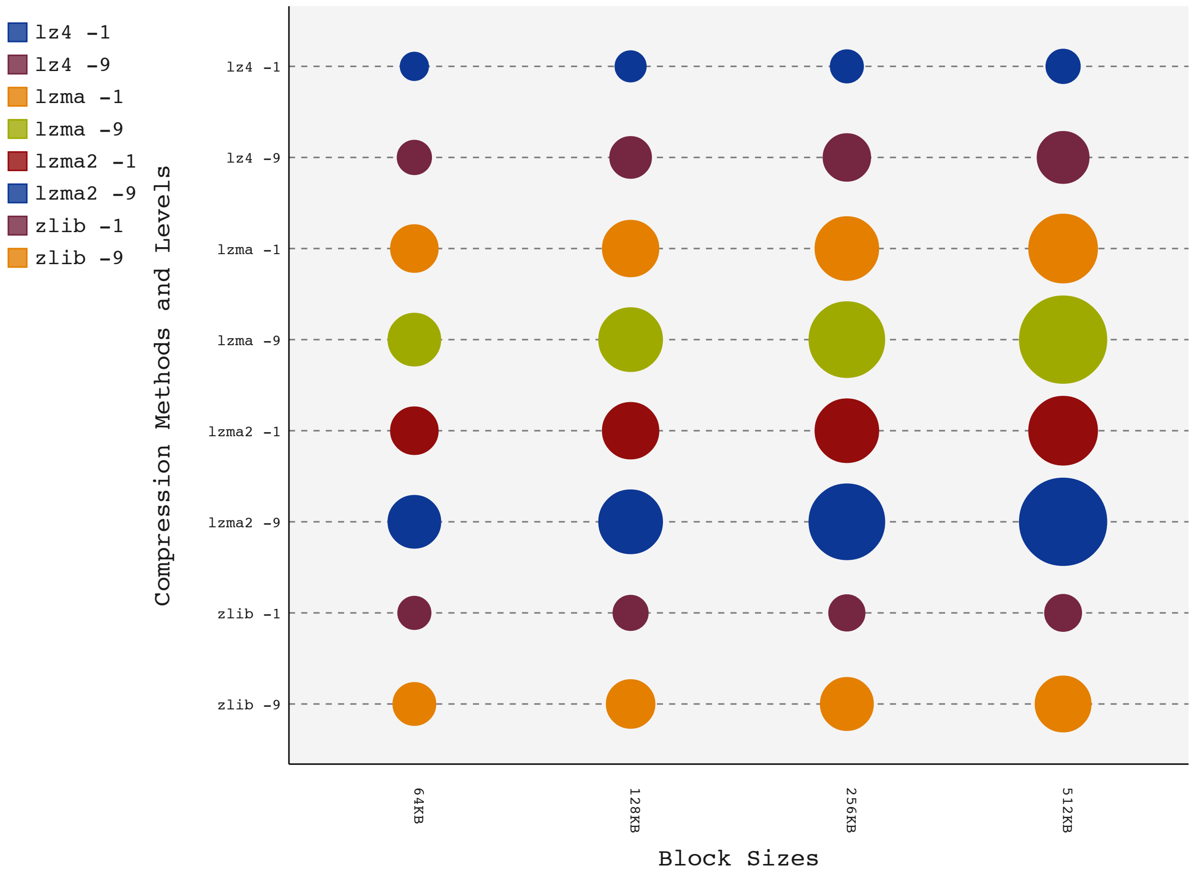


Figure 5.4: Average compression ratios in archive

Table 5.4 shows how significant the increase in compression yield is on the different block sizes. For example, the `64KB` block for `lzma2 -9` has a compression ratio of 6.3311, whereas that for the `512KB` block for the same method is 10.5311. This represents an increase of 66.34%[6] over the `64KB` block compression ratio. This shows that regardless of compression method, compression or compression level used, larger event data blocks have a better compression.

Table 5.4: Mean compression ratios in archive per method per block size

Compression Method	64KB	128KB	256KB	512KB
lz4 –1	3.3595	3.7089	3.9482	4.0915
lz4 –9	4.0952	4.9921	5.6942	6.2275
lzma –1	5.7068	6.7789	7.6546	8.2752
lzma –9	6.3270	7.6694	9.1167	10.5305
lzma2 –1	5.7099	6.7813	7.6568	8.2756
lzma2 –9	6.3311	7.6726	9.1213	10.5311
zlib –1	3.9527	4.1947	4.3291	4.4098
zlib –9	5.1308	5.8290	6.3600	6.7146

5.4.3 Compression Loss

The final part of evaluating the compression ratio results, involved comparing the results from the archiving system with the results of the command line tools.

From previous results it was determined that `xz` was the best command line tool and `lzma2` with `512KB` blocks was the best performing combination in the archiver. Table 5.5 shows the compression ratio achieved using the `xz` command line tool to compress the full length sample log file (Table 5.2), the compression ratio achieved by the archiver, and the loss in compression ratio.

Table 5.5: Compression ratio loss

Application Type	Command Line Ratio	Archive Ratio	Ratio Loss
`dns`	10.70	5.67	5.03 (47.04%)
`mail`	11.48	6.83	4.65 (40.50%)
`pcap-text`	10.77	6.71	4.06 (37.69%)
`firewall`	15.65	10.72	4.93 (31.52%)
`proxy`	9.96	5.85	4.11 (41.27%)
`windows-event`	96.52	48.47	48.04 (49.78%)

From Table 5.5 it is clear that there is a significant loss in compression ratio using the

[6](10.5311 - 6.3311) /6.3311

archiver instead of the command line tool. The loss ranges from 31.52% to 49.78% for the given data sets and compression method.

This loss can be attributed to the fact that the archiver uses smaller blocks which does not allow the compression methods to build bigger dictionaries. Furthermore the archiver introduces binary data, for example, the two 64 bit integer timestamps at the start of each event. As this has a high cardinality, the compression ratio is lower.

5.4.4 Weighted Archive Write Times

Figure 5.5 shows the weighted archiving times for each compression method, compression level and block size. Again the weighted times for each block agrees with the initial data collected during the compression method evaluation, with `lz4 -1` being the fastest.

