InnoDB Database Forensics:
Reconstructing Data Manipulation Queries from Redo Logs

Peter Frühwirt, Peter Kieseberg, Sebastian Schrittwieser, Markus Huber, and Edgar Weippl
SBA-Research
Vienna, Austria
Email: pfruehwirt,pkieseberg,sschrittwieser,mhuber,weippl@sba-research.org

***Abstract*—InnoDB is a powerful open-source storage engine for MySQL that gained much popularity during the recent years. This paper proposes methods for forensic analysis of InnoDB databases by analyzing the redo logs, primarily used for crash recovery within the storage engine. This new method can be very useful in forensic investigations where the attacker got admin privileges, or was the admin himself. While such a powerful attacker could cover tracks by manipulating the log files intended for fraud detection, data cannot be changed easily in the redo logs. Based on a prototype implementation, we show methods for recovering *Insert*, *Delete* and *Update* statements issued against a database.**

***Keywords*-InnoDB, digital forensics, databases, log files;**

I. Introduction and Background

When executing a SQL statement, the InnoDB storage engine keeps parts of the statements in several storage locations [16]. Thus, forensic analysis engaging with these locations can reveal recent activities, can help creating a (partial) timeline of past events and recover deleted or modified data [17]. While this fact is well known in computer forensics research and several forensic tools [7] as well as approaches [8], [10], [14], [23] exist to analyze data, the systematic analysis of database systems has only recently begun [11], [12], [15], [22]. Still, to this day, none of these approaches incorporate the data stored in InnoDB's redo logs, which not only constitute a rich vault of information regarding transactions, but even allow the reconstruction of previous states of the database.

Since version 5.5[1] InnoDB is the default storage engine for MySQL databases. It is transaction-safe and supports commits, rollbacks and crash-recovery [3], [21]. Transaction-safe means that every change of data is implemented as an atomic mini-transaction (mtr), which is logged for redo purposes. Therefore, every data manipulation leads to at least one call of the function `mtr_commit()`, which writes the log records to the InnoDB redo log. Since MySQL version 5.1, InnoDB compresses the written data with a special algorithm [5][2]. In our research, we disassembled the redo log files, which are used internally for crash-recovery, in order to identify and recover transactions for digital forensic purposes.

In Section II we describe the general structure of the log files that are used in the course of our analysis, in Section III we detail our approach for identifying recent operations, as well as using the redo information for recovering overwritten data. Section IV gives a detailed demonstration on the capabilities of our forensic method by analyzing example log entries. In Section V we conclude our work and give an outlook to future plans regarding the development of additional methods for recovering more complex statement types.

II. Log File Structure

A. General Structure

As default behavior, InnoDB uses two log files `ib_logfile0` and `ib_logfile1` with the default size of five megabytes each if MySQL is launched with the `innodb_file_ per_table` option activated [4]. Both files have the same structure and InnoDB rotates between them and eventually overwrites old data. Similar to the data files [9], the log files are separated into several fragments (see Figure 1):

1) One *Header block* containing general information on the log file.
2) Two *Checkpoints* securing the log files against corruption.
3) Several *Log Blocks* containing the actual log data.

The header block combined with the two checkpoints and padding is often referred to as *file header* and is exactly 2048 bytes long. Each log block contains a header, a trailer and several log block entries. Since each log block is exactly 512 bytes long, log block entries can be split and stored in two log blocks (see the description of the log block header for further information).

B. Header Block

The first part of the log file consists of the header block, which contains general information about the file. This block has a fixed length of 48 bytes and starts at offset 0x00, i.e.

[1] See http://blogs.innodb.com/wp/2010/09/mysql-5-5-innodb-as-default-storage-engine/

[2] See Appendix A for a description of the algorithm

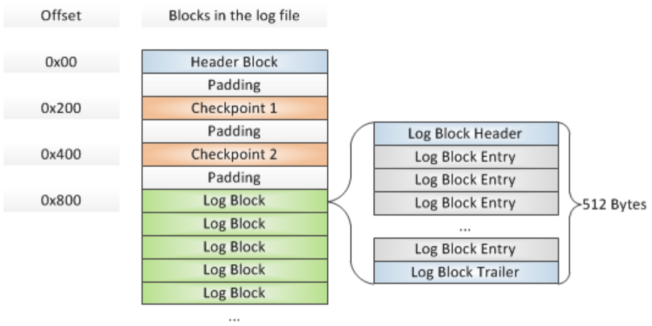

Figure 1. Structure of the log files

at the beginning of the file header. Table I gives an overview on the contents of the header block.

