Investigating File Encrypted Material Using NTFS $logfile

Niall McGrath and Pavel Gladyshev

Digital Forensic Investigation Research Group
University College Dublin

Abstract. When an encrypted file is discovered during a digital investigation and the investigator cannot decrypt the file then s/he is faced with the problem of how to determine evidential value from it. This research is proposing a methodology for locating the original plaintext file that was encrypted on a hard disk drive. The technique also incorporates a method of determining the associated plaintext contents of the encrypted file. This is achieved by characterising the file-encryption process as a series of file I/O operations and correlating those operations with the corresponding events in the NTFS $logfile file. The occurrence of these events has been modelled and generalised to investigate file-encryption. This resulted in the automated analysis of $logfile in *FindTheFile* software.

Keywords: NTFS $logfile file, MAC Times, Encryption.

1 Introduction

Law enforcement agencies (LEA) encounter encryption in relation to many crimes. The distribution of illegal material [1] [2] is an example of the many offences associated with file encryption. The use of encryption in general has been cited [3] as a major hurdle in digital investigations. When file-encrypted material is investigated and the file cannot be decrypted, cracked nor bruteforced; there is no formal method or technique to extract evidential value. As a result this research presents a methodology which identifies the original plaintext filename that was encrypted, while also displaying the plaintext contents of the file. This is irrespective of the file being "deleted" or not. A typical scenario that occurs is where an encrypted bundle is transmitted to a buyer or intended recipient of illegal material. Encryption software has a common feature of giving the option of deleting the original plaintext file after encryption. This naturally increases the complexity of a digital investigation but does not restrict it; how to recover deleted files is outlined in [4]. The NTFS $logfile file ($logfile) is the fundamental evidence artefact upon which the proposed methodology here is based on.

1.1 Problem Description

The main problem with investigating encrypted material is not being able to establish an evidential link between the encrypted file and the original plaintext file and also

M. Rogers and K.C. Seigfried-Spellar (Eds.): ICDF2C 2012, LNICST 114, pp. 183–203, 2013.

not being able to view the plaintext contents. The approach taken here to solve the problem is to observe the process of encryption and then characterise the sequence of events. Extracting event information from $logfile is central to the approach. This leads to formulating a methodology, where it is modelled, generalised, automated and then applied formally in a case-study.

1.2 Related Work

Cryptopometry methodology can only be used to investigate illegal material when it is encrypted and exchanged using public-private key encryption like OpenPGP or X.509 [5]. *Cryptopometry* also does not reveal the plaintext contents of the encrypted file. It is however elaborated on how the computer forensic investigator can use *Cryptopometry* to identify encrypted material, examine it and extract evidential value from it in [5]. Typically this scenario is where a distributor encrypts the illegal material and posts it into a newsgroup or interest group via anonymous re-mailer or via an instant messenger system. The accomplice who is subscribed to that group receives encrypted material and can decrypt it. The anonymity of all involved parties is preserved and the content cannot be decrypted by bystanders [5].

2 Background Information

In order for an application to encrypt a file's contents the underlying actions that take place can be categorised according to 1) type of I/O event i.e. Read or Write, 2) the processes and the sequence of threads that govern execution and 3) where in the stack does this executable file get called and in what mode i.e. user or kernel. In addition there are numerous NTFS design goals outlined in [6] but the specific components that are of interest in this paper are: $logfile and how it is updated by the Log file service and Master File Table (MFT). ObjectId and how the Distributed Link Tracking (DLT) service and how they facilitate forensic examinations are also of interest.

2.1 I/O File Processing

The steps of the I/O file open process along with the principles of I/O request packet (IRP) processing are detailed in [6]. It is outlined that the runtime library function calls the CreateFile function, and then the kernell32.dll-windows subsystem is called which in turn calls the native NtFileCreate function in Ntdll.dll. The transition into kernel mode (where NtCreateFile in Ntoskrnl.exe is called) and the subsequent commands to the Object Manager and the I/O manager and finally the transition back to user mode are listed.

2.2 NTFS

The journal file for the windows operating system is called $logfile. The $logfile is used to recover from system crashes and unexpected conditions. It has the standard

file attributes and stores the log data in the $DATA attribute. The file is organised into 4,096 byte pages consisting of two parts: the restart and the logging area. The restart area contains information on how to start the recovery after a system failure [7]. There are two types of information recorded here. These are "Redo" and "Undo" information. Redo information is how to reapply one sub-operation of a fully logged ("committed") transaction to the volume if a system failure occurs before the transaction is flushed from the cache. Undo information is how to reverse one sub-operation of a transaction that was only partially logged ("not committed") at the time of a system failure [9]. These "Redo" and "Undo" operation codes are used in a composite manner to form the series of log records that are written to the $logfile when a file operation is performed. The hexadecimal (0x) composite operation codes are used such as 0x0E/0x0F, 0x02/0x00, 0x08/0x00, and 0x14/0x14 for file creation, delete, extending, truncation, information setting and renaming [9]. NTFS guarantees that the transaction will appear on the volume, even if the operating system subsequently fails. A table of values that represent update records for each of the following transactions is presented in [9]: Initializing (0x02), de-allocating (0x03) file record segments, writing the end of file record segments (0x04), creating (0x05) and deleting(0x06) attributes, updating resident (0x07) and non-resident (0x08), setting attribute sizes (0x0B) adding (0x0E) and deleting (0x0F) index entry allocation and setting (0x15) and clearing bits in $bitmap (0x16).

In NTFS the primary data structure is the MFT and every file will have at least one entry in the MFT. The MFT holds information about the files and directories in MFT entries. These MFT entries store attributes; where an attribute is a data structure containing a specific type of information such as a file's filename. NTFS take the form of reading and writing attributes for a given file e.g. the $DATA attribute which is common to every file in the file system [6]. The $STANDARD_INFORMATION attribute contains the timestamp information for each file. This attribute determines the MAC times for a file when the properties of a file are viewed. There is also a $FILE_NAME attribute that contains the MAC time information as it relates to the filename for a given file. Link files are created when a file is opened [8]. Also an ObjectID is described as an attribute that uniquely identifies a file or directory on a volume. This is listed as the location of a file at some point in time; it is made up of a VolumeID and an ObjectID. The ObjectID of a file can be queried using the command line tool *fsutil* [7].