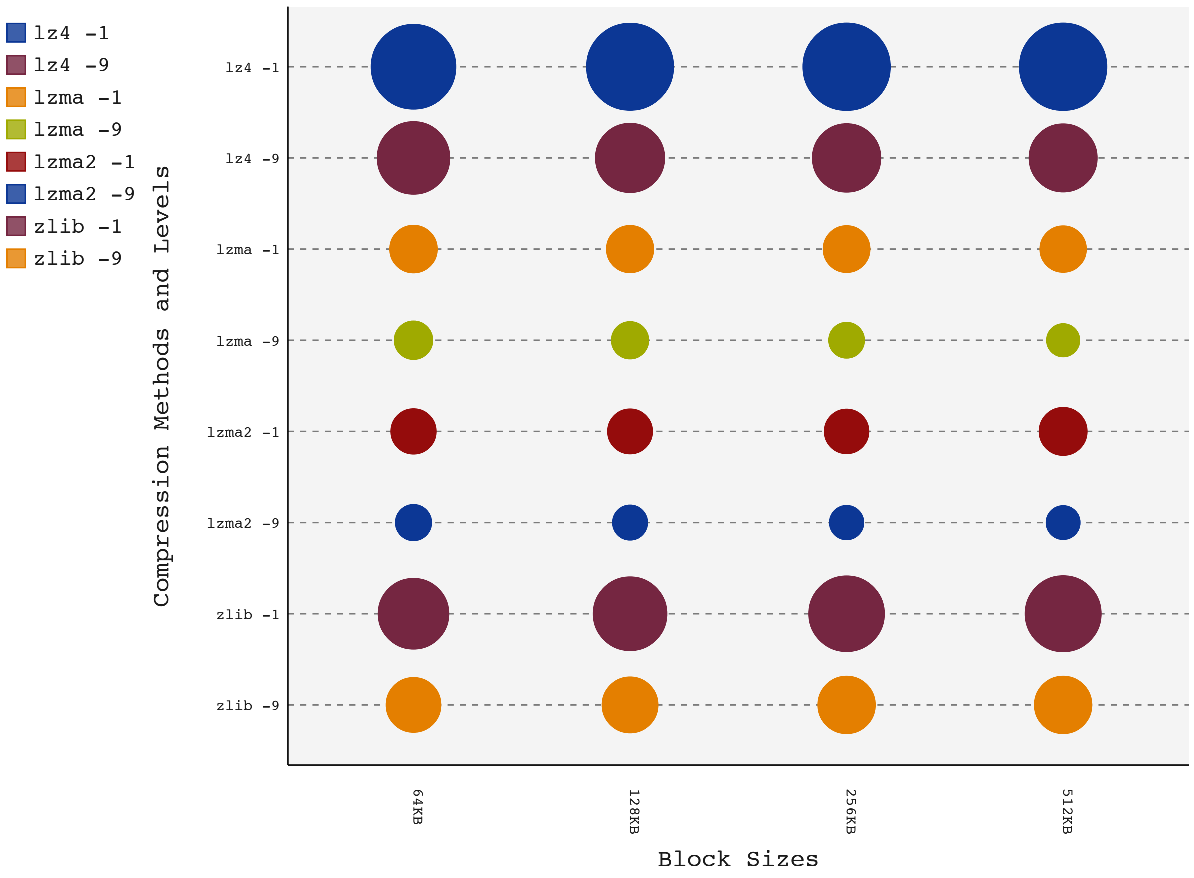


Figure 5.5: Weighted times for archiving sample logs

5.4.5 Average Archive Write Times

Line breaking, event breaking and timestamp parsing add processing overhead when ingesting the sample log data into the archive, which increases the time needed to ingest the log data into the archive. The average write times reflect the influence of the line breaking, event breaking, timestamp extraction and serialising event data to disk.

Apart form `lzma -9` and `lzma2 -9`, all the other compression methods and levels are very close to each other. So the speed advantages of some compression methods like `lz4` is largely negated by the processing overhead introduced. The slow compression speed of `lzma -9` and `lzma2 -9`, however still exceeds the introduced overhead.

The processing overhead also negates any time gained when using different block sizes. Only `lzma -9` and `lzma2 -9` show a marked decrease in compression time as the block size increases.

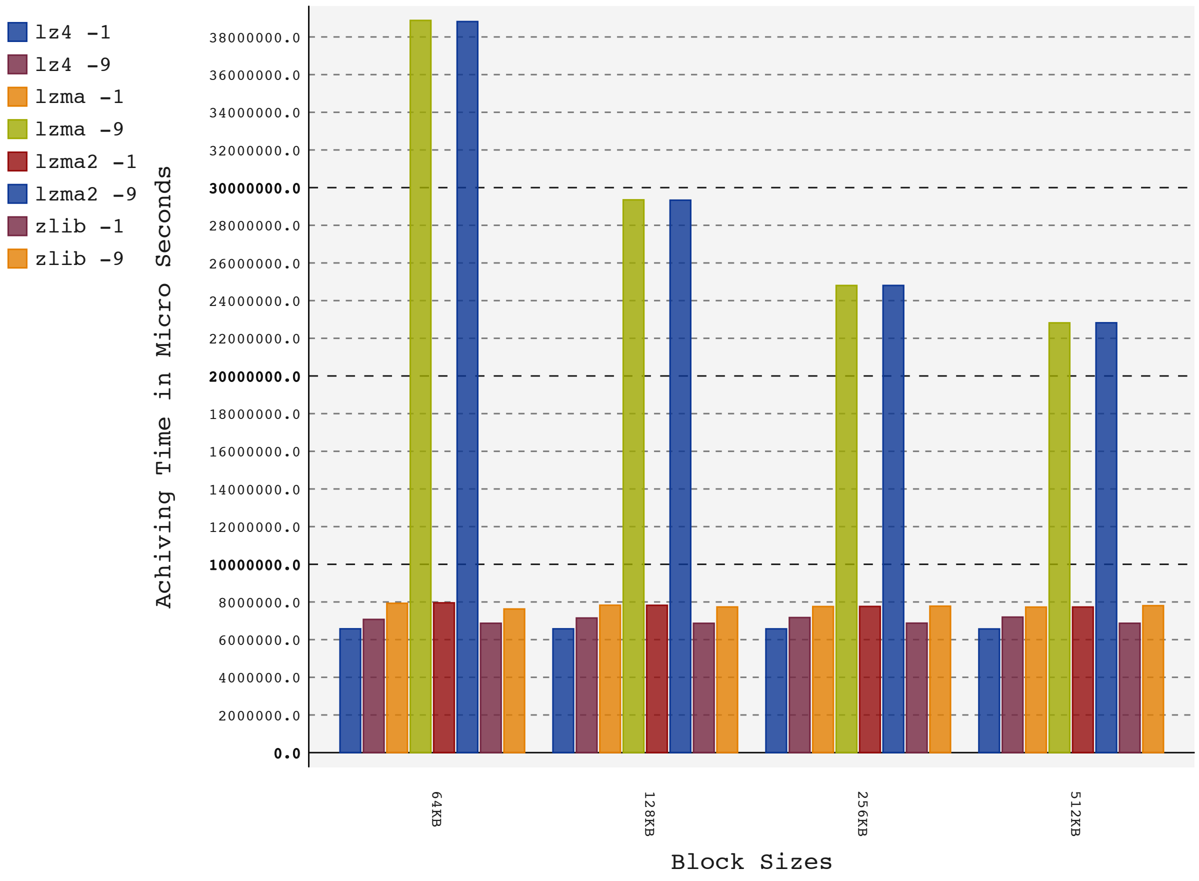


Figure 5.6: Average times for archiving sample logs

5.4.6 Weighted Block Retrieval Times

Figure 5.7 shows the weighted values for the block retrieval times. This again confirms that `lzma` and `lzma2` are the slowest decompression methods while `lz4 -1` is the fastest. This confirms that the compression profiles are the same for the archive and the command line tools.

