ANTS ROAD: A New Tool for SQLite Data Recovery on Android Devices

Lamine M. Aouad, Tahar M. Kechadi, and Roberto Di Russo

Centre for Cybersecurity and Cybercrime Investigation
University College Dublin, Ireland
{lamine.aouad,tahar.kechadi}@ucd.ie

Abstract. Recovering deleted information is one of the most important probative elements in a forensic investigation that involves a mobile phone. In this paper, we present a new tool implementing an innovative method, based on a low-level analysis, to recover deleted data from SQLite databases on Android devices, taking as an initial example text messages. The paper then proposes a generic framework for deleted data recovery that can be used with a range of SQLite databases on a variety of Android systems and devices. Indeed, although our initial aim was to recover deleted SMSs, we realized along the way that, with the appropriate changes, the initial implemented method can be applicable to the extraction of deleted information from any SQLite database file.

1 Introduction

In the last decade or so, the world of mobile phones has gone through a tremendous change, transforming the devices from simple phones to pocket computers, mostly referred to as smart phones. Prior to this change, using a mobile phone meant by and large emitting and receiving calls or text messages (SMSs). Nowadays, with the spread of new hardware and software technologies, a mobile user can also surf the Web, chat, and do more or less everything he or she can do with a desktop or a notebook computer, and even more. The capability of these devices is still growing, as the number of their users. Indeed, they are today the highest-selling consumer electronic devices.

The smart phones proliferation has brought new issues in terms of forensics evidence acquisition. They handle a large amount of personal data, including text messages, communications logs, contacts, multimedia, geo-location information, etc. These could potentially help answering crucial questions in a criminal investigation. However, the huge variety of devices and the lack of standards imply that there is no unified method of accessing, extracting, or retrieving this data. Surely, a huge contribution to the smart phones market growth was brought by Android OS, by which an impressive amount of relatively low-price devices has been sold. According to data from Google, the activation rate is projected to reach a million per day by mid August of this year (2012), and if it continues we could see 1.5 million per day by the end of 2013. Android accounts for 68% share of the global smart phone market (2^{nd} quarter of 2012).

These devices store most of the information in database files, which keep track of information deleted by the user, to a certain extent. However, this information cannot be accessed by traditional databases browsers and tools, and need alternative techniques.

This paper presents a new low-level analysis method for the recovery of deleted information from SQLite database files on Android devices. We particularly applied it to the recovery of deleted SMSs, then generalize it to other databases and devices. The next section presents related work and surveys some of the existing tools. Section 3 will then present the proposed method. Section 4 shows the initial evaluation supporting few databases from different Android releases and devices, along with a discussion and future work. Concluding remarks are then given in section 5.

2 Related Work

Android OS stores the information in a set of database files. These files are managed by SQLite, a database engine, also used by Mozilla Firefox, Thunderbird, Skype, Apple's iOS and Blackberry, among others. According to the SQLite documentation [2]: *"when you delete information from an SQLite database, the unused disk space is added to an internal free-list and is reused the next time you insert data. The disk space is not lost, but neither it is returned to the operating system"*.

About this deleted information, it adds: *"if you do not have a backup, recovery is very difficult. You might be able to find partial string data in a binary dump of the raw database file. Recovering numeric data might also be possible given special tools, though to our knowledge no such tools exist. (...) Recovery is also impossible if you have run vacuum since the data was deleted. If vacuum has not been run, then some of the deleted content might still be in the database file, in areas marked for reuse. But, again, there exist no procedures or tools that we know of to help you recover that data"*.

In a nutshell, the vacuum operation, mentioned earlier, rebuilds the database. It simply copies the contents of the database into a temporary database file and then overwrites the original with the contents of the temporary file. This procedure allows to reclaim the free marked space. It ensures that each table is stored contiguously and it may also reduce the number of partially filled pages. In auto-vacuum capable databases, the Database Management System executes the command automatically. The application designer or the database administrator has no control on the rebuilding operation. This feature is to keep in mind in the deleted records recovery. Indeed, since the user cannot know the last time that the vacuum command has been executed, he/she cannot know how many records can be recovered. Also, as a result of this operation, the recovery on the same database at different times will not necessarily return the same result set.

