Journal of Cyber Forensics and Advanced Threat Investigations

Volume 1(1-3), 27-41. https://doi.org/10.46386/ijcfati.v1i1-3.17

Making the Invisible Visible – Techniques for Recovering Deleted SQLite Data Records

Dirk Pawlaszczyk[1], Christian Hummert[2]

Abstract

Forensic analysis and evidence collection for web browser activity is a recurring problem in digital investigation. It is not unusual for a suspect to cover his traces. Accordingly, the recovery of previously deleted data such as web cookies and browser history are important. Fortunately, many browsers and thousands of apps used the same database system to store their data: SQLite. Reason enough to take a closer look at this product. In this article, we follow the question of how deleted content can be made visible again in an SQLite-database. For this purpose, the technical background of the problem will be examined first. Techniques are presented with which it is possible to carve and recover deleted data records from a database on a binary level. A novel software solution called FQLite is presented that implements the proposed algorithms. The search quality, as well as the performance of the program, is tested using the standard forensic corpus. The results of a performance study are discussed, as well. The article ends with a summary and identifies further research questions.

Notes for Practice

- Carving algorithms and techniques to recover (deleted) data from SQLite databases.
- Introduction of a novel open-source software solution called FQLite.

Keywords

FQLite, SQLite, Digital Forensics, Data Recovery, Digital Traces, Free-Page List.

Submitted: 10/12/2020 — **Accepted: 27**/01/2021 — **Published: 15**/02/2021

Corresponding author [1] Email: pawlaszc@hs-mittweida.de Address: Hochschule Mittweida – University of Applied Sciences, Technikumplatz 17, D-09648, Mittweida, Germany, ORCID ID: 0000-0001-7485-7478
[2]Email: christian.hummert@zitis.bund.de Address: Central Office for Information Technology in the Security Sector (ZITiS) Zamdorfer Str. 88, Munich, 81677, Germany, ORCID ID: 0000-0002-9932-3779

1. Introduction

SQLite is a widely used, lightweight database system. The self-contained SQL database engine runs on any smartphone as well as most computers. It implements a simple, compact, and file-based database management system (DBMS), which can be accessed via a program library. No other server software is required, and the database can be easily integrated into any application. It is prevalent and used in different software products. For example, Safari, Mozilla Firefox, and Google Chrome use SQLite version 3 databases to manage user data such as history, cookies, and downloaded files (Sanderson, 2018). According to an estimate of the domain sqlite.org, there are more than one trillion database installations in active use today1. In digital forensics, the detailed analysis of such databases is of great importance. Whether a suspect deleted call-lists, chat protocol, browser history, or configuration settings: Today, this data is almost always stored in an SQLite DBMS. Many free programs support reading the database format (Tamma et al., 2018). Unfortunately, these tools only show active records. Deleted information cannot be viewed or restored with most of these products. The recovery must then often be made manually. Alternatively, someone can use commercial and costly forensic solutions.

This article will first examine how a recovery of deleted data records on a low-level is carried out in general. In this context, two algorithms for the recovery of deleted data sets within SQLite databases are presented in detail. Beyond that, a novel free, open-source tool will be presented to recover slack spaces within the database. The search quality and the performance of the solution have been tested using the standard forensic corpus. The results of this study will be discussed.

[1] http://www.sqlite.org/mostdeployed.html

2. Technical Background

Before we turn to recover deleted records, let us first discuss some basics of building and managing SQLite databases. When considering how to reclaim data that have been removed or overwritten, we first have to talk about the way SQLite organizes its dataset.

In relational databases, all data is managed in tables. A table, in turn, is composed of several pages. Similar to blocks in a file system, a page is used to store data (Haldar, 2015). It can, therefore, contain multiple records. Each page has a variable size - the default size is 4096 bytes. The maximum size for an SQLite page is 65536 bytes. The pages are numbered. The first page in the database has the number 1. It is reserved exclusively for the database header. The 100-byte header record is usually followed by a description of the database schema. To represent a table together with their pages, the system uses a balanced tree data structure (B-tree) under the hood. This is a straightforward and highly efficient way to store data (Comer, 1979). The layout of B-tree pages in SQLite is similar to the layouts used in many other relational DBMS (Wagner et al, 2015).

As usual with trees as a data structure, a distinction is made between inner nodes and leaf nodes. The special feature is that only the leaf nodes can contain data records. The inner nodes are therefore only used for linking (see Figure 1). There is a separate tree for each table. The data records are accessed exclusively via the root node, traversing to each leaf. An active page of a database must be assigned to a node. In SQLite, a total of 4 different page types are used.

The one-byte flag at offset 0 of each page indicates the type. A value of 2 (0x02) means the page is an interior index b-tree page, whereas a value of 10 (0x0a) means the page is a leaf index b-tree page. We can ignore these values in our case since we are interested in data pages only. A value of 5 (0x05) means the page is an interior table b-tree page whereas the value of 13 (0x0d) means the page is a leaf table b-tree page. The link between node and page is realized via the page number. If a page is removed, then, similarly to the deletion of a file from the filesystem, the affected page is only "marked" as deleted. More precisely, the leaf node assigned with this page must only be cut from the tree (page number is removed). If we cannot find a page number in the table tree anymore, it is considered as deleted and can be filled with new data. All pages released in this way are saved in the free-page list (see next section). By inserting new content into the database, the released areas are actually overwritten. Thus, slack space within the database can emerge. These areas are of particular forensic interest.

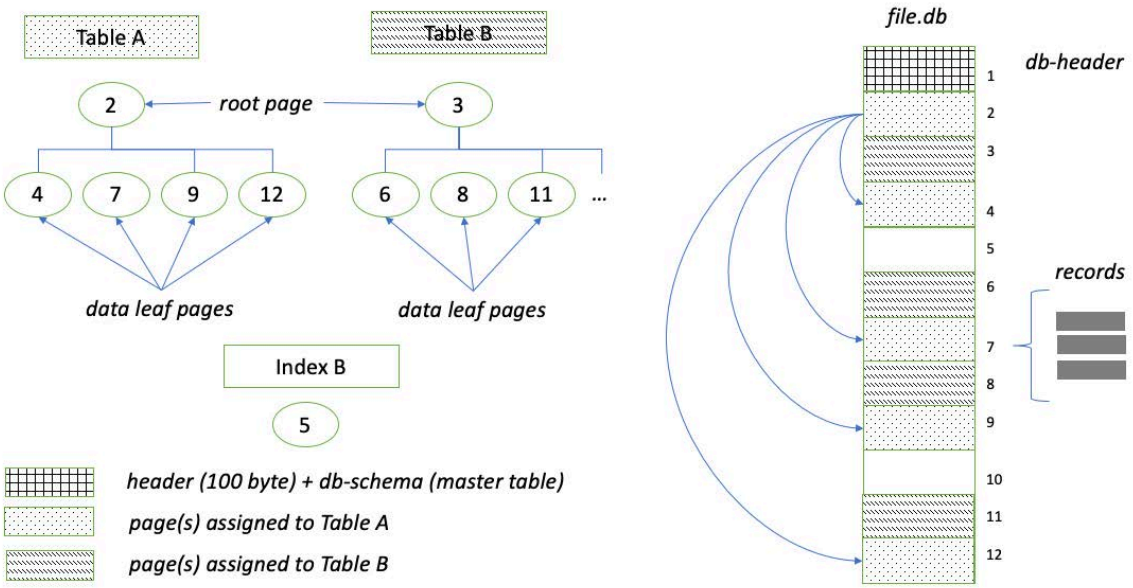

Figure 1. Structure of a SQLite3 database.

