FOX-IT

Extracting SQLite records

Carving, parsing and matching

22 July 2011

Ivo Pooters
pooters@fox-it.com

Pascal Arends
arends@fox-it.com

Steffen Moorrees
Moorrees@fox-it.com

1 Android data structures

The Android operating system stores most of the application data and operating system data in Sqlite databases. Application data may consist of configuration settings and user data. Applications may access and store the information through so-called content providers. These content providers provide an interface to applications and regulate where data is stored and which data can be accessed. Under the hood Android stores application data in the application directories on the user partition. Each application has its own space on the user partition in the folder *data/<com.app.name.here>*. The Sqlite databases are stored in a sub folder called *databases.* Often (but not always!) the databases can be recognized by the file extension *.db*.

For a listing of the relevant databases and other files on an Android device, the reader is referred to the document about Android and YAFFS2.

2 About SQLite

This section describes the SQLite file format and workings to the extent that it is relevant for carving SQLite records. We acquired this information from the references listed in section 4. The user is referred to these sources for more information about SQLite.

An SQLite database file consists of one or more pages. The size of the pages is a power of 2 between 512 en 65536 bytes. The actual page size is denoted in the header of the database file.

Each page is of one of the following types:

- The lock-byte page
- Freelist page
  - Trunk page
  - Leaf page
- B-tree page
  - Table interior page
  - Table leaf page
  - Index interior page
  - Index leaf page
- Payload overflow page
- Pointer map page

Writes to the database are always an integer multiple of the page size. Reads are also an integer multiple of the page size with the exception of a 100 byte read of the database header.

The first 100 bytes of the database file are used for the database header. The first 16 bytes of the header contains the magic-string “SQLite format 3” including the null terminator at the end.

Sqlite has the ability to reserve a small number of bytes at the end of each page for use by extension programs. This is called the *reserved space* or *reserved region* size. The size is stored as a 1-byte integer at offset 20 in the database header. Usually this value is 0.

The lock-byte page, freelist pages and pointer map pages will not further be described, since these are not relevant for carving SQLite records.

2.1 Binary tree pages

Sqlite uses a binary tree layout of pages to store indices and table content. Indices are stored in the index b-tree pages and table content is stored using the table b-tree pages. Each table in an SQLite schema is represented by exactly one table b-tree. Each index in an SQLite schema is represented by exactly on index b-tree.

The index b-tree pages and interior table b-tree pages are not relevant when carving for SQLite records, so they will not be further discussed here. For a full understanding of these pages, the reader is referred to the sources listed in section 4.

A b-tree consists of one root b-tree page, interior b-tree pages and leaf b-tree pages. A root is actually an interior b-tree page without a parent b-tree page. Interior b-tree pages hold keys and pointers to child b-tree pages. Leaf b-tree pages hold keys and the corresponding content. An SQLite program can find the value for a certain key by following the pointers from the root page down until it encounters a leaf page. For table b-tree pages, the keys represent the row ids of the table rows.

