SQLite Forensics:
Exposition on the SQLite File Format and Techniques for Data Carving

ST2602: Computer Forensics
Singapore Polytechnic

Ku Wee Kiat (P1030284)
Jeremy Heng (P1000720)
DISM 3B21/22

November 23, 2012

Abstract

The paper will study the SQLite File Specification and propose techniques for fingerprinting and identifying SQLite files in non-coherent data and the recovery of stored information from a SQLite database. It explains how a database file is laid out on the disk and the various physical and logical structures that comprise a SQLite database. In addition, we put forward reasoning for the value of the paper's propositions with regard to its importance in computer forensics. Python scripts accompany this paper as proof-of-concept pieces for the rough implementation of the techniques presented.

Contents

1 Introduction

1.1 Overview

The paper will study the SQLite File Specification and propose techniques for fingerprinting and identifying SQLite files in non-coherent data and the recovery of stored information from a SQLite database. It explains how a database file is laid out on the disk and the various physical and logical structures that comprise a SQLite database. In addition, we put forward reasoning for the value of the paper's propositions with regard to its importance in computer forensics.

1.2 Introduction to SQLite

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. This property makes it a very favourable for use in situations where a client side data store is desired. SQLite is used extensively by web browsers like Google Chrome and Firefox to store metadata and user information such as cookies.

Smartphone operating systems like Android, Symbian, and the iOS use SQLite to store contact lists and SMS messages. This wide usage of SQLite makes it a valuable subject for both forensics investigators and researchers.

The entire library is distributed and included in applications as a series of C source files or a large 'amalagamated' source and header file for inclusion into an application. It is often statically compiled into the application because of its extremely small size and as a result, performs SQLite operations within its process memory.

1.3 Motivations

Since SQLite is so widely used, the authors of this paper intended for this paper to further their, and ultimately, the reader's understanding of the SQLite file format with the intention to develop techniques to fingerprint and carve SQLite files in numerous data sources.

A side objective was to enable researchers to better develop open source towards this goal as we noticed a large number of commercial SQLite forensics tools when compared to an almost non-existent open source scene.

1.4 Summary of Conclusions

We had concluded that it was possible to carve table records given data containing a full or partial SQLite file to a degree of corruption. As a proof-of-concept for some techniques put forward, we include Python scripts for use on a sample database file one may create themselves using a supplied set of SQL commands in sqlite3.

2 SQLite File Format

This section will describe the SQLite format to an extent that is relevant to carving of SQLite data.

2.1 SQLite Header

Every SQLite database begins with a 100 byte header containing crucial metadata. An SQLite file may be identified almost intuitively by its magic signature, "SQLite format 3\x00".

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Magic Numbers "SQLite format 3\x00"																Page Size		Wr	Re
Rs	Ma	Mi	Le	Change Counter				Database Size				First FTP				Total FTP			
Schema Cookie				Format Number				Default Page Cache Size				Largest Root B-Tree Page				Text Encoding			
User Version				Incremental Vacuum Mode				Reserved for expansion											
												Version-Valid-For				SQLite Version Number			

As only a few fields of the header are relevant to our ends, we will focus solely on those. Please refer to the official file format specification for descriptions of the other fields.

The first 16 bytes of the header correspond to the string "SQLite format 3\x00" and is one of the identifying features of SQLite. Other important fields are: Page size (offset 16), Maximum embedded, minimum, and leaf payload fractions (offsets 21, 22, and 23 respectively), and the In-header database size (offset 28).

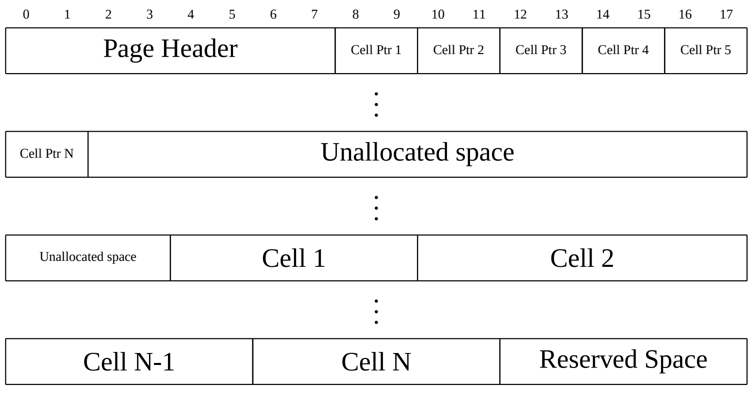
The magic numbers and payload fractions allow us to fingerprint database files while the page size and in-header database size (total number of pages) aid in our carving technique.

