SQLite Forensic Analysis Based on WAL

Yao Liu[1], Ming Xu[1(✉)], Jian Xu[1], Ning Zheng[1], and Xiaodong Lin[2]

[1] Internet and Network Security Laboratory, School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China
{141050061,mxu,jian.xu,nzheng}@hdu.edu.cn
[2] Faculty of Business and Information Technology, University of Ontario Institute of Technology, Oshawa, Canada
xiaodong.lin@uoit.ca

Abstract. SQLite database is an important source of evidence in forensic investigations. Write-Ahead Logging (WAL) was introduced to ensure data integrity and improve performance in SQLite databases. However, few attentions have been paid to utilizing it for forensic purposes, particularly in deleted record recovery. Without using WAL, prior recovery methods have been ineffective. This paper addresses techniques for SQLite forensic analysis based on WAL. Specifically, based on the storage mechanisms of SQLite and the structure of the WAL, both the original SQLite database and WAL are first constructed by extracting and analyzing all valid pages. SQLite history versions are then produced by using two reconstructed files above. Deleted records can then be recovered and tampered behaviors can be detected by comparing different versions of the reconstructed history file. Experimental results show that the proposed method can reconstruct history versions, recover deleted records and detect tampered behaviors effectively.

Keywords: Digital forensics · SQLite · Reconstruction · History versions · Recovery · Tamper detect

1 Introduction

SQLite is an open source, embedded relational database. Originally released in 2000, it was designed to provide a convenient means for applications to manage data. In addition to it's widely used in embedded devices, various communication applications and mobile Apps use SQLite (e.g., SMS, Contacts, Call History, E-mail Client, and third-party apps). It is estimated that several millions devices currently use this standard [1].

It is not surprising that the data stored in the SQLite database has grown from simple contacts lists ten years ago to several gigabytes of potentially sensitive and personally identifiable information (PII) today. Thus, forensic analysis of SQLite database has become essential and critical to investigating authorities, where recovery of deleted records is the most common task. Specifically, it

R. Deng et al. (Eds.): SecureComm 2016, LNICST 198, pp. 557–574, 2017.
DOI: 10.1007/978-3-319-59608-2_31

is very common to extract deleted records from the unallocated space (e.g., free space and free block) of database files during an investigation.

Flash memory capacity has increased considerably. To improve the performance of the SQLite database, a new WAL (Write-ahead Log) option is used in place of a rollback journal when SQLite is operating in WAL mode. This was made available in version 3.7.0 and later [2–5]. While some operations (e.g., insert, delete, update, etc.) are performed, original content is retained in the database file. The changes append to a separate WAL. Before a commit condition (i.e., checkpoint [3–5]) occurs, all operations are not committed immediately and the changes are not made to the actual data in the database. Therefore, it is difficult to recover recently deleted records from the database file until a checkpoint is made.

Our current work gives a detailed illustration of the procedure to reconstruct SQLite history versions through the original database file and WAL. The proposed method features two main utilities through the analysis of reconstructed history versions. The first is to restore deleted records, and the second is to detect tampered behaviors. Both are achieved by comparing different versions of the history file.

The remainder of this paper is outlined as follows: Sect. 2 introduces the background information concerning SQLite database and WAL. The methods to reconstruct SQLite history versions are given in Sect. 3. Section 4 evaluates the experiment and demonstrates the utility of our proposed method using two case studies. Sections 5 and 6 discusses some practical issues and related work. The paper ends with the discussion of the future work and conclusions in Sect. 7.

2 Background

SQLite is a SQL database engine widely used in embedded devices. It is especially popular in the applications of mobile devices where it is the de facto database to manage user data. This database utilizes either a rollback journal or a WAL to ensure the atomic operations. The behavior of the synchronous WAL and database is called checkpoint, which is automatically executed by SQLite. This section describes SQLite database features associated WAL and checkpoint mechanisms.

2.1 SQLite Database

Structure of SQLite. Typically, each SQLite database consists of fixed-size pages. These can be either table B-tree page, index B-tree page, free page or overflow page. The page size is defined in the first 100 bytes of the database file. All pages are of the same size and are comprised of multi-byte fields. The most significant fields are magic header string(aka file signature), text encoding, parameters of Btree structures, and incremental-vacuum settings. These important fields are stored in big-endian format as shown in Fig. 1.

Magic header string is used to effectively identify SQLite database files. Every valid SQLite database file begins with the following 16 bytes (in hex): 53 51 4c 69 74 65 20 66 6f 72 6d 61 74 20 33 00. This byte sequence corresponds to the UTF-8 string "SQLite format 3\0" including the null terminator character at the end.

