SPaRe: Efficient SQLite Recovery Using Database Schema Patterns

Suchul Lee[1], Sungil Lee[2], and Jun-Rak Lee[3]
[1]Korea National University of Transportation
157 Railroadmuseum-ro, Uiwang-si, Gyeonggi-do 16106, Korea (Uiwang Campus)
[e-mail: sclee@ut.ac.kr]
[2]National Security Research Institute
Yuseongdaero 1559, Yuseong, Daejon, Korea
[e-mail: silee@nsr.re.kr]
[3]Kangwon National University
Joongang-Ro, Samcheok, Kangwon, Korea
[e-mail: jrlee@kangwon.ac.kr]
*Corresponding author: Jun-Rak Lee

Received May 28, 2015; revised May 16, 2016; accepted January 16, 2017;
published March 31, 2017

Abstract

In recent times, the Internet of Things (IoT) has rapidly emerged as one of the most influential information and communication technologies (ICT). The various constituents of the IoT together offer novel technological opportunities by facilitating the so-called "hyper-connected world." The fundamental tasks that need to be performed to provide such a function involve the transceiving, storing, and analyzing of digital data. However, it is challenging to handle voluminous data with IoT devices because such devices generally lack sufficient computational capability. In this study, we examine the IoT from the perspective of security and digital forensics. SQLite is a light-weight database management system (DBMS) used in many IoT applications that stores private information. This information can be used in digital forensics as evidence. However, it is difficult to obtain critical evidence from IoT devices because the digital data stored in these devices is frequently deleted or updated. To address this issue, we propose **S**chema **Pa**ttern-based **Re**covery (SPaRe), an SQLite recovery scheme that leverages the pattern of a database schema. In particular, SPaRe exhaustively explores an SQLite database file and identifies all schematic patterns of a database record. We implemented SPaRe on an iPhone 6 running iOS 7 in order to test its performance. The results confirmed that SPaRe recovers an SQLite record at a high recovery rate.

Keywords: SQLite, Recovery, Database Schema, Internet of Things, Digital forensics.

1. Introduction

The Internet of Things (IoT) has emerged in recent times as one of the most dominant information and communication technologies (ICT) [1]. The IoT has consistently been highlighted as one of the top 10 strategic technologies in the Gatner, Inc. reports since 2012. Indeed, we are at the threshold of the so-called “hyper-connected world,” where many objects that surround us will be on computer networks in one form or another. This phenomena is now called the Internet of Everything (IoE)—a network of networks where billions of connections create unprecedented opportunities as well as new risks [2]—as it embraces capabilities such as context awareness, greater processing power, and energy independence, as well as more people, places, and new types of information.

The primary constituents of the IoT, such as smartphones, tablets, wearable devices, smart TVs, and automobiles, communicate with one another. The components of the IoT together provide comprehensive human welfare services through closely connected networks with no mutual interference, where this activity goes largely unrecognized by most people. The fundamental tasks that need to be executed in order to provide such a function involve the transceiving, storing, and analyzing of digital data.

It is challenging to handle the voluminous data involved in IoT by using IoT devices because such devices generally lack sufficient computational capability. Since IoT devices are designed to be portable and energy efficient, they typically use slow CPUs and limited memories, and are sometimes even not equipped with power supplies [3]. Since a large amount of efficient storage is difficult to install on IoT devices, SQLite is often used as primary storage. SQLite [4] is a prominent light-weight database management system (DBMS) used in many IoT applications, such as Android [5], iOS [6], and Skype [7]. Furthermore, such commonly used Web browsers as Firefox, Chrome, and Safari use SQLite. SQLite is a self-contained, serverless, and transactional database engine that many light-weight applications incorporate as their primary storage.

Digital forensics [8] is a branch of forensic science encompassing the recovery and investigation of material found in computer devices. Recent advances in Internet technology have caused the definition of digital forensics to be expanded to encompass an investigation into all devices capable of storing digital data, such as smartphones, surveillance cameras, digital recorders, sensors, network switches, and black boxes. Legally critical evidence, such as the video of a crime, the footprint of a suspect, or a penetrating (hacking) route, can be found through such digital data.

However, it is difficult to obtain critical evidence from IoT devices because digital data stored in such devices is frequently deleted or updated. For example, smartphones save private information, such as recent phone call records, emails, text messages, and access history in their primary nonvolatile storage, i.e., the DBMS. Such data can be accidentally deleted by smartphone users. Moreover, criminals (or suspects) may try to hide their identity or digital footprint by erasing records that implicate them in an illegal activity. Fortunately, there are several techniques to recover deleted DBMS data.