2.2 Pages

The largest structure in an SQLite file is a page. Pages come in a variety of types: Freelist Pages, Lock-Byte Pages, B-tree Pages, Payload Overflow Pages, and Pointer Map Pages. Pages have a default page size of 1024 bytes.

Only B-tree Pages and Payload Overflow Pages are of carving value, the rest are not relevant to our ends and hence in the remainder of this paper, it is these two types of pages we will focus our attention on.

There are 4 types of B-tree pages: table interior, table leaf, index interior and index leaf. Table leaf pages are of carving value because they store table content.



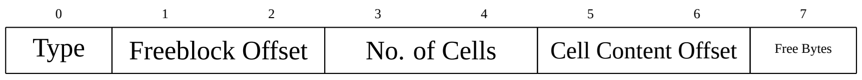
A table leaf page is comprised of: the 100 byte SQLite header (only on first page), an 8 byte page header, a cell pointer array, unallocated space, and cells.

Page headers contain metadata about the page. Cell pointer arrays are 2 byte big endian integer offsets arranged in ascending key value that point to cells at the end of page. There will always be an equal number of cell pointers and cells. Cells correspond to rows in the logical database table. The payload of each cell contains a record, the contents of the row (i.e. logical columns).

The reserved region is an unused space reserved for use by extensions to store page specific data. The size of reserved regions are specified by the 1-byte unsigned integer in offset 20 of the 100-byte database header. The size of the reserved region is usually 0.

2.2.1 Page Header

The B-tree page header is 8 bytes (or 12 bytes) long depending on the type of the B-tree page, which is detemined by the one byte value at offset 0 of the page header.

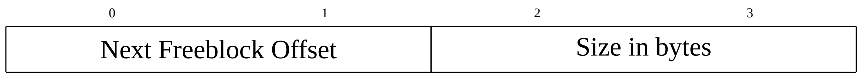


The type field is a flag. The value of this flag determines the type of B-tree page: 0x02 – interior index page, 0x05 – interior table page, 0x0A – leaf index page, and 0x0D – leaf table page. The freeblock offset points to the first freeblock in the page and is 0 if there are none. The number of cells describe how many cells are contained within the page. This also tells you how many entries are there in the cell pointer array. The cell content offset points to the start of the cells at the end of the page.

In interior B-tree pages offsets 8-12 contain the pointer to the right-most pointer. However, we need not concern ourselves with interior B-tree pages for the scope of this paper hence it is not included in the diagram above.

2.2.2 Freeblock

A freeblock is a 4 byte structure within a page that identifies unallocated or unused space within the B-tree page. Multiple freeblocks form a chain.



Freeblocks are chained together with the freeblock having the smallest offset at the beginning and ending with the freeblock with the largest offset. The first freeblock may be discovered by looking at offset 1 of the page header. The freeblock offset field points to the next freeblock or is 0 if it is the last freeblock.

A freeblock requires at least 4 bytes of space. Anything that is isolated and has less than 4 bytes of unused space within the cell content area is a fragment. The total number of fragmented bytes is stored in the 5th field (bytes 8-11) of the B-tree page header. Total number of fragmented bytes should not be more than 60 in a well-formed b-tree page.

In a b-tree page, the size of the unallocated region in addition to the total size of all freeblocks and fragmented free bytes equals the total amount of freespace.

SQLite may perform an operation similar to defragmentation on file systems to a b-tree page. It will reorganise the b-tree page to reduce or remove the number of freeblocks or fragmented free bytes in the page. The unused bytes will be contained in the unallocated space region. Cells with data are at the same time moved as tightly as possible to the end of the page.

2.2.3 Cell Format

A cell within a page contains a record from a logical table in an SQLite database. This is the level we are most interested in carving. Cells are variably sized.