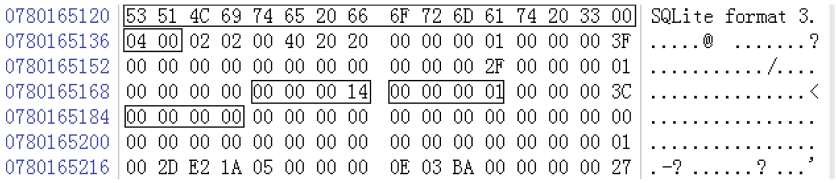


Fig. 1. Main fields of SQLite database file header

For database in auto-vacuum mode or incremental-vacuum mode, there is another significant page called pointer bitmap page. Pointer bitmap pages are extra pages inserted into the database to make operations of auto-vacuum and incremental-vacuum more efficient. The decision to use pointer bitmap page depends on a non-zero largest root B-tree page value. This is located at byte offset 52 in the database header.

SQLite Master Table. The SQLite master table (or SQLite temp master table in the case of a TEMP database) is a system table that contains information about all tables, views, indexes, and triggers in the database. It stores the complete database schema [6]. The structure of the SQLite master table can be created using the following SQL statement: CREATE TABLE sqlite master (type TEXT, name TEXT, tbl name TEXT, rootpage INTEGER, sql TEXT), the value of each filed is described in Table 1.

Table 1. SQLite master fields

No	Field	Value
1	type	Table, index, trigger or view
2	name	Table name for a table
3	tbl_name	Same to second filed (for table or view), or related to table name (for index or trigger)
4	rootpage	0 (for trigger or view), or rootpage number (for table or index)
5	SQL	Creating sentence for specific type of objects

B-tree Structure. SQLite databases are stored in segments, called pages. A SQLite database is composed of multiple B-trees, with each B-tree occupying a

minimum of one page. One B-tree is needed for each table and index. These are referred to as table B-trees and index B-trees. Each table or index in a SQLite database has a rootpage which defines the location of its first page. The root pages for all indexes and tables are kept in the SQLite master table [6].

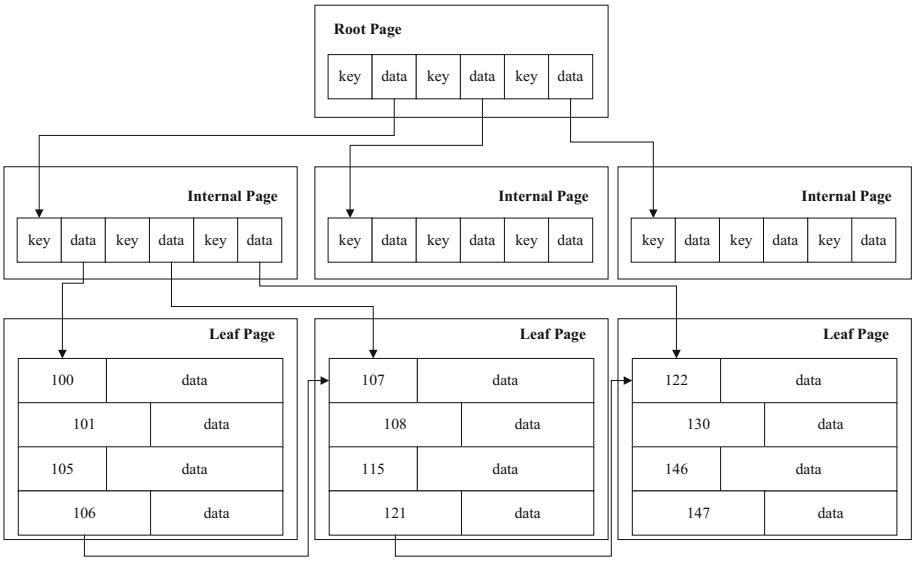


Fig. 2. Table B-tree structure of SQLite database (Owens, 2010)

A B-tree page is either an internal page or a leaf page. In the case of a table B-tree as shown in Fig. 2 (Owens, 2010), a leaf page contains keys and each key has associated data. An internal page contains K keys together with K pointers to child B-tree pages. A pointer in an internal B-tree page is just the 31-bit integer page number of the child page. Whereas for index B-tree, its structure is almost identical with table B-tree. The most obvious difference is that the internal pages contain not only keys but also index records. Furthermore, an internal page contains K keys together with K+1 pointers to child B-tree pages.

In an internal B-tree page, pointers and keys logically alternate. All keys within the same page are unique and are logically organized in ascending order from left to right. The ordering is logical, not physical. The actual location of keys within the page is arbitrary. For any key X, pointers to the left of X refer to B-tree pages on which all keys are less than or equal to X. Pointers to the right of X refer to pages where all keys are greater than X. Taking Fig. 2 as an example, the first data in the first internal page have referred to a leaf page, all keys in this leaf page are less than or equal to 106.

Within an internal B-tree page, each key and pointer to its immediate left are grouped into a structure called a cell [4]. The cell is the basic element to organize data within a B-tree page. The right-most pointer is held separately.