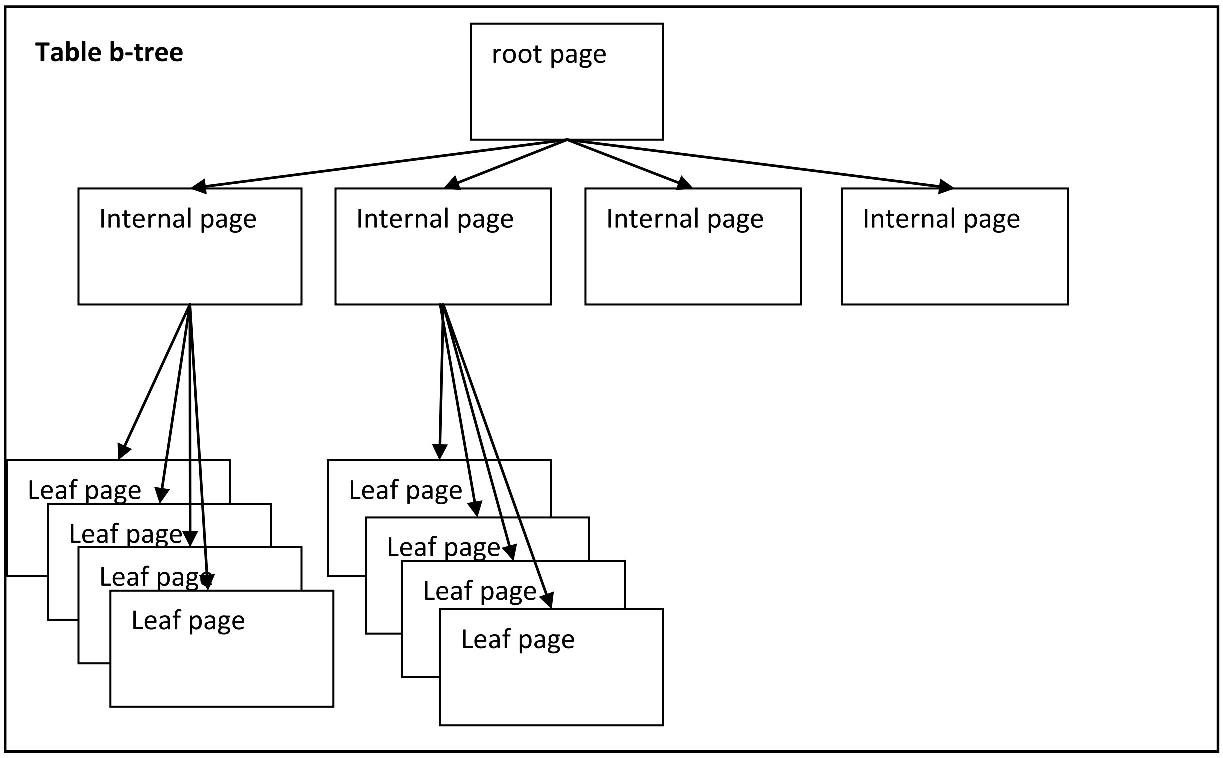

Figure 1. B-tree page hierarchy

Keys and pointers (interior pages) or keys and content (table leaf b-tree pages) are stored in 'cells'. The arbitrary length section of a cell is called the 'payload'. For index b-tree pages, the payload is the key. For interior table b-tree pages, there is no payload and for the leaf table b-tree pages, the payload is the content.

2.1.1 B-tree page format

A b-tree page (interior or leaf) is divided into the following regions:

1. The 100-byte database file header (only on page 1)
2. The 8 (leaf) or 12 byte (interior) b-tree page header
3. The cell pointer array
4. Unallocated space
5. The cell content area
6. The reserved region (usually zero)

This is depicted in Figure 2.