0	1	2	3	4
Payload Size	Row ID	Payload		Overflow Page

The payload size and row id is represented as a varint (please see the section on variable length integers for more information). Thus, each of these fields may range from 1 to 9 bytes in length. The payload length is also variable. The overflow page number is encoded as a 4 byte big endian integer.

2.2.4 Cell Payloads or Records

Cell payloads are comprised of a header and a body.

0	1
Header	Body

The header contains the header size as well as a variable number of serial types describing the content of the body. The body contains actual record data.

0	1	2	3	4
Header Size	stype 1	stype 2	···	stype N

All fields are encoded as varints. Each serial type corresponds to a column in the body. A table as follows describes the length and method of interpretation of the serial types when mapping to values in the body.

0	1	2	3	4
Value 1	Value 2		···	Value N

Serial types may be decoded using the following table:

Serial Type	Content Size	Meaning
0	0	NULL
1	1	8-bit twos-complement integer
2	2	Big-endian 16-bit twos-complement integer
3	3	Big-endian 24-bit twos-complement integer
4	4	Big-endian 32-bit twos-complement integer
5	6	Big-endian 48-bit twos-complement integer
6	8	Big-endian 64-bit twos-complement integer
7	8	Big-endian IEEE 754-2008 64-bit floating point number
8	0	Integer constant 0.
9	0	Integer constant 1.
10, 11		Reserved
N$\geq 12 and even$	(N-12)/2	A BLOB that is (N-12)/2 bytes in length
N$\geq 13 and odd$	(N-13)/2	A string in the database encoding and (N-13)/2 bytes in length.

2.2.5 Overflow Pages

A page's size is fixed while a record (or payload) size varies, a record's size can be too large to fit in one leaf table b-tree page. For example, if the page size is 1024 bytes and the record's (payload) is more than 1024 bytes, the record will not be able to fit within a single leaf table b-tree page and will thus be divided to fit into the available payload space in the page and the remainder in one or more overflow pages.

These pages form a linked list. The first 4-byte big endian integer 'overflow page no.' field in the overflow page indicates the next overflow page in the chain. If its the last overflow page, the field is 0. The fifth byte onwards to the last usable byte can be used for the payload. Each overflow page belongs to only one record.

2.3 Variable Length Integers

A variable-length integer or “varint” is a static Huffman encoding of 64-bit twos-complement integers requires fewer bytes to represent small positive values, a common range of values found in a database (e.g. row ids, small integers, etc). A varint is between 1 and 9 bytes in length and consists of either zero or more bytes which have the high-order (most significant) bit set followed by a single byte with the high-order bit clear, or nine bytes, whichever is shorter.

If the varint is less than nine bytes long, the lower seven bits of the bytes are used, to reconstruct the 64-bit twos-complement integer. When the varint is nine bytes long, the lower seven bits of each of the first eight bytes and all 8 bits of the ninth byte are used in reconstruction. Varints are stored in big-endian order.

Please see the appendix for an implementation of a varint decoder.

3 Carving

With the theoretical foundation in SQLite file formats laid out, we are in a position to begin proposing techniques for carving information from binary sources such as core dumps, file system images, and RAM memory images.

3.1 Carving Sources

First, we must identify a sample of souces whereby data carving of records may be carried out when performing SQLite forensics. An important point to note is that in order to carve successfully, one must understand how a source is structured. Hence, in order to ensure that the techniques listed in this section will yield positive results, a level of contiguity must exist. This level of contiguity should be, at the minimum, the size of a page.

A unique feature of SQLite that makes understanding the file format more valuable for forensics applications when compared to the more common server-client database model is its embedded property. The engine is often compiled into applications and all database operations is handled by the application itself. SQLite may be found embedded in microcontrollers, web browsers, user applications, mobile applications.

What this implies is that full SQLite database files or partial pages may be found in stack dumps, working memory, or core dumps when an application utilising the database is run or crashes. When SQLite files or pages are found in such locations, knowing techniques for carving data by hand is essential as the data or structures presented might be corrupted (e.g. remnants from a previous stack frame not quite overwritten).