3 Observation towards a Framework

Using three different encryption packages a file was encrypted. The encryption process was monitored using *Process Monitor. Process Monitor* is a system activity monitoring tool which monitors the flow of IRPs between various applications and the NTFS driver. It is an example of a passive filter driver. The output of *Process Monitor* while encryption is taking place is followed. Since three encryption packages were observed, it would be superfluous to illustrate all three here as the events are repetitive; therefore the events of one package (PrivateFile) are illustrated below. The first operation of note in Fig 1 is a file system *QueryOpen* executed on the plaintext file to be encrypted. The *QueryOpen* is initiated by the encryption software *exe*

process. A file handle is created and results in a successful retrieval of file attributes like the MAC times along with allocated size. The stack trace of this event originates from Kernel mode (ntkrnlpa.exe) to user mode (msvbvm60.dll). Next there is a file system *CreateFile* call for read access to the file plaintext file. This results in an *opened* status. Similarly the stack trace originates in kernel mode and traverses to user mode. Finally there is a *CloseFile* instruction whose job is to close down and free up the previous IRP associated resources. The file is now ready to be read.

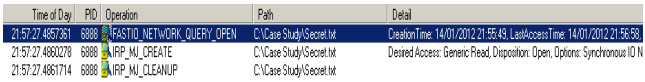

Fig. 1. Initial File System Events with Plaintext file

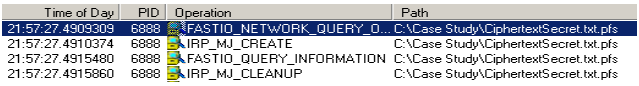

Fig. 2. File System Events with Ciphertext file

Similarly there are file system calls (*QueryOpen, CreateFile and CloseFile)* executed on the ciphertext file. The name of this file is inputted by the user or the software automatically populates the filename field by just appending the new file extension to the original plaintext file name, Fig 2. There is also a *QueryStandardInformationFile* call to query allocation size and determine if the entry is a directory or not; the file is now prepared to be written to. There are file system calls (*CreateFile & CloseFile)* but in addition there are calls to read the contents of the plaintext file and also to write the plaintext contents to the first temporary file, Fig 3.

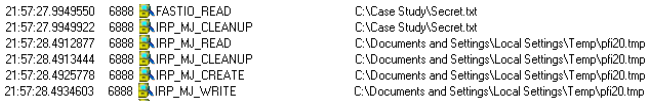

Fig. 3. Temporary file1 system events

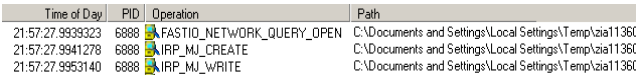

Fig. 4. Temporary file2 system events

Subsequently the plaintext contents are re-written to a second temporary file where the contents of this are encrypted, in Fig 4. Next there is a new handle created to the ciphertext file and then the encrypted contents of temporary file 2 are written into the designated ciphertext file. Please see Fig 5. The two temporary files during the encryption process are deleted. This is achieved by the file system calling a

setDispositionInformationFile call, while passing a boolean variable *Delete* set to true. There is a file system *QueryOpen* called on the plaintext file and this returns the MAC times of the file. Then there is also a *QueryInformationVolume* where the volume- create time and volume serial number are returned. There is also a flag (*SupportObjects*) returned to indicate whether *Objects* are supported or not, this is a reference to the DLT service mentioned earlier. Since the value returned here is *true* there is a subsequent file system control call to retrieve the *ÒbjectID.* This is the objectid of the birth volume i.e. volume id of where the plaintext file was originally created.

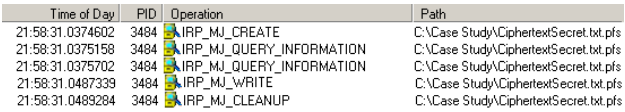

Fig. 5. Encrypted material written to ciphertext file

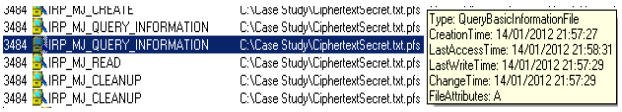

Fig. 6. Timelines and timestamps

When a file is accessed or read for file encryption purposes, the last access time attribute of the plaintext file will indicate the approximate creation time of the encrypted file. The approximation is caused by the difference in time or lag between the plaintext file contents being read, buffered in a temporary file then written to a second temporary file where the encryption is carried out. Once encryption is completed, the ciphertext data is written to the output file. The timestamp for when the IRP_MJ_CLOSE (fileClose) executes on the ciphertext file indicates the last access timestamp, as can be seen in Fig 6. The ciphertext file create timestamp is later than the last access timestamp for the plaintext file, in addition the ciphertext file last access timestamp is later than that of the plaintext file.

4 Characterise the Encryption Process

As can be seen from the observations above the flow of the encryption process can be characterised as a series of file I/O actions. The flow of data through the encryption process is depicted in Fig 7. The overall result of observing *Privatefile* encryption is that four files are created (when plaintext file in not deleted); two temporary files, a lnk file and the ciphertext file. The different interfaces (files) that the data flows through can be seen and it is at these “touch-points” along the process flow where evidence can be retrieved. This is because particular events, as indicated in [7] that occur at each of these points are recorded chronologically in the $logfile e.g. each mft update is recorded in $logfile and its entry is preceded with the following string “FILE0”.