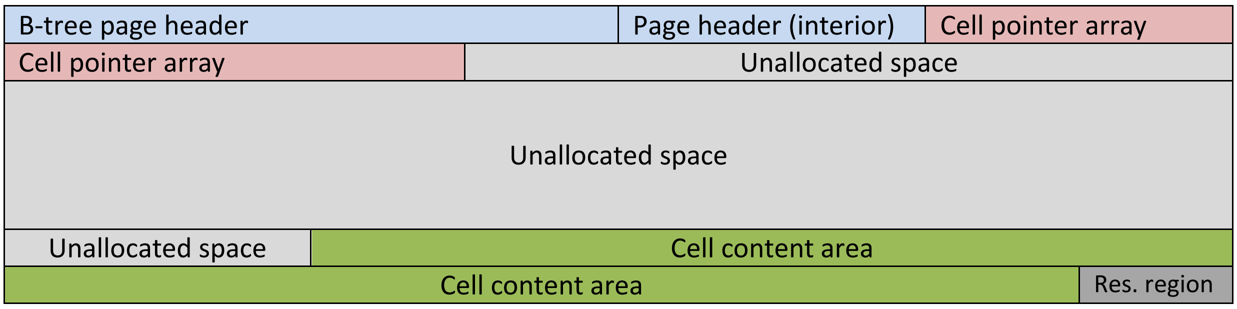

Figure 2. Sqlite b-tree page layout

The cell pointer array contains pointers to the actual cells in the b-tree page. Each pointer is a 2-byte integer which contains the offset in the b-tree page to the first byte of the cell. The cell pointers are arranged in key order.

The cell content region contains the actual cells. Sqlite aims to place the cells as far to the end of the page as possible to reserve space for the cell pointer array. The area between the last cell pointer and the first cell is the unallocated space.

2.1.2 B-tree page header

The b-tree page header is formatted as follows:

1. Byte 0: A flag indicating the type of b-tree page: 0x02 indicates interior index page, 0x05 indicates interior table page, 0x0A indicates a leaf index page, 0x0D indicates a leaf table page.
2. Bytes 1-2: Byte offset into first freeblock.
3. Bytes 3-4: Number of cells on page.
4. Bytes 5-6: Offset into first byte of the cell content area.
5. Byte 7: Number of fragmented free bytes within cell area content
6. Bytes 8-11: Right most pointer (interior b-tree pages only).

A freeblock is a structure used to identify unallocated space in a b-tree page inside the cell content area. Freeblocks are chained by a 2-byte integer pointer at the start of every freeblock. The integer represents the offset into the page to the next freeblock, or zero if it is the last freeblock. The third and fourth byte of a freeblock represent the length of the freeblock including the 4-byte header. In a well-formed b-tree page there will be at least one cell before the first freeblock (otherwise, it should be part of the unallocated region).

Fragmented bytes are groups of 1, 2 or 3 bytes of unallocated space in the cell content area. If the group is 4 bytes or more, it is a freeblock.

2.2 Leaf table binary tree page

The actual content of Sqlite tables is stored in the leaf table b-tree pages.

For a leaf table b-tree page, the page header is depicted in Figure 3.

0x0D	Offset 1st freeblock	Number of cells	Offset cell content	Num freebytes

Figure 3. Leaf table b-tree page header

Following the page header is the cell pointer array containing pointers to the cells in the cell content area.

Each Sqlite table record is stored in one cell. The table b-tree leaf cell has the following format:

1. A varint representing the total number of bytes of payload, including any overflow.
2. A varint which is the integer key, or row id.
3. The first portion of the payload that does not spill to overflow pages.
4. A 4-byte big-endian integer representing the page number of the first overflow page. This value is only present if payload is spilled to overflow pages.

Of course, regularly the table record content is too large to fit into one leaf page. Say, when a table record stores a text of 2000 characters and the database page size is 1024 bytes. The single record is too big to fit into a leaf (even if it would be the only record in the leaf page). In such a case, the content is cut into parts such that the first part fills the available payload space in the leaf page and the subsequent parts fit into overflow pages. If the payload size is less than U-34 then the whole payload is stored in the leaf page. Where U is the total page size minus the reserved region size.

Sqlite overflow pages are chained by a pointer stored in the first four bytes of each overflow page. This four byte big-endian integer represents the page number of the next page in the chain or zero in case of the last overflow page. Each overflow page is dedicated to exactly one record.