A leaf B-tree page has no pointers. It however, uses the cell structure to hold keys for index B-trees or both keys and contents for table B-trees. Data is also contained in the cell.

Cells are placed from bottom to top of the page. Since sizes are unfixed, the offset of its first byte is also recorded. These offset values are placed from top to bottom of a page. This bi-growing design makes it easy to add new record without defragmentation.

2.2 Write-Ahead Log

Beginning in version 3.7.0, SQLite introduced the option of using WAL mode to implement atomic commit and rollback. In this mode, the original content is preserved in the database and changes are appended into a separate WAL. A commit occurs when a special record indicating a commit is appended to the WAL. Thus a commit can occur without writing to the original database. This allows readers to work with unaltered original databases while changes are simultaneously being committed into the WAL. Multiple transactions can be appended to the end of a single WAL.

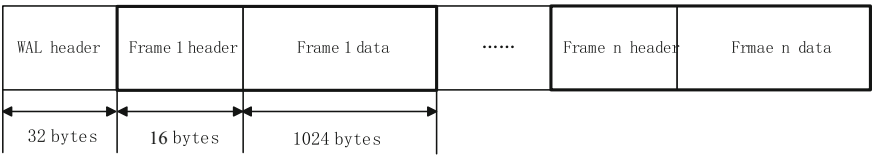


Fig. 3. WAL file structure

The logical structure of the WAL is provided in Fig. 3. It consists of a file header and several frames. The number of frames can be ranged from 0 to a lot, indicated in bold black box. Each frame is divided into a frame header and frame data. When checkpoint (is) triggered, valid data stored in WAL would be transformed into the database. WAL can be reused with new frames overwriting prior frames after checkpoints. WAL always grows from the beginning to the end of the sequence. The checksum and counter appended to each frame indicate that whether the frame is effective or not.

WAL has a 32-byte length header which includes eight 32-bit big-endian unsigned integers as shown in Table 2. Among them, the database page size corresponds with that of the database. Salt-1 is a random number that increases by 1 after a checkpoint. Salt-2 is a random number which is replaced by another random number after a checkpoint.

Frames trail after WAL file headers. Each frame consists of 24-byte frame header and frame data. Frame headers consist of six 32-bit big-endian unsigned integers as shown in Table 3. Page Number indicates which page of the database is recorded here. Salt-1 & 2 may match or differ to the corresponding value in the file header, and determine whether data in this frame is effective or not.

Table 2. WAL file header structure

Offset	Size	Description
0	4	Magic number: 0x377f0682 or 0x377f0683
4	4	Format version: Currently 3110100
8	4	Page size: Default size is 1024 bytes
12	4	Checkpoint number
16	4	Salt-1: Random integer incremented with each checkpoint
20	4	Salt-2: A different random number for each checkpoint
24	4	Checksum-1: First part of a checksum on the first 24 bytes of header
28	4	Checksum-2: Second part of the checksum on the first 24 bytes of header

Table 3. WAL frame header structure

Offset	Size	Description
0	4	Page number: Logical page number from database file
4	4	File size: For commit records, the size of the database files in pages after the commit. For all other records, zero
8	4	Salt-1: Copied from the WAL header
12	4	Salt-2: Copied from the WAL header
16	4	Checksum-1: Cumulative checksum up through and including this page
20	4	Checksum-2: Second half of the cumulative checksum

2.3 Checkpoint Mechanism

Similar to several mainstream databases (e.g., SQL Server, MySQL, Oracle, etc.), SQLite supports atomic transaction commit when reading and writing records. Transactions define boundaries around a group of SQL commands such that they either all successfully execute together or not at all.

By default, SQLite does a checkpoint automatically when the WAL reaches a threshold size of one thousand pages or more in size, or when the last database connection on a database file close. That is to say, the occurrence of a checkpoint means to transfer all the transactions that are appended in the WAL back into the original database.

With the use of checkpoints in WAL, all committed atomic transactions are appended into the WAL and will not be transformed into the database immediately. The transactions submitted at different times constitute different operation states. Taking each committed atomic transaction as granularity, several histroy versions can be achieved from WAL.

3 Methodology

In this section, a method is proposed to reconstruct SQLite history versions from Android devices. Figure 4 shows the framework for our reconstruction approach. The method consists of four main parts: Reconstructing SQLite database, extracting correspond WAL, combining original SQLite database with corresponding WAL to recreate SQLite history versions and forensic analysis for deleted record recovery and tamper detection.