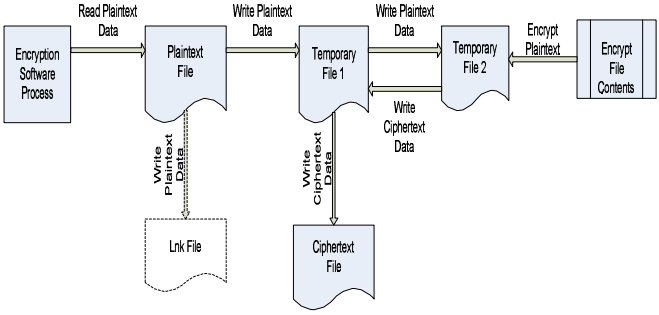

Fig. 7. Flow of data through

4.1 Sequence of Events That Constitute the Encryption Process

The sequence model in Fig 8 displays the order in which events take place. This is important in understanding the contents of the $logfile because a forensic picture of events can be constructed. The individual events listed here can be classified into various groups of event-types see table 1. This is needed in forming event sequence signatures.

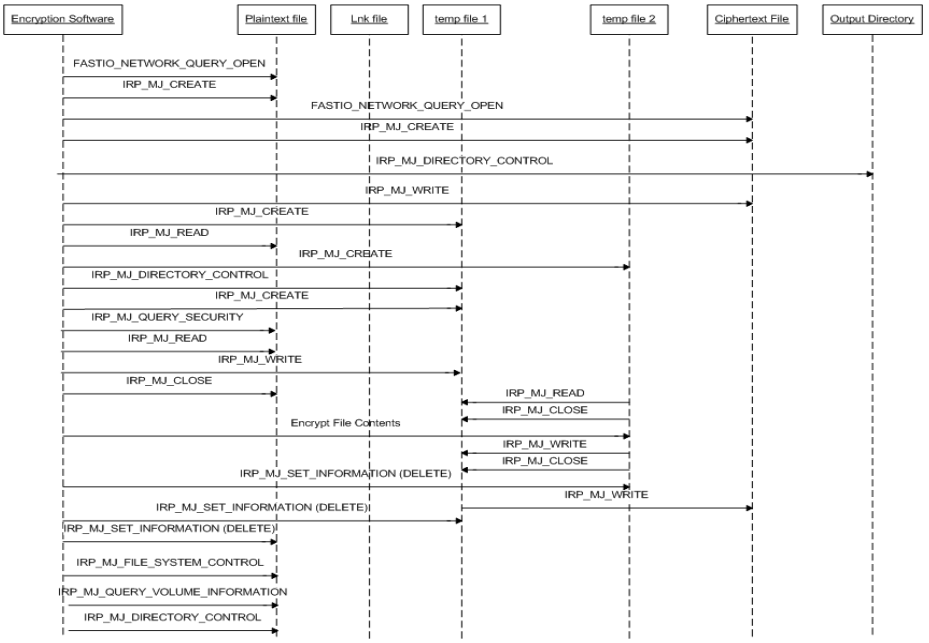

Fig. 8. UML Sequence Model (plaintext file not deleted)

4.2 Establishing an Event Sequence Signature of the Encryption Process

Three encryption software packages were studied here. It was found that in all three cases that the events which led to the creation of an encrypted file were consistent with each other. However it was noted that the frequency of certain events and their sequence varied slightly. Please see table 1 below where the individual event-types of the three encryption software packages are summarised. It was noted that when the plaintext file was deleted during encryption that there was no *lnk* file created. It was also noted that when plaintext file was not deleted that *Privatefile* was inconsistent in creating a *lnk* file. Under closer scrutiny this was understood to be conflicting with the anti-virus (AV) scan that was occurring at the time of experiment. Also noted was that *Meo* software didn't create any *lnk* nor temporary files and this was irrespective of any AV scans.

Table 1. Event-types

	Plaintext file deleted			Plaintext file not deleted		
Event -type	Privatefil	PGP	ME	Privatefil	PGP	ME
Create file for plaintext						
Create file for ciphertext						
Create lnk file	X	X	X			X
Create temp file(s)	(2)	(1)	X			X
Read plaintext file contents						
Write to temp file (s)	(2)	(1)	X			X
Write to ciphertext file						

5 Modeling Event Sequence Signature of the Encryption Process

Having observed the consistent occurrence of specific event types in table 1 during the file-encryption process, with different frequencies and event-sequences - this led to the question to see if the encryption process could be generalised using some formal approach. To this end a suitable formalism was introduced to define the encryption process as an event sequence signature.

5.1 Intrusion Detection Systems – Event Sequence Signature

In the design of Intrusion Detection Systems (IDS) there is a technique used to detect an intrusion which is called *anomaly detection*. The detection assumes attacks to be well-known sequences of actions. These actions are represented in the form of special patterns, called attack signatures. Attack signatures can be either mono-event or multi-event, depending on the number of steps in the corresponding attack scenarios. Defining multi-step attack signatures in a declarative form has been presented and it shows how temporal properties of multi-event attack signatures can be modelled in [10]. The presented model is based on high-level declarative Interval Temporal Logic (ITL). Temporal logic extends propositional logic with a notion of time by introducing special temporal operators e.g. always (D), sometime (◊), at the next

moment (D). Adding them allows true statements to be defined. The model in [10] is a slightly modified subset of ITL and it is called it *SigITL* (Signature ITL). Rules can be defined directly where a temporal formula that states a partially-ordered set of events in a multi-event signature. Each event is regarded as an atomic proposition of *SigITL* which are assumed to be mutually exclusive. Then temporal properties like sequence, any order, partial-order, exclusive choice, non-occurrence and repetition are defined.

