bring2lite: A Structural Concept and Tool for Forensic Data Analysis and Recovery of Deleted SQLite Records

By
Christian Meng and Harald Baier

From the proceedings of
The Digital Forensic Research Conference
DFRWS 2019 USA
Portland, OR (July 15th - 19th)

Digital Investigation 29 (2019) S31–S41

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2019 USA — Proceedings of the Nineteenth Annual DFRWS USA

bring2lite: A Structural Concept and Tool for Forensic Data Analysis and Recovery of Deleted SQLite Records

Christian Meng, Harald Baier*

da/sec Biometrics and Internet Security Research Group, Hochschule Darmstadt, Darmstadt, Germany

ARTICLE INFO

Article history:

Keywords:
Database forensics
Mobile device forensics
Data discovery
Tool testing and development
SQLite
Write-ahead log

ABSTRACT

As of today mobile applications such as WhatsApp or Skype make use of the SQLite database format to store its data. From an IT forensics point of view it is therefore essential to extract all information related to an SQLite database. In this paper we focus on digital traces in the scope of SQLite that belong to deleted database data. We suggest to make use of a structural approach: we analyse the deletion behaviour of SQLite depending on different database parameters, which affect the erasure of database data. As SQLite pragmas are an important parameter during deletion, we examine the erasure behaviour dependent on the pragmas in use. Based on the results of our analysis, we develop a concept to parse and process deleted records. Our concept is implemented within our tool `bring2lite`, which is evaluated with respect to a common dataset of SQLite files. The overall result is that `bring2lite` achieves the highest recovery rate of approximately 53% compared to eight competing programmes.

1. Introduction

Modern mobile operating systems make use of the SQLite database format to store application data. SQLite is popular for mobile applications, because the database is very fast and uses only few resources (Lite documentation - fi). For instance, the mobile messenger application WhatsApp relies on SQLite as storage format (WhatsApp is one of the most used communication channels worldwide with over 1.5 billion users each month (Statista, 2017)).

However, SQLite is not restricted to mobile applications. For example, popular web browsers like Firefox or Chrome and the mail client Thunderbird rely on SQLite. Besides application data SQLite is used to store meta data such as browser history, bookmarks, and login credentials, too.

From an IT forensics point of view it is therefore essential to extract all information related to an SQLite database. While the extraction of allocated database information is straightforward, the acquisition of deleted SQLite data is difficult, though.

In this paper we focus on digital traces in the scope of SQLite that belong to deleted database data. To stay time and resource efficient (especially on mobile devices), SQLite does not delete database records instantly by default. Instead it labels them as deleted. This behaviour is similar to the erasure process in file systems and offers the opportunity to recover some deleted SQLite data from the SQLite database files.

In the field of recovering deleted SQLite records, some commercial and open source tools have already been published to cover this topic. Sample common tools evaluated in this paper are the Forensic Browser for SQLite from Sanderson Forensics (Sanderson Forensics), Undark from Paul Daniels (Daniels) or Stellar Phoenix Repair for SQLite from Stellar Data Recovery (Stellar Data Recovery). We will show that our tool `bring2lite` is superiour to these tools with respect to recovering deleted SQLite data. Additionally some research papers address the deletion of SQLite records, e.g (Jeon et al., 2011) (Aouad et al., 2013) (Ramisch and Rieger, 2015) (Su and Xi, 2017). In Section 2 we present this related work and discuss their deficiencies.

An important aspect of current SQLite implementations is the use of the journal file format *Write-Ahead Log (WAL)*. WAL stores the latest changes to the SQLite database and is a richful source to recover deleted SQLite information. In contrast to our approach the majority of both the tools and the academic papers do not include the information provided by WAL files. A further relevant aspect in the context of deleted SQLite data is the inspection of the unallocated area within the SQLite database file. As of today this aspect is only treated by one academic paper (Su and Xi, 2017) and extended by our approach.

In order to improve the recovery rate of deleted SQLite data we suggest to make use of a structural approach: we analyse the

* Corresponding author.
E-mail address: harald.baier@h-da.de (H. Baier).

https://doi.org/10.1016/j.diin.2019.04.017

deletion behaviour of SQLite depending on different database parameters, which affect the erasure of database data. As SQLite pragmas are an important parameter during deletion, we examine the erasure behaviour dependent on the used pragmas.

Based on the results of our analysis, we develop a concept to parse and process deleted SQLite records. Our concept is implemented within our tool `bring2lite`. `bring2lite` is implemented in Python and publicly available.[1] In order to show the strength of our approach, we provide an evaluation of `bring2lite` and competing tools with respect to a common dataset of SQLite files. The overall result is that `bring2lite` achieves the highest recovery rate of approximately 53% compared to its eight competitors.

The rest of the paper is organised as follows: in Section 2 we review related work. Then Section 3 provides background information on the SQLite data format. Section 4 presents our structural analysis on data deletion in SQLite followed by Section 5, where we present our concept and algorithm to extract deleted SQLite data. Section 6 introduces our tool `bring2lite` followed by its evaluation in Section 7. We conclude our paper in Section 8 and a point to future work.

2. Related work

This section describes the current research in the field of SQLite with a focus on recovering deleted SQLite data. To cover all work that is related to this paper, we first give an overview of general approaches to restore deleted records. Beyond that, further methods are shown which do not only focus on the main file of the database or the typical structures in the database itself. We point out that in contrast to our approach all of the subsequent related work only considers a single part of recovering deleted records. In order to evaluate our tool with its competitors, Section 2.6 presents an SQLite database corpus and a standardised method of testing forensic SQLite tools.

There are further sources, which deal with the digital forensic analysis of SQLite databases. However, the focus of these sources is the actual forensic analysis rather than the recovery of deleted records. For instance, the recent book of Sanderson (2018) is an excellent reference to the general digital forensics aspects of SQLite, however, the recovery of deleted data is not yet at the core of this book. Anglano et al. (2017) conduct a regular forensic analysis of SQLite databases and only refer to other publications in the scope of deleted records. Pereira (2009) follows the approach to extract records on the partition level from temporary files of a moz_places database of the firefox web browser. Afonin and Katalov (2016) focus on SQLite records that contain deleted text messages, call logs, and chat entries. However, their process is very complex and requires the use of specialised software or SQLite specific expertise.

2.1. A recovery method of deleted record for SQLite database

Sangjun, Jewan, Keunduck and Sangjin (Jeon et al., 2011) describe a basic method to recover deleted records from an SQLite database. The method uses the schema table to collect information about the cell structure within the b-tree leaf pages. With this information the method searches for possible datatypes in the cells and tries to match them. To prove that their method works, the authors make use of an SMS database crafted by themselves. Nine records were recovered with the developed method from this specially crafted database. However, the authors do not consider SQLite data of journal files and unallocated space. Furthermore, their evaluation is superficial. The authors describe a tool called *SQLiteRecover*, however, it is not publicly available.

2.2. A tool for SQLite data recovery on android devices

The research paper (Aouad et al., 2013) written by Aouad, Kechadi and Di Russo focuses on recovering deleted records from the Short Message Service (SMS) and Multimedia Messaging Service (MMS) databases in Android devices (Aouad et al., 2013). makes use of a similar approach to our method, but with two important differences. The first difference is to shrink the size of the file pages that need to be processed by analysing the *pointer map page*, however, this only occurs if SQLite uses the vacuum pragma. The second change is that the processing of the SQLite structures are customized to match the fields of an SMS database file, that is (Aouad et al., 2013) does not develop a generic algorithm for all SQLite databases. To prove the viability of their method, the authors of (Aouad et al., 2013) present a specific evaluation and configure a setup with two Android devices. They were able to recover *95%* of the deleted records on the first Android device and *75%* on the second device, respectively. After one week of additional usage of the devices, the authors again extract deleted records. In this case, the results of the first device did not change, but the reconstruction of the data on the second smartphone only recovered *25%*. However, neither the source code nor an executable software of the concept used in (Aouad et al., 2013) is available.