Several approaches to recover deleted SQLite data have been proposed [9-13]. The pioneering research in SQLite recovery was conducted by Pereira *et al*. [9], who introduced a recovery scheme for the Firefox Web browser. The scheme leveraged a journal file that is utilized in data transfer in SQLite. Lee *et al*. proposed a method to recover deleted data in the overflow zone [10]. This method enabled the recovery of deleted data located in an unallocated area of an SQLite database file.

However, prevalent approaches to SQLite data recovery cannot recover an individually deleted record. Since they rely on the structure of an SQLite database file, deleted records can be recovered only in batches. Recovering a separately deleted record is particularly important for digital forensics because criminals (or suspects) might typically steal a specific item of record instead of the entire data to avoid arousing unnecessary suspicion among law-enforcement agencies. However, it is difficult to recover an individually deleted record for the following reasons: (i) The record often cannot be accessed via a logical SQLite pointer. We refer to such data as unreachable. (ii) Even the part of an SQLite database file where a deleted record was previously located can be physically replaced. We refer to such deletions as irreversible.

In this paper, we propose **S**chema **Pa**ttern-based **Re**covery (SPaRe), an SQLite recovery tool that leverages a database schema pattern. In particular, SPaRe exhaustively explores an SQLite database file and identifies each schema pattern of a database record. The identified schema patterns are then utilized to locate records that are unreachable and partially irreversible. SPaRe parses each scratch of a database file to determine if it contains a record by comparing it with the SQLite database schema pattern identified beforehand. We implemented SPaRe on an iPhone 6 running iOS 7 and tested it through extensive experiments. The results revealed that SPaRe can accurately recover every record found in a database file.

The remainder of this paper is organized as follows: Section 2 contains a review of related work in the area, whereas Section 3 and 4 contain details of the data structure of SQLite, where a key principle needed to understand our recovery scheme is introduced. Section 5 is devoted to a description of the design and implementation of SPaRe, and Section 6 contains an assessment of our scheme. Finally, we offer our conclusions in Section 7.

2. Related Work

Related work regarding SQLite recovery broadly falls in the following three areas.

2.1 Data Recovery using *.Journal* File

SQLite creates a *.journal* file as an intermediate product while processing transactional database queries. This file temporarily stores data for intermediate database processing before an actual record is written to the SQLite database file. Pereira *et al*. [9] proposed a method that exploits this file to recover a deleted record. To the best of our knowledge, this pioneering work was the first attempt in SQLite recovery. The work in this study inspired several approaches that utilize a transactional SQLite recovery technique. Unlike [9], our approach relies on the recognition and matching of a database schema pattern.

2.2 Recovery by leveraging SQLite Database File Structure

Jeon *et al*. [11] thoroughly analyzed an SQLite database file structure and noted that deleted records are re-located into unallocated space. Although the method proposed in [11] quickly explores the entire unallocated space through a pointer in the SQLite page header, it cannot identify a deleted record located in the free block. A deleted record in the free block is typically generated by the separate deletion of a record. The recovery of a record of this type is particularly important because the deletion of all data seldom occurs in practice. However, our approach can facilitate access to the free block, and exhaustively explores every database file scratch to find all recoverable records. One may object that an exhaustive search of all the memory is expensive. However, we believe that accuracy and completeness are more significant than computational complexity, especially for information security and digital forensics.

2.3 SQLite Logging File

SQLite writes each transactional operation to a separate logging file. The goal of this logging function is to guarantee integrity during each transaction. Delta-WAL [13] leverages the logging file to perform SQLite recovery. In SQLite, if one or more records are updated, all data, i.e., all records, is re-written. This wasteful update mechanism is improved upon in [13]. Only updated record(s) are written to the logging file, and thus the size of the file is reduced in Delta-WAL [13]. However, Delta-WAL focuses on preserving the integrity of a database transaction and, hence, is orthogonal in purpose to our work here. Our proposed scheme can be extended to utilize a logging file. This will provide an additional opportunity to recover an irrecoverable record. The consideration of this extension is postponed until future work.