Offset	Length	Interpretation
0x00	4	Group Number of the log file
0x04	8	First log sequence number (lsn) of this log file
0x0C	4	Archived log file number
0x10	32	This field is used by InnoDB Hot Backup. It contains the ibbackup and the creation time in which the backup was created. It is used for displaying information to the user when `mysqld` is started for the first time on a restored database.

Table I
INTERPRETATION OF THE HEADER BLOCK

C. Checkpoints

InnoDB uses a checkpoint system in the log files. It flushes changes and modifications of database pages from the doublewrite-buffer [2], [6], [13] into small batches, because processing everything in one single batch would hinder the processing of SQL statements issued by users during the checkpoint process.

Crash Recovery: The system of checkpoints is vitally important for crash recovery: The two checkpoints in each log file are written on a rotating basis. Because of this method there always exists at least one valid checkpoint in the case of recovery. During crash recovery [1], [18] InnoDB loads the two checkpoints and compares their contents. Each checkpoint contains an eight byte long *log sequence number* (`lsn`). The `lsn` guarantees that the data pages contain all previous changes to the database (i.e. all entries with a smaller `lsn`). Therefore, each change that is not written to the disk has to be stored in the logs for crash recovery or rollbacks. InnoDB is forced to create the checkpoints in order to flush data to the disk [18].

Location in the log files: The two checkpoints are located in the log files `ib_logfile0` and `ib_logfile1` at addresses 0x200 and 0x400 respectively. Every checkpoint has the same structure with a fixed length of 304 bytes. A detailed explanation of the checkpoint structure can be found in Table II. When flushing the log data to the disk, the current checkpoint information is written to the currently unfinished log block header by the method `log_group_checkpoint()` [19].

Offset	Length	Interpretation
0x00	8	Log checkpoint number
0x08	8	Log sequence number of checkpoint
0x10	4	Offset to the log entry, calculated by `log_group_calc_lsn_offset()` [19]
0x14	4	Size of the buffer (a fixed value: $2 \cdot 1024 \cdot 1024$)
0x18	8	Archived log sequence number. If `UNIV_LOG_ARCHIVE` is not activated, InnoDB inserts FF FF FF FF FF FF FF FF here.
0x20	256	Spacing and padding
0x120	4	Checksum 1 (validating the contents from offset 0x00 to 0x19F)
0x124	4	Checksum 2 (validating the block without the log sequence number, but including checksum 1, i.e. values from 0x08 to0x124)
0x128	4	Current fsp free limit in tablespace 0, given in units of one megabyte; used by ibbackup to decide if unused ends of non-auto-extending data files in space 0 can be truncated [20]
0x12C	4	Magic number that tells if the checkpoint contains the field above (added to InnoDB version 3.23.50 [20])

Table II
INTERPRETATION OF THE CHECKPOINTS

D. Structure of the Log Blocks

The log file entries are stored in the log blocks (the log files are not organized in pages but in blocks). Every block allocates 512 byte of data, thus matching the standard disk sector size at the time of the implementation of InnoDB [24]. Each block is separated into three parts: The log block header, data and the log block footer. This structure is used by InnoDB in order to provide better performance and to allows fast navigation in the logs.

In the following subchapters, we discuss the structures of header and trailer records, in Section III we demonstrate how to reconstruct previous queries from the actual content of the log blocks.

1) Log Block Header: The first 14 bytes of each block are called the *log block header*. This header contains all the information needed by the InnoDB Storage Engine in order to manage and read the log data. After every 512 bytes InnoDB automatically creates a new header, thus generating a new log block. Since the log file header containing the header block, the checkpoints and additional padding is exactly 2048 bytes long, the absolute address of the first log block header in a log file is 0x800.

Offset	Length	Interpretation
0x00	4	Log block header number. If the most significant bit is 1, the following block is the first block in a log flush write segment. [20].
0x04	2	Number of bytes written to this block.
0x06	2	Offset to the first start of a log record group of this block (see II-D3 for further details).
0x08	4	Number of the currently active checkpoint (see II-C).
0x0C	2	Hdr-size

Table III
INTERPRETATION OF THE LOG BLOCK HEADER

As described in Section II-C, the currently active log block always holds a reference to the currently active checkpoint. This information is updated every time log contents is flushed to the disk.