For an exact forensic examination, we further need to take a closer look at the header information of the database. The database is managed in a single file on disk only. All data records are stored in a unique binary format. Thus, we have to identify and decode the fields that are essential for our investigation. The head of each SQLite database has a well-defined structure. Each byte has a meaning. In Table 1, we can see the structure and function of each header-field within the first 40 bytes of the database file. This information is taken from the official web page of the SQLite project (sqlite.org, 2020). With this in mind, it is straightforward to identify an SQLite database. We only need to compare the beginning of the file with the header string

specified in Table 1. However, the table contains even more interesting information that can help us to find and recover deleted records. Beginning at offset 16, we can find a two-byte value that represents the page size of the database. It is a big-endian integer and must be a power of two. Unused or removed pages in the database file are normally stored on a data structure called free-page list. At offset 32, we can find a 4-byte big-endian integer, which indicates the beginning of this list. It contains the offset of the first page of the list.

Table 1. Header fields of the on-disk database file format (sqlite.org, 2020).

offset	length	field content
0	16	The header string: "SQLite format 3\000"
16	2	The database page size in bytes. Must be a power of two between 512 and 32768 inclusive, or the value 1 representing a page size of 65536.
18	1	File format write version. 1 for legacy; 2 for WAL.
19	1	File format read version. 1 for legacy; 2 for WAL.
20	1	Bytes of unused "reserved" space at the end of each page.
21	1	Maximum embedded payload fraction. Must be 64.
22	1	Minimum embedded payload fraction. Must be 32.
23	1	Leaf payload fraction. Must be 32.
24	4	File change counter.
28	4	Size of the database file in pages.
32	4	Page number of the first freelist trunk page.
36	4	Total number of freelist pages.

If the value is zero, the list is empty. The 4-byte big-endian integer at offset 36 represents the total number of entries on the free-page list. Together with the page size, we now have everything we need for an analysis of the slack spaces.

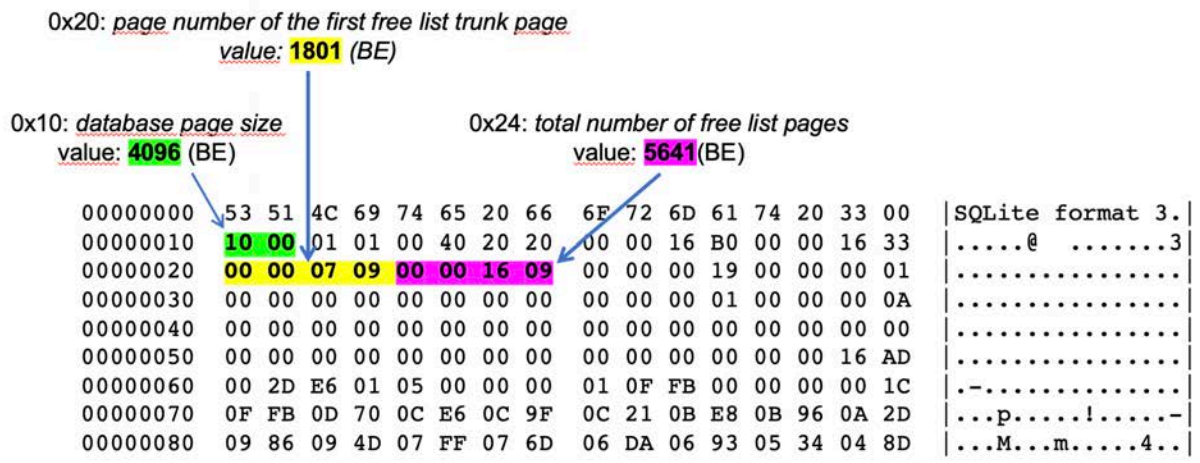

Figure 2. Example of the database header with freelist information.

3. Methods and Algorithms

1.1. Examine the free-page list

Let us first look at the case where one or more pages have been completely deleted from the database. To determine whether there are any deleted pages in the database, we have to search for the free page list (Haldar, 2015), (sqlite.org, 2020). From a technical point of view, all the database's free memory pages are linked together in a list. As already mentioned, some of these pages may contain old, actually deleted data records. Starting from the beginning of the database file, we first have to find the free page list's jump address. If the start address for the list has found, as shown in Figure 2, the entry address of the first free memory page can then easily be determined using the following equation:

$$\text{Jump to address} = ((\text{4 Byte BE in offset 32})-1) * \text{page size} \quad (1)$$

In the above example, the list, therefore, starts at the beginning:

$$\text{jump to hex} = (0x0709 - 1) * 0x1000 = 0x708000$$
$$\text{jump to decimal} = (1801 - 1) * 4096 = 7372899$$

The offset value calculated in this way leads directly to the first element in the freelist. From here, you can iterate over the subsequent entries.

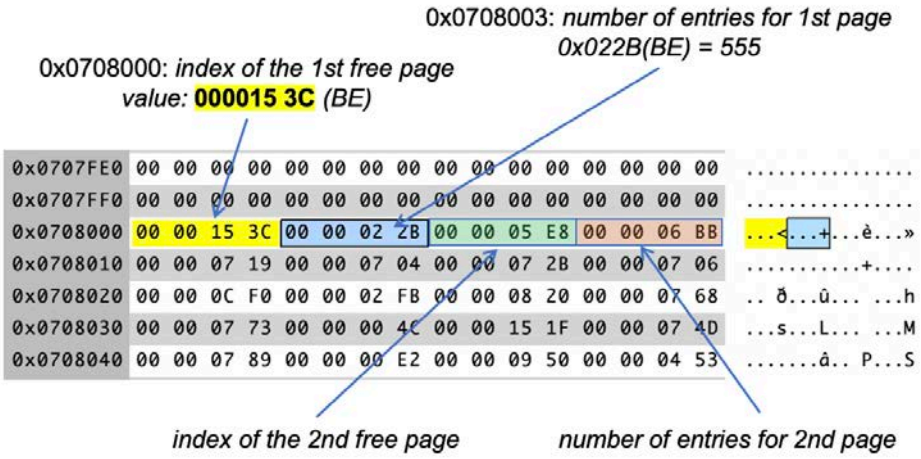

Figure 3. Start of the freepage-list

As before, this value must be multiplied by the page size to determine the start address of a particular page. Now we can move the file pointer to the first page. Each page begins with a header of between 8 and 12 bytes (see Figure 4). The b-tree page header is 8 bytes in size for leaf pages and 12 bytes for interior pages. All multibyte values in the page header are big-endian (sqlite.org, 2020). The actual cell pointer array with the offset values of the data records follows the header directly. The offset value is a two-byte integer value and is given relative to the free page's beginning. We can again iterate through the field with a loop. The number of cells can be found at offset 3 - 4. Active data records are stored in the cell content region.

It should be noted that this area behaves similarly to a stack within the memory allocation. It grows "downwards" in the direction of lower addresses (Haldar, 2015). Thus, a record added last to the page resides at a lower page address than the older entries.