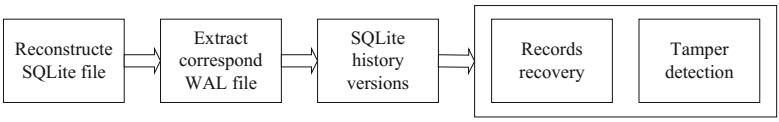


Fig. 4. Framework of reconstruct approach

3.1 Reconstructing SQLite Database

Pre-processing. Pre-Processing is intended to extract the real master table from phone image. Based on the storage mechanism of SQLite, only the tables and indexes have actual storage space. Thus, these information are the main consideration when analyzing SQLite master table. In addition to SQL fields, other fields are also the focus of our consideration. Prior to this, we define a candidate set for the SQLite master table. Considering various Android devices are designed according to the Goggle's official source code. There is a slight discrepancy between different mobile phones. We obtained the master table structure from Android source code as the initial candidate set [7]. For those customized mobile phones, we need to adjust candidate sets on an individual basis. Using short message databases as an example, Fig. 5 shows a typical master table structures.

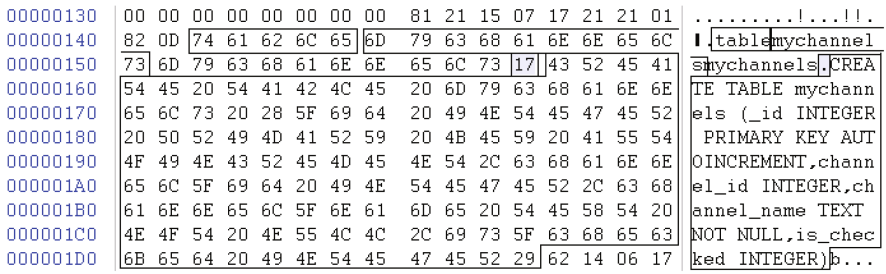


Fig. 5. Example of master table structure

Algorithm 1. Extracting and Analyzing Master Pages Algorithm

Input: image file, candidate master table records
Output: result1 = {offset, sql, field type order}

```
/*default page size is 1024bytes*/
for block ∈ image file do
    page[ ] ← block
end for
if page[0] = 0x0D or page[0 − 15] = magic number then
    /*recognize B-tree pages*/
    if data block ∈ table name from candidate master table then
        while i < total records in each page do
            extract all fields within pages and store in database
            i++
        end while
    end if
end if
offset ← offset + page size
for extracted page ∈ image file do
    extract sql and analysis field type order f1
end for
return result1
```

Several steps are necessary to retrieve a true master table. First, a phone's image and candidate master table are taken as an input. Second, the image is then traversed to recognize all pages belonging to a true master table. Third, pages are analyzed to extract all SQL statements and field types of tables and indexes. The simplified algorithm is presented in Algorithm 1.

Among them, set $result1$ contains three fields, which are used to describe the internal information of a master page. Each field is explained as follows:

$$(offset, sql, field\ type\ order)$$

where:

- *offset* is the address of a specific page belonging to a master table;
- *sql* is the creating sentence for specific types of objects;
- *field type order* is a string which contains all filed types within a specific page;

Extracting and Analyzing Pages. For a given phone image and a real master table, we can get all the B-tree pages that belong to the specified database. This stage of the algorithm is shown in Algorithm 2.

In the extracting phase, the purpose is to acquire pages that match the given conditions. These conditions are used to determine whether a page belongs to a specified table. As described in Algorithm 2, two main match conditions are used: The first is to confirm the basic structure of the database pages (e.g., if we locate a data page when traversing the image, it must begin with 0x0D. Additionally, cell sums should not exceed the page size defined in first 100 bytes of the database

Algorithm 2. Extracting All Pages Algorithm

Input: result1, image file
Output: result2 = {offset, table name, row head, row tail}

```
/*default page size is 1024 bytes*/
if page[0] = 0x0D or page[0] = 0x05 or page[0] = 0x0A or page[0] = 0x02 then
    /* recognize four type of pages*/
    if the fix offset value conform to page structure then
        for each page locate one record and acquire field type order f2
    end if
end if
if f1 page ∈ f2 then
    offset, table name, row head, row tail
end if
offset ← offset + page size
return result2
```

file). The second match condition is to compare field types extracted above with pages use d for analysis from the image.

Among them, set *result*2 contains four fields, which are used to describe the internal information of a page. Each field is explained as follows:

$$(offset, table\ name, row\ head, row\ tail)$$

where:

- *offset* is the address of a specific page within the image file;
- *table name* is the page belonging to;
- *row head* is the minimum value of the row id within a specific page;
- *row tail* is the maximum value of the row id within a specific page;

Due to the fact that the field types only have three values in SQLite database tables, this causes more pages to be located than previously expected which hinders analysis. This is addressed in the processing phase by using B-trees. B-tree assists in maintaining unique keys which are sorted in order for sequential traversing. Besides, taking into account the allocation mechanism of Ext file system and the use of WAL, we can make further filter out of extracted pages. The processing algorithm is presented in pseudo-code shown in Algorithm 3.