5.2 Modeling Event Sequence Signature for the Encryption Process

By applying the SigITL specified in [10], the events in table 1 are grouped and modelled according to the artefact or "touch-point" that is recorded in $logfile. In order to formally define the modelling rules, let the following be mutually exclusive atomic propositions of SigITL: A = Read Plaintext file process, B = Create and Write to *tmp* file (repetition), C = Create *lnk* file process, D = Create and Write Ciphertext file, E = Delete Plaintext file (if selected by user; non-occurrence), Z_1 = User selection: decision to delete plaintext file and Z_2 = User selection: decision not to delete plaintext file. The temporal events like sequence, non-order, non-occurrence and mutually exclusive are defined below. If the events must occur in a fixed sequential order, then they are expressed as follows:

$$\Diamond A; \Diamond B; \Diamond C; \Diamond D; \Diamond E \qquad \text{or} \qquad \Diamond (A;B;C;D;E)$$

Equation 1. Expressing an ordered sequence of events

When the events occur in no fixed order then they can be expressed using the "∧" operator. So the events in table 1 can be summarised as :

$$\Diamond A;(\Diamond B \wedge \Diamond C \wedge \Diamond D \wedge \Diamond E)$$

Equation 2. Sequence of events with deletion of plaintext file

However it was observed that proposition B must follow A and E must occur last. Propositions C and D occurred in no fixed order other than after A and B but before E. The non-occurrence of an event between two others can also be expressed using the "D¬" operator. The occurrence of at least n repetitions of a particular event type can also be expressed. In this case the proposition B is expressed as B^n where n= 0, 1, 2 since it was observed that B can occur zero times, once or twice, now the following sequence signature model is arrived at:

$$\Diamond A; (\Diamond B^n \wedge D\neg E) \wedge (\Diamond CD\neg E) \wedge (\Diamond D \wedge D\neg E); \Diamond E$$

Equation 3. Events with non-occurrence and with repetition

Alternatively when there is no deletion of the plaintext file the events can be modelled as:

$$\Diamond A; \Diamond B^n \wedge \Diamond C \wedge \Diamond D$$

Equation 4. Events with no deletion of plaintext file

But the exclusive choice between two or more alternative events is represented by the operator $\oplus$. Since there is one decision to be made between Z_1 or Z_2 then there is an exclusive choice between Equation 3 and Equation 4. This is modelled: $\Diamond Z_1$; $(\Diamond A ; (\Diamond B^n \wedge D \neg E) \wedge (\Diamond CD \neg E) \wedge (\Diamond D \wedge D \neg E); \Diamond E) \oplus (\Diamond Z_2; (\Diamond A; \Diamond B^n \wedge \Diamond C \wedge \Diamond D))$. Equivalently, this true statement is now expressed as:

$$\Diamond((Z_1; (A ; (B^n \wedge D \neg E) \wedge (CD \neg E) \wedge (D \wedge D \neg E); E) \oplus (Z_2; (A; B^n \wedge C \wedge D)))$$

Equation 5. Model of event sequence signature

Equation 5 represents the generalised event sequence signature that occurs during file-encryption. This leads to the ability of recognising the occurrence of file-encryption and the subsequent analysis and investigation of encryption by using $logfile.

5.3 Constraint Satisfaction (CS) and Backtracking

Now that the event sequence signature can be modelled and generalised for the file encryption process it will provide a basis to automate the methodology. The main components of the methodology will consist of identifying the atomic propositions (A, B, C, D, E, F, Z_1 & Z_2) listed above.

To classify the type of model that Equation 5 represents is not that complex as it clearly represents a constraint satisfaction problem (CSP). In general CS is the process of finding a solution to a set of constraints that impose conditions that the variables must satisfy [11]. The general CSP consists in finding a list of values x = (x[1],x[2], …, x[n]), that satisfies some arbitrary constraint i.e. a boolean function. In this research x = (A, B, C, D, E, Z_1, Z_2). Backtracking is an important tool for solving CSPs. Backtracking recursively builds candidates to the solutions [12], and abandons each candidate as soon as it determines that it cannot be completed to a solution. Backtracking forms the basis of the automated solution to the methodology, this is implemented in *FindTheFile*, see section 8.

6 Methodology

6.1 Identify the Encrypted File to Be Investigated

As described [9] when a file or a folder is created then a series of log records are written out to the $logfile. The hexadecimal series 0B/0B→08/00→0B/0B→07/07→1B/01 was observed to occur a number of times when the ciphertext file and other files are created during encryption. After an image of the HDD is taken and the $logfile is exported for analysis the file name of the encrypted file under investigation is determined.

6.2 Determine BirthVolumeID of Ciphertext File and VolumeID

The *BirthVolume ID* of the encrypted file is determined and then matched with the *VolumeID* of the volumeID of the volume used. It can be concluded that the encrypted file was created on the same volume of forensically acquired volume under investigation. So updates or modifications to the ciphertext file would be in the $logfile.

6.3 Determine $FILE_NAME of Ciphertext File

The final occurrence of the ciphertext file name is searched for in the $logfile as a unicode string. This provides a starting point from which to step backwards in the $logfile, backtracking will be used here. Using the hexadecimal series referred to in 6.1 the $FILE_NAME attributes are searched for in the $logfile. These names are the Win32 name and the DOS name, see [9] for more detail. By analysing the last occurrence of $FILE_NAME attribute in the $logfile, the timestamps can be extracted. There are the three MAC times and the MFT modification time displayed here. Please see next step in 6.4 below, from this it can be seen when the ciphertext file was created.

6.4 Examine the Timestamps

NTFS timestamps contain the last modified, last accessed created and the MFT modified times of a file. These form part of the NTFS $FILE_NAME attribute of a file. These hexadecimal values are decoded to give date and time in UTC. The creation date of the encrypted file is given to be at the time when there is IRP_FILE_CLOSE was executed on the ciphertext file. This time closely approximates the last access time of the plaintext file.

6.5 Determine Where the Add/Delete Index Entry

For the newly created files the 0x0e/0x0f log record is included in $logfile as this indicates when a file is added/deleted from the index entry. This value is used to determine if plaintext file is deleted or not. This index entry includes a $FILE_ NAME attribute.

6.6 Determine Other Files Created during Process

Using the hexadecimal patterns outlined in 6.1 and 6.5 the temporary files created during encryption are identified. During the *Privatefile* encryption a temporary file is used where the data is *readin* and buffered from the plaintext file. Then the data is written from this temporary file to a second temporary file, where it is encrypted. The encrypted material is written to the cipherext file from there. If the plaintext file is not deleted then a *.lnk* file is created, this links back to plaintext file.