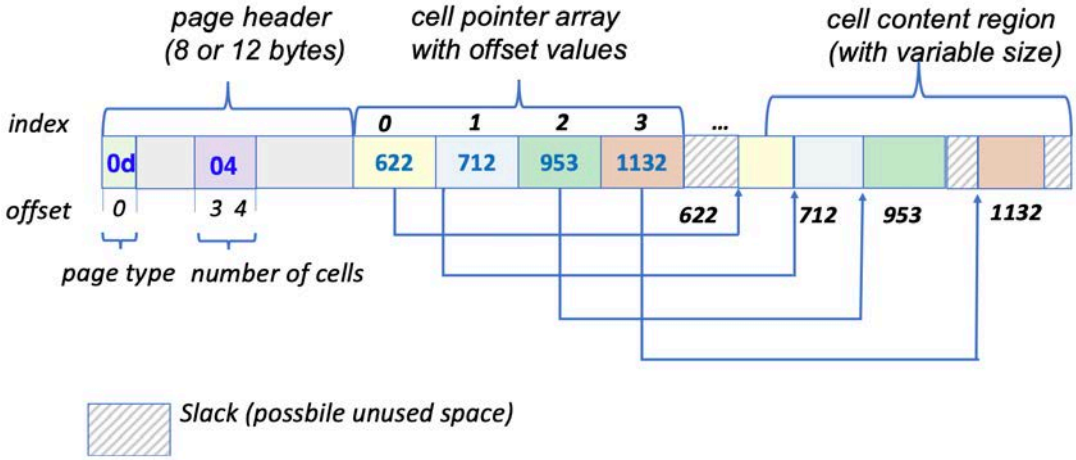

Figure 4. Structure of a data page in SQLite

When dealing with a particular record, we must analyze the structure of a record more closely. Figure 5 shows an example of a data record from a free page. In this case, it is a browser cache entry from the table *moz_bookmarks*.[2]

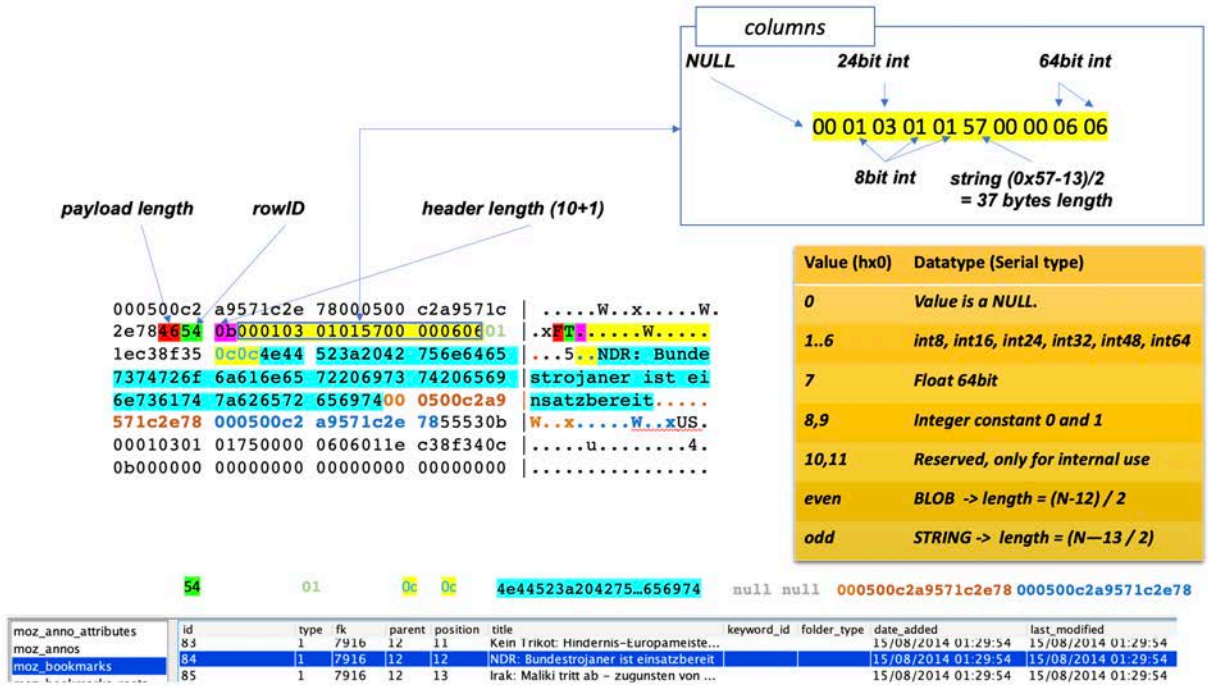

Figure 5. Structure of a data record in SQLite including serial types

The format of a cell record depends on the page the cell is located. In the above example, a leaf cell (header 0x0d) is shown. The record is split into a header and a body. The header begins with a single varint-value. SQLite makes intensive use of such base 128 encoded numbers on binary level (sqlite.org, 2020). A varint is a way of compressing down integers into a smaller space than is usually needed. By default, computers use fixed-length integers. When storing small numbers, much memory is wasted because of this behaviour. This is different for varint values. The lower 7 bits of each byte are used to encode the actual number. The most significant bit is used to indicate, whether or not more bytes are coming. In our example, the header cell has a binary value [0]0001101. The header length is the size of the header in bytes including the size varint itself. Accordingly, we have 12 (13-1) header bytes to follow. If the leftmost byte were set to one, we would have to consider one or more bytes to determine the actual length. Hence, the magnitude is the total number of bytes in the header. There are two more interesting values in the header. The first one is again a varint and holds the total number of bytes of payload. In our example, the payload has a length of 0x46 (70) bytes, the initial portion of the payload does not spill to overflow pages. This value is then followed by a field that is known as rowID. However, these are of secondary importance for the understanding of the following explanations.

The header size varint and serial type varints will usually consist of a single byte (Table 2). The serial type varints for large text and BLOBs might extend to two or more-byte varints. However, that is not the rule. The actual column values immediately follow the header. Columns with values 0,8,9,10 and 11 are NULL. The string column size can be calculated using the following formula: length = (N – 13) / 2. In our example, the string length is 37 because of (0x57-13)/2. The string terminator char \0 is not stored. It should be noted that SQLite does not have an explicit data type for representing time and date information. Very often, timestamps are represented in standard UNIX time, i.e., with a 64bit long number.

To summarize, an algorithm can be derived from what has been said so far (see Figure 6). It consists of two main loops - the first one iterates the free-page list. We go through the cell pointer array for the respective page to recover data cells within the inner-loop. Accordingly, the worst-case time complexity is O(n*m), whereas n represents the number of free pages, and m represents the maximum number of cells within a page. Since the size of a page is limited, the number of cells is also limited from above. The algorithm has linear time complexity. At this point, it should be mentioned that the SQLite Auto-VACUUM

[2] The table belongs to a Mozilla browser.

function must be disabled. Otherwise, the database would be cleaned up regularly, and the slack areas would be released to save space.

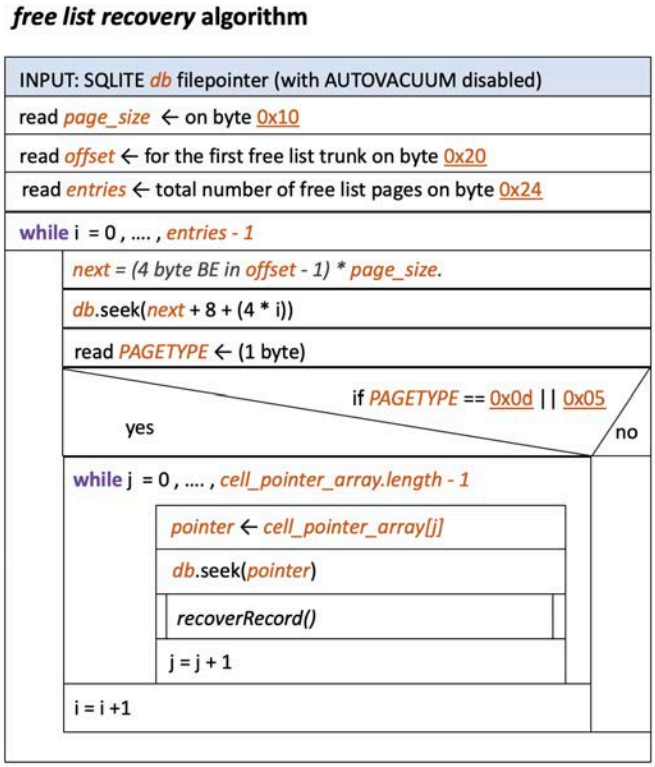

Figure 6. The freepage list recovery algorithm.

1.2. Carving for more artefacts

So far, we have tried to grab the low hanging fruits. However, what happens if no free-page list exists? Other places in the database also could contain slack space. Whenever individual records are deleted from a table, they can occur. Hence, we have to carve for the hidden records inside all data pages. Unfortunately, this results in another problem. Before we can recover those data, we first have to find the deleted records. Thus, we need to examine the slack areas more closely.