2.3. Recovery of SQLite data using expired indexes

Ramisch and Rieger (2015) describe how indexes can be used to restore deleted records. Their basic idea is that expired records sometimes can be unrecoverable from the regular SQLite structure, but can still exist in the b-tree index leaf page. In the cells of these pages the indexed values are redundantly stored to the corresponding cell within the table b-tree leaf pages and therefore can be used to restore deleted records. The method described in (Ramisch and Rieger, 2015) first extracts index structures and uses them to enrich extracted data from freeblocks which are stored in the table b-tree leaf pages. The extraction itself works similar to the methods from the previous two research papers but without the component of enrichment. The tool was tested on multiple not specified databases and with the “database Envelope Index”, which contains message metadata from Apple Mail.” ((Ramisch and Rieger, 2015)). In these cases the tool worked well and could extract all deleted records correctly. The method works without the feature of processing payload which exceeds more than one page. Likewise the previous two sections, the authors of (Ramisch and Rieger, 2015) implemented a tool based on the provided method, but do not give access to any sort of source code or an executable.

2.4. Key technologies for mobile phone forensics and application

Su and Xi (2017) use a different pre-processing and framework approach in comparison to the other related work (Su and Xi, 2017). offers a method to first decrypt encrypted SQLite databases and afterwards extract deleted records. The first stage of the framework regards different encryption methods that are used by manufacturers and SQLite itself. The second stage is a similar method to the previous delineated approaches. In the first step, the authors extract the so-called “affinity types” and compare them to the types within the cells. If two field combinations match each other, the cell is ready for an extraction. To validate the reliability of the method, a database with four records is created. Afterwards, three of these records are deleted and one additional record is inserted. As the last step, another record of the last two records is deleted. With this prepared

[1] https://github.com/bring2lite/bring2lite.

database and the described algorithm all records can be recovered. Although the authors of (Su and Xi, 2017) describe the parsing process of non-allocated area as well as the freeblock structure, their provided method does not cover the freelist structure as a part of the extraction process of deleted records nor the WAL file.

2.5. SQLite forensic analysis based on WAL

In (Liu et al., 2017) Yao, Ming, Jian, Ning and Xiaodong introduce an alternative method, focusing on processing the Write-Ahead Log (WAL) file in addition to the main SQLite file. As a result of their research, a tool is developed which processes the main database and the corresponding WAL file in at most five steps. Similar to other related work, the method analyses the page type as well as the serial types. To test the functionality of the developed tool, two databases were crafted. On the first database, the checkpoint to release the data from the WAL to the SQLite database was not triggered and on the second, the checkpoint was triggered. The tool in (Liu et al., 2017) was able restore *100%* of the records on the first and *80%* on the second database. The authors show that their results are superiour to the approach of (Jeon et al., 2011) (see Section 2.1). In contrast to our tool `bring2lite` the described extraction approach in (Liu et al., 2017) only addresses the area of extracting deleted records from the WAL file.

2.6. A standardised corpus for SQLite database forensics

A fundamental problem in science in general and in IT forensics in special is the comparability of concepts and tools. The main reason is that an accepted ground truth is often missing. The paper of Nemetz, Schmitt and Freiling (Nemetz et al., 2018) offers a solution concerning this problem with respect to SQLite. To achieve a comparability between the amount of tools, a forensic corpus was created which contains standardised SQLite databases (Nemetz et al., 2018). describes criteria that need to be considered if such an SQLite corpus is designed and crafted. These criteria mention that a corpus has to be representative, complex, heterogeneous, annotated, available, distributed and maintained. As a result the researchers present a collection of 77 databases which are separated into 5 categories. One of these categories is designed to test tools with respect to recovering deleted records. The authors of (Nemetz et al., 2018) create a forensic corpus and apply a set of six tools. The comparison shows that no tool is able to recover all deleted records from the crafted databases. It is one main goal of our paper to show the effectiveness of available tools (including our method implemented within `bring2lite`) with respect to this ground truth of SQLite databases. A subsequent work (Schmitt, 2018) addresses the problem of anti-forensics in the scope of SQLite and tests the robustness of tools to handle anti-forensic measures.

3. Background on the SQLite specification

In this section we provide some fundamentals on the SQLite specification, which are necessary to understand our approach. This section bases on the standard book on SQLite (Allen and Owens, 2010) and the official SQLite specification (Lite documentation - fi) and its derivative documents. We first introduce pages and pragmas in Section 3.1 and then give some details on the actual SQLite file format in Section 3.2.

3.1. SQLite pages and pragmas

Before an SQLite database file can be created, certain configurations have to be adjusted. First, the page size needs to be set. A page is the smallest unit, which can be addressed, that is the page size defines how much new space will be allocated if the current file size is too small. A page is similar to a cluster on file system level and a sector on partition level, respectively.

A second important configuration parameter are *pragmas*. A pragma basically is an option, which typically is customized before an SQLite database is created for the first time. The relevant pragmas in the scope of our paper are the following (see (Lite documentation — pr)):

- *secure_delete*: The pragma secure_delete is configurable with the three settings 0, 1 and FAST. A special setting of this pragma is the option FAST which leaves forensic artefacts in the so called freelists. If this pragma is set to 1, all deleted records and pages will be overwritten by zeros (see (Lite documentation — pr)).
- *auto_vacuum*: The second pragma is called auto_vacuum. If this pragma is turned on, the database deletes unused pages and will not keep these pages in a freelist (see (Lite documentation — pr)).
- *journal_mode*: The last relevant pragma is journal_mode which enables or disables journaling. The pragma has six options which are DELETE, TRUNCATE, PERSIST, MEMORY, WAL and OFF. All of these options, omitting the WAL and OFF switch, will create an additional file with the ending '-journal'. If the pragma is set to WAL, a file with the '-wal' ending will be created. The WAL file will never be deleted, but as it has a fixed size SQLite overwrites its pages once the WAL file end is reached (see (Lite documentation — pr)).

All three pragmas affect the conditions under which SQLite keeps deleted records in any structure in both the main database file and any optional journal file.

3.2. SQLite file format

An SQLite file is organised in a so-called *b-tree structure* and is divided into an SQLite header and an SQLite body (Lite documentation - fi). The header comprises the first 100 bytes of the file and it provides basic information such as a header string, page size, size of the database file, text encoding. At offset 100 of the SQLite file, the SQLite body starts with its first page. This page contains a structure called *SQLite master table*. The master table holds essential information about the table and index schemas of the database and is the entry point to each table and index through the page number of its first page.

In the context of this paper, pages belong to one out of four different types. Each type has its own task and structure. The types are the following:

Freelist trunk page is organised in 4 bytes page pointers. The first 4 bytes pointer will point at the next freelist trunk page. Every subsequent pointer shows the page number of a freelist leaf page. The pointer to the first freelist trunk page is stored in the SQLite header at offset 32.

Freelist leaf page is a free page which can be allocated to store new records. This type of page holds no allocated data or any other type of content.