6.7 Use the "FILE0" Entry in $logfile to Step Backwards

Each MFT entry starts with the ascii signature string *FILE0* (or 0x46494C4530). By backtracking using the "FILE0" string and by examining the log details of the newly created and updated files (temp files, lnk, ciphertext and plaintext) the touch-points or the interfaces are revealed. This indicates the chronology and the sequence that would take place i.e. when the plaintext file was last updated with an MFT update on access time and also the newly created files' MFT update entries.

6.8 Determine the Original Plaintext File Name

The unicode string value of the plaintext file name can easily be extracted from the $FILE_NAME attribute in the $logfile. By backtracking in $logfile and passing each file or touch-point will lead to determining the original plaintext file name. There are Win32 and DOS $Filename attributes. When deletion occurs the process follows a different series of log record entries i.e. 0F/0E→03/02→16/15→0B/0B→08/00→0B/0B→07/07→1B/01. For deletion the composite pattern 0F/0E→03/02→16/15 precedes the 0B/0B→08/00→0B/0B→07/07→1B/01 composite pattern. Indicating that deleting (0x0F) and adding (0x0E) index entry allocation, de-allocating (0x03) and the initializing (0x02) of file record segments. The first part of the composite is for "Redo" and the second part is for "Undo" operation.

6.9 Examine the contents of the Plaintext file

The contents of the $DATA attribute of the plaintext file can be located and the hexadecimal values extracted. The contents will remain even if the file in question was deleted during the encryption process. This is because the $bitmap attributes and the data is just marked as de-allocated in NTFS. Depending on the size of the plaintext file, then the data can be stored residently on $mft or non-residently. The hexadecimal pattern 07/00→07/00 indicates what the resident data is otherwise the addresses of the clusters or run of clusters where non-resident data is specified in [7].

6.10 Determine BirthVolumeID of Plaintext File

This is to validate that the plaintext file identified was encrypted on the same volume. This would be carried out after identifying and locating the original plaintext file.

7 Case Study

The overall objective of this case study is to validate the methodology to see if it can establish an evidential link between the encrypted file and the original plaintext file and also view the plaintext contents. The name of the ciphertext file under investigation is *Secret.txt.pfs*. If the file name is obfuscated, renamed or its attributes changed then these activities can be tracked and traced in the $logfile – so evidence

of this would be detected. This activity was observed to have the 0x05/0x06 composite pair in the $logfile.

7.1 Determine BirthVolumeID of Ciphertext File and VolumeID

Determine the BirthVolumeID by using *fsutil*. A *fsutil* query is executed against the file and the resulting BirthVolumeID is outputted. The identical volumeID is confirmed to occur in $logfile for the ciphertext file. The BirthVolumeID in Fig 9 and the volumeID in Fig 10 are the same so it can be concluded that the encryption took place on this volume.

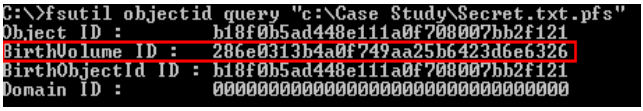

Fig. 9. BirthVolumeID of Ciphertext File extracted using fsutil

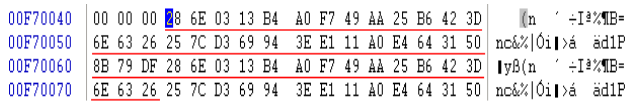

Fig. 10. VolumeID of Ciphertext file in $logfile

7.2 Determine $FILE_NAME of Ciphertext File

Find the last occurrence of the ciphertext file name in $logfile and backtracking search is initiated from this point, please see Fig 11. This is part of the $FILE_NAME attribute.

7.3 Examine the Timestamps

As can be seen from Fig 11 the four timestamps are create, last modified, mft modified and last accessed– in this order. The *0x 9EADACA345DCCC01* value represents the time *Thu, 26 January 2012 16:14:54 UTC,* this represents the last modified, mft modified and last accessed times. The last access time of plaintext file will closely approximate this time.

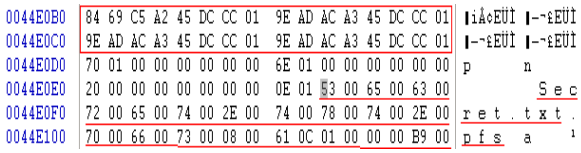

Fig. 11. Ciphertext file name and timestamps

7.4 Other Files Created during the Encryption Process

Using the same pattern 0B/0B→08/00→0B/0B→07/07→1B/01 (the series 0B/0B→07/07→1B/01 is used to close transactions) and by tracking in the $logfile it was seen that the other files are created i.e. .lnk and two temporary files. The *.lnk* file is created if the plaintext file is not deleted during encryption.

7.5 Plaintext File - (Irrespective If It is deleted)

The plaintext file name is determined to be *Secret.txt*. After the encryption process has accessed and opened the plaintext file for reading, the last access time recorded in the $logfile is 0x1E55289745DCCC01, when decoded is *Thu, 26 January 2012 16:14:33 UTC,* Fig 12.

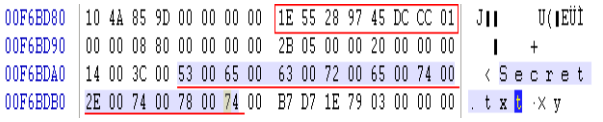

Fig. 12. Plaintext file-last access date

7.6 Plaintext Contents

Even if the plaintext file is deleted or not-deleted; the plaintext contents remains in the $logfile, as can be seen in Fig 13. This is subject to the clusters and sectors not being overwritten by other data that might be added later. Note: The plaintext contents is also available in the $MFT – this only occurs when the plaintext file is not deleted during the encryption process.

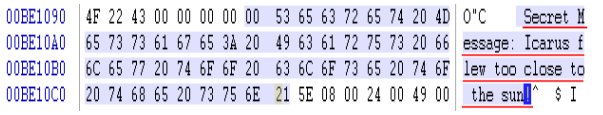

Fig. 13. Plaintext content in $logfile

7.7 Determine BirthVolumeID of Plaintext file

This is a validation step and *fsutil* query is executed against the identified plaintext file and the resulting BirthVolumeID is outputted. The outputted BirthVolumeID is identical to what is extracted from the $logfile, please see Fig 14 and Fig 15.