However, from the beginning: Dropping a table or deleting some rows from the database naturally creates gaps, which are replaced when new records are inserted. When a record is deleted, parts of its header are overwritten. More precisely, the first 4 bytes are filled with new information (Jeon et al, 2012). The first two bytes store the offset of the next free block within the current page. The next 2 bytes provide information about the length of the free block and thus indirectly the size of the formerly deleted data. Similar to a free page list, free blocks within a page are managed as a simple linked list. The offset of the first list element is on offset 3 and 4 in the page header. From here, we can follow the link to the next data fragment. If the value stored in the next-free block field is 0, then there are no more free blocks within this page. Unfortunately, the information on free blocks is not always reliable. Sometimes a table is deleted with a SQL-DROP statement. In this case, the specification for free blocks is overwritten too. Hence, no cell pointer tells us where to find all the deleted records.

As pointed out, a record is overwritten only partially. Since all length specifications are stored as varint values without a fixed length, sometimes more and sometimes fewer header fields are overwritten. When carving for hidden entries, we have to consider this. Not in every case, a complete recovery is possible, as we can see in Figure 7. If only the payload length bytes and rowid bytes are missing a complete recovery is always possible. A missing length specification for the header length byte is also not a big problem, since we can derive this information indirectly from the remaining header bytes. However, it becomes difficult if the length specification for the first column has been overwritten too. Sometimes it is possible to deduce the length of the first column from neighboring records. Most tables in a typical SQLite database schema are rowid tables (Sanderson, 2018), (sqlite.org, 2020). Moreover, they start with a varint key in the first column. So, we could make an educated guess in this case. If more bytes of the header get replaced, there is no chance for an automatic recovery, due to the vast of possible length values.

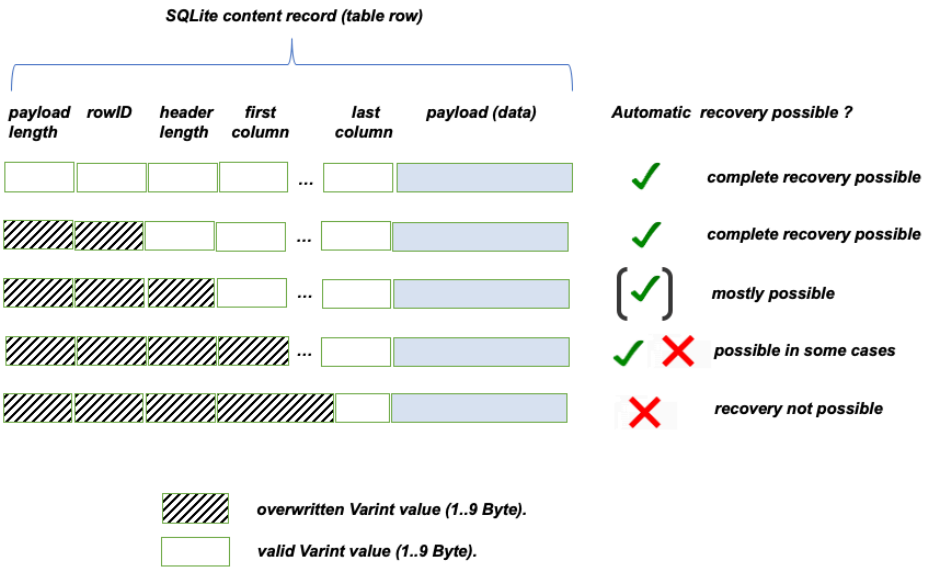

Figure 7. Deleted records and their structure.

Besides the free block areas, there is also the so-called unallocated space (Figure 8). It follows after the record header directly. Usually, this space is empty and contains only zeros by default. However, sometimes, fragments of deleted records can be found here. If, for example, the last inserted record is deleted within a page, the cell pointer is overwritten. In this case, the offset of the active content area is moved towards the next regular record. SQLite also reorganizes the contents of pages in the background to avoid free blocks from time to time. This defragmentation process also causes content to be moved to the unallocated area. Accordingly, it is much more complex to keep track of which parts of a page are allocated or free at any given time.

To summarize, all memory areas that do not contain active records and do not belong to the page header or the cell pointer region can contain potentially deleted artefacts. Nevertheless, where to begin? Fortunately, each record begins with a listing of serial types. It is situated in the record header (as described in the last section). If we first analyze the database schema, we may be able to derive some pattern from it. We will use this information to search for artefacts. It works as a signature or fingerprint.

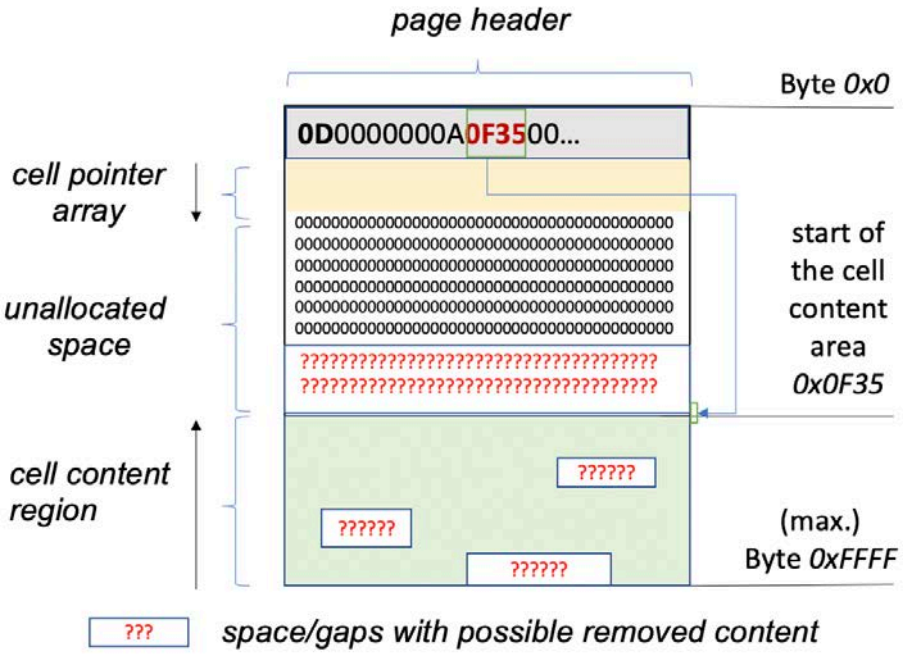

Figure 8. Possible slack for deleted artefacts.

This procedure is based on the assumption that at least most of the header information is still intact. In this way, we can perform a search in the uncharted regions of the data page. In the following, we will discuss a practical method that is not guaranteed to be perfect but sufficient to deliver good detection results in most cases. Our heuristic makes an educated guess about the header of a record. Therefore, we first analyze the database schema to derive a search pattern for each table row.