Table b-tree interior page holds pointers to its children b-tree pages and to the page number of the right-most child (the child page with the largest number), which is held separately. Similar to a freelist trunk page, the pointer is 4 bytes in size.

Table b-tree leaf page is the key structure to store records in an SQLite database. This type of table b-tree page is the only one that can contain active data.

Each table b-tree leaf page is divided into a *page header* and a *page body* as shown in Fig. 1. This type of page is the only structure

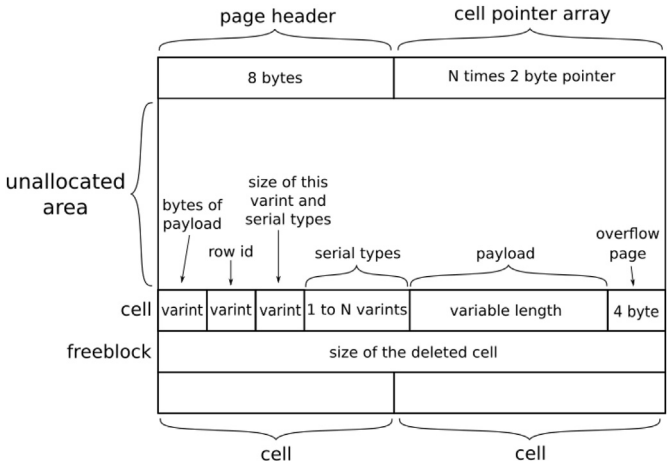

Fig. 1. Layout of a regular table b-tree leaf page, which stores active and deleted records.

within an SQLite file, which holds regular user or application data. The page header comprises 8 bytes and is followed by the *cell pointer array*. The cell pointer array stores pointers (each of which is 2 bytes long) to every cell within the same page. SQLite stores a cell as far as possible to the end of a page in the *cell content area*. Please note that the first cell pointer addresses the last cell of the page, and the last cell pointer points to the first cell in cell content area. The region between the last pointer of the cell pointer array and the first cell in the cell content area is the *unallocated area*.

One important pointer in the page header is the *freeblock pointer* at offset 1 in the page header. Freeblocks contain removed cells in the cell content area and are organised as a chain: the freeblock pointer is the entry point to the chain (i.e., the first freeblock), and each freeblock contains a pointer to its successor. A freeblock is separated in three areas: (1) a 2-byte pointer to the next freeblock, (2) the length of the current freeblock encoded in 2 bytes and (3) the actual free area with deleted content. If a freeblock is the last in a freeblock chain, the first field will be set to zero. For our considerations, the third area of a freeblock is of special interest.

Algorithm 1. Calculation of a varint based on a byte sequence.

```
   Data: byte-sequence (e.g., a serial type)
   Result: bitstring of varint
1  varint = 0;
2  index = 0;
3  while byte-sequence [index] >= 128 do
4  |   varint = varint || (byte-sequence[index] − 128);
5  |   index += 1;
6  end
7  if byte-sequence [index] < 128 then
8  |   varint = varint || byte-sequence[index];
9  end
10 return varint;
```

The smallest entity within an SQLite database is the *cell*. Numbers in a cell are encoded by a *variable-length integer* or shortly a *varint*. The length of a varint is 1–9 bytes, its value can be calculated as shown in Algorithm 1. The calculation algorithm iterates over the input byte array (e.g., the serial type of the cell) and goes over all bytes until a byte with most-significant bit equal to zero is found (that is the numeric value of the byte as unsigned integer is smaller than 128). The most-significant bit of all preceding bytes is removed to get the actual representation of the underlying 64-bit number.

Each cell starts with two varints to represent the number of bytes of the payload and the row id, followed by a byte array, and ends with an optional 4 byte pointer, which refers to an overflow page if one is needed. The byte array stores a varint, which holds the header length of the byte array, serial types and the actual payload. All serial types besides TEXT and BLOB are encoded with a defined value. A BLOB is defined by a number that is greater than or equal to 12 and even, while a TEXT is defined by a number that is greater than or equal to 13 and odd. Finally, the size of the record stored in the payload can be calculated with the serial types as explained in (Lite documentation - fi).

4. Structural analysis on SQLite record deletion

In this section we perform a structural analysis to learn about the deletion behaviour of SQLite. Our central goal is to observe how SQLite removes records under different conditions, i.e. to learn the reality of SQLite deletion. More precisely we analyse, which data structures still provide information if content is deleted from the database. From Section 3 we know that deleted content may be found in data structures of a table b-tree leaf page (unallocated area, freeblocks), a freelist page, and finally artefacts in a journal file, e.g., a WAL file.

As discussed in Section 3.1 SQLite uses pragmas to configure the deletion of database entries. In order to learn about how SQLite deletes records under different conditions and in different data structures, various scenarios are defined and examined. These scenarios are labelled with S1 to S6, where we start with an easy scenario and turn to more complex ones.

As we define each scenario and each pragma configuration on our own, no sample SQLite files are available, that is we have to generate our SQLite files to observe the deletion behaviour on our own. In Section 4.1 we introduce our six scenarios and explain our generation method of our test file set. Then in Section 4.2 we present and review the results of our structural analysis.

4.1. Scenarios and test files

We explain our approach to generate the test files to observe the deletion behaviour of SQLite. We have to generate these SQLite database test files for our structural analysis on our own for two reasons: first a database, which contains realistic data and is grown over months or even years, contains a huge and complex amount of data. In this case it is highly probable that we overlook a structure and misinterpret the deletion behaviour and hence do not reproduce reality. Second – and similar – when we define the deletion behaviour on our own, we know about the actual ground truth.

We generate our SQLite test database files for our structural analysis with the help of a self developed tool. This tool produces SQLite database files by reading two different configuration text-files: the first one defines the database schema and the second one the actual database records. We only generate one table. This table contains columns for *id*, *name*, *surename* and *zip*.

Based on this information an SQLite statement script is initialised. We use this script and hence generate an SQLite test file for every scenario and all combinations of pragmas, respectively. In all we test 12 pragma combinations for each of the 6 scenarios, hence in all we create 72 database files for our structural analysis. Sample results for some pragma combinations are listed in Table 1.

Our six scenarios are as follows (from easy to more complex):

Scenario S1 Insert 1 record, delete it.
Scenario S2 Insert 3 records, delete 1 record.

Scenario S3 Insert 3 records, delete all.
Scenario S4 Insert records until a second page will be created, delete the record on the second page.
Scenario S5 Insert records until a second page will be created, delete all records on the first page.
Scenario S6 Insert records until a second page will be created, delete all records.

The first scenario represents the simplest among all. Only one record is inserted and deleted afterwards to empty the database again. The second and third scenario are an extended version of the first scenario as three records are inserted, resepectively. The second scenario deletes only one and the third deletes all three records. That way it can be examined how SQLite tries to optimise the structure when only one record is deleted and the rest is still in use. Besides, it can be considered what happens if all records are deleted and the used page, which holds multiple deleted records, will be set free.

The last three scenarios (i.e., S4 to S6) address the aspect of deleting a whole table b-tree leaf page and especially how SQLite performs if a whole table b-tree leaf page is inserted and deleted afterwards. In this particular case, 159 records are necessary to fill an entire table b-tree leaf page. This number of records depends on the concrete data types and content of the records.

4.2. Results of structural analysis

In this section we present the results of our structural analysis on base of the scenarios and different pragma combinations as explained in Section 4.1. To clarify again our main goal is to identify, under which conditions a recovery of deleted SQLite data is possible.