3.2 Abstract of Carving Procedure

The overall procedure for performing SQLite forensics may be summarised:

1. Detecting a SQLite database
    (a) Locate start of SQLite Database (constants in header)
    (b) Parse header to find page size

2. Carving
    (a) Determine tables schema
        i. Locate page one, and parse the page header
        ii. Locate cells using the cell pointer array
        iii. Parse cell records
    iv. Correspond fields in the record to the sqlite$_{m}asterschema$
    (b) Extract table records
        i. Locate B-tree leaf table pages, and parse page headers
        ii. Locate cells using the cell pointer array
        iii. Parse cell records
            A. Follow pointers to overflow pages if the capacity of the page is inadequate to hold the record.

A simple hierarchy of the structures we will encounter when carving a sqlite database is:

SQLite Database [Leaf Table B-tree Page [Cell [Record]], OverFlow Page]

It is clear that in order for us to obtain the data with fidelity in "Record" we have to begin by first unpeeling the outer layer, SQLite Database and continue on the Leaf Table B-Tree Page, then on Cell, and finally, the Record which will require further parsing in order to obtain the data from a database table row.

However, carving may still be carried out directly on the Cell level if the forensics operator recognises patterns and data structures in the raw data such as readable strings.

3.3 Identifying a SQLite Database

A SQLite Database can be identified with the following constant characteristics:

1. A Magic Number "0x53 0x51 0x4c 0x69 0x74 0x65 0x20 0x66 0x6f 0x72 0x6d 0x61 0x74 0x20 0x33 0x00"
2. Page size at offset 16 is always a 2 byte big endian integer that is a power of 2 and by default is 1024 (0x04 0x00)
3. The 3 bytes starting from offset 21 of a database header file is a constant "0x40 0x20 0x20"
4. 100 byte header size

This is assuming that the database is well-formed. If the magic number has been tampered with, this method of identifying a SQLite database file will not work. This step of identifying a SQLite database in a binary blob or filesystem image can be ignored in case of a non-well-formed database.

It is interesting to note that in the case where a database header cannot be found, but Leaf Table B-tree Pages can be identified, records may still be extracted. If there is a need to know the constant size of a page, the default value of 1024 bytes may be assumed or a value that is a power of 2 between 512 to 65536 bytes.

3.4 Identifying Leaf Table B-Tree Pages

From what we learned in the earlier sections of this paper, a Leaf Table B-Tree Page can be identified using the following characteristics

1. 1-byte Constant “0x0D” which indicates the start of the page
2. Fixed page size as defined in database header
3. Offset 1st freeblock is less than page size constant
4. Number of Cells is more than 0 and less than Page Size/4
5. Offset to Cell Content area is equal to 0 or between 8 and Page Size.

The page can be carved starting from “0x0D” to a size of the Page Size constant as defined in the database header. Assumption of Point 4 is that the minimum size of a Cell is 4 bytes. The above points make the assumption that the Leaf Table B-Tree Page is well-formed.

3.5 Identifying and Carving Cells from Pages

After identifying pages, the cells in the page can be identified using the cell pointer array which is after the 8-byte b-tree page header. The cell pointer array contains pointers that are made of 2-byte integers which is the offset in the b-tree page to the first byte of each cell. According to the cell format, the first varint of the cell indicates the payload size and the next varint is the rowid. Therefore, from the cell offset in the cell pointer array, the first 2 varints will indicated the payload size and rowid respectively.

The SQLite File Format Specification provides a formula to calculate for any payload overflows:

```
UsableSize = TotalPageSize - ReservedSize
TotalPageSize and ReservedSize are defined in the database header.
MaxLocalSize = UsableSize - 35
If PayloadSize <= MaxLocalSize then entire payload in page
else if PayloadSize > MaxLocalSize then
    MinimumLocal =  ((UsableSize-12)*32/255)-23
    LocalSize = MinimumLocal + ((PayloadSize-MinimumLocal) %
                (UsableSize -4))
LocalSize is the size of the payload stored in the cell.
Therefore, PayloadSize - LocalSize = OverFlowSize.
```

3.6 Carving and Parsing Records

After, the 2 varints of the Cell is the Record where the contents of a database table row resides. From the carved Cells, the record can be parsed and data extracted. In the case that the cells are not from a leaf table b-tree page, automated tools should be able to catch exceptions rising from errors during parsing.