3. Analysis of SQLite Database File Format

3.1 SQLite Database File Structure

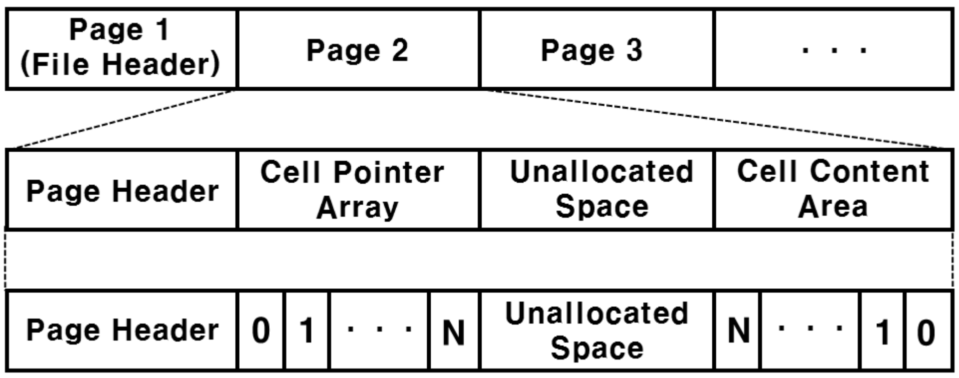


Fig. 1. SQLite Database File Structure

Fig. 1 shows the structure of an SQLite database file, which is divided into several areas of equal size, each of which is called a page. The first page is reserved as an SQLite database file header. Except for the first page, each page consists of a page header, a cell pointer array, unallocated space, and cell content area, as shown in **Fig. 1**. A page is implemented with a B-tree data structure. A database record is physically stored in a B-tree node, called a cell. A cell is located in the cell content area, and SQLite accesses it via a cell pointer located in the cell pointer array. Note that a cell is placed in reverse order (see the index in the cell content area). Therefore, the unused area in a page, i.e., the unallocated space, can be located in the middle. The size of the unallocated space is variable.

3.2 SQLite Data Type

The current implementation of SQLite supports the following data type: null, signed integer, real (IEEE 754-2008 64-bit floating-point number), and text (string and blob). Each data type is designed to consume a minimal amount of memory because SQLite is often utilized in light-weight applications. **Table 1** summarizes SQLite's data types.

Table 1. SQLite data type

Storage class	Serial types	Content Size (Bytes)
Null	0	0
Signed Integer	1~6	1~4, 6, 8
Real	7	8
Text (Blob)	N ≥ 12 and even	(N-12)/2
Text (String)	N ≥ 13 and odd	(N-13)/2

3.3 SQLite Cell Structure

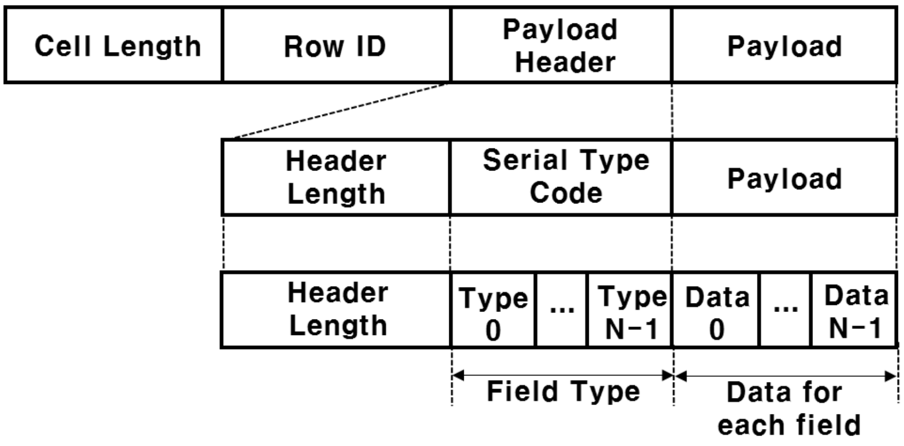


Fig. 2. SQLite cell structure

As described in Section 3.1, an SQLite record is stored in a cell. A cell consists of cell length, a cell identifier, a payload header, and the payload, as shown in **Fig. 2**. An SQLite record is composed of one or more fields. Consider the SQLite database table "telephone book." This table consists of three fields: (i) "abbreviated number" of integer type, (ii) "name" of text type, and (iii) "phone number" of text type. The type of each field, e.g., integer and text, is specified in the serial type code, as shown in **Fig. 2**.

The basic idea underlying SPaRe is that an SQLite record is written to the database file according to the serial type code. This serial type code is generated according to a pre-defined rule, as summarized in **Table 2**. Specifically, the length of the serial type code of the telephone book is 3 bytes because the table consists of three fields. Note that there is no unique serial type code for a table. There are numerous serial type codes for the telephone book, including 0x010C0D, 0x060D0D, and 0x010D0D. We refer to them as schema patterns of a record for the telephone book. Interpreting a schema pattern reveals information regarding the start/end point of each field of record.

Table 2. Form of representation of each data type

Storage class	Representation form		Length of representation form
	Hexa	Decimal	
Null	0x00	0	1
Signed Integer	0x01 ~ 0x06	1~6	1
Real	0x08	8	1
Text(Blob)	N ≥ 0x0C and even	N ≥ 12 and even	1 ~ 9
Text(String)	N ≥ 0x0D and odd	N ≥ 13 and odd	1 ~ 9

4. Deletion and Insertion Operations in SQLite

4.1 Deletion of Record

The deletion of an SQLite record can be implemented as below.