2.2.1 Varint

A variable-length integer or "varint" is a static Huffman encoding of 64-bit twos-complement integers that uses less space for small positive values. A varint is between 1 and 9 bytes in length. The varint consists of either zero or more bytes which have the high-order bit set followed by a single byte with the high-order bit clear, or nine bytes, whichever is shorter. The lower seven bits of each of the first eight bytes and all 8 bits of the ninth byte are used to reconstruct the 64-bit twos-complement integer. Varints are big-endian: bits taken from the earlier byte of the varint are the more significant and bits taken from the later bytes. Listing 1 provides an algorithm outline for parsing varints.

```
pos = 0
value = 0
while  pos < 9 and not complete:
   byte = varint[pos];
   if MSBset(byte) and pos < 8:
        value.appendbits(byte[1:])
   else if MSBset(byte) and pos == 8:
        value.appendbits(byte)
        set complete
    else:
         value.appendbits(byte[1:])
        set complete
    pos ++
if not complete:
    error!
```

Listing 1. Outline algorithm for parsing varint

2.2.2 Cell and payload format

The cell format for a cell in a leaf table b-tree page is depicted below:

Payload size (varint)	Row id (varint)	Payload	Overfl page num (4-b int)

The payload is in *record format*. The record format defines a sequence of values corresponding to columns in a table. Each record describes both the serial type of the columns as well as the value of the columns.

A record contains a header and a body in that sequence. The header begins with a varint representing the size of the header in bytes (including the varint itself). Following are one or more varints representing the column type for each column in the SQLite table. So, one varint per column. These varints are called *serial-type* numbers.

The serial type numbers and their meaning are listed in Table 1.

Serial type	Content size	Meaning	Our reference
0	0	NULL	ST_NULL
1	1	8-bit twos-complement integer	ST_INT8
2	2	Big-endian 16-bit twos-complement integer	ST_INT16
3	3	Big-endian 24-bit twos-complement integer	ST_INT24
4	4	Big-endian 32-bit twos-complement integer	ST_INT32
5	6	Big-endian 48-bit twos-complement integer	ST_INT48
6	8	Big-endian 64-bit twos-complement integer	ST_INT64
7	8	Big-endian IEEE 754-2008 64-bit floating point number	ST_FLOAT
8	0	Integer constant 0. Only available for schema format 4 and higher.	ST_C0
9	0	Integer constant 1. Only available for schema format 4 and higher.	ST_C1
10,11		*Not used. Reserved for expansion.*	
N≥12 and even	(N-12)/2	A BLOB that is (N-12)/2 bytes in length	ST_BLOB
N≥13 and odd	(N-13)/2	A string in the database encoding and (N-13)/2 bytes in length. The nul terminator is omitted.	ST_TEXT

Table 1. SQLite serial types

Note that for serial types 0, 8, 9, 12, 13 the value is zero bytes in length. Whenever a value for a certain column is not set, then the values is stored as serial-type ST_NULL and no bytes are used for the value.

The record format for a leaf table b-tree page is depicted in Figure 4.

Header					Body			
Header size (varint)	Stype col 1 (varint)	Stype col 1 (varint)	…	Stype col n (varint)	Value col 1	Value col 2	…	Value Col n

Figure 4. SQLite record format

3 Methodology

The goal is to extract and examine table records of SQLite databases from a file system image independent of the file system type. This problem can be subdivided into three sub problems:

1. Identifying SQLite table records
2. Parsing SQLite table records
3. Identifying the table and database to which the SQLite records (used to) belong

The carving method described in this document has been implemented in a python program named *extractAndroidData.py*.

3.1 Identifying SQLite table records

SQLite table rows are stored in the record format (we will refer to these as records) in cells in table b-tree leaf pages. It is difficult to identify the cells or records from a table b-tree leaf page directly, as there are no identifying header bytes. However, it is possible to identify table b-tree leaf pages with high probability. Table b-tree leaf pages can be recognized by analyzing the first 8 bytes of the SQLite page. To do this, one must first know the SQLite page size of the database(s). The page size is stored in the database file header at offset 20. Usually when carving for database records, the database file header is not available. For our method we assume the SQLite page size is known beforehand from reference testing and it is smaller or equal to the file system page size. We define the page size as *PS*. Sqlite pages are aligned with file system pages, so it is trivial to determine where the SQLite pages page headers may be located.