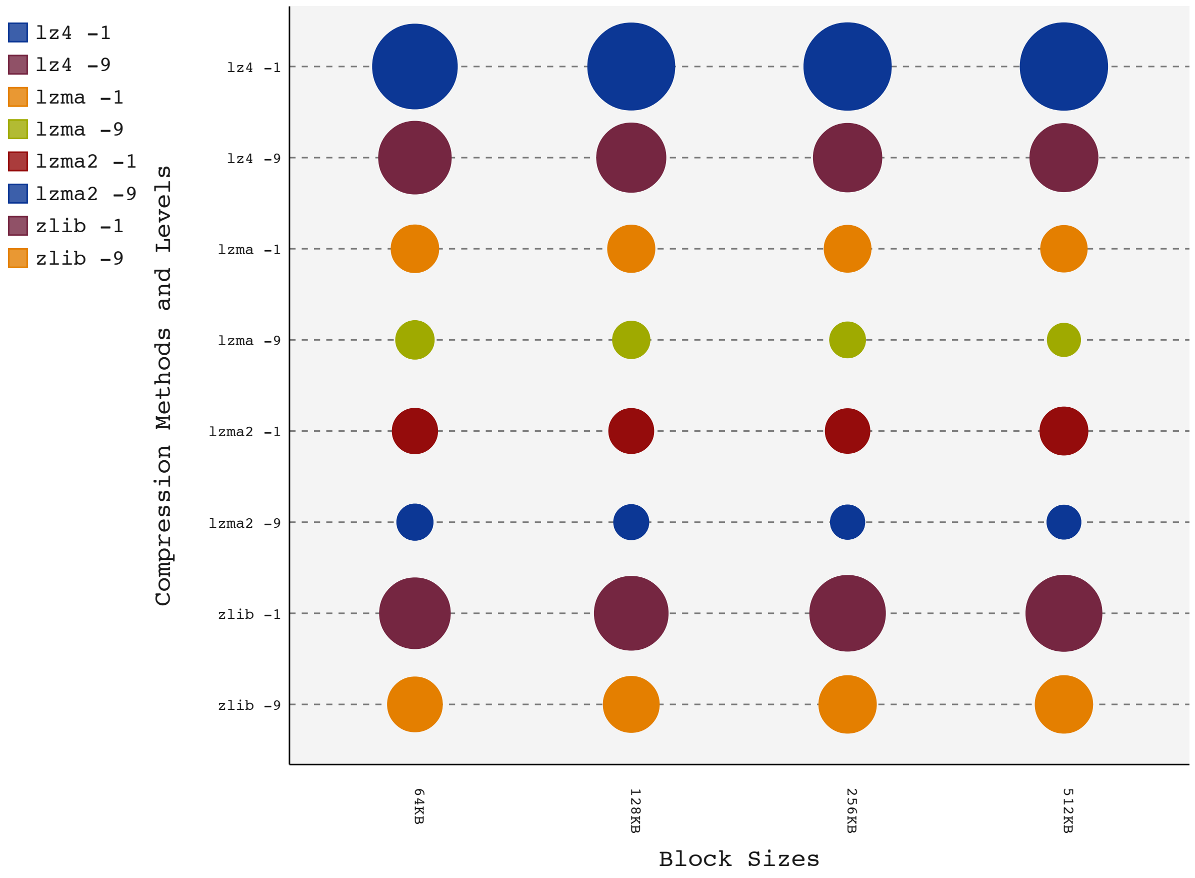


Figure 5.7: Weighted times for reading blocks from archives

5.4.7 Average Block Retrieval Times

The average block retrieval times are shown in Figure 5.8. Again `lz4` is the fastest algorithm by a large margin, with `lzma` and `lzma2` the slowest two compression methods. This figure shows that, except for `lzma` and `lzma2`, there is no significant gain in retrieval based on the block sizes and even for the exceptions the gain is not very large.