- In the first method, the record is first replaced with zeros. Deletion of this type can be found in current SQLite implementation incorporated in Web browsers, such as Firefox, Safari, and Chrome. If a record is deleted in this manner, no prevalent method can recover the record because no record physically remains in the database file.
- The second method involves physically removing a record, e.g., its raw format. This is typically used for the removal of a particular record in iOS.
- The final method involves freeing the allocated memory and leaving a record in the SQLite database file. Skype, the well-known online chat application, adopts this method. Further, the group removal of text messages or phone call records in iOS is implemented in this manner.

The current deletion implementation for SQLite is summarized in **Table 3**.

Table 3. Implementation of deletion in SQLite applications

Implementation Method	Example Applications
Replacement with zero (0)	Firefox, Safari, Chrome
Physical removal	iPhone OS (iOS) – Typically used for the removal of an individual record
De-allocation of memory area (Free memory)	SkyPe, iPhone OS (iOS) – Group removal of text messages and phone call records

Note that a record that is not physically erased from the database file can be recovered. Here, we detail how a record is deleted after this fashion. Suppose N records are stored in a database as shown in **Fig. 3**. If one or more records are removed (Cases I and II in **Fig. 3**), the corresponding cells are re-assigned as free blocks (see the shaded block in **Fig. 3**). Further, this block remains unchanged until a new record is inserted into it. However, if all data is deleted, the entire page except the page header is re-assigned as unallocated space (Case III in **Fig. 3**). The authors of [11] have shown that a record can be recovered in Case III. In this paper, we show that a record that is deleted in the manner of Cases I and II can be recovered as well.

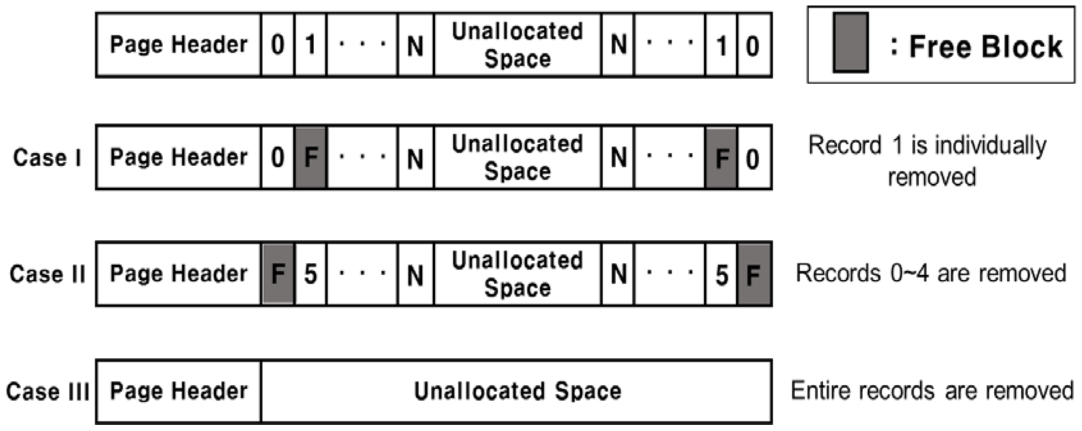


Fig. 3. Deletion of records in SQLite

4.2 Insertion of Record

We now describe the insertion of an SQLite record. This answers the question: "how can a recoverable record exist for a deleted item?"

When a new record item is inserted into the database, SQLite first checks if there is any free block that can store the record, i.e., if there exists a free block of size greater than or equal to the new item. If so, the record (partially) replaces the corresponding free block (Case I in **Fig. 4**). The remainder of the free block is reduced (see F'). Part of the record originally contained in F is now irrecoverable, but there may be a record in F'. Obviously, SQLite can neither locate each removed record, nor figure out how many records have not been replaced in F'. On the contrary, if a free block is unavailable, the new record is inserted into unallocated space (Case II in **Fig. 4**). Note that the data structure of a database file, i.e., each cell pointed to by the corresponding cell pointer, is not broken in any case.