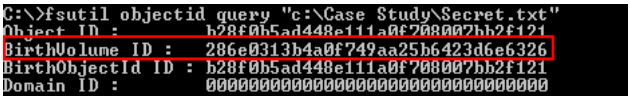

Fig. 14. BirthVolumeID of Plaintext File using fsutil

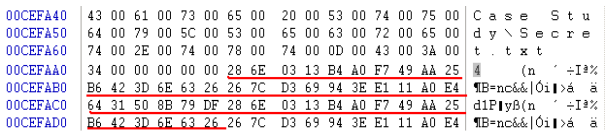

Fig. 15. BirthVolumeId from $logfile

7.8 Result of Investigation

The name of the plaintext file that was encrypted was revealed to be *Secret.txt.* The ciphertext file name is *Secret.txt.pfs* (as was known) and the secret message which was encrypted is "*Secret Message: Icarus flew too close to the sun!"*. The result of this case study is that it validates the methodology. It validates that the objectives of being able establish an evidential link between the encrypted file and the plaintext file while also revealing the plaintext contents of the encrypted file are met.

8 Automation of Methodology: *FindTheFile* Parser

Since the event sequence signature of the encryption process can be modeled and the occurrence of the event sequence signature can be classified as a constraint satisfaction problem- this provided the framework to the approach taken to automate the methodology and using the backtracking algorithm for searching event signatures. The use of backtracking is justified as it is the formally recognised solution to a CSP. As a result the parser was built using the JAVA high-level language. The JAVA language was selected as the language of implementation because of its platform cross-compatibility and its extensive library of APIs. The parser was called *FindTheFile* and the central class for parsing is the *StringTokeniser.* This class was instantiated for each activity that is carried out in the encryption process e.g. Create, Write, Delete file. These activities have corresponding hexadecimal entries in the $logfile that facilitates identification of these activities. These hexadecimal entries are used to initialise each *StringTokeniser* class with the appropriate tokeniser e.g. for newly created files the hexadecimal composite of *0x0e/0x0f* log record would be used as a tokeniser to indicate the action of adding a newly created file. The *Runtime* class from the java.io.* library is also used to call the external tool called *fsutil.*

8.1 Implementing the Backtracking and Recursion in Java

In order to apply backtracking to the data of a particular instance of a problem that is to be solved the following procedural parameters: root, reject, accept, first, next, and output are implemented. *FindTheFile* takes the instance data X as a parameter and would do the following: root(X): return the partial candidate at the root of the search tree, reject(X,c): return true only if the partial candidate *c* is not worth completing, accept(X,c): return true if c is a solution of X and false otherwise, first(X,c): generate the first extension of candidate c, next(X,s): generate the next alternative extension of a candidate, after the extension s and finally output(X,c): use the solution c of X, as

appropriate to the application. These steps are the pseudo code for implementing the backtracking solution (with recursion). This solution is the searching for the occurrence of an event sequence signature which is modeled by equation 5. The use of the *iterator* JAVA class was used at the core of this processing. This subsequently facilitated the automation of the methodology.

8.2 Recognising Temporary and Link Files

In order for the parser to be able to separate various created files (temporary and link) from the target file (plaintext file), the parser uses straightforward rationale. The rationale is that a temporary file- no matter what software package creates it, it will always be deleted. So the parser identifies a temporary file by the sequence of events that occur in the $logfile i.e. the file is created, processed (written to or read from) and then deleted. There are hexadecimal operator codes to indicate these actions in the $logfile. This eliminates any problems that might arise with *FindTheFile* not recognising file-naming conventions of a specific encryption software package might have. In the case of recognising the link file – the rationale is simply; a link file will always have a .lnk extension and this never changes no matter what encryption software is used.

8.3 FindTheFile

This parser was successfully implemented and was run against the case study. The $logfile was loaded and the ciphertext file under investigation was entered and then the parser was started, please see Fig 16. The output panel was generated with the results of the investigation. The results of the parser demonstrate that the search located all files that are associated with the ciphertext file under investigation in the $logfile, while backtracking (with recursion) was successfully implemented. It also indicates the original plaintext file name and shows with a check-box if it was deleted during the encryption or not. A text field is also outputted with the plaintext contents of the original file, regardless if the file was deleted or not. The timestamps (in UTC) display creation date of the encrypted file and the last accessed time of the plaintext file that was encrypted, see Fig 17.

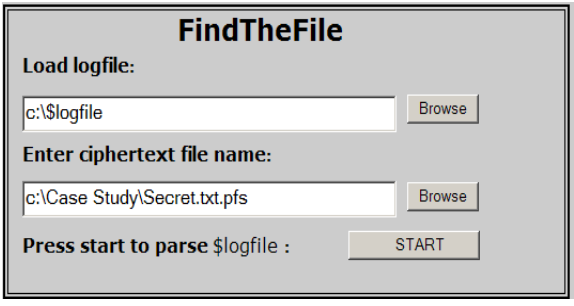

Fig. 16. FindTheFile: Initial Screen

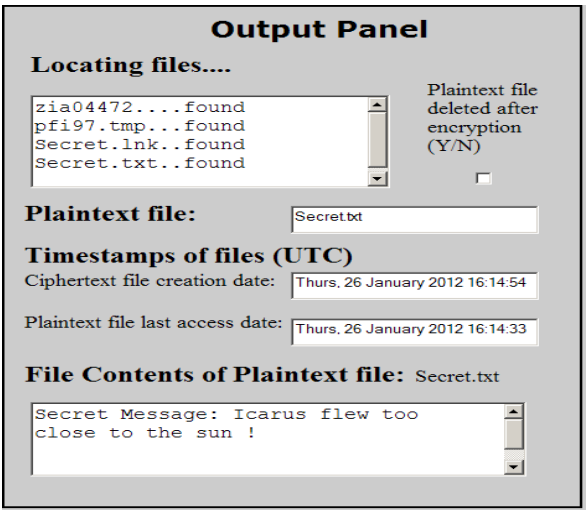

Fig. 17. FindTheFile: Output Panel

9 Performance Evaluation

9.1 Introduction

As stated previously the research objective was to develop and evaluate a methodology for the investigation of encrypted material using the $logfile where: a) there is an evidential link established between encrypted file and the plaintext file and, b) the plaintext contents of the original file is revealed. Receiver Operator Curve (ROC) analysis is carried out to evaluate how well the methodology performed.