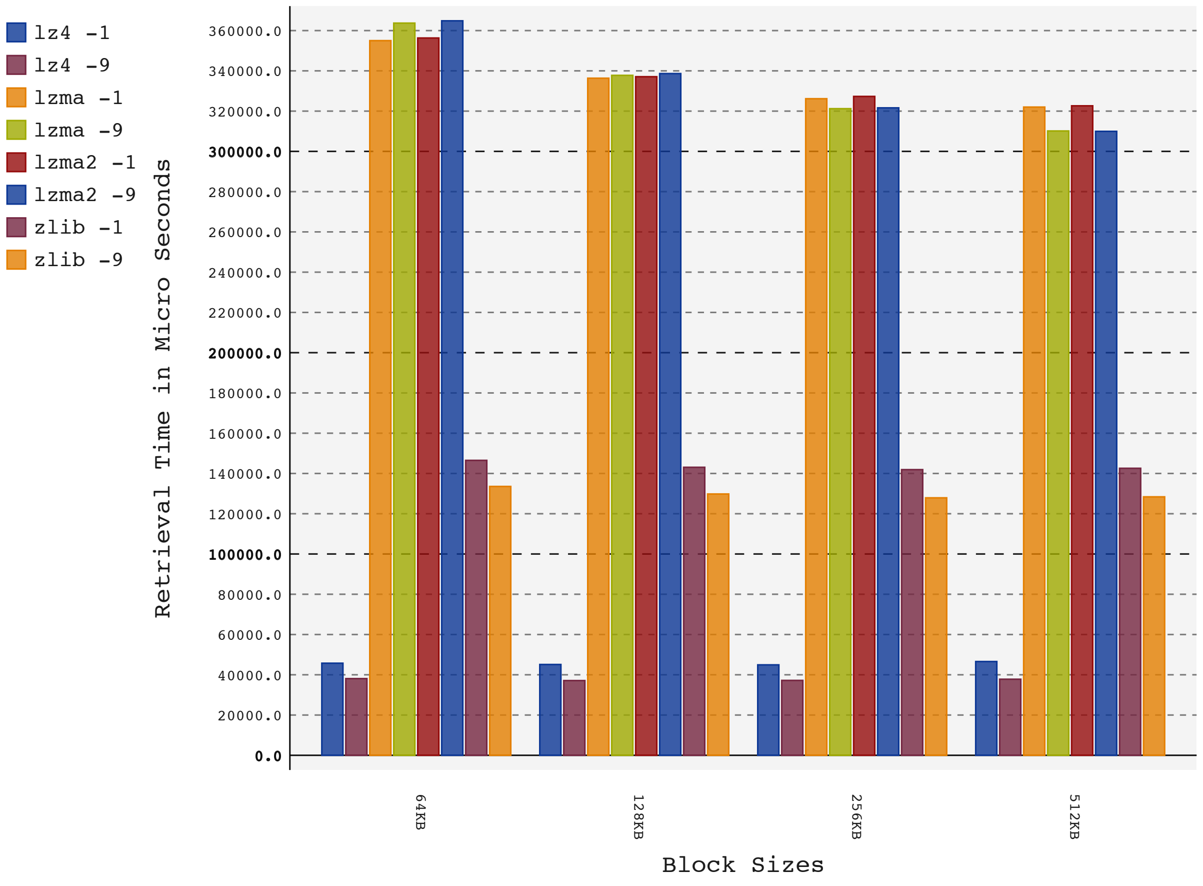


Figure 5.8: Average block retrieval times

5.4.8 Random Access Read Times

Random access is one of the major objectives of the implementation. Reading times for events, therefore, are measured for both sorted and randomly ordered entry identifiers. The sorted times were averaged over all the log sample files while the randomly ordered times were averaged over three random retrieval times and then the calculation was averaged by the number of log sample types.

Figure 5.9 depicts the average read times of single events in a sequential order, thus limiting the number of compressed data blocks read from disk and decompressed in memory. The chart shows a small variance in performance between the compression methods, with `lz4` and `zlib` showing the best performance. The detailed times can be found in Table D.1, which shows that there is no substantial change in read times for each compression method for different event data block sizes. For example `lzma2 -1` takes `1.83`, `1.81`, `1.80` and `1.80` seconds for `64KB`, `128KB`, `256KB` and `512KB` event block sizes, respectively.

Figure 5.10 shows the average read times for single events in random order; the random

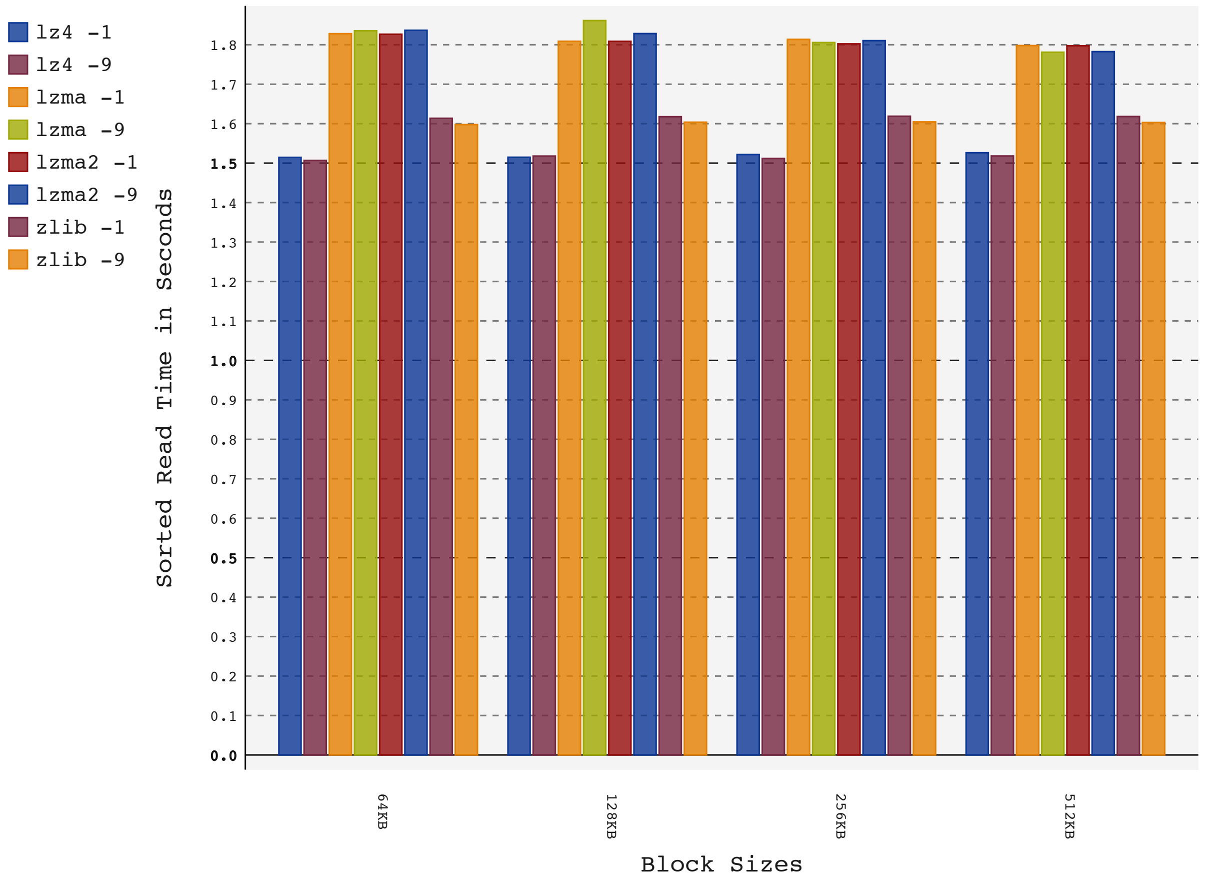


Figure 5.9: Average retrieval time based on sorted entry identifiers

nature of the access forces an almost continuous reload of event data blocks from the storage and decompression thereof each time. The chart shows a significant increase in the time needed to retrieve all the events. For `lzma2 -9` using a `512KB` block, according to Table D.2, the read times for sorted and random access increase from `1.79` seconds to `714.74` seconds, respectively. This is due to the significant increase in the number of event data blocks read. Take, for example, the `mail` log sample using `512KB` blocks (Table 5.6); the sorted access is `25` page reads while the three random accesses required `59,260`, `59,273` and `59,302`, respectively.

Figure 5.10 also shows that performance loss is further increased as the block sizes increases. For `lzma2 -9` the average read times are `126.31`, `229.57`, `416.68` and `714.74` seconds for `64KB`, `128KB`, `256KB` and `512KB` event block sizes respectively. The random nature of the retrieval negates any possible benefit of the event data block caching (Section 5.1.3) used.