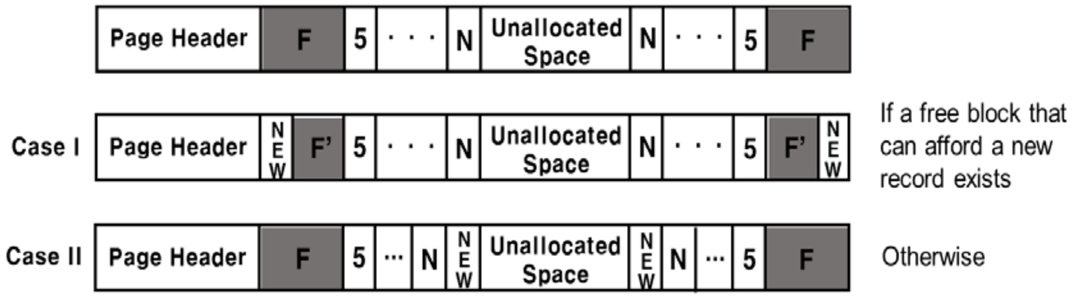


Fig. 4. Insertion of records in SQLite

5. SPaRe: Schema Pattern based Recovery

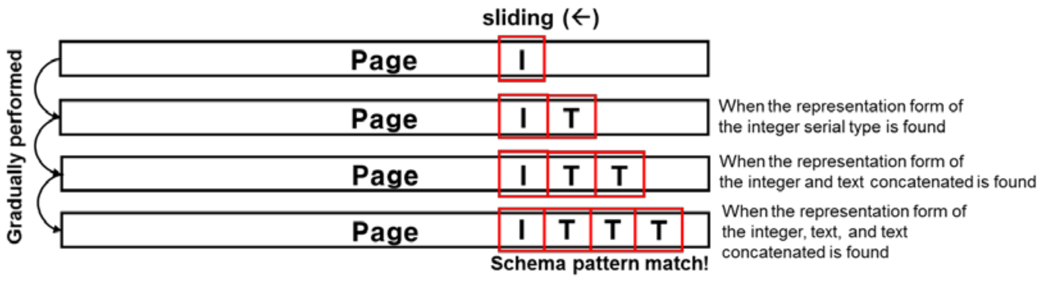

sliding (←)
Gradually performed
Page
I
T
When the representation form of the integer serial type is found
When the representation form of the integer and text concatenated is found
When the representation form of the integer, text, and text concatenated is found
Schema pattern match!

Fig. 5. SPaRe schema pattern parsing

In this section, we describe SPaRe in detail. Consider an SQLite database table "message." This table consists of four fields: (i) date, of integer type, (ii) address, of text type, (iii) msg, of text type, and (iv) data, of text type.
To describe a schema pattern, we use an acronym for each data type: N, I, R, and T are abbreviations for null, integer, real, and text types, respectively. For example, the schema pattern for "message" table is ITTT.
A given record is written in reverse order as described in Section 4. Therefore, SPaRe inversely travels a database file in an exhaustive manner and identifies every schema pattern matched. In order to be matched with a schema pattern, every serial type code should be found in reverse order in the given file. **Fig. 5** shows an example of matching for schema pattern ITTT. Note that comparing against serial type T will not occur until the matching against integer serial type I is successful. We refer to this matching process as "schema pattern parsing."
However, schema pattern parsing is not simple because a schema pattern has no unique representation form, as described in Section 3.3. Further, a hexa-string can have multiple meanings. Suppose 0x02 is found on a page; 0x02 may or may not imply integer type code I, and can be numerical value "2" found in any text. This ambiguity while interpreting a hexa-code increases the complexity of an algorithm. Fortunately, a database record typically has a primary key. The primary key is typically integer type, and is the first field of a database table.
If a schema pattern is found, SPaRe can infer where the corresponding record is stored (see Section 3.3). SPaRe does not know whether the record has been deleted, and returns every record matched. Thus, some of the recovered records may be (partially) broken.
Algorithm 1 describes the pseudo-code of SPaRe's parsing algorithm.

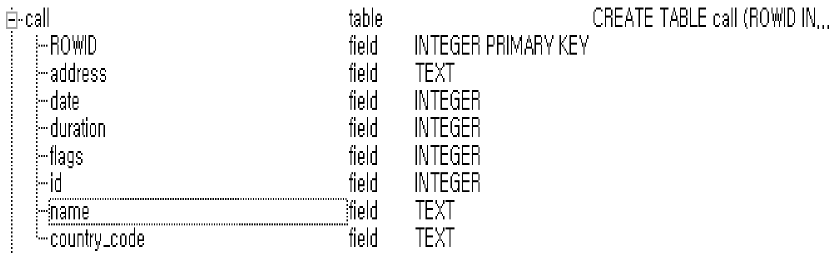
Algorithm 1. Schema pattern parsing algorithm

```
Function Identification of record

Input
   page page
   schema pattern $C_0C_1…C_{N-1}$ /* length of schema pattern is N */

Output
   set of records R
```

```
Begin
  /* exhaustive search in reverse order */
  R ← φ
  L ← length[page]          /* length of page is L */
  For page[position], position ← L-N to 0 do
    parsed ← true;
     iter ← 0;
    While iter ≤ N-1 do
      If Serial_Type_Code(page[position+iter]) != $C_{iter}$ then
        parsed ← false;
        break;
       End If
       iter ← iter + 1;
    End While
     If parsed == true
       R ← R ∪ Record pointed via page[position]
End

Return R;
```