3.1.1 Identifying table b-tree leaf pages

A chunk of size *PS* is identified as SQLite table b-tree leaf page when:

1. Byte 0 is equal to 0x0D and
2. (offset to 1st freeblock) Byte 1,2 is an integer between 8 and *PS* and
3. (number of cells) Byte 3,4 is an integer between 1 and *PS/5* and
4. (offset to cell content area) Byte 5,6 is an integer between 8 and *PS*. or equal to 0

Ad 2. One could assume well-formedness of the b-tree page and say that the offset to the 1st freeblock should be larger than the value of byte 5,6 (the offset to the cell content area) or 0. For performance this did not appear necessary, so we chose not to make it more strict than necessary.

Ad 3. We can assume that the minimum size of a record is 5 bytes. So the number of cells in a page should be no more than *PS/5*.

This heuristic is not bullet-proof, there will be cases where the chunk passes the above-mentioned criteria, but is not a SQLite table b-tree leaf page. One should take this into account when parsing the page.

3.1.2 Carving cells from the pages

Once the table b-tree leaf pages are identified the cell pointer map of each page can be read to determine the location of the cells in the page.

From the cell offset, the first two varints can be parsed and decoded to the payload size and row id respectively. If the payload size is less than or equal to *usableSize* – 35, then we can assume that the entire record is in this leaf page and pass the payload to the parser as such.

If the payload size is larger than *usableSize-35,* The following formula calculates how many bytes are spilled to overflow pages (from the SQLite specification):

$$\begin{aligned}
&\mathit{usableSize} = \mathit{total\ page\ size} - \mathit{reserved\ region\ size}\\
&\mathit{minLocal} = ((\mathit{usable\ size} - 12) * 32/255) - 23\\
&\mathit{maxLocal} = \mathit{usable\ size} - 35\\
&\mathit{localSize} = \mathit{minLocal} + ((\mathit{payloadSize} - \mathit{minLocal})\ \%\ (\mathit{usableSize} - 4))
\end{aligned}$$

If *localSize* is larger than *maxLocal* then *minLocal* bytes are stored in this leaf page and the rest is spilled. Else, *localSize* bytes are stored in this leaf page and the rest is spilled.

Note that this is different from the official SQLite specification which states that *maxLocal* bytes are stored on the leaf page when *localSize* is larger than *maxLocal*. However we observed that this is not the case on the image we investigated in the DFRWS2011 challenge. We did not test this for other SQLite databases.

Further research is required to find a method to track down the overflow pages belonging to a certain cell. Due to time constraints we have not researched this and do not include the data in the overflow pages.

3.2 Parsing SQLite table records

When all SQLite table cells have been identified and extracted, it is time to parse the cells into meaningful information. During the parsing process the program should allow for the possibility that the page was not a table b-tree leaf page after all. Any unexpected parsing errors could indicate that it is not such a page. Analyzing the payload portion is done from left to right:

1. Determine the payload header size from the first varint.
2. For each column determine the serial type until the end of header is reached.
3. For each encountered serial type, determine the expected byte length as *n* of the value and parse *n* bytes as the value for that column. The parser should take into account that the actual available bytes may be less than the payload size determined by the cell in case of overflow payload. If the payload is spilled and cannot be recovered, a dummy string is stored as the column value.

The result is a list of recovered records. A recovered record stores the column serial types and column values and the row id.

3.3 Identifying the database and table

For a given SQLite database the database header page and other pages will likely be stored on spread-out locations in a file system. Without knowledge of the file structure, it is very difficult to relate one database page to another page of the same database. As a consequence the records cannot directly be related to pages containing the database schemas.

Before reading on, the reader should understand the difference between SQLite serial types and SQLite data types. SQLite uses serial types to indicate the type of data stored on record level (i.e. how they are actually stored on disk). SQLite uses data types to indicate the type as defined in a table schema. A mapping exists to map SQLite data types to SQLite serial types and is listed in Table 2. Note that the mapping is not one-to-one. Values of any data type which are not set, will be stored as ST_NULL serial type.