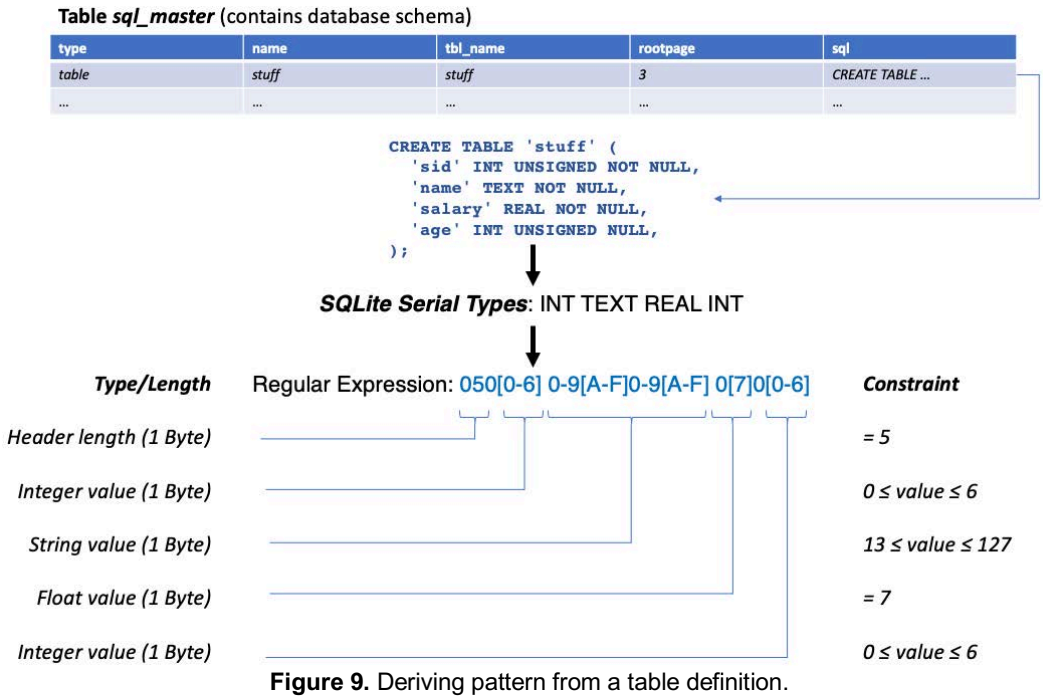

Figure 9. Deriving pattern from a table definition.

Figure 9 shows an example of a search pattern derived in this way. We will start with the header byte. It is always at the beginning of the header. The value corresponds to the number of columns bytes in the table increased by one for the length counter itself. Thus, the header specification begins with the value 0x5 (decimal '05') since the table has four columns in the above example. Integer valued columns are always mapped with a value between 0x0 (no value) or 0x6 (INT64). A floating-point number is always represented binary by the hex value 0x07 (decimal '07') and has a fixed size of 8 Bytes. Finally, yet importantly, we have a text column in the above example. Unlike numerical values, strings do not have a fixed length. In the example, we assume that the length for the name field does not exceed 77 bytes. If the length of the text column is larger, the length may be represented by two or more header bytes, since all header bytes are represented as varint values. In this case, the pattern would have to be adjusted accordingly. If this information is combined, a search pattern can be derived from it.

We repeat the pattern building step for each table definition that could be found. The actual search process is straightforward. First, the data page to be carved is converted into a hex-string. Then a pattern matching is performed, using the regular expressions created previously. Again, we use the pattern to identify the probable (header) start of a previously deleted record. In our search, we concentrate only on the slack areas. This implies that we have previously identified the regular records and the header. The FQLite-tool presented in the next section uses a Boyer-Moore algorithm for the actual search. Rather than first translating a regular expression into a deterministic finite automaton, the implementation matches the fly's regular expression. Naturally, a search based on a pattern carried out in this way can also lead to false-positive results. Therefore, each match is subsequently checked for plausibility. The test is passed if: (a) the (probably) complete header information could be reconstructed, (b) all constraints are fulfilled and (c) the computed payload length derived from the header information does fit into the actual cap (no overlapping with a regular record nor payload exceeds page size). Only if all three conditions are met, the record is restored. Since regular expressions do not support range checking for numbers, we need to define and check them with additional constraints. The constraints can be directly derived from the binary information (see Figure 7). In this way, the entire page is checked until either no further hits are found, or there are no slack areas. A flowchart of the recovery process discussed can be found in Figure 10.

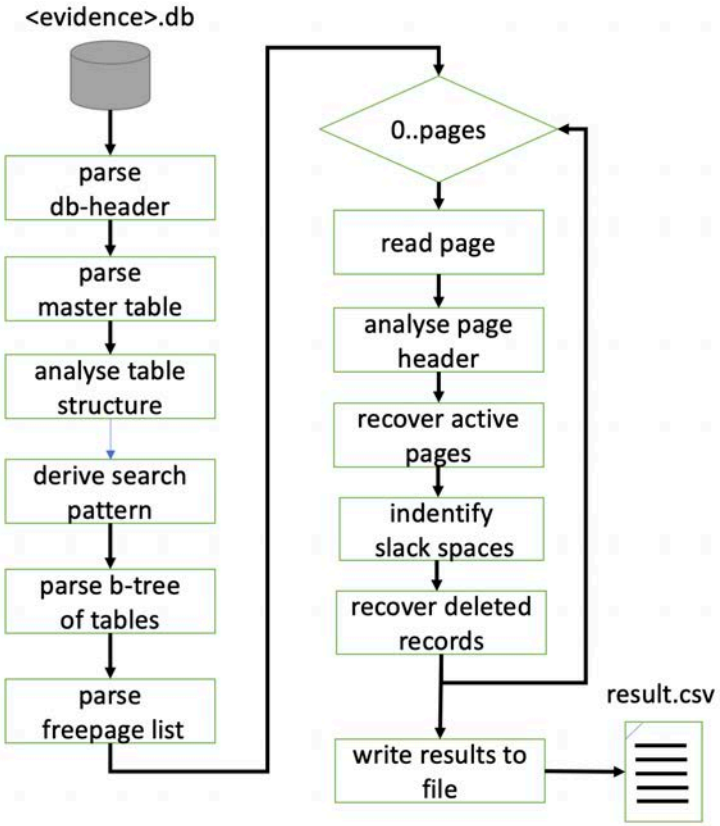

Figure 10. Flowchart of recovery process.

4. Related Work

Due to its popularity, various projects have started to gather information from SQLite databases over the last decade, in an overview of the current state of research in the field of database forensics. A good general introduction on the forensic investigation in SQLite is a book published by Paul Sanderson (Sanderson, 2018). However, it does not focus on the carving of deleted records. A somewhat older publication by Jeon et al. from 2012 already mentions important key issues when recovering data in SQLite databases. The authors use the schema table to collect information about the database structure and the data leaf pages. However, the authors overlook that by deleting tables, this information is usually lost as well. The recovery of data from the unallocated area is not discussed. Unfortunately, no concrete algorithm is presented. Only a tool called SQLiteRecover is mentioned, but it is not publicly available. In a publication from 2012, Bagley et al. discuss the recovery of data from the Firefox database (Bagley, 2012). The paper proposes an algorithm to recover deleted SQLite entries based on known internal record structures. The search for URL entries in the unallocated space is done using some simple kind of pattern matching. However, in contrast to FQLite (see next section), only strings starting with the protocol "http(s)://" are searched. A re-construction of data records other than URLs is therefore impossible.

In Aouad et al. 2013, the authors propose a method to extract deleted message artefacts on Android devices. Unfortunately, the described algorithm is only tailored to specific message types. The authors do not provide a generic solution for arbitrary SQLite databases. Neither the program nor the source code is publicly available. The recovery of index records from SQLite databases is presented in Ramisch & Rieger, 2015. The idea behind their approach is to make use of entries from index-tables to restore deleted records. Removed records sometimes can be unrecoverable from the regular SQLite table, but can still reside in the b-tree index leaf page. FQLite does not evaluate this information at the moment. The quite promising approach is unfortunately not complemented by a corresponding freely available program. A script used to recover freelist pages from a database is presented in (DeGrazia, 2013). The program is written in Python and offers a simple graphical frontend. It is part of the evaluation section. It is limited to the recovery of free pages. Thus, it is comparable to the first algorithm presented in this article.