Among them, the set *result*3 contains three fields, which are used to describe additional information on the page. Each field is described as follows:

$$(validate\ list, root\ page\ list, offset\ list)$$

where:

- *offset* is the address of pages extracted in Algorithm 2;
- *validate list* is the list to mark valid pages;
- *root page list* is the list to mark root page;

Algorithm 3. Processing Extracted Pages Algorithm

Input: result2, image file
Output: result3 = {offset, validate list, root page list}

for *table name* ∈ *result*2 **do**
  get row head list, row tail list and offset list by table name
  for *row head value* ∈ *row head list* **do**
    if *value* = 1 **then**
      $result3 \leftarrow offset, validate = 1$
      $id \leftarrow the\ first\ id\ when\ value = 1$
    end if
  end for
end for
for *row head value* ∈ *row head list* **do**
  $i = 0$
  if $i < id$ **then**
    $result3 \leftarrow offset, validate = 0$
  end if
  if $i > id$ **then**
    if *value* = 1 **then**
      break
    else
      compare two adjacent row head values
    end if
    if $row\ head[i] = row\ head[i-1]$ **then**
      $result3 \leftarrow offset, validate = 1, rootpagelist = 1$
    else
      $result3 \leftarrow offset, validate = 0, rootpagelist = 0$
    end if
  end if
end for
return *result*3

Reorganizing Pages. After completing the previous two steps, we now have a series of valid table pages with the range of records and index pages with leaf order and key order. With this information, the size of a database file and the logical number of each page can be easily determined.

When analyzing extracted pages, we found that some tables within the database were empty. This was previously unconsidered during extraction and analysis. We filled empty table pages to reorganize the file. The main reorganization steps are described commonly as follows:

- Comparing rootpage number from master table pages and leaf order from interior pages (include tables and indexes) to obtain the most right leaf number. This is also the total page number of the database file.
- Creating an empty file which contains the total number of pages. Each known page is duplicated in at a corresponding location in the new file. Remaining

pages will be filled with empty data pages or index pages according to the master table.
– Inserting pointer map pages if they exist.

3.2 Extracting WAL File

Recovering WAL is based on the data blocks reorganize method. A data block represents a data frame as previously described. A frame is considered valid if and only if the following conditions are true [4]: First, the salt-1 and salt-2 values in the frame-header match salt values in the wal-header. Second, the checksum values in the final 8 bytes of the frame-header exactly match the checksum computed consecutively on the first 24 bytes of the WAL header and the first 8 bytes and the content of all frames up to and including the current frame. The main recovery steps are described as follows:

– Finding the WAL file header by magic bytes, saving all the 4-byte random value in a given set.
– Traversing the set, searching for a random value which belongs to the current WAL file using the field type characteristics of a specific table. For example, an SMS table. Pointer map pages are inserted if they exist.
– Finding the each frame of the WAL file. Locating and saving the offsets of the 4-byte random value in frames which matches the value previously mentioned.
– Reorganizing the frames by the constant size of data frame, which belongs to the current WAL file.

3.3 Reconstructing SQLite History Versions

Previous studies on the reconstruction of the SQLite history versions mainly discussed in [22,23]. They proposed a recovery method using YAFFS2 metadata. However, this method cannot be applied to newer devices with extensive use of ext4 file system.

Our method for reconstructing the SQLite history versions is closely related to the methods proposed in Sects. 3.1 and 3.2. As mentioned above, we have reconstructed the SQLite database and its corresponding WAL. Each frame header shows the page number of its related database file and whether the transaction was committed. The details can be seen in Sect. 3.

The occurrence of a transaction operation may affect multiple data pages. Each operation is not immediately written back to the database. Thus, we aim to locate which frame blocks contain commit markers. By individually considering each marker we can obtain a number of history versions of SQLite database.

4 Experiment and Evaluation

The following experimentation tests the method proposed in Sect. 3. All experiments were conducted on Android smartphones with WAL. SMS is one of the

most common applications that have adopted SQLite for storage and management of data. Furthermore, it also has a substantial amount of important user information which is of interest for forensic investigations [8]. In this paper, we attempt to recover deleted SMS messages as a case study for the proposed method.

The first stage of the experiment involved conducting a series of predefined activities on the device. This stage is divided into two experimental scenarios, in the first scenario, some activities are conducted as follows: restore the phone to the factory settings, add 150 records, delete 50 records and update 10 records. Similar to the first scenario, the second is just different in that the number of records used in add and delete operations are 250 and 100, respectively.

4.1 Collection of Data Image

There are two main methods to obtain data image: physical method and logical method [9,10]. In this paper, we focus on user data partition for acquiring data.

Rooting mobile devices for the purpose of data acquisition is often discussed in literature. While we believe it should generally be avoided [11], we have decided to root this device for expediency of data collection. We utilized the dd tool to acquire storage images for analysis:

- Put the mobile device in developer mode and connect it with the computer.
- Establish TCP communication between the device and computer using the $adbforward$ command.
- View the partition information using the $adbcatpartitions$ command.
- Copy and transfer the image from the device to computer using the dd and nc commands [12].