It is assumed that the schema of the SQLite database and its table structures are known beforehand when carving for its records. The schema can easily be acquired by examining reference databases of the same type. This knowledge can be used to create templates for relevant tables in the database. A template is a list of SQLite data types. SQLite data types can be any of *TEXT, INTEGER, REAL, BLOB* or *NULL*. Where NULL is not actually a SQLite data type, but a catchall data type defined by us to match NULL serial types.

data type	**Serial type**
NULL	ST_NULL
INTEGER	ST_INT8 – ST_INT64 ST_NULL
REAL	ST_FLOAT, ST_INT ST_NULL
BLOB	ST_BLOB, ST_NULL
TEXT	ST_TEXT, ST_NULL

Table 2. SQLite data type map

3.3.1 Defining templates

When defining a template one should take into account columns which may contain a NULL value. For example, a table with the following columns:

Column name	**data type**	**Nullable**
_id (prim key)	INTEGER	
Name	TEXT	
Protocol	INTEGER	
Fraction	REAL	no

The _id column is the primary key or row id. This means that SQLite will store the value as a NULL value in the payload. The actual value is stored as the row id in the cell. The Name column may be empty (NULL) when no value is set. The protocol value may also be NULL when no value is set. The fraction value will never be NULL, because the schema defines this column as not nullable. SQLite may store a REAL value as a ST_FLOAT serial type, one of the integer serial types (when there is no fractional part) or ST_NULL (when no value is set).

With this information the following template can be defined for this table:

Col #	SQLite data type
1	NULL
2	TEXT or NULL
3	INT or NULL
4	REAL or INT

When defining a template, the following aspects determine the data types per column:

- Does the schema define whether the column is nullable?
- Does the schema define a default value for the column?
- According to the schema, is the data type REAL? Then the serial type may be mapped to data type INT or REAL.

3.3.2 Effectiveness

The effectiveness of the template matching scheme is influenced by the following aspects:

- The number of columns in a table schema. A record from table A with 10 columns can be more easily matched to table A than a record from table B with 2 columns. There may exist many more tables with 2 columns with the same column types.
- The possibility of NULL values in columns. When a column can have a null data value (i.e. no value needs to be set) then the template will be less strict. For example, a record from a table with columns [INT (p. key), TEXT, TEXT, BLOB] where all values may be NULL can be interpreted as a record from a table with columns [INT (p. key), INT, INT, BLOB] where all values may be NULL.

The effectiveness can be increased when more is known about the values of columns. The templates could be extended by defining for each column a range of possible values. For example, consider the table:

Column name	data type	Nullable
_id (prim key)	INTEGER	
Username	TEXT	No
Password	TEXT	No
Host	TEXT	No

Knowing that the password is stored in clear text and the host is stored as an IP-address, the extended template could be defined as:

Col #	SQLite data type	Possible values
1	NULL	All integer values
2	TEXT	1 < size < 25
3	TEXT	1 < size < 25
4	TEXT	7 < size < 16 and regular expression [0-9\.]*

Due to time constraints this is not implemented in the program.

4 References

http://www.ssddfj.org/papers/SSDDFJ_V4_1_Lessard_Kessler.pdf

http://www.sqlite.org/fileformat.html

http://www.sqlite.org/fileformat2.html

http://forensicsfromthesausagefactory.blogspot.com/2011/04/carving-sqlite-databases-from.html

http://forensicsfromthesausagefactory.blogspot.com/2011_05_01_archive.html

http://forensicsfromthesausagefactory.blogspot.com/2011/05/sqlite-pointer-maps-pages.html

http://forensicsfromthesausagefactory.blogspot.com/2011/07/sqlite-overflow-pages-and-other-loose.html