9.2 ROC Analysis

ROC analysis incorporates binary classification which is the classifying of a given set of objects into two groups on the basis of whether they have some property or not. The medical community applies this to testing techniques and a typical scenario is the medical testing carried out to determine if a patient has a certain disease or not. ROC analysis is a useful technique for visualizing, organizing and selecting/evaluating classifiers based on their performance. When the area under the ROC curve (AUC) is computed it will indicate the measure of performance as a scalar of the chosen classifier. The classifier of interest is the finding the right plaintext file and its contents in $logfile. It is discussed [13] that when measuring the performance of medical and quality control tests, the concepts *sensitivity* (true positive rate - TPR), *specificity* (true negative rate - TNR) and *1-specifcity* (false positive rate - FPR) are used; these concepts are readily usable for the evaluation of any binary classifier. The number of true positives, false negatives, true negatives, and false positives always add up to 100% of the set. It is explained that in statistical hypothesis testing of an experiment, there will be a null hypothesis and an alternative hypothesis [13]. Based on the outcome of the experiment, it will be decided whether to reject the null

hypothesis or not. If the result of the experiment is statistically significant, then the null hypothesis is rejected in favour of the alternative hypothesis.

9.3 Experiment

The following experiment was carried out –where there are six encryption scenarios i.e. encryption with three different packages and for each package there are two options –deleting and not deleting the plaintext file. As a result there are six event sequence signatures to monitor and analyse, for each instance two text files were encrypted. Then this means there are twelve instances of Equation 5. These are numbered in column "No.", please see Table 2. The objective of the experiment is to determine a measure of performance of the methodology in terms of true-positives and false-positives. A classification variable (dichotomous) that would indicate results of instantiating the methodology was selected and this is called 'Successful' where Yes and No are the outcome. Then the binary representation of this is in column "Binary" where there are two classes 1= success and 0= failure.

9.4 Results and Observations

Using the "1"s listed in "Binary" column Table 2 as the list of true positives (TPs) which are also listed in column B of Table 3, then a list of false positives (FPs) can be created - take the TPs and replace "0" with "1" and vice-versa [13]. The FPs are listed in column C of Table 3. The TP rate is then calculated as being the proportion of files above this point that can be correctly investigated. This is calculated by summing the number of TPs above this point in the table and then dividing by the total number of TPs. These values are listed in column D of Table 3. Similar calculations are carried out for the FP rate in column E. The true negative (TN) rate is simply calculated by subtracting the FPR from 1 because FPR=1-Specificity. These calculations constitute the ROC data and are listed in Table 3.

Table 2. Evaluation for six event sequence signatures

No. (Instance of Eqn.5)	**Delete plaintext File** Yes = Y No = N	**Encryption Package**	**Successful** ✔ =Yes X=No	**Binary** ✔ =1 X= 0
1.1	Y	Privatefle	✔	1
1.2	N	Privatefle	✔	1
2.1	Y	Privatefle	✔	1
2.2	N	Privatefle	✔	1
3.1	Y	PGP	✔	1
3.2	N	PGP	✔	1
4.1	Y	PGP	✔	1
4.2	N	PGP	✔	1
5.1	Y	MEO	✔	1
5.2	N	MEO	✔	1
6.1	Y	MEO	X	0
6.2	N	MEO	✔	1

Table 3. Data to plot ROC graph

A	B	C	D	E	F
No. (Instance of Eqn.5)	**TP**	**FP**	**TP rate** (Sensitivity)	**FP rate** (1- specificity)	**TN rate** (Specificity)
1.1	1	0	0	0	1
1.2	1	0	0.091	0	1
2.1	1	0	0.182	0	1
2.2	1	0	0.273	0	1
3.1	1	0	0.364	0	1
3.2	1	0	0.455	0	1
4.1	1	0	0.545	0	1
4.2	1	0	0.636	0	1
5.1	1	0	0.727	0	1
5.2	1	0	0.818	0	1
6.1	0	1	0.909	0	1
6.2	1	0	0.909	1	0

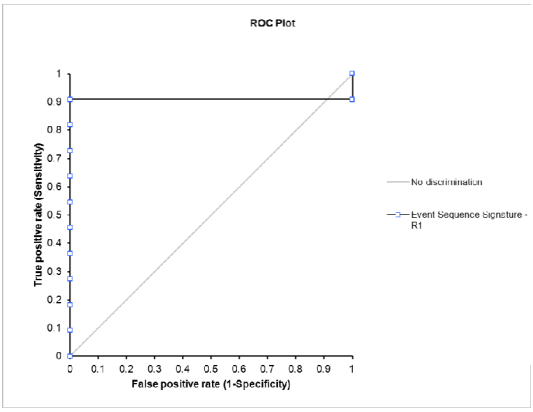

Fig. 18. ROC Plot

Using the data in the Table 3, the TPR as Y-axis and the FPR (recall that FPR = 1-*Specificity*) as X-axis were graphed to give the ROC plot in Fig 18. Then the AUC was computed by calculating the area for each row (where A is the TPR column and C is the FPR column) using the trapezoid rule [13]. Subsequently, a decision plot can be drawn up – not included here. The decision plot will allow the choice of the sequence event signature that minimizes the rate of false positives and aids in the selection of a specific value to use as a threshold that provides a desired trade-off between the true positive rate and the false positive rate.