6. Evaluation: A Case Study

5.1 Experimental Setup



```
⊟-call            table     CREATE TABLE call (ROWID IN...
  ├-ROWID         field   INTEGER PRIMARY KEY
  ├-address       field   TEXT
  ├-date          field   INTEGER
  ├-duration      field   INTEGER
  ├-flags         field   INTEGER
  ├-id            field   INTEGER
  ├-name          field   TEXT
  └-country_code  field   TEXT
```

Fig. 6. The database table schema for phone call records

We implemented SPaRe on iPhone 6 running iOS 7. iPhone 6 employs SQLite ver. 3.7. Many operational data types, including phone call records, usage history, text messages, and phone numbers, are written to the iPhone 6 SQLite database file.
To perform the experiment, we set several SQLite database tables: “phone call record,” “text messages,” “phone number record,” “usage history,” and “application data.” **Fig. 6** shows a snapshot of the “phone call record” table. The schema pattern of this table was IT II II TT. Moreover, the first field of the table was ROWID, of integer type, and was the primary key. For each table, we chose some records (not all records) and deleted them. We then performed recovery using the methods proposed in [10][11] as well as SPaRe.
We repeatedly performed 10-fold experiments and computed the average recovery rate.

5.2 Experimental Result

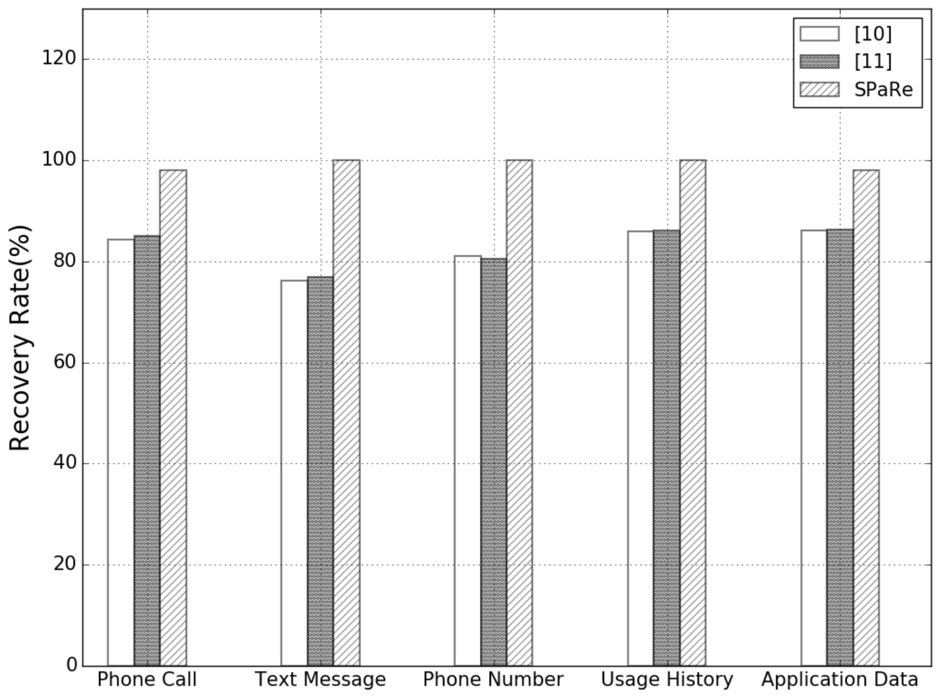


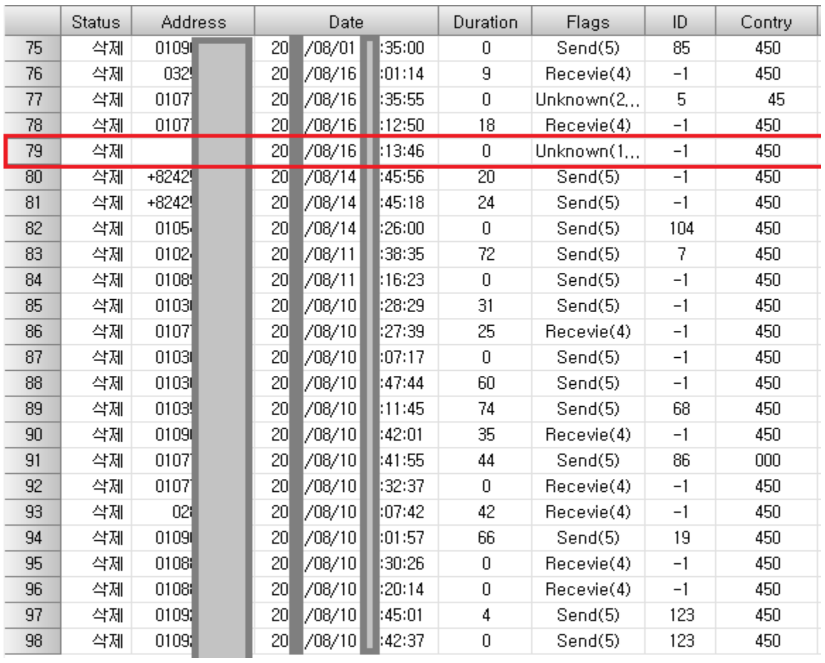
Fig. 7. The experimental results: average recovery rates for each scheme