In [1], the authors have reported the recovery properties of few database systems, including SQLite, by comparing their behavior in terms of deletion, update, insert, and vacuum operations. An important behavior of SQLite to mention here is that data is deleted logically, and not removed. Previous values of a record are however completely overwritten in many cases during an update. Deleted records are freed and subsequent insertions may overwrite the data. However, freed pages are not returned to the file system until vacuum is performed. The recovery rate was also fairly low, at about 400 to 500 records throughout the tested workload, which was up to 30000. Although this is a completely different use case, including about hundred operations of insertion, update, deletion and so on, it shows the challenges of recovering information from this database management system. In terms of software tools, Epilog [9] for Windows platforms, is the only dedicated tool to deleted data recovery from SQLite databases we could find. It includes three recovery algorithms that can be used on any SQLite database, regardless of the type of the data stored. Nevertheless, the tested version of Epilog, on few SMS databases, did not recover any of the deleted messages. This is more likely the result of the wide range of customization and differences in the database structure and fields across devices and OS versions.

On the other hand, there are many studies in the literature on the retention and recovery of deleted data from different underlying systems including file systems, memory, and even specific applications such as browsers and documents [11], [12], etc. There are also many forensic tools and methods performing physical and logical data acquisition, for Android and other devices, surveys on existing tools can be found in [3], [4], and few methods in [5], [6], [7], among many others. However, logical acquisition simply queries the databases, and therefore cannot extract information that are not accessible from an SQL browser. The existing literature is very limited, and deleted SMSs recovery from an Android device, and more generally any other deleted data source, can be considered as a relatively unexplored field. The existing forensic analysis support of database files, other than as physical memory dumps, is very limited. This work aims at covering this gap.

3 The Method

In order to extract deleted text messages from an Android device, we performed a low-level analysis on the related database file. Android stores all the information about SMSs and MMSs in the `mmssms.db` file. The data we are interested in resides in the `sms` table. For a forensically-sound analysis, we pulled out a copy of this database from the device.

3.1 SQLite Database Structure

Before explaining the method, let us present the structure of SQLite databases. It is composed of pages, most of which are organized in a B-tree structure. A page is a set of a fixed number of bytes, whose size is a power of 2, between 512 and 65536 inclusive. All the pages in the same database have the same size and they are numbered, starting from 1. Each page can have only a single use between the following.

– *Freelist pages*: pages that are not in active use anymore, and that are put in a linked list to be reused if additional pages are required.
– *B-tree pages*: a B-tree page can be an internal page or a leaf page. The content is stored only in the leaf pages, so it is on these pages that we focused our attention on. A B-tree page is either a table B-tree page or an index B-tree page. However, since the data is stored only in table B-tree pages, we focused only on these. The data stored in a B-tree table leaf page is organized in cells. Usually, but not always, each cell contains exactly one record.

- *Overflow pages*: sometimes the payload of a B-tree cell is too big to fill in a B-tree page. In these cases, the surplus is stored into an overflow page. The overflow pages form a chain, the first four bytes of every overflow page are a pointer to the next page, or have value 0 if the page is the end of the chain.
- *Pointer map pages*: pages whose aim is to make the auto-vacuum operation more efficient. They simply contain links between pages from child to parent. The first, and usually the only one, pointer map page is page 2. A pointer map page exists only if the database is auto-vacuum. It has been useful to our work to further shrink the set of candidate pages where to look for the deleted records.

3.2 Analysis Set Up

In every SQLite database, the first page contains the 100 bytes database header, that is divided into fields. The multibytes fields are stored in big-endian format. For our analysis, the most significant fields are listed in the following table. All offsets are intended from the beginning of the first page and all the sizes are expressed in bytes.

Offset	**Size**	**Description**
0	16	The header string ”SQLite format 3\0”.
16	2	The database page size in bytes.
52	4	If greater than 0, the database is auto-vacuum capable.
56	4	The database text encoding. A value 1 means UTF-8.

Before starting the analysis, it is necessary to check that the first 16 bytes match the string ”*SQLite format*”, followed by the null terminator character. If not, the database is not a valid SQLite database file, and this method cannot be applied. In our case, if the `mmssms.db` copy is not corrupted, it will pass the check.