It is pointed out that when the AUC is 1 the accuracy of the classifier is concluded to be excellent, when AUC is between 0.80 and 0.90 the accuracy of the test is

regarded to be good, while 0.70 to 0.80 indicates a fair accuracy level, 0.60 to 0.70 is regarded to be poor and anything else warrants test failure [13]. The AUC in this research has been calculated to be 0.91. Therefore it can be inferred that the methodology yielded an excellent result. The results verify the selection of the instance of Equation 5 variable (Success/Failure) as a classifier. In relation to Hypothesis Testing, if the Null hypothesis is H_0 (indicating randomness) and the alternative hypothesis is H_1 (indicating non-randomness), then where H_0: area ≤ 0.5. H_1: area > 0.5. So in relation to this research, the results of the experiment prove to be more powerful than a random rule as the AUC is 0.91. As a result H_0 is rejected and H_1 is accepted i.e. results from the methodology classifier are not random and are statistically significant. As a note - any points that lie under the line of no discrimination in Fig 18, would represent where no discrimination can be made between TPR or FPR this would be regarded as test failure.

10 Research Contribution

The research presents a unique and original method of investigating encrypted material and revealing the plaintext. This is achieved by characterising encryption as a series of file I/O operations rather than a mathematical or a theoretical problem. Then by following the various points along the I/O process flow evidence artefacts can be identified in the $logfile that lead to a successful investigation of encrypted material. A novel approach to the investigation of encrypted material is represented in the use of event sequence signature modelling which aided the classification of the presented problem as a constraint satisfaction problem. Then this provided the basis of implementing a successful backtracking search solution. The methodology was successfully automated by implementing a parser that parsed the $logfile for events and was able to output the results of the investigation. A side-channel attack is defined as any attack based on side-channel information e.g. implementation or physical details of a cryptosystem. The side–channel attack is not based on brute force or theoretical weaknesses of the cryptosystem that can be exposed through cryptanalysis techniques [14]. Therefore it can be inferred that this research effectively results in a side-channel attack on encryption.

11 Future Work

This methodology will be extended to investigate multiple file formats; image formats like JPEG in particular. This will result in an investigative technique to analyse steganographically generated image files (stego-files). Once this is in place emphasis will be placed on admissibility of all evidence produced by methodology-thus ensuring *Daubert* compliance. As an approach to achieve this it is necessary to

get the methodology tested in actual field conditions rather than just in laboratory conditions. Therefore, it is intended to recommend and promote the use of this methodology within a LEA environment. Use of the methodology in this way would identify benefits and drawbacks and they would form the basis of future work.

12 Conclusion

The main outcome of this work is a formal methodology. This methodology has been validated through the development of an automated system and also through its practical application on a case study. The performance of the methodology has been evaluated using a binary classification system of true-positives and negatives and has resulted in an excellent score. The objective of the methodology has been to investigate encrypted material while revealing the original plaintext file and its contents- this has been carried out successfully in a case study. The modelling of the initial problem and backtracking solution served as a framework to facilitate the generalisation of the process and subsequent automation – so the presented methodology was compatible and interoperable with all tested types of encryption. The methodology relies fundamentally on the evidential value that can be extracted from the $logfile. The main challenge in this research was that the data in the $logfile is transient as the contents gets rolled over on a cyclical basis – as fresh data gets written in to the log file, older data is flushed out. Unfortunately there is insufficient knowledge on the precise nature of the log-rotation cycle of $logfile in the public domain.

The following tools were used in this research: AccessData FTK Imager 2.5.1, WinHex 16.2 3, Process Monitor, Dcode, MS FSUTIL, Privatefile, Meo and PGP Desktop.

References

1. Carter, H.: Paedophiles jailed for hatching plot on internet (2007)
2. Joseh, S.: Hamas Terror Chat Rooms (December 11, 2007)
3. Siegfried, J., et al.: Examining the Encryption Threat, Computer Forensic Research and Development Center. International Journal of Digital Evidence (2004)
4. Bunting, S.: The Official EnCase Certified Examiner Guide. Wiley (2008)
5. McGrath, N., Gladyshev, P., Carthy, J.: Cryptopometry as a Methodology for Investigating Encrypted Material. International Journal of Digital Crime and Forensics 2(1) (January-March 2010); special edition of selected papers from e-Forensics (2009)
6. Russinovich, M.E., Solomon, D.A.: Windows Internals Covering Windows Server 2008 and Windows Vista. Microsoft Press, One Microsoft Way (2009)
7. Carrier, B.: File System Forensic Analysis. Addison Wesley, Boston (2005)
8. Parsonage, H.: The Meaning of Linkfiles in Forensic Examinations (2010)
9. Cho, G.-S., Rogers, M.K.: Finding Forensic Information on Creating a Folder in $LogFile of NTFS. In: Gladyshev, P., Rogers, M.K. (eds.) ICDF2C 2011. LNICST, vol. 88, pp. 211–225. Springer, Heidelberg (2012)

10. Nowicka, E., Zawada, M.: Modeling Temporal Properties of Multi-event Attack Signatures in Interval Temporal Logic. Wrocław University of Technology (2006)
11. Rossi, F., Van Beek, P., Walsh, T.: Constraint Satisfaction: An Emerging Paradigm. In: Handbook of Constraint Programming. Foundations of Artificial Intelligence. Elsevier, Amsterdam (2006)
12. Gurari, E.: Backtracking algorithms "CIS 680: DATA STRUCTURES: Chapter 19: Backtracking Algorithms" (1999), http://www.cse.ohio-state.edu/gurari/course/cis680/cis680Ch19.html#QQ1-51-128
13. Altman, D.G., Bland, J.M.: Diagnostic Tests – Sensitivity and Specificity. BMJ 308(6943), 1552 (1994) PMID 8019315
14. Chen, S., Wang, R., Wang, X., Zhang, K.: Side-Channel Leaks in Web Applications: A Reality Today, A Challenge Tomorrow. In: IEEE Symposium on Security & Privacy (May 2010), http://research.microsoft.com/pubs/119060/WebAppSideChannel-final.pdf