Table 5.6: Number of event data block reads from archive

Compression Method	Block Size	Event Count	Sorted	Random 1	Random 2	Random 3
dns	64KB	101,562	213	99,136	99,105	99,146
dns	128KB	101,562	107	96,818	96,722	96,805
dns	256KB	101,562	54	92,099	91,893	91,883
dns	512KB	1015,62	27	82,639	82,430	82,427
firewall	64KB	76,107	201	74,188	74,181	74,222
firewall	128KB	76,107	101	72,307	72,192	72,230
firewall	256KB	76,107	51	68,413	68,356	68,415
firewall	512KB	76,107	26	60,766	60,727	60,776
mail	64KB	74,178	199	72,265	72,343	72,379
mail	128KB	74,178	100	70,401	70,489	70,360
mail	262144	74,178	50	66,612	66,716	66,692
mail	**512KB**	**74,178**	**25**	**59,260**	**59,273**	**59,302**
pcap-text	64KB	76,864	201	74,925	74,942	74,856
pcap-text	128KB	76,864	101	73,044	73,026	72,928
pcap-text	256KB	76,864	51	69,216	69,252	69,095
pcap-text	512KB	76,864	26	61,709	61,664	61,413
proxy	64KB	47,056	185	45,744	45,777	45,801
proxy	128KB	47,056	93	44,415	44,495	44,568
proxy	256KB	47,056	47	41,862	41,969	42,012
proxy	512KB	47,056	24	36,803	36,848	36,974
winevent	64KB	7,758	163	7,525	7,519	7,509
winevent	128KB	7,758	82	7,271	7,287	7,272
winevent	256KB	7,758	41	6,806	6,830	6,770
winevent	512KB	7,758	21	5,906	5,880	5,888

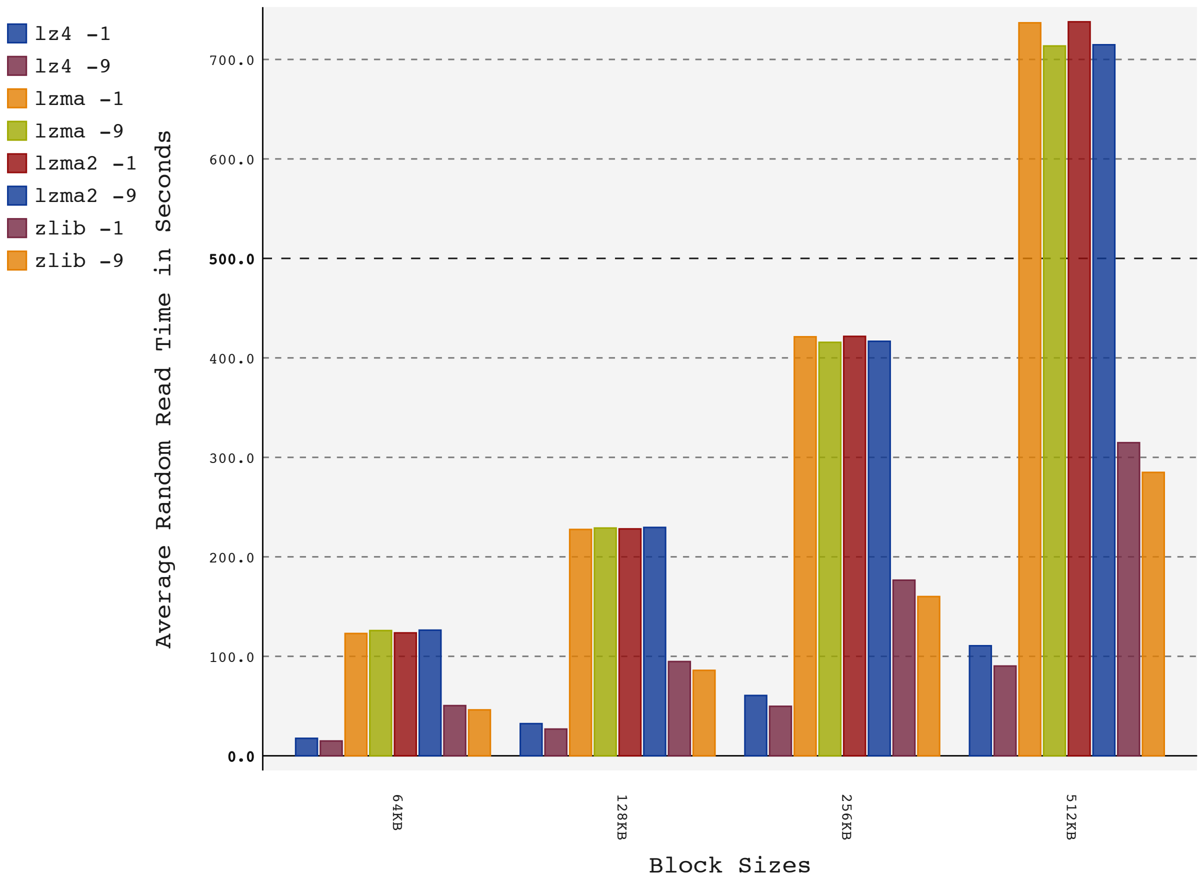


Figure 5.10: Average retrieval time based on randomly ordered entry identifiers

5.5 Summary

Corresponding with the results achieved in the previous chapter, `lzma2 -9`(`xz`) and `lzma -9` provide the best compression ratio. There is however a significant loss in compression ratio of between `31.52%` and `49.78%` depending on the log sample type. Compression ratio also improves with the use of larger event data block sizes.

Again `lz4 -1` provides the fastest read and write times. This holds true for accessing events in both a sequential and random manner. Random access also introduces significant increase in retrieval times as the event data block size used increases.

Chapter 6

Conclusion and Future Work

This chapter provides our conclusions based on the results obtained in the previous two chapters. Lastly, possible future work is discussed.

6.1 Contribution of this Work

The research objectives of this work were: to present a random access compressed archive for storing heterogeneous text security log data and to evaluate the impact of different compression methods and block sizes.

The research has shown that a pseudo-random access compressed archive can provide random access to event-broken security log entries with an entry identifier created at archival time. There is, however, a significant decrease in the compression ratio, and increase in compression and decompression times. This cost is offset by the ability to be able to randomly access individual entries thereby making it suitable for indexing in IR systems and other indexing implementations.

It was further shown that when performing batch retrieval from the archive, the event identifiers should be sorted so that the event data blocks are sequentially accessed. This should alleviate the very high retrieval time associated with pure random access to the archive.

Based on the experimental results, the compression method selected for this archive depends on the specific use case. In cases where compression is of the utmost importance, `lzma -9` with 512KB event data blocks should be used. In cases where random access dominates without the ability to sort the entry identifiers before retrieval, `lz4 -9` with 64KB event data blocks is the best choice. When a middle ground is required, `zlib -9` with a 64KB block offers the best solution. The 64KB event data block is used with `zlib -9` as it offers the best random retrieval time and there is no significant increase in compression ratio with bigger blocks that can offset the random retrieval time benefit.

This work provides a strong foundation for storing log event data in such a way that it can be used for full text indexing and analytics.