At offset 16, important information is located, stored in 2-bytes big-endian format, it is the page size. This is useful for dividing the database into pages, the working units of the first part of our work, that is the selection of a candidate set of pages where to look for the deleted records. The Android SMSs database has a page size of 1024 bytes. The 4-bytes big-endian integer at offset 52 indicates if the database is auto-vacuum. This information is quite important here. Indeed, if this field has value 0, the database is not auto-vacuum capable and it is not possibile to explore the pointer map stored in the second page. The `mmssms.db` file is auto-vacuum, which is the default.

At offset 56, there is a 4-bytes value that indicates the database text encoding. In our case, the relevant value is 1, which means UTF-8 encoding. Other values are 2 for UTF-16 little-endian and 3 for UTF-16 big-endian. For the other fields meaning, we refer the interested reader to the official SQLite documentation [2]. After the initial first page analysis, we went on with the analysis of the pointer map page that represents the starting point of the first selection of interesting pages, i.e. potential source of deleted text messages.

3.3 The Pointer Map Page Analysis

As already mentioned, in an auto-vacuum database, the second page represents a pointer map page, which aim is to facilitate moving the pages around in the database as part of performing the vacuum operation. It is a sort of lookup table that stores a 5 byte record for every page that follows the pointer map page. In these records the first byte indicates the page type, and the others 4 bytes, to read in big-endian format, are a reference to the parent page, indicated with `0x00 0x00 0x00 0x00` if it is null, or `0xVV 0xVV 0xVV 0xVV` (where `VV` stands for variable) otherwise.

- **`0x01`** `0x00 0x00 0x00 0x00`: a B-tree root page has no parent page.
- **`0x02`** `0x00 0x00 0x00 0x00`: a B-tree free page has no parent page.
- **`0x03`** `0xVV 0xVV 0xVV 0xVV`: the first page of an overflow chain. Its parent is the B-tree page containing the B-tree cell to which the overflow chain belongs.
- **`0x04`** `0xVV 0xVV 0xVV 0xVV`: a page that is part of an overflow chain, but that is not the first page. Its parent is the previous page in the overflow chain.
- **`0x05`** `0xVV 0xVV 0xVV 0xVV`: a page that is part of a table or index B-tree structure and is not a root page or an overflow page. Its parent is the page containing the parent tree node in the B-tree structure.

Interested readers can find a deeper analysis of the pointer map page in [2], or [8]. In our work, in order to perform an initial shrink of the candidate pages set, we kept only pages that are part of a table B-tree structure and pages in overflow chains. Then, we made a further selection in this set, keeping only the pages that are not child of the B-tree root page, because we empirically realized that they do not contain any useful data. By doing so, the searching set has been significantly reduced.

3.4 B-Tree Table Leaf Pages Analysis

Once we obtained the first candidate set, we started the pages analysis. First of all, we checked out the first byte of each page. If its value is `0x0D`, i.e. 13, it means that the page is a leaf node in the B-tree structure, so it contains data and it is a good candidate to contain deleted records.

The second and the third bytes of a page represent the relative offset of the first free space block inside this page. If this offset is zero, it means that in the page there is no free space, and then there cannot be any deleted record, since the space occupied by deleted records is considered to be free. We went on in the analysis selecting only the pages with at least one free space block, further shrinking the candidate set.

The next two bytes tell us the number of cells in the page (nPages), in our case the number of non-deleted SMS records. At relative offset 5, a 2-bytes field indicates the offset from the page starting of the first cell that contain a valid record, while the next byte is a null separator (`0x00`). Going on, there are (nPages) byte pairs, each one containing the relative offset of a valid content cell. Each 'pointed' cell containing a non-deleted SMS has a fixed structure, similar to the one shown in figure 1.