The following steps are to be followed in the parsing of the cell's payload, in other words the record:

1. Determine header size of payload from the first varint
2. After the varint till the end of the header, determine the serial type of each column
3. Determine expected size of value from each serial type encountered
4. Parse the expected size of value as that column's value from the body section of the payload.
5. A dummy string is stored as the value of a column that has overflows.

Following the above steps, a record from each cell is extracted.

4 Applications and Computer Forensics

As smartphone usage increases, so will the number of devices utilising SQLite. Similarly, the number of browsers and other applications that require a fast and light solution for storing data client side are growing exponentially. The information contained in these SQLite databases will be valuable to forensics investigators as they may contain a high quantity of personal user data like short messages, HTTP cookies or browsing histories. As forensics investigators do work on forensic images of filesystems or RAM, there is a need to make sense out of the data, i.e. carve SQLite records from these sources.

There has not been very detailed research on SQLite carving published nor are there any open source solutions to perform SQLite forensics. With the open sourced proof-of-concept code produced alongside this report, it is hoped that further research may be carried out and better tools can be developed from it.

4.1 Challenges and Asides

A challenge faced when carving SQLite databases from non-contiguous forensic images like filesystem images is in the extraction of a whole database, especially large sqlite databases. The header and different sections of the database might not take up a contiguous chunk of hard disk space so it is difficult to simply identify a SQLite database header and expect to extract a whole fully intact database.

However, it is possible to identify and extract individual B-tree pages but not to a hundred percentage accuracy as with all carving operations.

5 Thoughts

Our thoughts on the subject and the assignment after the completion of the project.

5.1 Ku Wee Kiat

There is very little open source tools dedicated to the carving of sqlite databases and data. In recent years, there has been a rise in interest in sqlite because of its special characteristics which led to widespread usage of sqlite db. While researching the topic of SQLite Forensics, the official sqlite documentation and especially the fileformat documentation has been a great help providing lots of details of the inside workings of sqlite.

I have learned a lot of new things, not just related to forensics or sqlite, just from reading the documentation. For example, b-trees, varints and how useful offsets and linklists are. Looking at the development of the SQLite carving PoC code was also a good learning experience and can be used in future projects.

Tasks Allocated Researching on the sqlite file format and possible forensics techniques.

5.2 Jeremy Heng

My first encounter with performing SQLite forensics, particularly within the realm of carving, was in a cyber security capture the flag competition hosted in Korea (Codegate 2012). There was a challenge that required you to carve deleted data from a cookie database generated by the Chrome browser. Since then, I've been rather interested in SQLite technology and have used the library myself for application projects.

During the competition, it was annoying how most of the SQLite data recovery tools we had found were proprietary. Carving by hand then was not an option as time was of the essence. This was a primary motivation to produce some source code to aid carving for this paper.

Throughout the course of this project, I have learnt many interesting things from a variety of domains. Of course, I have learnt about how SQLite stores its data but I did enjoy the esoteric things like Huffman encoding schemes and implementing them on my own. Data structures also took the centre stage with B-trees, B+trees, pointers, offsets, endianness, and structures playing a huge part of SQLite's internal workings.

Also, I have grown to respect D. Richard Hipp, the creator of SQLite, immensely for not only creating one of the best SQLite engines around but also for his altruism. SQLite is under the public domain and is open for anyone to use.

Tasks Allocated Proof-of-concept writing, research, development of identification and carving techniques.

A Test SQLite Database

We have created a test ("testdb") database for use with the demonstrative code. The following SQL commands will create an identical copy.

CREATE TABLE comftest (var1 integer not null, var2 integer not null, astring varchar(100) not null);

INSERT INTO comftest VALUES(1, 2, "This is a string for COMF.")

INSERT INTO comftest VALUES(3, 4, "This is yet another string for COMF.")

INSERT INTO comftest VALUES(5, 6, "ss")

INSERT INTO comftest VALUES(7, 3133731337, "That number is a really huge integer!")

B Variable Length Integer Decoder

This file implements an algorithm to decode variable length integers as used by the SQLite file format. It demonstrates the algorithm on a set of sample numbers used in documentation.