6.2 Future Work

Our research highlighted some concepts that could benefit from further research.

Implementation of an inverted index using the proposed archiving system will provide insight into the performance of the archive as the basis of a IR system, especially in a distributed environment where the archives can be used as shards (Roy, 2008), both on a single server as well as on individual nodes in a cluster.

While serialising the events to the archive (see Section 3.5.2) a number of binary elements are introduced to the log data that have a detrimental effect on the achievable compression ratio. Future researchers can expand on this work by separating these binary elements and text data within an event data block (see Section 3.5). The separation will add processing overhead when reading data from the archive, but the gain in compression ratio should offset the processing cost.

The archive evaluates several compression methods to compress whole blocks. An alternative would be to compress the text part of individual entries only using an algorithm like the Burrows-Wheeler Transformation (Burrows and Wheeler, 1994) with some variant of Huffman Coding (Huffman, 1952), as discussed in Sections 2.3.3 and 2.3.1. The results from this could provide better insight into the cost of compressing individual entries as opposed to big data blocks containing both binary and text data.

References

Berners-Lee, T. *RFC 2396: Uniform Resource Identifiers (URI).* 1998. Online.
Retrieved from `http://www.rfc-archive.org/getrfc.php?rfc=2396`
Retrieved on 20 November 2014.

Blanc, B. and Maaraoui, B. *Endianness or where is byte 0.* 2005. Online.
Retrieved from `http://3bc.bertrand-blanc.com/endianness05.pdf`
Retrieved on 8 August 2014.

Burrows, M. and Wheeler, D. J. *A block-sorting lossless data compression algorithm.* Technical report, Systems Research Center of Digital Equipment Corporation, 1994. doi:10.1.1.141.5254.

Büttcher, S., Clarke, C. L. A., and Cormack, G. V. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press, 2010. ISBN 978-0-262-02651-2.

Deutsch, L. P. *GZIP file format specification version 4.3.* 1996. Online.
Retrieved from `http://tools.ietf.org/html/rfc1952`
Retrieved on 2 November 2014.

Donovan, J. *Wireless Data Volume on Our Network Continues to Double Annually.* February 2012. Online.
Retrieved from `http://www.attinnovationspace.com/innovation/story/a7781181`
Retrieved on 26 November 2014.

Faith, R. *dictzip.* June 1997. Online.
Retrieved from `http://www.linuxcommand.org/man_pages/dictzip1.html`
Retrieved on 13 September 2013.

Fenwick, P. M. *The Burrows–Wheeler Transform for Block Sorting Text Compression: Principles and Improvements. The Computer Journal*, 39(9):731–740, 1996. doi:10.1093/comjnl/39.9.731.

Frakes, W. B. and Baeza-Yates, R. *Information Retrieval - Data Structures & Algorithms.* Prentice Hall, 1992.

Gantz, J. and Reinsel, D. *The digital universe decade-are you ready.* Technical report, EMC Corporation, 2010.

Gantz, J. and Reinsel, D. *The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east.* Technical report, EMC Corporation, 2012.

Huffman, D. A. *A method for the construction of minimum redundancy codes. Proceedings of the IRE*, 40(9):1098–1101, 1952. doi:10.1109/JRPROC.1952.273898.

ISO. *ISO 8601:2004 Data elements and interchange formats. Information interchange. Representation of dates and times.* International Organization for Standardization, 2005.

Kiselkov, S. *LZ4 Compression.* January 2013. Online.
Retrieved from `http://wiki.illumos.org/display/illumos/LZ4+Compression`
Retrieved on 3 September 2014.

Klyne, G. and Newman, C. *RFC 3339: Date and Time on the Internet: Timestamps.* 2002. Online.
Retrieved from `http://www.ietf.org/rfc/rfc3339.txt`
Retrieved on 19 May 2014.

Matsumoto, M. and Nishimura, T. *Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998. doi:10.1145/272991.272995.

Moffat, A. and Zobel, J. *Self-indexing inverted files for fast text retrieval. ACM Trans. Inf. Syst.*, 14(4):349–379, 1996. doi:10.1145/237496.237497.

Navarro, G. and Makinen, V. *Compressed full-text indexes. ACM Computing Surveys (CSUR)*, 39(1), 2007. doi:10.1145/1216370.1216372.

Press, G. *A Very Short History Of Big Data.* May 2014. Online.
Retrieved from `http://www.forbes.com/sites/gilpress/2013/05/09/a-very-short-history-of-big-data/2/`
Retrieved on 26 November 2014.

Roy, R. *Shard – A Database Design.* July 2008. Online.
Retrieved from `http://technoroy.blogspot.com/2008/07/`

`shard-database-design.html`
Retrieved on 8 January 2015.

Shafranovich, Y. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Technical report, Internet Engineering Task Force, October 2005.

Shannon, C. E. *A Mathematical Theory of Communication*. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, 2001. doi:10.1145/584091.584093.

Workgroup, D. *Dwarf Debugging Information Format Version 3*. 2005. Online.
Retrieved from `http://dwarfstd.org/doc/Dwarf3.pdf`
Retrieved on 9 August 2014.

York, K. *A Random Access Compressed File Layer*. July 2001. Online.
Retrieved from `http://www.drdobbs.com/article/print?articleId=184401415&siteSectionName=`
Retrieved on 12 September 2013.

Ziv, J. and Lempel, A. *A universal algorithm for sequential data compression*. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977. doi:10.1109/TIT.1977.1055714.

Zobel, J. and Moffat, A. *Inverted files for text search engines*. *ACM Computing Surveys (CSUR)*, 38(2), 2006. doi:10.1145/1132956.1132959.

Appendix A

Test Platform

The hardware used for all the experiments in this work is listed below. This is followed by the `bonny++` statistics of the hard drive that provides the base line for the hard drive performance.

The server is a **HP Proliant MicroServer** with the following specifications:

- AMD Turion II Neo N54L (2.2GHz, 15W, 2MB, 2 Core)
- (2 x 2GB) PC3–10600E (UDIMMs)
- 1 x 250GB 7200RPM HDD
- Embedded AMD SATA controller (with RAID 0, 1 4 Internal HDD Support)
- AMD SATA controller (RAID 0, 1)
- NC107i Gigabit Adapter
- 150W Power Supply
- 4 x LFF NHP SATA Ultra Micro Tower.