The payload length, the Row Id and all the payload header subfields are stored using a VarInt format. VarInt (*Variable Integer*) can take between 1 and 9 bytes, depending on the value stored. The Most Significant Bit (MSB) of each byte indicates if the next

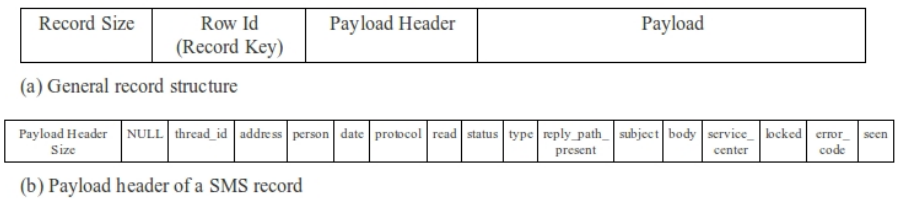

Fig. 1. SMSs record structure

byte is also part of the field (1) or not (0), while the remaining 7 bits are used to store the value itself. For the `mmssms.db` file it is enough to consider the case with at most two bytes VarInt, following this algorithm:

```
Let $x$ be the value of the first byte
**if** $x < 128$ **then**
    $result = x$
**else**
    Let $y$ be the value of the second byte
    $result = (x - 128) * 128 + y$
**end if**
```

Checking that the first byte is less than 128 is equivalent to checking that its MSB is 0. Indeed, if the x's most significant bit is 1, the result can also be computed by concatenating the latter 7 bits of the second byte to the latter 7 bits of the first one. Storing data in VarInt field allows to save space. Indeed, VarInts are big-endian that, using a static Huffman encoding, need less space for small positive values.

The data in the payload is stored in a serialized way: it is the Payload header that indicates how to identify each payload field. In fact, for each payload field there is a Serial Type Code, what we previously called "payload header subfields", that denotes its type of data, according to the following table. Note that we reported only the Serial Types that are relevant to the SMSs database.

Serial Type	**Meaning**
0	null field.
N $\leq$ 4	big-endian 8*N bit two's complement integer.
5	big-endian 48 bit two's complement integer.
6	big-endian 64 bit two's complement integer.
N $\geq$ 13 and odd	(N-13)/2 bytes string in the database encoding.

The payload header varies from one table to another. Each manufacturer can add custom fields to the shown structure. We can check how many fields are in the payload header simply by reading the payload header length. In a deleted SMS record there are some differences. Remembering that if in a page there is more than one free block, they are concatenated in a chain, the first two bytes form a pointer to the next free cell in the chain; a value zero indicates that the current block is the last one. The third and the fourth bytes represent the size of the block in bytes, including the header. Both fields are big-endian integers.

After this *pre-header*, we find the payload header. In our analysis, we realized that the payload header lacks the record size and the record key. Indeed, three different cases are possible:

– It can start with the payload header size,
– With the `NULL` byte that precedes the Serial Type Code list, or
– Directly with the first Serial Type Code, in this case the `thread_id`.

Since the payload header is an important information about the record, but it is not always available, we computed it in advance simply by taking the minimum size between all the valid record (non-deleted SMSs) size. This is a valid method since all the SMS record sizes differ at most by one byte, depending on whether the body Serial Type Code needs one or two bytes. In this computation, we considered only the complete records that can be recognized by a payload header with at least the 16 basic Serial Type Codes.

There are also record fragments or records containing only the body, without any other field, but we did not consider them neither in the computation, nor in the analysis. Applying this method before starting to collect data, we further reduced the candidate set to the pages that contain only complete SMSs. Knowing the payload header size, the data collection has been performed using each Serial Type Code for reading the specified number of bytes and interpreting them according to the above table. Besides the chains of deleted cells, built by the first two bytes of each free block, we also managed the *internal chains*. We define an internal chain as a chain of deleted SMSs inside the same cell and it can be recognized by the fact that the record size indicated in the first two bytes is much bigger than the actual payload content. A simple counter is enough to understand the transition from a record to the next one in the same cell.

Deleted SMSs could also be found in the space between the page header and the first valid content cell. When this happens, the first bytes pair after the page header is greater than 0 and different from the last cell offset (that is, the last bytes pair in the page header) and it represents the offset, from the beginning of the page, of a deleted SMS. This SMS can be considered the head of a chain and its first field indicates the offset of the next element, usually the same offset indicated in the page header as the first free space block inside this page. Based on these patterns, experimentally induced from a large set of test data, we carried out a set of evaluations described in the next section.

4 Evaluation

We tested the method on two different mobile phones, mounting two different versions of Android OS: the LG Optimus One with Android 2.3.3 and the Samsung Galaxy Gio with Android 2.3.5.