In all we have 72 different test settings. We present sample results of our structural analysis in Table 1, further results are available in Meng (2018). A scenario in Table 1 is marked with

- a "+", if all records of the according test file can be fully extracted,
- a "0", if some records leave traces and may be restored, and
- a "-", if no record of the according test file can be extracted, that is no trace can be found via a hex editor.

The rating is with respect to the main SQLite database file, if no additional journal file is generated. If a journal mode is activated, the rating in Table 1 depicts the recovery result of this additional file. While the "+" and "-" rating is unique, respectively, the neutral rating "0" only means that parts of the original records leave traces after deletion. For example S5 is rated as neutral in the second column of Table 1. In this particular case a freelist page holds only about 50% of the data within a freelist page.

We start our discussion with the first column in Table 1. The rating shows that all deleted records can be extracted, if the pragma combination secure_delete = 0 and auto_vacuum = 0 is used with no journal, that is we consider the recovery results from the main SQLite database file. Using a hexdump viewer we observe that besides the first 4 bytes of the deleted record the content is still available. In this case SQLite deallocates the content by generating a freeblock and changes the information about the first freeblock at offset 1 in the page header. If a deletion of multiple adjacent records is executed, the whole deleted area is defined as a single huge freeblock and may be restored, too.

Next we observe the deletion behaviour, if we only change the pragma secure_delete to the FAST setting and leave the remaining two pragmas unchanged (second column in Table 1). As no journal file is use, the recovery results are with respect to the main database file, too. The SQLite specification (Lite documentation — pr) describes the FAST setting as follows: "This has the effect of purging all old content from b-tree pages, but leaving forensic traces on freelist pages." Therefore we do not expect to recover traces in the table b-tree page itself, but in a freelist leaf page, if such a page was generated. A freelist is generated, if (1) at any time, some data of the database table was written on this page, (2) no more data of the table is stored on this page, and (3) the database allocated at least two pages for the table. In our test setting, only scenarios S5 and S6 generated freelist pages, hence only for these scenarios we were able to recover content.

The third column in Table 1 sets the pragma secure_delete = 1 and makes use of a WAL file as journal. Although the whole content in the SQLite database file is securely deleted (because SQLite writes zero bytes to the location of the deleted records), we are able to restore the whole content from the WAL file. All previously deleted records are still allocated in the last but one written frame of the WAL file. We point out that SQLite does not delete or change any frames of a WAL file, but continuously writes changes into a subsequent new frame, that is the whole page history in its previous states can always be recovered through the formerly written frames.

The fourth column in Table 1 makes use of a rollback journal in the PERSIST mode. The pragma secure_delete is set to 1 and auto_vacuum = 0 is used. The PERSIST journal mode does not delete the rollback journal file after completion of a transaction, it only wipes out the journal file header with zero bytes. Compared to a WAL file the rollback journal differs in terms of the number of pages that can occur within the rollback journal. More precisely a rollback journal stores every page only once. After deletion of the content in the SQLite database file we do not observe any indication about a deletion in the corresponding rollback journal page. In this test setting all records are readable without any limitations from the rollback journal file. Our hypothesis is that SQLite first inserts all records into the rollback journal and then directly deletes the processed data in the main database file.

Finally we point to the case of deleted content in the unallocated area. If a database is in long-time usage, the unallocated area between the cell pointer array at the beginning of each table b-tree leaf page and the cell content area can contain deleted data as well. For instance this happens if cells next to the unallocated area are

Table 1
Sample pragma combinations under which it is possible to restore deleted records.

Scenarios	secure_delete = 0/ auto_vacuum = 0/ journal_mode = OFF	secure_delete = FAST/ auto_vacuum = 0/ journal_mode = OFF	secure_delete = 1/ auto_vacuum = 0/ journal_mode = WAL	secure_delete = 1/ auto_vacuum = 0/ journal_mode = PERSIST
S1	+	−	+	+
S2	+	−	+	+
S3	+	−	+	+
S4	+	−	+	+
S5	+	0	+	+
S6	+	0	+	+

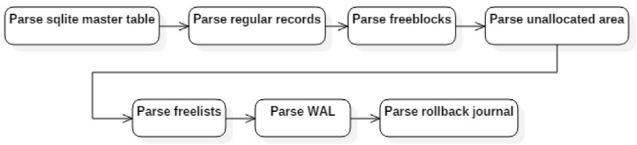

Fig. 2. Flowchart of the whole method that tries to recover deleted records from an SQLite database file.

deleted (that is cells, which have been allocated lastly). In such a case SQLite just changes the pointer of the beginning of the content area instead of using a freeblock.

5. Concept to extract deleted content

In this section we develop our concept to extract deleted data from an SQLite database. It is the result from our structural analysis from Section 4. Fig. 2 shows an abstract overview about the method that is the basis of our concept. The different processing steps are the extraction of SQLite data from the SQLite master table, regular records, freeblocks, unallocated areas, freelist pages and finally the journals WAL and rollback, respectively.

First the SQLite header needs to be processed before any of the other steps can be triggered. Then the SQLite master table is parsed. This table is always stored on the first page of the database file and hence is easy to find. The master table stores information about all tables and indexes of the database. Important information are the respective schema and the entry point to the table or index (the entry point is stored as the page number of the table or index root page). Using the master table, Algorithm 2 connects all pages of the database file to their respective schema. It is necessary to check if the SQLite master table is a table b-tree interior page or a regular table b-tree leaf page. If the first database page is a table b-tree interior page, all table b-tree leaf pages need to be gathered through the page pointers first and be parsed afterwards. The entry point to a table may be either a table b-tree leaf page (if the table is small and all data fits in one page) or a table b-tree interior page (if the data exceeds one single page). Based on the result of Algorithm 2, a key-value mapping that joins the pointers to the related SQLite schemas is possible.

Algorithm 2. Connects all pages with their corresponding schema.

```
   Data: schemas (array of all entry pointers), page-size
   Result: all schemas connected to their pages
1  result = [];
   /* loop over all entry pages                    */
2  for p in schemas do
3      if p > 0 then
4          page = extract_page((p-1) * page_size);
5          if page == interior b-tree page then
6              result[p].append(
               page.header.right_most_pointer);
7              numb_cells = page.header.number_of_cells;
8              cells = extract_cells(page, numb_cells);
9              result[p].append(cells);
10         else
11             result[p].append(p);
12         end
13     end
14 end
15 return result;
```

If all information from the SQLite master table is extracted, we parse the regular records. This can be easily done using the outcome of Algorithm 2 and the information of the cell pointer area in each b-tree page.

The next step is to recover deleted records from each active page of the database. These records are either stored in freeblocks or in the unallocated area. We first discuss the recovery of deleted data from freeblocks. Using the page header of an active page we jump to the first freeblock of this page and then traverse the chain of freeblocks. Due to the linked list concept of freeblocks this is straightforward. However, the first four bytes of each freeblock are used for building the linked list and hence the original content of these four bytes may not be recovered, i.e., the original varints, which were stored in these four bytes, are lost. Nevertheless due to Algorithm 2, we know the schema of the table and thus it is possible to make an assumption, which data types were held by the cell. A cell always starts with a varint to encode the length of the cell followed by a varint for the integer key (the rowid). After that, the actual record header starts with its varint encoded header length. Often each of these three varints is one byte long, respectively. After the record header varint SQLite enumerates a varint for each column (that is the serial type array). As the freeblock wipes out the first four bytes, mostly the first varint of the serial type array is not recoverable. Based on the table schema from the master table, the length of every serial type and the overall length of the serial type array can be calculated. We make use of the supposition that the wiped out first varint of the serial type array is always an integer.