Further carving functionalities are not offered. An example of a viral script is Undark von Paul Daniels (Daniels, 2020). The program written in C++ can only be operated from the command prompt. The database is searched from back-to-back

page by page. More detailed information about the database schema will not be evaluated. Found data records cannot be uniquely assigned to a table. In (Skulkin & Mikhaylov, 2018) the authors describe a simple method to recover data from a damaged or corrupted database. Without special tool support, they are able to save most of the data into another SQLite database. However, a search for deleted artifacts does not take place. Another interesting contribution focuses on the analysis of WAL files (Yao, 2016). This special file type is used to insert new records into an SQLite database. The authors only address the problem of extracting content from a WAL-file. Therefore, the range of applications is very limited. FQLite currently offers no support for this special file format. Thus, the contribution is a useful add-on to our work. A comparable approach can be found in (Shun, 2014). In (Pawlaszczyk, 2017) a program called MOZRT SQLite recovery tool is introduced. It was specially designed to recover data from the free-page list for browser caches of the MOZILLA browser.

A scientifically very well documented and descriptive article about carving deleted data in SQLite was written by *Meng and Baier* (Meng & Baier, 2019). The authors describe an approach that uses structural information from the database to make hidden content visible again. They focus on the inspection of the unallocated area within the SQLite database file. However, the authors evaluate the free block entries in the page header to recover deleted records. As already mentioned, these are unfortunately inaccurate in many situations and sometimes even overwritten. In contrast, our solution examines all areas that do not belong to the header or to active data sets. It should be emphasized that the authors have published all results as well as the source code. The concept is implemented within a tool called *bring2lite*. It was compared with FQLite in the evaluation chapter and was able to recover 53% of the deleted data records in the test. In our study, it thus took 2nd place.

In (Wagner, 2015) a tool is presented that can auto-detect internal DBMS storage mechanics for new databases and reconstruct the data structure and contents of known DBMS. The binary raw data of different database systems are evaluated. Their goal is to provide a generic tool that seamlessly supports many different databases. The authors verify the tool's ability to recover both deleted and partially corrupted data directly from the internal storage of different databases, including an SQLite database. Unfortunately, there is no information on where this tool can be obtained. According to the authors, it is still in an early stage of development.

In addition to the approaches presented here, there is a multitude of commercially available products. The publications of these tools usually praise the advantages of the respective tool in the recovery of data. Unfortunately, no statements are usually made about how exactly the search for artifacts works or how it was implemented algorithmically. *Sqlite Forensics Explorer*[3] or *Sanderson Forensics SQLite Forensic Toolkit*[4] to mention two examples offer a functionality comparable to the tool discussed here. *SQLabs SQLite Doctor*[5], *Stellar Phoenix Repair for SQLite*[6] as well as *SQLite Database Recovery*[7] are further examples. However, for most of the programs, a paid license must be purchased. In addition, the algorithms used internally are rarely documented. A comparison of the search quality of the addressed programs (see section Evaluation) showed that these products deliver rather mixed results with the automatic recovery of deleted content. Some programs completely failed to restore even a single deleted record.

5. The FQLite Recovery Tool

The algorithm and processing steps discussed in the last sections were implemented to check their quality and demonstrate practical use. To make the deleted information visible again, we use an App which is called FQLite. This program allows someone to recover deleted entries from an SQLite3 database. It, therefore, examines the database for entries marked as deleted. Those entries can be recovered and displayed (see Figure 11). The tool does not only recover and browse the content of freelist pages and deleted records. It also helps to create CSV-format export for the data found. It is 100% written with the Java standard class library. It runs out of the box on any platform with a Java compiler version 1.8 or higher. To ensure maximum transparency and easy verifiability of our results, the application is distributed as open-source. Accordingly, the source code is published under Mozilla Public License Version 2 and the GNU General Public License Version 3 or later. So, anybody can modify or redistribute it under the conditions of these licenses. We have opted for this way to ensure maximum transparency and traceability.

[3] https://www.sqliteviewer.org/
[4] https://sqliteforensictoolkit.com/
[5] https://sqlabs.net/sqlitedoctor_details
[6] https://www.stellarinfo.com/sqlite-repair.php
[7] https://www.systoolsgroup.com/sqlite-database-recovery.html

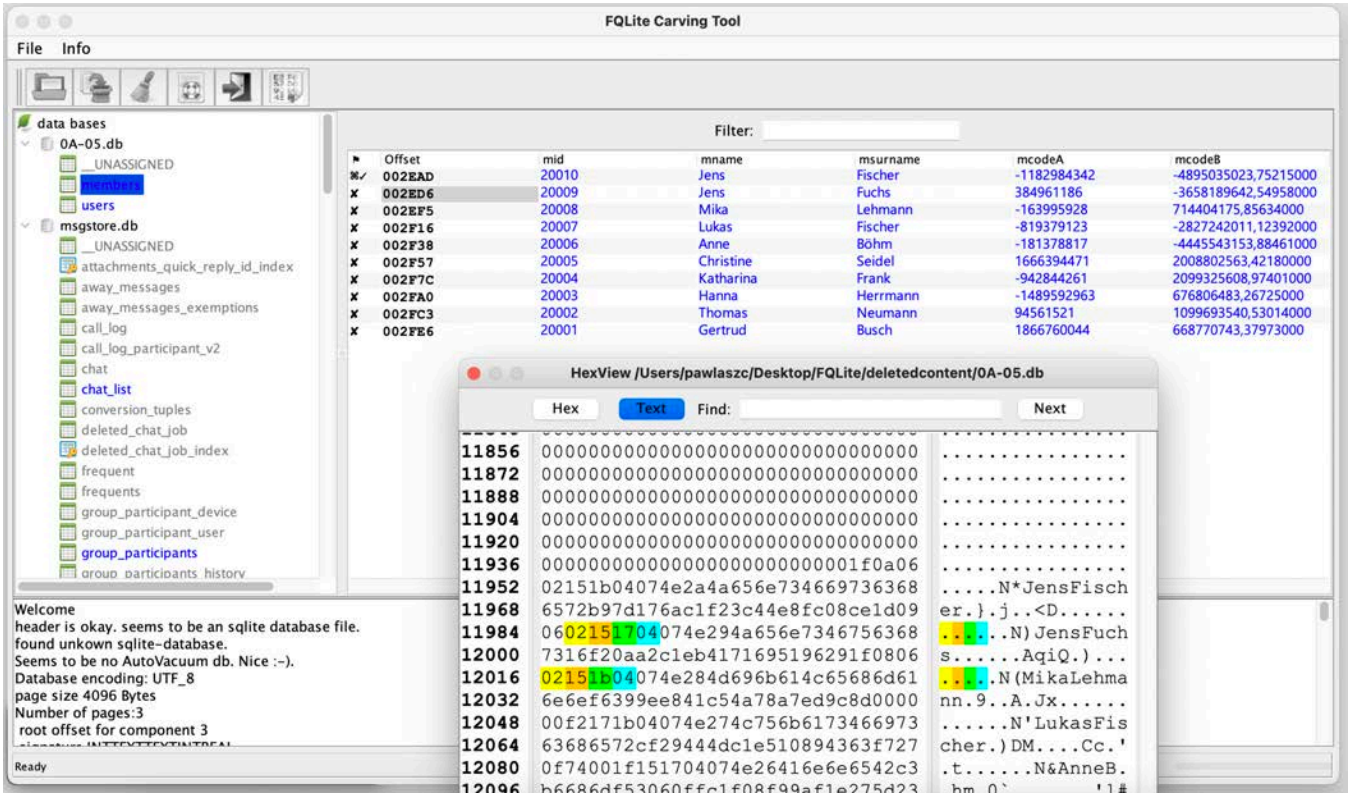

Figure 11. The FQLite Data Retrieval Tool.

The program can operate in two different modes. A simple and intuitively graphical frontend makes it possible to examine and export several SQLite databases simultaneously in GUI-mode. Found and restored records are managed via a table view. It is possible to jump to the location within the binary file at any time within a hex-view, to validate each match by hand. In the command-line interface (CLI) mode, we can automate the extraction process. This operational modus offers the highest performance and should be used for time-critical investigations. On the console, we only need to specify the input path of the database to be analyzed. The extraction process is entirely automatic.