In order to support dd and nc commands, the BusyBox [13] tool is required. It can be installed on the system partition to protect the integrity of the data partition. As a result, the image of data partition can be obtained bit-by-bit that is important for the next step in our investigation.

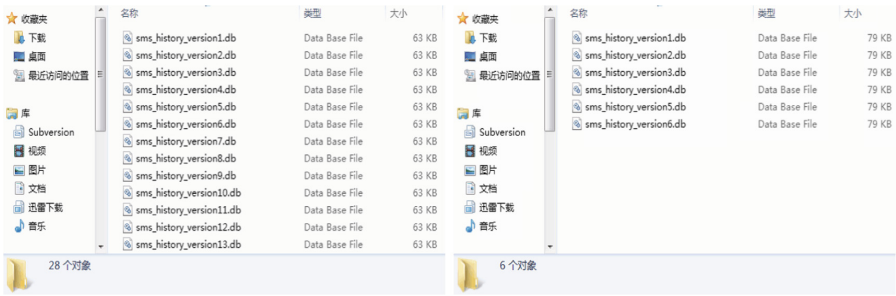
4.2 Reconstruction of SQLite History Versions

The aim of the experiments is to test the proposed method for reconstructing the history versions of SQLite database. Prior to this point, we have to reconstruct SQLite database and corresponded WAL, and evaluate the effectiveness of the two reconstructed files.

A common method in the literature to compare whether two files are identical is through their hash values [14]. The two original files are acquired from the image file using dd tool and their hash values are calculated separately (Only calculate the effective part for WAL). Then, these two files are reconstructed with the method described in Sect. 3. The hash values of these files are subsequently calculated by comparing the original files with reconstructed files, we

Table 4. The comparison results of SQLite and WAL

Case Num	Original valid file	Reconstructed file	Compared result
1	554f7e0b621c4b9c2c1d8df7d3faf5d4(SQLite)		√
	0cfdfcf0252254760a01620154b9036e(WAL)		√
2	c03ca026d59987f97a79af90747b7c47(SQLite)		√
	a076bb47b198b2b932c966c8f2bf51cf(WAL)		√



(a) Case I (b) Case II

Fig. 6. The result of reconstruct history files in two cases

can determine whether the experimental result was effective or not. The comparison results of the two files are displayed in Table 4. It can be observed that two files are reconstructed correctly.

In our research, we found the factors that affected the result of the experiments is checkpoint mechanism. We will discuss two situations according to whether the checkpoint is triggered.

The experimental results of the reconstructed history versions are displayed in Fig. 6. For the sake of simplicity, we named file based on the incremental version number. Situation I indicate that the checkpoint is not triggered, in contrast, situation II means the checkpoint is triggered. As a result, 28 history versions and 6 history versions are found respectively from the situation I and situation II. In situation II, a part of the records in WAL were transformed into database and are marked as invalid pages, and we reconstructed those as valid history versions.

4.3 Case Studies Using Reconstructed Files

As some common database operations (e.g., insert, delete, update, etc.) occurred in WAL and are transferred back into the database file when checkpoint conditions is reached. History version files contain valuable evidences, especially on operation of deleted and updated records. Through the comparison of different history version files, we can discern each operation about a database. Two case studies are discussed below.

Evaluation Criterion. The precision rate (define as Eq. (1)) and recall rate (define as Eq. (2)) are adopted as the criterions to evaluate the proposed method, and the F-value (define as Eq. (3)) is used to evaluate the quality of the recovery approach. In the below equations, the A refers t o the number of recovered records belonging to the SMS database; the B refers to the number of records that do not belong to the SMS database; and the C refers to the number of records which belong to the SMS database but were not recovered from the image file.

$$Precision(P) = \frac{A}{A+B}. \tag{1}$$

$$Recall(R) = \frac{A}{A+C}. \tag{2}$$

$$F - value = \frac{2 * P * R}{P+R}. \tag{3}$$

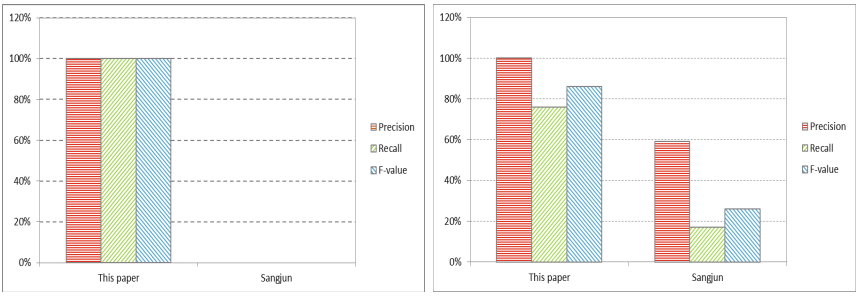
Case I: Recovering Deleted Records. Each history version file records one or more transaction operations. We can recover deleted records easily by comparing two different version files. As shown in Table 5, we recovered all deleted records in situation I and approximately 75% in situation II. The value of precision, recall, and F-value indicates that the proposed method can recover recently deleted records effectively