Algorithm 3. Extract deleted data from a freeblock.

```
   Data: serial types (based on the table shema), L
         (estimated value of header lengh), freeblock length
   Result: multiple possible solutions that could match the
           record based on the related shema
1  possible-solution = [];
2  length =
   calculate-length-of-freeblock-content(serial-types);
3  content = extract-content(L, length);
4  possible-solution.append(content);
5  return possible-solution;
```

To calculate the length of a cell header and thus find the start of its payload, let v be the length of the first three varints within a cell (length of the cell payload, rowid and record header length). Furthermore, let s be the number of all serial types that are defined within the schema of the current page. Let b be the numbers of data types with variable length (blob and text) fields that occur in the schema related to the current cell. This variable is zero if there is no text or blob field stored within the schema. Then the length L of the header is $L = v + s + b$.

The problem with this calculation is that there are two variables that are unknown in general. First the variable v can grow to a size of 27 bytes as each of the varints has a maximum size of 9 bytes. Second the size of b is not readable from the schema. We therefore first iterate over the variable v and check every possibility of b at each value of v until a certain threshold is reached. In each iteration step the extracted field of the cell, which is based on the serial types, is compared to the data types held in the corresponding schema of the current page. In this implementation the threshold is set to one for every additional text or blob field in the schema. This value can be set to a maximum of nine (maximum value of a varint) per variable length field to test all possibilities.

A comparable calculation of the exact length of a cell is proposed by Jeon et al. (2011). However, the computation in (Jeon et al., 2011) is not suitable in our opinion as Jeon et al. compute the whole cell length at once. However, this requires the knowledge of the cell

header length, which is not the case in advance.

If the comparison between assumed serial types and schema matches, these values are considered to be correct. Afterwards, the records of the freeblock can be extracted as shown in Algorithm 3. Through the serial types that are extracted from the previously described step, the length of the payload can be calculated, as shown in line 2 of Algorithm 3. This happens through the conversion between serial types and length predefined from SQLite. With the length of the payload and the variable *L*, the extraction of several fields can be processed. The final conversion into human readable values of every field is a post-processing step and not shown in Algorithm 3.

The return value of Algorithm 3 is realised as a list, because if the first three varints only sum up to three bytes, it is not possible to recover the integer type held within the first byte of the cell payload. In this case, all six possible integer values have to be iterated and stored as possible solutions. Furthermore, a freeblock can consist of more than one deleted cell. This is an additional possibility, in which more than one record can be added to the solution list.

Algorithm 4. Extraction of unallocated area from a table b-tree leaf page.

```
   Data: page (i.e. hexdump of currently processed page)
   Result: processed unallocated area
1  header-length = get-header-length(page);
2  cellpointerarray-length =
   get-cell-pointer-array-length(page);
3  stop = get-start-of-cell-content-area(page);
4  unalloc-area = extract-unalloc-area(header-length +
   cellpointerarray-length, stop);
5  deleted-record-pointer = NULL;
6  for x = 0 → length(unalloc-area) do
7  |  byte = extract-byte(x);
8  |  if byte != "" then
9  |  |  deleted-record-pointer = x;
10 |  |  break;
11 |  end
12 end
13 result = reduce-unalloc-area(unalloc-area,
   deleted-record-pointer);
14 return result;
```

Next we turn to the unallocated area. Algorithm 4 describes the extraction of the unallocated area from a regular table b-tree leaf page. First, it is necessary to calculate the values of header length and extract the number of cells within the cell pointer array to find the starting point. The stop point is where the cell content area starts. After the extraction of the unallocated area it has to be inspected for any data, that is bytes different from zero bytes.

We now turn to the extraction of freelist pages. The algorithms of regular b-tree leaf pages are also suitable to process freelist leaf pages if little adjustments are made. A freelist leaf page is not designated to contain any data at all. In other words, the previously described algorithms can process a freelist leaf page from the start to the point where data is found. Unlike freelist leaf pages, the freelist trunk pages still contain an array which holds pointers to the freelist leaf pages. With the trimmed unallocated area from Algorithm 4, a processing of the extracted cells as in the regular page processing algorithm and freeblocks as in Algorithm 3 can be released. There are different options how these types of unallocated areas can be processed with these algorithms. The concrete implementation in this paper always tries to extract a freeblock header and then the serial types. Subsequently, the content will be extracted based on the found serial types.

For the extraction of WAL and rollback journal files, we refer to (Meng, 2018).

6. The tool bring2lite

As proof of concept we implement a tool called `bring2lite`. This tool implements the algorithms of Section 5 and adds a few functions for usability reasons. `bring2lite` is implemented in python. Digital forensic experts often make use of command line tools like dd (Rubin et al. Kemp), file (Darwin) or strings (strings). The command line interface of these tools accepts a user input via flags and presents the results on the terminal. Hence we decide to make use of a simple user interface based on command line input, too. Furthermore `bring2lite` informs its users about the current progress of its processing status. Finally `bring2lite` generates a cryptographic hash. It provides all extracted information via a folder structure within the chosen destination folder.

Key advantages of `bring2lite` compared to general purpose tools like strings are as follows: first, `bring2lite` is able to connect recovered data to its according database schema and table. Hence the context and the data types of each record field are known. Second as our tool is aware of the data types it is able to recover numbers from their varint encoded representation. There is no need for any post processing step to find out the actual value.

We will present sample interaction with and information on our tool in what follows. We assume to work on a Linux operating system and work in the directory `bring2lite` in the user's home directory, where the tool is installed. We call this folder the main directory of `bring2lite`.

6.1. Design of bring2lite

In this section we explain the main design of `bring2lite`. In the main directory of the tool we find a readme file, a setup python script and the subdirectory `bring2lite`, where the actual Python scripts and classes are stored. The content of the `bring2lite` subdirectory is shown, too.

```
~/bring2lite$ ls
bring2lite  __init__.py  README.md  setup.py

~/bring2lite$ ls bring2lite
classes  debug.log  __init__.py  main.py
```

Over all steps the SQLite database is handled in its binary format. The first three steps of our programme are as follows: (1) gather basic information about the SQLite database file from the SQLite header, (2) process and extract all shemas from the SQLite master table and connect them to the corresponding database page and (3) loop over all pages of the processed SQLite database. Our main goal is to encapsulate the functionality of the third processing step into different parts and classes to master the complexity of the whole extraction process.

An overview of the class structure of `bring2lite` is depicted in Fig. 3. The classes are stored in the classes directory of the `bring2lite` subfolder. They are as follows:

```
~/bring2lite$ ls bring2lite/classes
gui.py       parser.py          report_generator.py
__init__.py  sqlite_parser.py   potentially_parser.py
__pycache__  journal_parser.py  visualizer.py
WAL_parser.py
```

The parsing and extraction logic is implemented in one of the five parser classes, where Parser is the parent class and the remaining four parser classes its inheritents. In order to process a database file, the relevant parser class calls a function parse(), which is already defined in the super class Parser (the PotentiallyParser calls the function parse_page() instead). Using such a class design implementation the page processing logic can be placed in one single location and all pre- and post-processing operations are able to be done in the particular classes depending on the actual database file (SQLite, journal, WAL). Sample parsed and extracted data is `file_size`, `header_size` or `page_size` that is required to process every database file type. Additional fields like file_change_counter or checkpoint_sequence are individual for journal or WAL files and hence to its corresponding parser.