---

readvarint.py

```
def main():
    vals = [[0x2B],
            [0x8C, 0xA0, 0x6F],
            [0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
             0xFF, 0xFF, 0xFF, 0xFF, 0x01],
            [0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
             0xFF, 0xFD, 0xCD, 0x56],
            [68, 1, 6, 23, 21],
            [1, 6, 23, 21]]

    print "Signed test:"
    for i in vals:
        res = decode_varint(i)
        print "%s: %d (%d bytes)" %
        (", ".join(map(hex, i)), res[0], res[1])

def decode_varint(bl):
    def twos_comp(val, bits):
        if( (val&(1<<(bits-1))) != 0 ):
            val = val - (1<<bits)
        return val

    bit_set = 0
    mask = 0
    no_of_bytes = 0
    for i in range(9):
        no_of_bytes += 1
        if i != 8:
            to_add = (bl[i] & 0b01111111)
            bit_set = (bit_set << 7) + to_add
            if not bl[i] & (1 << 7):
                break
        else:
            bit_set = bl[i] + (bit_set << 8)

    return (twos_comp(bit_set, 64), no_of_bytes)


if __name__ == "__main__":
    main()
```

C Reading pages from SQLite database

This file demonstrates how one might parse SQLite pages. We do not actually use corrupted data. To obtain our pages, we simply split a test SQLite database along its page size.

---

readvarint.py

```
PAGE_SIZE = 1024

import pprint
```

```
def main():
    data = file("testdb").read()
    no_of_pages = len(data)/PAGE_SIZE
    pages = []
    for i in range(1, no_of_pages+1):
        pages.append(data[PAGE_SIZE*(i-1):PAGE_SIZE*i])
    for i in pages:
        print "Dumping page: "
        print read_page([ord(j) for j in i])

def read_page(bl):
    # Create page dictionary
    page = {}

    # Set the page header offset
    fst_pgb = "SQLite format 3\x00" == "".join(map(chr, bl[:16]))
    phos = (100 if fst_pgb else 0)

    # Read header
    pg_hdr = parse_page_header(bl[phos:phos+8])
    pg_hdr['offset'] = phos
    page['hdr'] = pg_hdr

    # Read cell pointer array
    previous_cell = PAGE_SIZE
    ca_begin = phos + 8
    ca_end = phos + 8 + (phos+8+pg_hdr['number_of_cells']*2)
    cell_array = bl[ca_begin: ca_end]
    p_cell = []
    for i in range(0, pg_hdr['number_of_cells']*2, 2):
        current_offset = tbl(cell_array[i:i+2])
        cell_size = previous_cell-current_offset
        p_cell.append((current_offset, cell_size))
        previous_cell = current_offset

    page['cptr_array'] = p_cell

    # Read cells
    cells = []
    for i in p_cell:
        offset, length = i
        cells.append(bl[offset:offset+length])
    page['cells'] = cells

    # Return the page dictionary
    return page
```

```
def parse_page_header(bl):
    if len(bl) != 8:
        return None
    pg_hdr = {}
    pg_hdr['type'] = bl[0]
    pg_hdr['first_freeblock_offset'] = tbl(bl[1:3])
    pg_hdr['number_of_cells'] = tbl(bl[3:5])

    c_off = tbl(bl[5:7])
    pg_hdr['cells_offset'] = (65536 if not c_off else c_off)
    pg_hdr['number_of_freebytes'] = bl[7]
    return pg_hdr

def tbl(bl):
    total = 0
    for i in range(len(bl)):
        total += bl[::-1][i] << (i*8)
    return total


if __name__ == "__main__":
    main()
```

D Reading cells

This file demonstrates how one might parse cells obtained from SQLite pages.