Table 5. The result of delete operation

No	Image size (KB)	Total records	Deleted records	Recover records	Precision	Recall	F-value
1	3251200	150	50	50	1	1	1
2	3251200	250	100	76	1	0.76	0.86

To confirm the effectiveness of our approach, we compare it to results of similarly proposed schemes in [18]. Since the compared recovery method restores records from the SQLite database file, the file of SMS database was first retrieved from two images. The result is described in Fig. 7. In situation I, there are no records being restored by Sangjun's method. Moreover, 17 records were restored from situation II. The P is 1, the R is 0.068, and the F-value is 0.127. It is clear that our proposed method has an improvement over the precision rate, recall rate and F-value.

Case II: Detecting Tampered Behaviors. Consider a scenario where a suspect commits a crime. Incriminating evidence was transmitted somehow, perhaps via SMS. To avoid incarceration the criminal attempts to destroy all electronic evidence. Fortunately, we can recover deleted records to retrieve evidence if the



(a) Situations I (b) Situations II

Fig. 7. The result of recover deleted records in two situations

suspect simply delete SMS messages using built-in SMS deletion functionality. Knowing this, the suspect may simply tamper with the content of evidence and do not delete records. In this case, traditional recovery method will not work effectively. However, we can detect these tamper behaviors by comparing two different version files.

The compared results of tamper operation are shown in Table 6. As we can see, these results are similar to those when recovering deleted records in case I. For all 10 tampered records, we can detect 10 records in situation I and 8 records in situation II. The value of precision, recall and F-value indicate that the proposed method can detect tampered behaviors effectively.

Table 6. The result of tamper operation

No	Image size (KB)	Total records	Tamper records	Detect records	Precision	Recall	F-value
1	3251200	150	10	10	1	1	1
2	3251200	250	10	8	1	0.8	0.89

5 Discussion

In this section, we discuss some practice issues and considerations related to our method to reconstruct original SQLite database and WAL.

For expediency in the collection of data for this research, we decided to gain root privileges and utilize the *dd* tool to acquire storage images for analysis. In practical cases, however, the physical acquisition method is recommended to ensure data integrity.

If the metadata of the file system allows normal to access, we may be able to retrieve the directory structure of the file system and extract two files directly

through existing tools [15,16]. In the case that a file system is damaged, the above method will not work well (e.g., system crash, human factors, etc.). In this case, we need to reconstruct SQLite database and WAL from the whole physical image as Sect. 3 has described.

When we extracting and analyzing pages to reconstruct the SQLite database file, there are a number of pages to filter out which do not belong to the current database. Through further analysis, we found that this is due in part to frequent deletion of the databases during several different cases studies. We also verified that the restore factory settings created the same problem. These results also prove that the restore factory settings do not erase all data, we are still able to recover some potential evidence from the image file.

6 Related Work

Although there are previous works on the recovery of SQLite records and history versions, substantial research on forensic analysis of SQLite based on WAL is limited. This section will review the existing researches on the analysis of SQLite.

The earliest research of database records recovery can trace back to 1983. Haerder [17] suggested a method that restored the deleted records using the transaction file. This method can be applied to traditional database on the PC when the information of deleted records is included in the transaction file.

Sangjeon [18] explained the management mechanism of the SQLite database when records were deleted. Sangjeon later proposed a method to recover deleted records from free blocks. However, the approach can only restore the deleted records from unallocated space within the page.

A year later, Lamine [19] presented a new tool to recover deleted records from SQLite databases based on a low-level analysis. In order to perform further shrink of the candidate page sets, they used pointer map page [4] to keep only pages that are part of a table B-tree and pages in overflow chains.

Other publications deal with the recovery of deleted records, which are not in the database file itself, but in the unallocated disk space. Pereira [20] researched the forensic analysis of SQLite databases of Mozilla Firefox. In contrast to other applications, no expired data remain in the Firefox database file. Therefore Pereira proposed a carving method for single records.

Shu, Zheng, and Xu [21] presented a new recovery method for Firefox history records based on the SQLite WAL file. An effective algorithm was proposed to reconstruct the whole data frame in WAL file from the unallocated space based on its structure, and the history records are extracted from the data frames according to the content of the records.

Xu, Yang, Wu, et al. [22,23] proposed a method to recover files, reconstruct the file system, and their previous history versions (Taking the SQLite database as a case study) using YAFFS2 metadata. However, since ext4 file system is widely used in android phones and Linux systems, this method cannot be applied to newer devices.