Each interaction with the exception of the results generated by `bring2lite` between a user and the tool is encapsulated in the GUI class.

6.2. Sample usage of bring2lite

We show a sample use of `bring2lite`. A sample command line call is as follows:

```
~/bring2lite$ python3.6 ./bring2lite/main.py \
-filename ./db/0B-02.db --out ./result
```

In our example we invoke the main python script main.py of our tool `bring2lite` via the python interpreter. The tool works on the SQLite database as stated by the –file switch. In our example we investigate the database file 0B-02.db from the corpus (Nemetz et al., 2018) as explained in Section 7 (the database file resides in the folder db). The switch –out of `bring2lite` is used to define the folder, where the tool deposits its output.

```
~/bring2lite$ ls -l results drwxr-xr-x cm cm 16384 Mr
27 20:34 freeblocks
drwxr-xr-x cm cm 16384 Mr 27 20:34 freelists
drwxr-xr-x cm cm 16384 Mr 27 20:34
regular-page-parsing
drwxr-xr-x cm cm 16384 Mr 27 20:34 schemas
drwxr-xr-x cm cm 16384 Mr 27 20:34 unalloc-parsing
```

`bring2lite` generates at most five different folders in the output directory results to store recovered SQLite information from different categories of data structures. If an output folder is not generated, `bring2lite` did not succeed to find a data structure of that category. The folder freeblocks is used to store reconstructed information from freeblocks of database pages. The directory freelist occurs if `bring2lite` processes any freelist page. The folder regular-page-parsing holds database information from allocated (i.e. non-deleted) SQLite data. Recovered information on the schemas of the SQLite database are written to the directory schemas. Finally the folder unalloc-parsing holds restored SQLite data from the unallocated space within an SQLite page. In our particular example all five folders are generated, i.e. our tool was able to find deleted data in all implemented SQLite categories as well as regular non-deleted content.

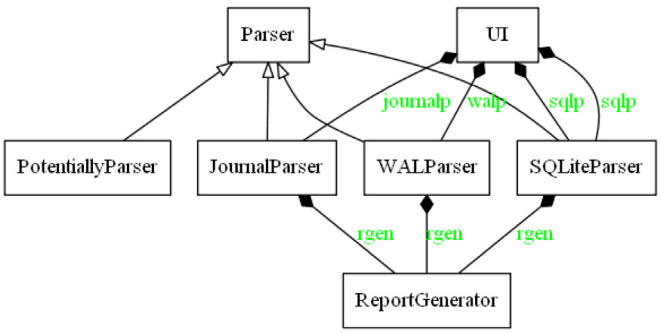

Fig. 3. Class structure of `bring2lite`.

bring2lite writes its data to a log file in each of these five subfolders. The name convention of a log file is N-page.log, where N is the page number of the SQLite file, where the data originally resided, that is every log file stands for a single page from the SQLite database. For instance if an investigator wants to examine all recovered information from the unallocated area of the first page, he has to access the file 1-page.log in the unalloc-parsing folder. A sample output is as follows:

```
~/bring2lite$ ls -l results/unalloc-parsing
-rw-r--r-- cm cm 2896 Mr 27 20:34 1-page.log
```

The processed database 0B-02.db only stores deleted records within the unallocated space of the first page. Our tool was not able to find any other record stored in an unallocated area of another database page, because no second file was created. We next look at the content of the recovered information from the unallocated space.

```
~/bring2lite$ less results/unalloc-parsing/
1-page.log
INT,TEXT,TEXT,INT,REAL
20010,Luisa,Kuhn,-1407291853,4892744407.93914
20009,Christian,Schulze,527030628,4362154905.
38727
20008,Zoe,Schubert,-603005252,4007666590.16147
20007,Luca,Scholz,1643805150,1166617011.72898
++++++++++++++++++++++++++++
```

The first line of a log file shows the schema of the processed page or an error text if no schema could be extracted. Then one extracted record is written to each line. Finally a separation line of plus characters is generated to separate the outcome of two extraction processes.

7. Evaluation

In this section we evaluate our tool `bring2lite` with respect to database files from the forensic corpus of Nemetz et al. (2018). We compare our tool to competing ones, too. We decided to make use of the corpus provided by (Nemetz et al., 2018) because it is a standardised benchmark for extracting deleted data from SQLite database files. Our central evaluation result is depicted in Table 2.

7.1. Evaluation corpus and tested tools

In our opinion the corpus in (Nemetz et al., 2018) comprises two segments: the first one is able to test the general functionality of a digital forensic SQLite recovery tool. This covers aspects such as encoding (e.g., which UTF-encoding is supported), how a tool handles special characters, or column names with extremely high length. The second segment is designed specifically to test the ability of a tool to recover deleted records from regular SQLite database file structures. This segment is divided into five categories and every category is named with a hexadecimal digit starting from 0A, that is the corpus contains the categories from 0A to 0E. Every database file within a category is numbered starting with 01, e.g.,

Table 2
Recovery results of our tool `bring2lite` compared to other tools tested with the forensic corpus from (Nemetz et al., 2018).

Case	Tools tested by the creators of the forensic corpus (Nemetz et al., 2018)						Tools tested by the authors of this paper		
	Undark	SQLite Deleted Records Parser	SQLabs SQLite Doctor	Stellar Phoenix Repair forSQLite	SysTools SQLite Database Recovery	Sanderson Forensic Browser for SQLite	Sqlite Forensic Explorer	Autopsy SQLite Deleted Records Plugin	bring2lite
0A-01	20/20*	0/20	0/20	0/20	0/20	0/20	0/20	0/20	20/20
0A-02	9/20*	20/20*	0/20	0/20	0/20	0/20	0/20	0/20	1/20
0A-03	20/20*	0/20	0/20	0/20	0/20	0/20	0/20	0/20	20/20
0A-04	15/20*	10/20*	0/20	0/20	0/20	0/20	0/20	0/20	13/20
0A-05	11/20*	20/20*	0/20	0/20	0/20	0/20	0/20	0/20	3/20
0B-01	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	4/10
0B-02	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	4/10
0C-01	0/7	0/7	0/7	0/7	0/7	7/7	7/7	0/7	6/7*
0C-02	0/10	0/10	0/10	0/10	0/10	10/10*	9/10	0/10	8/10*
0C-03	0/7	7/7	0/7	0/7	0/7	2/7	4/7	0/7	6/7*
0C-04	0/10	10/10*	0/10	0/10	0/10	1/10*	8/10	0/10	8/10*
0C-05	0/10	10/10*	0/10	0/10	0/10	10/10*	9/10	0/10	10/10
0C-06	0/7	0/7	0/7	0/7	0/7	0/7	5/7	0/7	6/7*
0C-07	0/10	0/10	0/10	0/10	0/10	0/10	10/10	0/10	9/10*
0C-08	0/10	10/10*	0/10	0/10	0/10	0/10	6/10	0/10	7/10*
0C-09	5/10*	10/10*	0/10	0/10	0/10	0/10	2/10	0/10	0/10
0C-10	11/20*	20/20*	0/20	0/20	0/20	0/20	2/20	0/20	5/20
0D-01	0/5	2/5*	0/5	0/5	0/5	0/5	1/5	0/5	1/5
0D-02	0/5	1/5*	0/5	0/5	0/5	0/5	1/5	0/5	1/5
0D-03	0/5	0/5	0/5	0/5	0/5	0/5	1/5	0/5	0/5
0D-04	0/5	2/5*	0/5	0/5	0/5	0/5	0/5	0/5	1/5*
0D-05	0/5	0/5	0/5	0/5	0/5	0/5	0/5	0/5	0/5
0D-06	1/10*	5/10*	0/10	0/10	0/10	0/10	0/10	0/10	0/10
0D-07	0/5	5/5*	0/5	0/5	0/5	0/5	5/5	0/5	3/5*
0D-08	0/5	5/5*	0/5	0/5	0/5	0/5	3/5	0/5	3/5*
0E-01	3/7	2/7	0/7	0/7	0/7	3/7	0/7	0/7	5/7*
0E-02	0/5	0/5	0/5	0/5	0/5	0/5	0/5	0/5	3/5*
Sum	**95/278**	**139/278**	**0/278**	**0/278**	**0/278**	**33/278**	**73/278**	**0/278**	**147/278**

0D-02 is the second SQLite database file in the fourth category. In all 27 files with 278 deleted records are inspected.