Fig. 7 shows the results. We observed that SPaRe successfully recovered at least 98% of the removed data, whereas the other schemes [10][11] recovered at most 86% of the deleted records. Particularly for the “text message,” “phone number,” and “usage history” tables, all records removed were recovered by SPaRe. Further investigation of the compared schemes revealed that every record that had not been recovered had been re-located in the free block. This shows that SPaRe can recover a record located in the free block whereas state-of-the-art schemes cannot.

We also observed that, SPaRe was unable to recover some records in the “phone call record” and “application data” tables. **Fig. 8** shows a screenshot of the recovered records in the “phone call record” table, where we see that some records were incorrectly recovered.

For example, there is a partially recovered record (see index 79). SPaRe failed to recover the address and the flag fields in this case. This is the representative case of partial recovery. We manually analyzed the database file, and found that the record of index 79 was in part replaced by another record, resulting in skewed data.

Further, SPaRe might have misinterpreted a text string as a schema pattern. We did not observe this case in our experiment. This happens rarely and, thus, does not significantly affect the accuracy of recovery.



	Status	Address	Date	Duration	Flags	ID	Contry
75	삭제	0109	20 /08/01 :35:00	0	Send(5)	85	450
76	삭제	032	20 /08/16 :01:14	9	Recevie(4)	-1	450
77	삭제	0107	20 /08/16 :35:55	0	Unknown(2...	5	45
78	삭제	0107	20 /08/16 :12:50	18	Recevie(4)	-1	450
79	삭제		20 /08/16 :13:46	0	Unknown(1...	-1	450
80	삭제	+8242	20 /08/14 :45:56	20	Send(5)	-1	450
81	삭제	+8242	20 /08/14 :45:18	24	Send(5)	-1	450
82	삭제	0105	20 /08/14 :26:00	0	Send(5)	104	450
83	삭제	0102	20 /08/11 :38:35	72	Send(5)	7	450
84	삭제	0108	20 /08/11 :16:23	0	Send(5)	-1	450
85	삭제	0103	20 /08/10 :28:29	31	Send(5)	-1	450
86	삭제	0107	20 /08/10 :27:39	25	Recevie(4)	-1	450
87	삭제	0103	20 /08/10 :07:17	0	Send(5)	-1	450
88	삭제	0103	20 /08/10 :47:44	60	Send(5)	-1	450
89	삭제	0103	20 /08/10 :11:45	74	Send(5)	68	450
90	삭제	0109	20 /08/10 :42:01	35	Recevie(4)	-1	450
91	삭제	0107	20 /08/10 :41:55	44	Send(5)	86	000
92	삭제	0107	20 /08/10 :32:37	0	Recevie(4)	-1	450
93	삭제	02	20 /08/10 :07:42	42	Recevie(4)	-1	450
94	삭제	0109	20 /08/10 :01:57	66	Send(5)	19	450
95	삭제	0108	20 /08/10 :30:26	0	Recevie(4)	-1	450
96	삭제	0108	20 /08/10 :20:14	0	Recevie(4)	-1	450
97	삭제	0109	20 /08/10 :45:01	4	Send(5)	123	450
98	삭제	0109	20 /08/10 :42:37	0	Send(5)	123	450

Fig. 8. The experimental results for phone call record—data were anonymized for privacy

6. Discussions

We summarize the advantages of SPaRe from the perspective of digital forensics. Unlike previous approaches, SPaRe can recover a record in the free block. A record located in the free block is often generated via the deletion of an individual record. Since none of the cell pointers indicates the position of this record, it is unreachable and, thus, cannot be recovered using a method that leverages an SQLite database structure. Only exploring every scratch of a database file can solve this issue.

Another advantage of SPaRe is that it can recover a partially broken record because SPaRe directly recovers a record from a database file. If a schema pattern is not broken, SPaRe can recover the corresponding record in any case, irrespective of its integrity. SPaRe can even recover a record if the file system is physically damaged.