Generally, most of these works focus on recovering deleted records from the database file or carving single records from unallocated disk space. This paper details the reconstruction of SQLite history versions using the original database and WAL, and analyses two case studies using reconstructed history versions.

7 Conclusion

SQLite database is widely used to store messages, call history, browser history and much more, both on desktop computers as well as mobile devices. Therefore, it has tremendous forensic potential data and drawn more attention to digital forensic investigators and analysts.

When a SQLite database is in WAL mode, these potential forensic data will first present in WAL, and then be written to the database periodically. That is, WAL is also the source of evidence that we should study.

In this paper, we first reconstructed the original SQLite database and WAL. Then a method based on the original database and WAL to reconstruct SQLite history versions was proposed. The experimental results show that the proposed method can reconstruct history versions correctly.

Based on reconstructed files, we also demonstrated the utility of our proposed method. Our proposed method is capable of recovering deleted records and detecting tampered behaviors. It is evident that our method can recover recently deleted records and detect tampered behaviors effectively.

The widely use of large capacity flash memory provides us several opportunities and challenges. Large capacity will store more useful information. However much more effort is required to recover data and perform forensic analysis. Our future research will look at creating tools to ease forensic analysis on large capacity data storage.

Acknowledgment. This work is support by Natural Science Foundation of China under Grant Nos. 61070212 and 61572165, the State Key Program of Zhejiang Province Natural Science Foundation of China under Grant No. LZ15F020003.

References

1. Most widely deployed SQL Database. http://www.sqlite.org/mostdeployed.html
2. Rollback Journals. https://www.sqlite.org/tempfiles.html#rollb-ackjrnl
3. Write-Ahead Logging. https://www.sqlite.org/wal.html
4. The SQLite Database File Format. https://www.sqlite.org/filefo-rmat2.html
5. SQLite Source Code. https://www.sqlite.org/download.html
6. Xu, M., Yao, Y., Ren, Y.Z., Xu, J., Zhang, H.P., Zheng, N., Ling, S.: A reconstructing android user behavior approach based on YAFFS2 and SQLite. J. Comput. **9**(10), 2294–2302 (2014)
7. Master Table Structure. https://github.com/android/platform_packages_providers_telephonyprovider
8. Short Message Service. https://en.wikipedia.org/wiki/Short_Message_Service

9. Hoog, A.: Android Forensics: Investigation, Analysis and Mobile Security for Google Android, 1st edn. Syngress Publishing, Waltham (2011)
10. Breeuwsma, M.I.: Forensic imaging of embedded systems using JTAG (boundary-scan). Digit. Invest. **3**(1), 32–42 (2006)
11. Martini, B., Do, Q., Choo, R.K.-K.: Conceptual evidence collection and analysis methodology for Android devices. ArXiv e-prints, June 2015
12. Netcat(windows). http://www.securityfocus.com/tools/139
13. BusyBox. https://play.google.com/store/apps/details?id=stericson.busybox
14. DFRWS Challenge Report. http://sandbox.dfrws.org/2006/garfinkel/part1.txt
15. Proposed Methodology for victim android forensics. https://viaforensics.com/viaforensics-articles/viaforensicsaflgical-tool-android-forensic-investigations.html
16. ViaForensics. https://viaforensics.com/products/viaextract
17. Theo, H., Andreas, R.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15**(4), 287–317 (1983)
18. Jeon, S., Bang, J., Byun, K., et al.: A recovery method of deleted record for SQLite database. Pers. Ubiquit. Comput. **16**(6), 707–715 (2011)
19. Aouad, L.M., Kechadi, T.M., Russo, R.: ANTS ROAD: a new tool for SQLite data recovery on android devices. In: Rogers, M., Seigfried-Spellar, K.C. (eds.) ICDF2C 2012. LNICST, vol. 114, pp. 253–263. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39891-9_16
20. Pereira, T.M.: Forensic analysis of the firefox 3 internet history and recovery of deleted SQLite records. Digit. Invest. **5**(3–4), 93–103 (2009)
21. Xu, M., Shu, W.X., Zheng, N.: A history records recovering method based on WAL file of firefox. J. Comput. Inf. Syst. **10**(20), 8973–8982 (2014)
22. Xu, M., Yang, X., Wu, B.B., Yao, Y., Zhang, H.P., Xu, J., Zheng, N.: A metadata-based method for recovering files and file traces from YAFFS2. Digit. Invest. **10**(1), 62–72 (2013)
23. Wu, B., Xu, M., Zhang, H., Xu, J., Ren, Y., Zheng, N.: A recovery approach for SQLite history recorders from YAFFS2. In: Mustofa, K., Neuhold, E.J., Tjoa, A.M., Weippl, E., You, I. (eds.) ICT-EurAsia 2013. LNCS, vol. 7804, pp. 295–299. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36818-9_30