Further user intervention is not required. In this way, the program does support batch-processing. Another essential feature is the support of multi-core processors for the data acquisition process.

6. Evaluation

In order to evaluate the quality of the proposed algorithms, various tests have been executed. First, the detection rate should be validated. This is undoubtedly the most critical indicator for the quality of a forensic tool. Since we use a heuristic, false-positive or false-negative results may occur when searching for deleted records. Mostly the latter should be avoided in a forensic investigation if possible. The standard corpus for forensic investigations was used for the experiments (Nemetz et al., 2018). This framework was designed to evaluate and benchmark forensic analysis methods and tools. It comprises 77 SQLite database files. The dataset is divided into different categories. Some test the correct representation of regular records and database structures. Others can be used to validate the detection of deleted or overwritten records.

Accordingly, the first 50 database files of the test set to focus on a special feature of SQL statements and some internal characteristics of the SQLite file format, while 27 databases feature explicitly deleted artefacts that may be recovered. The results for the first four categories (regular data and structures) are depicted in Table 2, whereas the detailed results of category V (recoverable content) are summarized in Table 3. FQLite was able to read, analyze all databases of the test set successfully. This cannot be taken for granted. A case study with the same corpus in 2018 with six different tools showed a very mixed result (Nemetz et al., 2018). None of the tested programs managed to do this. The best program was able to read 66 of 77 databases. However, 26 out of 66 were incomplete or contained errors, when looking at all categories.

The differences between the individual tools become particularly clear when comparing the deleted records' detection rate with the results achieved by FQLite in the same test (see Table 4). In this test, only the models of category 5: Deleted &

overwritten contents of the individual tools are compared. The 27 databases contain exactly 278 deleted records besides the regular ones. It is naturally of special interest to recover exactly these records for the quality of a forensic tool. The best-tested tool bring2lite manages to recon 52.9% of the deleted records. FQLite, however, was able to recover all(!) deleted records, at least in this test.

Table 2. Header fields of the on-disk database file format (Nemetz et al, 2018).

Category I: Keywords & identifiers								
File:	**01-**	**01-**	**01-**	**01-**	**01-**	**01-**	**01-**	**01-**
Result:	✓	✓	✓	✓	✓	✓	✓	✓
File: *.db	**01-09**	**01-10**	**01-11**	**01-12**	**01-13**	**01-14**	**01-15**	**01-16**
Result:	✓	✓	✓	✓	✓	✓	✓	✓
File: *.db	**01-17**	**01-18**	**02-01**	**02-02**	**02-03**	**02-04**	**02-05**	**02-06**
Result:	✓	✓	✓	✓	✓	✓	✓	✓
File: *.db	**02-07**	**03-01**	**03-02**	**03-03**	**03-04**	**03-05**		
Result:	✓	✓	✓	✓	✓	✓		
Category II: Encoding and character sets								
File: *.db	**04-01**	**04-02**	**04-03**	**04-04**	**04-05**	**04-06**		
Result:	✓	✓	✓	✓	✓ *	✓ *		
Category III: Database elements								
File: *.db	**05-01**	**05-02**	**05-03**	**05-04**	**06-01**	**06-02**	**06-03**	**06-04**
Result:	✓	✓	✓	✓	✓	(✓	✓	✓
Category IV: Tree and page structures								
File: *.db	**07-01**	**07-02**	**07-03**	**07-04**	**08-01**	**09-01**		
Result:	✓	✓	✓	✓	✓	✓		

The results of the comparison tools were taken from (Meng & Baier, 2019). We have only added the result achieved from FQLite. Of course, these test results do not allow a general statement to be made about the individual tools' search quality. Nevertheless, a tendency is recognizable. FQLite with its algorithms, performs very well and surprisingly large distances to the other programs in this test.

Table 3. FQLite results for removed content.

Category V: Deleted & overwritten contents							
0A-01 20/20	**0A-02** 20/20	**0A-03** 20/20	**0A-04** 20/20	**0A-05** 20/20	**0B-01** 10/10	**0B-02** 30/30	**0C-01** 20/20
✓	✓	✓	✓	✓	✓ *	✓	✓
0C-02 40/40	**0C-03** 20/20	**0C-04** 40/40	**0C-05** 40/40	**0C-06** 20/20	**0C-07** 40/40	**0C-08** 40/40	**0C-09** 10/10
✓	✓	✓	✓	✓	✓	✓	✓
0C-10 20/20	**0D-01** 10/10	**0D-02** 10/10	**0D-03** 15/15	**0D-04** 10/10	**0D-05** 10/10	**0D-06** 10/10	**0D-07** 25/25
✓	✓	✓	✓	✓	✓	✓	✓
0D-08 25/25	**0E-01** 17/17	**0E-02** 19/19					
✓	✓	✓					

However, there were also some problems with the acquisition results. For example, in test databases 04-05 and 04-06, all data records were recognized. The test data encoded with UTF16LE and UTF16BE contain some characters which are represented by surrogate pairs (4 instead of 2 bytes). In our test, they were not displayed correctly. In test case 06-02, a virtual

table with spatial coordinates is created. A virtual table is an object presenting an SQL table interface but which is not stored in the database file, at least not directly. Hence, not all data records could be completely restored. For this special case, a corresponding plugin is needed. Another corpus database contains a data record that is distributed over several overflow pages due to its size. In this particular case, this long data record was deleted. The program recognizes the record and can partially restore it. However, one of the overflow pages has already been overwritten after deletion and reused as an internal data page (type 0x5). Therefore, the restored data record contains overwritten data areas. However, the program itself cannot recognize that some of the data have been changed while no longer belongs to the original record. In another test case, a table was deleted (database 0B-01.db). A new table was then created with the same column types but different table and column names.

Table 4. FQLite recovery results compared to other tools
tested with the forensic corpus (results for tools 1 to 9 are taken from (Meng & Baier, 2019)).

Forensic tools 1 to 5				
Undark	SQLite Deleted Record Parser	Stellar Phoenix Repair for SQLite	SysTools SQLite Database Recovery	Sanderson Forensic Browser for SQLite
95/278 34,2%	139/278 50%	0/278 0%	0/278 0%	33/278 11,9%
Forensic tools 6 to 10				
SQLabs SQLite Doctor	Sqlite Forensic Explorer	Autopsy SQLite Deleted Records Plugin	bring2lite	FQLite
0/278 0%	73/278 26,3%	0/278 0%	147/278 52,9%	**278/278 100%**

A previously deleted page was partially overwritten with data records for the new table. As a result, there were suddenly records from two different tables in one data page. FQLite was able to recover all records - deleted and new. However, the deleted records, actual data of the old table, were erroneously assigned to the new table. We have not really found a solution to this problem. In this case, this could only be detected with the help of the SQL script that is included with every database in the test corpus. In practice, this information will unfortunately rarely be available to the investigator. On the other hand, such a constellation is not very likely to be encountered in practice.

Finally, there was a problem with the recognition of records. Instead of the expected 20 records, FQLite showed 21 recovered records. We initially assumed a false-positive hit. In fact, FQLite engine has worked perfectly correctly. One of the records was apparently moved to a new page after some data was deleted by the de-fragmentation process of SQLite. More precisely, a copy was created. Thus, the record was suddenly twice in the database in two different pages. Apart from these exceptions, all other databases of the corpus could be restored correctly. A very encouraging result.

Since forensic investigations are time-critical, the duration of the extraction process is also a crucial measure. In the second series of tests, the possibility of parallelizing the data acquisition process was tested. It is reasonable to assume that a significant speedup can be achieved when carving for artefacts concurrently. If we analyze a database page, there are usually no dependencies to other pages within the database.

An exception is the so-called overflow pages, which are used when there is not enough space in a page to store a data set. The SQLite file format states that overflow pages are chained together using a singly linked list. To answer this question, we have used a slightly adapted model of the *chinook.db* database[8]. Since there is no suitable benchmark for such questions at the moment, we used an example that is frequently used for training situations. The database in our test contained about 2,048 million entries. For us, it was primarily interesting to see how the execution time is reduced when we perform our analysis in parallel on several processor cores.