---

readvarint.py

```
def main():
    test = [32, 1, 4, 1, 1, 65, 1, 2, 84, 104, 105,
             115, 32, 105, 115, 32, 97, 32, 115, 116,
             114, 105, 110, 103, 32, 102, 111, 114, 32,
             67, 79, 77, 70, 46]
    badtest = "BADDATA" * 10
    print decode_tleaf_cell(test)
    try:
        print decode_tleaf_cell([ord(i) for i in badtest])
    except Exception:
        print "Bad t-leaf cell."


def decode_tleaf_cell(bl):
    def pop_vstack(vbl):
        """Pops off a varint off the stack and
        returns a tuple of the value and the
        remainder."""
        value, vlen = decode_varint(vbl)
        return (value, vbl[vlen:])

    def pop_vstack_rec(st, vbl):
        type_table = {}
        type_table[0] = lambda b: (None, vbl)
        type_table[1] = lambda b: (twos_comp(b[0], 8), vbl[1:])
        type_table[2] = lambda b: (twos_compl(b[:2], 16), vbl[2:])
        type_table[3] = lambda b: (twos_compl(b[:3], 24), vbl[3:])
        type_table[4] = lambda b: (twos_compl(b[:4], 32), vbl[4:])
        type_table[5] = lambda b: (twos_compl(b[:6], 48), vbl[6:])
        type_table[6] = lambda b: (twos_compl(b[:8], 64), vbl[8:])
        type_table[7] = None # Will throw error. Not implemented.
        type_table[8] = lambda b: (0, vbl)
        type_table[9] = lambda b: (1, vbl)
        type_table[10] = None
        type_table[11] = None

        if st < 12:
            return type_table[st](vbl)
        elif st % 2 == 0:
            pass # handle blob
        else:
            str_len = (st-13)/2
            return ("".join(map(chr, vbl[:str_len])), vbl[str_len:])


    payload_len, bl = pop_vstack(bl)
    rowid, bl = pop_vstack(bl)
    overflow_page_no = bl[payload_len:] or None

    pstack = bl[:payload_len]
    payload_hdr_len, pstack = pop_vstack(pstack)
    payload_hdr_st = []
    for i in range(payload_hdr_len-1):
        res, pstack = pop_vstack(pstack)
        payload_hdr_st.append(res)
    cell = {’payload_len’: payload_len,
            ’rowid’: rowid,
            ’payload_hdr_len’: payload_hdr_len,
            ’payload_hdr_st’: payload_hdr_st,
            ’payload’: []
            }
    for i in payload_hdr_st:
        res, pstack = pop_vstack_rec(i, pstack)
        cell[’payload’].append((res, i))

    return cell

def twos_comp(val, bits):
    if( (val&(1<<(bits-1))) != 0 ):
        val = val - (1<<bits)
    return val

def twos_compl(bl, bits):
    total = 0
    for i in range(len(bl)):
        total += bl[::-1][i] << (i*8)
    return twos_comp(total, bits)

def decode_varint(bl):

    bit_set = 0
    mask = 0
    no_of_bytes = 0
    for i in range(9):
        no_of_bytes += 1
        if i != 8:
            to_add = (bl[i] & 0b01111111)
            bit_set = (bit_set << 7) + to_add
            if not bl[i] & (1 << 7):
                break
        else:
            bit_set = bl[i] + (bit_set << 8)

    return (twos_comp(bit_set, 64), no_of_bytes)

if __name__ == "__main__":
    main()
```

References and Reading List

[1] I. Pooters, P. Arends, and S. Moorrees, “Extracting sqlite records: Carving, parsing and matching,” *FOX-IT*, July 2011.

[2] D. R. Hipp, “Sqlite’s use of temporary disk files,” *SQLite Documentation*, 2012.

[3] D. R. Hipp, “The sqlite database file format,” *SQLite Documentation*, 2012.

[4] D. R. Hipp, “Requirements for the sqlite database file format,” *SQLite Documentation*, 2012.

[5] RaySprite, “Sqlite database file format,” *RaySprite*, 2012.

[6] D. R. Hipp, “Well-known users of sqlite,” *SQLite Documentation*, 2012.

[7] R. Drinkwater, “Carving sqlite databases from unallocated clusters,” *Forensics from the sausage factory*, Apr. 2011.

[8] D. Eindbazen, “Codegate 2012 – forensics 300,” *De Eindbazen*, Feb. 2012.

[9] FLOSS, “Floss weekly 26,” *FLOSS*, Mar. 2008.

[10] M. Owens, *The Definitive Guide to SQLite*. Definitive Guide, Apress, 2006.