However, SPaRe requires exhaustively searching a database file. To implement this, several heuristics can be applied. A possible one involves parsing each serial type in speculative order. Consider a schema pattern TITT. As shown in **Table 3**, parsing serial type I requires fewer comparisons than serial type T. Thus, SPaRe first parses serial type I in advance, and then parses the other serial type codes. We observed that this simple heuristic can significantly improve the performance of SPaRe.

Above all, we believe that computational complexity is not a critical factor in digital forensics, where acquiring critical evidence is most crucial.

We summarize the differences between SPaRe and certain previous approaches in **Table 4.**

7. Conclusion

In this paper, we proposed SPaRe, an SQLite recovery scheme that leverages a schema pattern of a database table. We implemented SPaRe on iPhone 6 running iOS 7 and tested it. The results revealed that SPaRe recovered data from the entire spectrum of recoverability, including (i) data belonging to all unallocated areas (i.e., unallocated space plus free blocks), (ii) partially replaced data, and even (iii) data written on physically damaged storage.

Table 4. Comparison with the state-of-the-art schemes [11]

SQLite Area		**[10]**	**[11]**	**SPaRe**
Unallocated space		recoverable	recoverable	recoverable
Free block	Individually deleted record	Partially recoverable	Partially recoverable	recoverable
	Consecutive records	irrecoverable	irrecoverable	recoverable
Replaced or physically removed		irrecoverable		

8. Acknowledgement

This was supported by Korea National University of Transportation in 2017. The study is supported by 2015 Research Grant from Kangwon National University (No.201510096).

References

[1] Microsoft, “Make IoT real with the Internet of Your Things,” Article(CrossRef Link)

[2] Kortuem, Gerd, et al. "Smart objects as building blocks for the internet of things." *Internet Computing, IEEE 14.1 (2010): 44-51*, December 2009. Article(CrossRef Link)

[3] Sundmaeker, Harald, et al., Vision and challenges for realising the Internet of Things, 2010. Article(CrossRef Link)

[4] A. Grant and O. Mike. The Definitive Guide to SQLite. Apress LP, 2010. Article(CrossRef Link)

[5] Enck, William, et al. "A Study of Android Application Security," *USENIX security symposium*, Vol. 2. 2011.

[6] Apple iPhone, Article(CrossRef Link)

[7] Baset, Salman A., and Henning Schulzrinne. "An analysis of the skype peer-to-peer internet telephony protocol," in *Proc. of IEEE INFOCOM 2006*. Article(CrossRef Link)

[8] Casey, Eoghan. Digital evidence and computer crime: forensic science, computers and the internet. Academic press, 2011. Article(CrossRef Link)

[9] M. T. Pereira, "Forensic Analysis of the Firefox 3 Internet History and Recovery of Deleted SQLite Records," *Digital Investigation*, Vol. 5. No. 3, pp.93-103, 2009. Article(CrossRef Link)

[10] K. Lee, S. Yang, W. Hwang, K. Kim, T, Jang, and G. Son, A Recovery Scheme for the Deleted Overflow Data in SQLite Database", Journal of KIIT, Vol. 10, No. 11, pp.143-153, 2011.

[11] S. Jeon, J. Bang, K. Byun, and S. Lee, "A Recovery Method of Deleted Record for SQLite Database," *Personal and Ubiquitous Computing*, Vol. 16, No. 6, pp.707-715, 2012. Article(CrossRef Link)

[12] S. Lee and H. Yum, "A Recovery Method of Deleted Record Using The Schema Pattern Analysis for SQLite Database," *The workshop for digital forensic technique*, 2011. 8.

[13] J. Lee, M. Shin, Y. Jang, and S. Park. "A Novel Recovery Scheme for SQLite Based on Logical Logging," *Journal of KIIT*, Vol. 12, No. 11, pp. 181-192, Nov. 30, 2014. Article(CrossRef Link)

Suchul Lee received the BS and Ph. D degrees in computer science from Seoul National University, Seoul, South Korea in 2008 and 2014, respectively. He is currently an assistant professor in the Department of Computer Science and Information Engineering at Korea National University of Transportation, Uiwang, Korea. From 2014 to 2016, he was a member of research staff in National Security Research Institute, Daejon, Korea. His research interests include computer and information security, wireless and mobile systems, Internet applications, such as 802.11, cognitive radio, and traffic analysis with an emphasis on system performance optimization.

Sungil Lee received the MS degree in information security from Soonchunhyang University, Asan, Chungnam, South Korea in 2005. He is currently a senior member of research staff in National Security Research Institute, Daejon, Korea. His research interests include computer security and forensics.

Jun-Rak Lee received the B.S. M.S. and Ph.D. degrees in mathematics from In-ha University in 1984, 1986, and 1991, respectively. Since 1995, he has been with Kangwon National University as a professor in the College of Humanities and Social Sciences. His research interests include mathematical analysis, performance optimization, linear algebra, information security, and computer networks.