In order to compare `bring2lite` to other competing tools (i.e., tools to recover deleted records from SQLite database files), we proceed as follows: first, we adopt results of Nemetz et al. (2018), that is we cite the recovery performance of six tools tested in (Nemetz et al., 2018). Second we tested our tool `bring2lite` and two further ones against the corpus of (Nemetz et al., 2018) (we tested the Sqlite Forensic Explorer (SQLite) and Autopsy SQLite Deleted Records Plugin (McKinnon, 2017) on our own, as these tools are freely available to us).

7.2. Evaluation results

The overall evaluation result is given in Table 2. Tools that are located under the left heading were tested by Nemetz et al. (2018) and we only cite their recovery performance. All tools that are located beneath the right heading are tested by the authors of this paper. Each entry in Table 2 shows the number of items, where all deleted data successfully was recovered for that file, in relation to all items in the test database file. Nemetz et al. (2018) introduced a category labelled with a star: (Nemetz et al., 2018) make use of this labelling if *"some elements correctly, some wrongly processed (errors)"*

The first category of the test corpus is labelled by 0 A. The key test setting is to create one or more tables, insert in all 20 elements, and finally launch a drop statement. From Table 2 we see that five SQLite test database files belong to this category. Additionally we see that some tools cannot handle this type of delete operations. For instance programmes such as the *Forensic Browser for SQLite* or *Sqlite Forensic Explorer* do not process the SQLite master table or the unallocated area within a table b-tree leaf page. Furthermore we extract from Table 2 that the tools *SQLite Deleted Records Parser* or *Undark* seem to perform better than our tool `bring2lite` for some database files within this category. However, all such results are marked with a star and hence only provide partial recovery – in contrast to our tool. For instance by examining the source code of SQLite Deleted Records Parser we see that the SQLite Deleted Records Parser simply eliminates the non-printable characters and returns the printable characters as a result. With that approach of processing SQLite database files, only ASCII characters are correctly extracted. However, data types such as integer and floating point numbers need a manual post-processing. We point out that tools like strings (strings) work in a similar way.

The poor performance of `bring2lite` with respect to database file 0A-02 is due to the following fact. 0A-02 contains a freelist page, that is there is no information about freeblocks available. `bring2lite` parses a freelist page sequentially from the beginning to the end, and it does not distinguish between a regular record and a freeblock. As the database 0A-02 was created trough a randomised deletion SQLite statement, a freeblock was created subsequent to the first regular database entry, and `bring2lite` is not able to process this freeblock and all subsequent records.

The second category of the test corpus is labelled by 0B. The test setting is to create one or more tables, insert 10 elements, followed by a drop, create and insert statement, respectively. From Table 2 we see that two SQLite test files belong to this category. The first two categories differ from each other as the databases in this category trigger a create table operation after the drop has finished. No tool can parse any records except from `bring2lite`, which recovers four records per file. Every record of the two databases was found in the unallocated area of the SQLite database.

The third category 0C shows the results of all tested tools based on databases, where create, insert and subsequent delete operations are used. All these operations put deleted records in regular SQLite structures. Concerning the results of this category, the *SQLite Deleted Records Parser* supposedly recovers more records than `bring2lite`. However, the SQLite Deleted Records Parser only partly recovers records. Furthermore as can be seen in database files 0C-02, 0C-05, 0C-06 or 0C-07, respectively, the tool

bring2lite achieves better results. The database file worth to be discussed is 0C–05. By looking at the results of this database file, the tool bring2lite is the only one that can reach an extraction rate of 100%. All other tools have partly recovered records and are consequently labelled with a star or have a lower recovery rate compared to bring2lite.

The second last category 0D comprises SQLite database files, which create a table, insert new records, delete some of them followed by a further insert statement. From Table 2 we see that nearly all tools have problems to process this type of databases. The tool *SQLite Deleted Records Parser* reaches a formal recovery rate of 44.4%, but all results are labelled with a star. These results are to be considered as critical, because this category is the closest to reality among all presented categories. Our tool bring2lite can only reach a recovery rate of 20.0% and only the minority of the recovered records are errorless.

Finally, the last category 0E shows the behaviour of the different tools in the context of databases with deleted records and overflow pages. In this type of databases, bring2lite reaches a recovery rate of 66.7%. The best recovery rate reached by its competitors is 25.0%. However, not all deleted records can be fully recovered, because of the overwritten overflow page pointer. Without the predefined evaluation setting (i.e., partly extracted records are considered, too), the recovery result of our tool decreases to 33.3%.

7.3. Final result, discussion and limitations

The last row in Table 2 shows the overall result after five categories. In all 27 files are inspected, and 278 records have to be recovered. Based on the evaluation criteria from (Nemetz et al., 2018), our tool bring2lite performs the best followed by the SQLite Deleted Records Parser and Undark:

1. bring2lite recovers 147 out of the 278 records, that is a recovery rate of 52.9%.
2. *SQLite Deleted Records Parser* extracted 139 out of the 278 records and hence reaches a recovery rate of 50.0%.
3. *Undark* recovers 95 out of the 278 records, that is a recovery rate of 34.2%.

It is important to mention that with respect to particular databases (e.g., category 0D) other tools seem to perform better. This is due to different reasons: some databases only contain text and therefore a printable string extraction performs better than our structure based approach.

Another problem that is not covered by the current version of bring2lite is as follows: if a freeblock contains more than one cell bring2lite only recovers the first record from this cell. This massively limits the recovery rate of bring2lite, because a lot of deleted records within the forensic corpus of (Nemetz et al., 2018) fall into this type of structure. By observing the worst case recovery rate of bring2lite when processing database file 0C-03, it can be seen that an improvement of the freeblock extraction method would increase the amount of restored records to 100% without the star label and hence the performance of SQLite Deleted Records Parser.

Some minor limitations concern preprocessing steps before using bring2lite and hence are relevant to the usability of bring2lite. First unlike other tools, bring2lite is not able to decrypt databases that are encrypted. The research paper by (Su and Xi, 2017) offers a solution to this problem, and it is possible to include this feature in a future release of bring2lite. A second issue is that every database has to be collected manually and copied in a dedicated folder to make bring2lite to work on these files.

More important is the current evaluation setting as proposed by Nemetz et al. (2018). For instance the tool *SQLite Deleted Records Parser* does not extract all records without an error for any of the files in category 0D. However, the actual recovery performance in terms of 'meaningfulness' (i.e., is it possible for the investigator to extract the relevant information from the recovered record) is not considered. This is the main limitation of the current evaluation methodology and must be improved in a next step.