[8] https://www.sqlitetutorial.net/sqlite-sample-database/

The test system used was a MacBook Pro 2017 with a 2.8GHz Quad-Core Intel Core i7, 16GB RAM, and a 256GB Apple SSD. The system has a number of 4 physical cores. A total of more than 100 test runs were conducted. The results were used to calculate the average time required. The tests show a moderate relative speedup of 2,36 when scaling the system from 1 to 4 cores. The tests were performed with version 1.0 of the program. In order to better understand what the application requires the most computing time for, a profiler was used. Surprisingly, only about 30% of the execution time was used for the actual search process. The largest part of the computing time (around 63%) was needed to decode the recovered records and concatenating them into UTF8 strings.

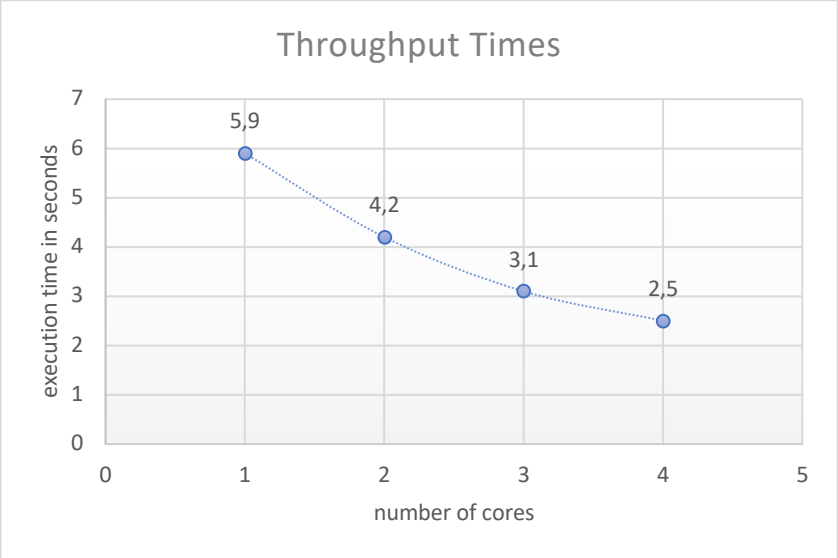

Figure 12. Obtained Throughput Times.

7. Conclusions

Database forensics is an upcoming research discipline over the last decade (Chopade & Pachghare, 2019). The use of carving techniques in the field of databases, for many years, an established method in digital forensics, is a relatively new approach. In this article, we presented some techniques for carving and acquisition of deleted data in SQLite databases. Besides an algorithm for the recovery of deleted data pages, a novel heuristic for detecting deleted records on a binary level within the database's slack areas was introduced. As proof of concept, an open-source application named FQLite was presented, which implements all proposed techniques. The benchmark scope has proved that the program can handle almost all pitfalls of the SQLite data format. In particular, FQLite performs better in this field than many other forensic solutions currently on the market. It could achieve the highest recovery rate within our test. However, it must be emphasized that the application is explicitly not meant to be a complete forensic tool. The range of functions is limited to searching for (deleted) data records and their export into a CSV format. Functions like picture extraction or case handling are not implemented at the moment. Nevertheless, the results achieved are very encouraging.

Planned future improvements are related to an error-free display of UTF16 encoded data. Full support of virtual tables is also planned for one of the next program versions. As a new feature, support for WAL files will be made available in addition to the standard database files. Beyond that, we are working on a further improvement of the algorithm's recognition quality and further speed up the acquisition process. Finally, we plan to harden our tool against anti-forensic measures like those described by Schmitt (Schmitt, 2018).

Declaration of Conflicting Interest

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The project in part has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 832800.

References

Aouad, L.M., Kechadi, T.M., Russ, R.D.: Ants road (2012). A new tool for SQLite data recovery on android devices. In: M. Rogers, K. Seigfried-Spellar (Eds.), ICDF2C 2012, Vol. 114 of LNICST, Springer, pages 253-263. doi: 10.1007/978-3-642-39891-9_16

Bagley, R., Ferguson, R. I., Leimich, P. (2012). On the digital forensic analysis of the Firefox browser via recovery of SQLite artifacts from unallocated space. 6th International Conference on Cybercrime Forensics Education & Training (CFET).

Chopade, R., Pachghare, V.K. (2019). Ten years of critical review on database forensics research. In: Digital Investigation Volume 29, pages 180-197. doi: 10.1016/j.diin.2019.04.001

Comer, D.(1979). Ubiquitous b-tree. ACM Comput Surv,11 pp. 121-137, doi: 10.1145/356770.356776

Daniels,PL (2020). Undark - a SQLite deleted and corrupted data recovery tool. project homepage. http://pldaniels.com/undark/ (last accessed 03/03/2020)

DeGrazia, M. (2013). Python Parser to Recover Deleted SQLite Database Data. URL: http://az4n6.blogspot.be/2013/11/python-parser-to-recover-deleted-sqlite.html

Haldar, S. (2015). SQLite Database System Design and Implementation (Second Edition). pages 256 (2015).

Jeon, S., Bang, J., Byun, K., Sangjjn, L. (2012). A recovery method of deleted record for SQLite database. Pers Ubiquit Comput 16, 707–715. doi: 10.1007/s00779-011-0428-7

Liu, Y., Xu, M., Xu, J., Zheng, N., Lin, X. (2016). SQLite Forensic Analysis Based on WAL. In: Security and Privacy in Communication Networks 12th International Conference, SecureComm 2016, Guang-zhou, China, 2016, Proceedings

Meng, C., Baier, H. (2019). bring2lite: A Structural Concept and Tool for Forensic Data Analysis and Recovery of Deleted SQLite Records. Digital Investigation: Volume 29, Supplement, July 2019, pages 31-41, (2019). doi: 10.1016/j.diin.2019.04.017

Nemetz, S., Schmitt, S., Freiling, F. (2018). A standardized corpus for SQLite database forensics. In: Digital Investigation, vol. 24, Supplement, 2018, pages 121-130. doi: 10.1016/j.diin.2018.01.015

Pawlaszczyk, D. (2017). Digitaler Tatort, Sicherung und Verfolgung digitaler Spuren. In: Labudde D., Spranger M. (eds) Forensik in der digitalen Welt. Spring. doi: 10.1007/978-3-662-53801-2_5

Ramisch, F., Rieger, R. (2015). Recovery of SQLite Data Using Expired Indexes. IMF '15: Proceedings of the 2015 Ninth International Conference on IT Security Incident Management & IT Forensics 2015 pages 19–25. doi: 10.1109/IMF.2015.11

Sanderson, P. (2018). SQLite Forensics. Independently published, ISBN 978-1980293071, 315 pages (2018).

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

ShuN., W., Zheng, M. Xu (2014). A history records recovering method based on WAL file of firefox, In: Journal of Computational Information Systems 10(20):8973-8982, 2014. doi: 10.12733/jcis12145

Skulkin, O., Mikhaylov, V.K. (2018). Forensic Analysis of Damaged SQLite Databases, forensic focus.com, March 2018.

sqlite.org: Database File Format (2020). Official Webpage https://www.sqlite.org/fileformat.html. (last access 08.10.2020).

Tamma, R., Skulkin, O., Mahalik, H., Bommisetty, S. (2018). Recovering deleted SQLite records. In: Practical Mobile Forensics - Third Edition, pages 176-189 (2018).

Wagner, J., Rasina, A., Grier, J. (2015). Database forensic analysis through internal structure carving. Digital Investigation. Volume 14, Supplement 1, August 2015, pages 106-S115 (2015). doi: 10.1016/j.diin.2015.05.013