Self-Monitoring, Analysis and Reporting Technology[1] tool in FreeBSD shows the following information for the hard disk:

[1] www.smartmontools.org

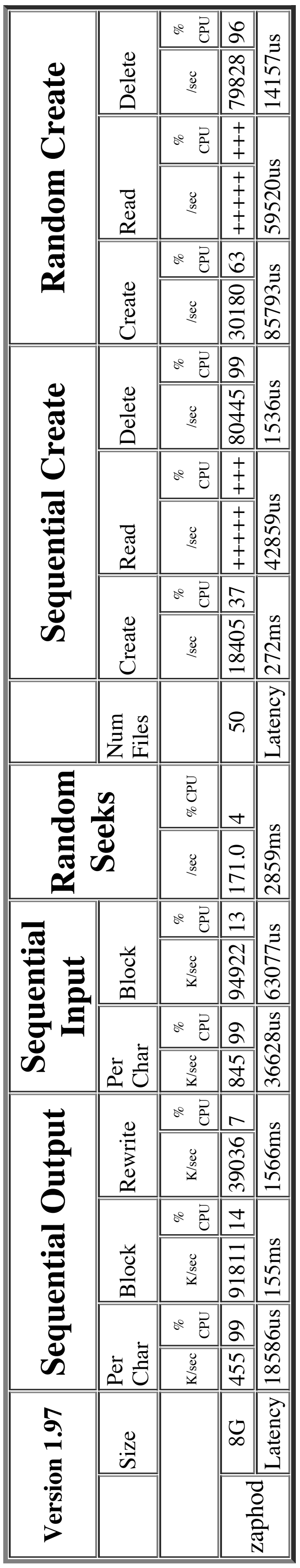
```
/usr/local/sbin/smartctl -i /dev/ada0
smartctl 6.3 2014-07-26 r3976 [FreeBSD 10.0-RELEASE-p3 amd64] (local build)
Copyright (C) 2002-14, Bruce Allen, Christian Franke, www.smartmontools.org

=== START OF INFORMATION SECTION ===
Model Family:     HP 250GB SATA disk VB0250EAVER
Device Model:     VB0250EAVER
LU WWN Device Id: 5 000c50 05081fc40
Firmware Version: HPG9
User Capacity:    250,059,350,016 bytes [250 GB]
Sector Size:      512 bytes logical/physical
Rotation Rate:    7200 rpm
Form Factor:      3.5 inches
Device is:        In smartctl database [for details use: -P show]
ATA Version is:   ATA8-ACS T13/1699-D revision 6
SATA Version is:  SATA 2.6, 3.0 Gb/s (current: 3.0 Gb/s)
Local Time is:    Tue Nov 25 08:37:24 2014 SAST
SMART support is: Available - device has SMART capability.
SMART support is: Enabled
```

The benchmarking tool `bonnie++ v1.97`[2] is used to obtain statistics for the hard disk. Listing A.1 shows the command executed. Figure A.1 contains the statistics obtained for the hard disk.

```
bonnie++ -s 8g -u root -n 50 -d /tmp/bonnie
```

Listing A.1: bonnie++ command

[2] http://www.googlux.com/bonnie.html



Version 1.97		Sequential Output						Sequential Input				Random Seeks			Sequential Create						Random Create					
	Size	Per Char		Block		Rewrite		Per Char		Block				Num Files	Create		Read		Delete		Create		Read		Delete	
		K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	/sec	% CPU		/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU
zaphod	8G	455	99	91811	14	39036	7	845	99	94922	13	171.0	4	50	18405	37	+++++	+++	80445	99	30180	63	+++++	+++	79828	96
	Latency	18586us		155ms		1566ms		36628us		63077us		2859ms		Latency	272ms		42859us		1536us		85793us		59520us		14157us	

Figure A.1: bonnie++ results

Appendix B

Log File Samples

The appendix contains extracts from the sample logs used in Chapters 4 and 5.

B.1 PF Firewalls: firewall

```
Mar  3 10:24:55 11.123.215.66 10: 20:24.715440 rule 0/0(match): block in on em1: 11.130.45.170.3813 > 86.107.252.66.7799: UDP, length 32
Mar  3 10:24:55 11.123.215.66 10: 20:24.718632 rule 0/0(match): block in on em2: 192.16.0.207.2291 > 11.171.147.232.445: tcp 12 [bad hdr length 16 - too short, < 20]
Mar  3 10:24:55 11.123.215.66 10: 20:24.737414 rule 0/0(match): block in on em1: 11.201.194.84.1496 > 188.240.68.129.6432: UDP, length 46
Mar  3 10:24:55 11.123.215.66 10: 20:24.739743 rule 0/0(match): block in on em2: 192.16.1.41.4661 > 192.16.191.146.445: tcp 12 [bad hdr length 16 - too short, < 20]
Mar  3 10:24:55 11.123.215.66 10: 20:24.749981 rule 0/0(match): block in on em2: 192.16.0.207.2292 > 11.171.29.200.445: tcp 12 [bad hdr length 16 - too short, < 20]
Mar  3 10:24:55 11.123.215.66 10: 20:24.754977 rule 0/0(match): block in on em2: 192.16.1.41.4628 > 192.16.40.153.445: tcp 12 [bad hdr length 16 - too short, < 20]
```

Listing B.1: Extract from PF Firewall logs

B.2 Windows Security Event Logs: windows-event

```
04/16/2014 02:57:29 PM
LogName=Security
SourceName=Microsoft Windows security auditing.
EventCode=4634
EventType=0
Type=Information
ComputerName=exp-srv-sql12.dummy.com
TaskCategory=Logoff
OpCode=Info
RecordNumber=21372209
Keywords=Audit Success
Message=An account was logged off.

Subject:
       Security ID:          dummy\spadmin
       Account Name:         spadmin
       Account Domain:       dummy
       Logon ID:             0x3EF89597

Logon Type:                  3

This event is generated when a logon session is destroyed. It may be positively
correlated with a logon event using the Logon ID value. Logon IDs are only
unique between reboots on the same computer.

04/16/2014 02:57:29 PM
LogName=Security
SourceName=Microsoft Windows security auditing.
EventCode=4672
EventType=0
Type=Information
ComputerName=exp-srv-sql12.dummy.com
TaskCategory=Special Logon
OpCode=Info
RecordNumber=21372208
Keywords=Audit Success
Message=Special privileges assigned to new logon.
```

```

Subject:
       Security ID:          NT AUTHORITY\SYSTEM
       Account Name:         SYSTEM
       Account Domain:       NT AUTHORITY
       Logon ID:             0x3E7

Privileges:           SeAssignPrimaryTokenPrivilege
                      SeTcbPrivilege
                      SeSecurityPrivilege
                      SeTakeOwnershipPrivilege
                      SeLoadDriverPrivilege
                      SeBackupPrivilege
                      SeRestorePrivilege
                      SeDebugPrivilege
                      SeAuditPrivilege
                      SeSystemEnvironmentPrivilege
                      SeImpersonatePrivilege
```