8. Conclusion and future work

In this paper we investigated digital traces in the scope of SQLite that belong to deleted database data. In order to improve the recovery rate of deleted SQLite data we performed a structural approach and analysed the deletion behaviour of SQLite depending on different database parameters, which affect the erasure of database data. Based on the results of our structural analysis, we proposed a concept to parse and process deleted SQLite records. Our concept is implemented within our tool bring2lite.

In the field of recovering deleted SQLite records, some commercial and open source tools have already been published to cover this topic. Sample common tools evaluated in this paper are the Forensic Browser for SQLite from Sanderson Forensics (Sanderson Forensics), Undark from Paul Daniels (Daniels) or Stellar Phoenix Repair for SQLite from Stellar Data Recovery (Stellar Data Recovery). A central result is that our tool bring2lite is superiour to these tools with respect to recovering deleted SQLite data. Additionally some research papers address the deletion of SQLite records, e.g (Jeon et al., 2011) (Aouad et al., 2013) (Ramisch and Rieger, 2015) (Su and Xi, 2017). In order to show the strength of our approach, we provide an evaluation of bring2lite and competing tools with respect to a common dataset of SQLite files. The overall result is that bring2lite achieves the highest recovery rate of approximately 53% compared to its eight competitors.

An important aspect of current SQLite implementations is the use of the journal file format *Write-Ahead Log (WAL)*. In contrast to our approach the majority of both the tools and the academic papers do not include the information provided by WAL files. A further relevant aspect in the context of deleted SQLite data is the inspection of the unallocated area within the SQLite database file. As of today this aspect is only treated by one academic paper (Su and Xi, 2017) and extended by our approach.

Important future work in the scope of this paper is to test bring2lite against anti-forensic measures of SQLite as proposed by Schmitt (2018). We plan to test the robustness of our tool against the anti-forensic measures of (Schmitt, 2018). Furthermore, the processing of freeblocks needs improvement. From an investigator's point of view, our current implementation is obstructive, because relevant information may not be extracted while still being present in SQLite structures. Additionally, SQLite can hold other data types than an integer in the first column of a table.

Further future work is to improve the accuracy of processing overflow pages. If a pointer to an overflow page is overwritten with active content, this page could still be intact, but not extractable via the regular SQLite structure. Before this problem can be tackled, a research of whether the overflow page will stay within the database if the original record is deleted or not, is a prerequisite. Because these pages cannot be reached through a structure, another solution of processing these pages is needed. For this solution it is essential to search and recognise this sort of pages and parse record artifacts if they are found.

Acknowledgement

This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State

Ministry for Higher Education, Research and the Arts (HMWK) within CRISP (www.crisp-da.de).

References

Afonin, O., Katalov, V., 2016. Mobile Forensics - Advanced Investigative Strategies: Master Powerful Strategies to Acquire and Analyze Evidence from Real-Life Scenarios. Packt Publishing, Birmingham, UK.

Allen, G., Owens, M., 2010. The Definitive Guide to SQLite, second ed. Expert's voice in open source, Apress, New York.

Anglano, C., Canonico, M., Guazzone, M., 2017. Forensic analysis of telegram messenger on android smartphones. Digit. Invest. 23, 31–49. https://doi.org/10.1016/j.diin.2017.09.002. http://www.sciencedirect.com/science/article/pii/S1742287617301767.

Aouad, L.M., Kechadi, T.M., Russo, R.D., 2013. Ants road: a new tool for SQLite data recovery on android devices. In: Rogers, M., Seigfried-Spellar, K. (Eds.), ICDF2C 2012, Vol. 114 of LNICST. Springer, p. 253263.

Daniels, P.. Undark - github. https://github.com/inflex/undark.

Darwin, I.. File - linux man page. https://linux.die.net/man/1/file.

Jeon, S., Bang, J., Byun, K., Lee, S., 2011. A recovery method of deleted record for SQLite database. Personal Ubiquitous Comput. 16 (6), 707–715. https://doi.org/10.1007/s00779-011-0428-7.

SQLite documentation - file format. https://sqlite.org/fileformat2.html.

SQLite documentation — pragma page. https://www.sqlite.org/pragma.html.

Liu, Y., Xu, M., Xu, J., Zheng, N., Lin, X., 2017. SQLite forensic analysis based on wal. In: Deng, R., Weng, J., Ren, K., Yegneswaran, V. (Eds.), Security and Privacy in Communication Networks. Springer International Publishing, Cham, pp. 557–574.

McKinnon, M., 2017. Autopsy-Plugins - Github. https://github.com/markmckinnon/Autopsy-Plugins/tree/master/Parse_SQLite_Del_Records.

Meng, C., 2018. Forensic Data Analysis and Recovery of Deleted SQLite Records. Master's thesis, Hochschule Darmstadt, Germany.

Nemetz, S., Schmitt, S., Freiling, F., 2018. A standardized corpus for SQLite database forensics. Digit. Invest. 24, S121–S130. https://doi.org/10.1016/j.diin.2018.01.015.

Pereira, M.T., 2009. Forensic analysis of the Firefox 3 Internet history and recovery of deleted SQLite records. Digit. Invest. 5 (3), 93–103. https://doi.org/10.1016/j.diin.2009.01.003. http://www.sciencedirect.com/science/article/pii/S1742287609000048.

Ramisch, F., Rieger, M., 2015. Recovery of SQLite data using expired indexes. In: 2015 Ninth International Conference on IT Security Incident Management & IT Forensics. IEEE. https://doi.org/10.1109/imf.2015.11.

Rubin, P., MacKenzie, D., Kemp, S.. Dd - linux man page. https://linux.die.net/man/1/dd.

Sanderson, P., 2018. SQLite Forensics. Independently published.

Sanderson Forensics. Sanderson forensics - forensic browser for SQLite. https://sandersonforensics.com/forum/content.php?198-Forensic-Browser-for-SQLite.

Schmitt, S., 2018. Introducing anti-forensics to SQLite corpora and tool testing. In: 11th International Conference on IT Security Incident Management IT Forensics (IMF), pp. 89–106. https://doi.org/10.1109/IMF.2018.00014.

SQLite Forensic explorer. http://www.acquireforensics.com/products/sqlite-forensic-explorer/.

Statista, 2017. Number of monthly active whatsapp users worldwide. from april 2013 to december. https://www.statista.com/statistics/260819/number-of-monthly-active-whatsapp-users/.

Stellar Data Recovery. Homepage of stellar data recovery - stellar phoenix repair for SQLite. https://www.stellarinfo.com/sqlite-repair.php.

strings. Linux man page. https://linux.die.net/man/1/strings.

Su, Q., Xi, B., 2017. Key technologies for mobile phone forensics and application. In: 2017 2nd International Conference on Multimedia and Image Processing (ICMIP). IEEE. https://doi.org/10.1109/icmip.2017.15.

Christian Meng. Christian Meng (M.Sc.) is a graduate of Darmstadt University of Applied Sciences (HDA), where he specialized in the fields of information security and digital forensics. From 2016 to 2019 he teached as a lecturer in the subjects of IT security and programming at HDA. He currently works as a security consultant for usd AG.

Harald Baier. Harald Baier holds a professorship for internet security at Darmstadt University of Applied Sciences. He is a Principal Investigator at the Centre for Research in Security and Privacy Darmstadt (CRISP) and the spokesman of his university in the board of directors of CRISP. His current research interests cover digital forensics, malware detection as well as security protocols and infrastructures. He currently serves as dean in his faculty.