On the Optimus One we recovered 22 deleted SMSs: 10 incoming, 10 outcoming, 1 draft and 1 of unknown type. On the Samsung Galaxy Gio, we recovered 6 deleted

SMSs: 3 draft, 2 outgoing and 1 incoming. Manually analyzing page by page the databases, we found 23 deleted SMSs on the Optimus One and 8 on the Galaxy Gio. The method recovered 22 out of 23 (95% of the deleted records) in the former, and 6 out of 8 (75%) in the latter case. In both cases, however, the unrecovered SMSs were partially filled or corrupted, i.e. partially overwritten by other SMSs. This means that the proposed method recovered all the complete deleted SMSs that were still present in the database files.

After one week of use, we repeated the tests. Since we did not delete any additional SMSs, the results on the Optimus One have not changed. On the Galaxy Gio, on the other hand, we deleted all the received and sent SMSs and the proposed method recovered only 2 SMSs, 1 draft and 1 outgoing. This indicates that an auto-vacuum operation has been performed on it. We also applied the vacuum command manually to the copies of the acquired databases. No deleted SMSs have been recovered. This confirmes that the vacuum operation, both manual and automatic, remove all the deleted data that might have still been in the database. These results highlight how strong is the link between the deleted SMSs recovery, and more generally the deleted records recovery from an SQLite database, and the unpredictability of the vacuum operation execution.

4.1 Discussion

The method proposed in this paper implements an efficient way to recover complete deleted SMS records from a SQLite database on Android phones. Nevertheless, the applicability of this work remains subject to the vacuum operation. On the one hand, auto-vacuum is useful to our goal because it allows us to navigate the pointer map page and shrink considerably the candidate set of pages on which to carry on the deleted SMSs lookup. Indeed, if the database file was not auto-vacuum capable, its second page would not contain a valid pointer map and the lookup would have been performed on the whole set of the database pages, which can be quite big. This would have resulted with a considerable loss of efficiency. On the other hand, working with an auto-vacuum enabled database means that we do not have the control of the time of the last clean up. As a result, it is also possible that two recoveries made at different times on the same database return two different result sets.

4.2 Additional Use Cases

To understand what part of the deleted SMSs extraction can be reused, we analyzed two additional databases. Particularly, we tested other Android databases, namely `browser.db` and a proprietary database, WhatsApp's `msgstore.db`. In the browser database, we are interested in the `bookmarks` table that, despite its name, contains the whole Web history, recording for each visited link whether or not it is saved as a bookmark. While the general record structure is the same as mentioned for the SMSs, the payload header changes. We can see that in figure 2. As for the SMS record, we represented only the basic payload header, but each manufacturer can add custom fields to the shown structure. The information we are targeting is:

Payload Header Size	NULL	title	url	visits	date	created	description	bookmark	favicon	thumbnail	touch_icon	user_ entered

Fig. 2. History browser item payload header

- `title`: item title shown on a browser tab,
- `url`: item url (what we actually see in the browser address bar),
- `visits`: how many times the url has been visited by the user,
- `date`: date/time of the last visit,
- `bookmark`: it indicates whether or not the url is saved as a bookmark.

The second database that we studied is `msgstore.db`. It is the database used to store messages sent and received by WhatsApp, a proprietary instant messaging App that uses the Web to send messages. In this database, we are interested in the `messages` table that contains all the sent and received messages. Again, the general record structure is the same as previously discussed, while the payload header, shown in figure 3, changes. In this case, however, the payload header is well-defined and it does not depend on the device manufacturer. For this database, we are targeting the following information.

- `key_remote_jid`: for incoming messages it is the sender id, for outcoming ones the recipient id,
- `status`: message status. It indicated if the message is incoming or outgoing, or if it has an unknown value,
- `data`: message body content,
- `timestamp`: the timestamp when the message has been created,
- `sendTimestamp`: the timestamp when the message has been sent,
- `receivedTimestamp`: the timestamp when the message has been actually received by the target user,
- `receivedServerTimestamp`: for outgoing messages, the timestamp when the message has been received by the WhatsApp server,
- `receivedDeviceTimestamp`: for outgoing messages, the timestamp when the message has been received by the recipient device.