Listing B.2: Extract from Windows Security Event logs

B.3 BIND: dns

```
16-May-2012 09:47:38.099 queries: info: client 93.186.18.197#22608: query: ns1.async.org.za IN A -
16-May-2012 09:47:38.478 queries: info: client 208.69.34.10#42480: query: ns1.async.org.za IN A -
16-May-2012 09:47:38.753 queries: info: client 216.32.180.10#49518: query: dsgschool.com IN MX -
16-May-2012 09:47:39.037 queries: info: client 208.69.34.10#41436: query: phantom.moria.org IN A -
16-May-2012 09:47:39.233 queries: info: client 154.32.107.18#36428: query: ns1.async.org.za IN AAAA -
16-May-2012 09:47:39.895 queries: info: client 218.248.255.197#37554: query: gauntlet.mithral.co.za IN AAAA -E
```

Listing B.3: Extract from BIND logs

B.4 Squid Proxy Logs: proxy

```
1380042813.978 29679 196.23.167.67 TCP_MISS/200 4629 CONNECT sites.google.com:443 - DIRECT/155.232.240.59 -
1380042813.978 27726 196.23.167.67 TCP_MISS/200 4629 CONNECT ssl.gstatic.com:443 - DIRECT/74.125.233.47 -
1380042813.978 27726 196.23.167.67 TCP_MISS/200 4629 CONNECT ssl.gstatic.com:443 - DIRECT/74.125.233.47 -
1380042813.978 86265 196.23.167.67 TCP_MISS/200 3325 CONNECT fbcdn-profile-a.akamaihd.net:443 - DIRECT/197.80.130.17 -
1380042813.978 86575 196.23.167.67 TCP_MISS/200 235430 CONNECT fbstatic-a.akamaihd.net:443 - DIRECT/197.80.130.25 -
1380042813.978 85246 196.23.167.67 TCP_MISS/200 3136026 CONNECT fbcdn-video-a.akamaihd.net:443 - DIRECT/165.165.46.25 -
1380042813.978 76970 196.23.167.67 TCP_MISS/200 3181 CONNECT fbcdn-profile-a.akamaihd.net:443 - DIRECT/197.80.130.27 -
1380042813.978 110368 196.23.167.67 TCP_MISS/200 46916 CONNECT www.facebook.com:443 - DIRECT/66.220.152.19 -
```

Listing B.4: Extract from Squid Proxy logs

B.5 Postfix Mail Logs: mail

```
Feb 23 00:00:06 dryder postfix/qmgr[61060]: 0CFE717B86C: removed
Feb 23 00:00:12 dryder postfix/smtpd[84554]: connect from unknown[190.239.81.198]
Feb 23 00:00:13 dryder postfix/smtpd[84554]: NOQUEUE: reject: RCPT from unknown[190.239.81.198]: 450 4.1.8 <veuw@hxhg.net>: Sender address rejected: Domain not found; from=<veuw@hxhg.net> to=<unknown@sacschool.com> proto=SMTP helo=<XIOMI-PC>
Feb 23 00:00:14 dryder postfix/smtpd[84554]: lost connection after RCPT from unknown[190.239.81.198]
Feb 23 00:00:14 dryder postfix/smtpd[84554]: disconnect from unknown[190.239.81.198]
Feb 23 00:00:30 dryder postfix/smtpd[84688]: lost connection after QUIT from static.85-10-211-71.clients.your-server.de[85.10.211.71]
```

Listing B.5: Extract from Postfix mail logs

B.6 PCAP Files: pcap-text

```
10:46:22.607165 IP 196.21.0.65 > 196.21.42.85: ICMP echo request, id 256, seq 6702, length 44
10:46:23.394343 IP 114.37.199.14.1276 > 196.21.42.117.445: Flags [S], seq 511233496, win 65535, options [mss 1440,nop,nop,sackOK], length 0
10:46:23.633206 IP 196.21.0.65 > 196.21.42.86: ICMP echo request, id 256, seq 6958, length 44
10:46:24.603639 IP 196.21.0.65 > 196.21.42.87: ICMP echo request, id 256, seq 7214, length 44
10:46:25.603922 IP 196.21.0.65 > 196.21.42.88: ICMP echo request, id 256, seq 7470, length 44
10:46:26.389151 IP 114.37.199.14.1276 > 196.21.42.117.445: Flags [S], seq 511233496, win 65535, options [mss 1440,nop,nop,sackOK], length 0
10:46:26.603507 IP 196.21.0.65 > 196.21.42.89: ICMP echo request, id 256, seq 7726, length 44
```

Listing B.6: Extract from PCAP file

Appendix C

Additional Information for the Compression Method Evaluation

Table C.1: Compression ratio weighted values

Compression Method		Weighted Total
bzip2	–1	195
bzip2	–9	259
gzip	–1	106
gzip	–9	187
lz4	–1	56
lz4	–9	111
lzma	–1	258
lzma	–9	335
lzop	–1	32
lzop	–9	126
xz	–1	226
xz	–9	304
xz	-e	353
Total		2548

Table C.2: Compression time weighted values

Compression Method		Weighted Total
bzip2	–1	153
bzip2	–9	123
gzip	–1	309
gzip	–9	251
lz4	–1	344
lz4	–9	235
lzma	–1	242
lzma	–9	56
lzop	–1	355
lzop	–9	155
xz	–1	211
xz	–9	58
xz	-e	56
Total		2548

Table C.3: Decompression time weighted values

Compression Method		Weighted Total
bzip2	–1	54
bzip2	–9	30
gzip	–1	245
gzip	–9	284
lz4	–1	287
lz4	–9	318
lzma	–1	185
lzma	–9	183
lzop	–1	306
lzop	–9	320
xz	–1	104
xz	–9	107
xz	-e	125
Total		2548

Appendix D

Random Access Retrieval Times

Table D.1: Average retrieval time based on sorted entry identifiers in seconds

Compression	Method	64KB	128KB	256KB	512KB
lz4	–1	1.51	1.51	1.52	1.53
lz4	–9	1.51	1.52	1.51	1.52
lzma	–1	1.83	1.81	1.81	1.80
lzma	–9	1.84	1.86	1.81	1.78
lzma2	–1	1.83	1.81	1.80	1.80
lzma2	–9	1.84	1.83	1.81	1.78
zlib	–1	1.61	1.62	1.62	1.62
zlib	–9	1.60	1.60	1.60	1.60

Table D.2: Average retrieval time based on randomly ordered entry identifiers in seconds

Compression	Method	64KB	128KB	256KB	512KB
lz4	–1	17.57	32.32	60.64	110.62
lz4	–9	14.90	26.88	49.75	90.23
lzma	–1	122.94	227.53	421.19	736.82
lzma	–9	125.88	228.95	415.63	713.54
lzma2	–1	123.55	228.10	421.62	737.82
lzma2	–9	126.31	229.57	416.68	714.74
zlib	–1	50.44	94.73	176.55	314.75
zlib	–9	46.16	85.90	160.07	284.89