Payload Header Size	NULL	key_ remote_jid	key_ from_me	key_id	status	needs_push	data	timestamp	media_url	media_mime_type	media_size	media_name	...

...	latitude	longitude	thumb_image	remote_resource	received_ timestamp	send_ timestamp	receipt_server_ timestamp	receipt_device_ timestamp

Fig. 3. WhatsApp message payload header

As in the SMS records recovery, for both the Web history items and the WhatsApp messages, the payload length, the Row id and all the payload header subfields are stored using a VarInt format. The rules to compute these values are those we discussed in the previous section. However, this is not the only analogy with the deleted SMSs recovery. Indeed, we realized that most of the work made can be reused, from the pointer map analysis to specific details such as the internal and the external chains management. The only aspect that changes is the field extraction itself. Since the payload header and, consequently the payload itself, is different from database to another, we need to redefine for every database how to extract each field. In practice, using the discussed method, we need to establish how to rebuild each record field from a byte group, whose size is indicated by the related Serial Code Type. Following this process, and implementing it in a modular way, we could obtain, with a relatively small effort, a generalized module to recover deleted information from a variety of databases.

4.3 Future Work

In order to recover more deleted information, one direction would be in trying to retrieve the previous versions of the database files and then apply the proposed method to each of them. However, since the auto-vacuum execution overwrites every time the previous database version with the same name, this should be thought of beforehand and probably integrated in the development process to facilitate potential forensic investigations. The analysis has been made on SQLite general assumptions, we have already showed how we investigated ways to reuse this work to recover deleted information from other Android databases. The following step is to consider a wider range of information stored by SQLite databases in a variety of systems and devices. Indeed, many other phones operating systems use SQLite to store data, including Apple's iOS and Blackberry. This method is not limited to Android systems and devices, and it should be sufficient to check the page size and to adapt the matching Serial Code Types/Payload Field to the specific table structure to obtain a specific-purpose software that runs for any auto-vacuum valid SQLite database. We will validate this generalization in our future work.

5 Conclusion

In this paper, we proposed and implemented a method for deleted information recovery from SQLite databases. The initial target was database files under Android OS. We discovered a set of patterns related to deleted information recovery on these databases and validate it with a set of use cases. This work has a potentially wide range of applications in the important area of mobile digital forensics. It also aims at setting blueprints for a wider range and deeper analysis involving additional databases and operating systems. Indeed, there is a gap in the literature in this area, and this work is addressing it by proposing a method and a tool implementing and documenting the acquisition and analysis of deleted information using a popular database engine.

References

1. Stahlberg, P., Miklau, G., Levine, B.: Threats to privacy in the forensic analysis of database systems. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD 2007 (2007)
2. The SQLite Official Documentation, `http://www.sqlite.org`

3. Hoog, A., Gaffaney, K.: iPhone forensics. Via Forensics White paper (2009)
4. Hoog, A.: Android Forensics - Investigation, Analysis and Mobile Security for Google Android. Elsevier (2011)
5. Aouad, L., Kechadi, T., Trentesaux, J., Le Khac, N.-A.: An Open Framework for Smartphone Evidence Acquisition. In: Peterson, G., Shenoi, S. (eds.) Advances in Digital Forensics VIII. IFIP AICT, vol. 383, pp. 159–166. Springer, Heidelberg (2012)
6. Aouad, L., Kechadi, T.: Android Forensics: A Physical Approach. In: The 2012 International Conference on Security and Management (July 2012)
7. Quick, D., Alzaabi, M.: Forensic analysis of the Android file system YAFFS2. In: Australian Digital Forensics Conference (December 2011)
8. Rob, P., Coronel, C.: Database Systems: Design, Implementation and Management. Thomson Course Technology (2009)
9. The Epilog SQLite forensic tool, `http://www.ccl-forensics.com/Software/epilog-from-ccl-forensics.html`
10. Drinkwater, R.: Forensics from the sausage factory - An analysis of the record structure within SQLite databases. Technical report (May 2011)
11. Carrier, B.: File System Forensic Analysis. Addison Wesley (2005)
12. Rosenblum, M.: Understanding data lifetime. Stanford University (2006)