2) Log Block Trailer: The log block trailer only contains a checksum for verification of the validity of the log block.

Offset	Length	Interpretation
0x00	4	Checksum of the log block contents. In InnoDB versions 3.23.52 or earlier this did not contain the checksum but the same value as LOG_BLOCK_HDR_NO [20].

Table IV
INTERPRETATION OF THE LOG BLOCK TRAILER

3) Splitting log entries over log blocks: In case a log entry is too big to fit into the remaining space left in the currently active 512-byte log block, it is split over two log blocks. To this end, the currently active block is filled up until the last four bytes that are needed for the log block trailer. A new log block is then generated, holding a log block header and the remaining contents of the split log entry. The offset at position 0x04 and 0x05 in the log block header is used to specify the beginning of the next log entry, i.e. the byte after the end of the split entry (see Table 2). This is needed in order to identify the beginning of the next entry without having to refer to the log block before, thus enhancing navigation in the log file drastically.

III. QUERY RECONSTRUCTION

In this section we demonstrate how to reconstruct executed queries on the basis of information derived from the log files described in the last chapter. As several parts of the data are stored in a compressed form (see Appendix A), it is not always possible to give an exact length definition for each field, since the length of these fields is determined by the decompression routine. These values are marked with a circle symbol (○) in the field “length”. Length definitions containing an asterisk are defined by other fields in the log entry, whereas the number before the asterisk refers to the field where the length was defined.

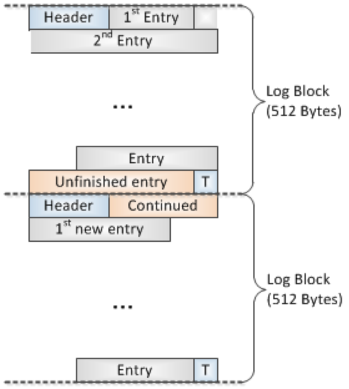

Figure 2. Splitting a log entry over two log blocks

In this paper, we focus on the analysis of InnoDB’s new compact file format, which is recognized by the prefix *mlog_comp_* in the log types. Older versions of InnoDB logs need much more space and are not in the scope of this paper.

In our analysis, we focus on three different basic statements, *Insert*, *Delete* and *Update*, since they form the majority of all log entries. Furthermore they are of main interest in most cases of forensic analysis.

Descriptions of the log entries: Since the lengths, amounts and the positions of the relevant fields inside the log entries are highly variable, we refrain from giving any offsets for the data fields in question. In order to provide a certain amount of clarity, the fields are numbered in ascending order and fields being of the same type (e.g. a variable number of fields containing length definitions) are given the same value.

A. Statement Identification

All log entries can be identified by their *log entry type* which is provided by the first byte of each entry. A complete list of all existing log entry types can be found in the source code [3]. However, for our forensic analysis, all information needed can be harvested from only a few, distinctive log entries (see Table IX).

1st byte	Name	Description
0x14	mlog_undo_insert	Identifies data manipulation statements.
0x26	mlog_comp_rec_insert	Insertion of a new record.

Table V
DISTINCTIVE LOG ENTRIES

For every data manipulation statement, InnoDB creates at least one new log entry of the type mlog_undo_insert.

[3] innobase/include/mtr0mtr.h

This log type stores the identification number of the affected table, an identifier for the statement type (*Insert*, *Update*, *Delete* . . .), as well as additional information that is largely depending on the specific statements type.

Field nr.	Length	Interpretation
1	1	Log entry type (always 0x14).
2	◦	Tablespace id.
3	◦	Page id.
4	2	Length of the log entry.
5	1	Data manipulation type.
. . .	variable	Rest of the log entry, depending on the data manipulation type.

Table VI
GENERAL STRUCTURE OF A MLOG_UNDO_INSERT LOG ENTRY

The most important field for the identification of the statement is the field holding the data manipulation type. In our analysis, we focus on the values for this key parameter shown in Table VII.

Data manipulation type	Description
0x0B	*Insert* statement.
0x1C	*Update* statement.
0x0E	Mark for Delete.

Table VII
ANALYZED VALUES FOR THE DATA MANIPULATION TYPE

The form of each `mlog_undo_insert` log entry is very much depending on the content of the actual statement it represents. Therefore, there is no general structure for the log entries, but every type of entry is represented differently, to allow an economical form of storing the log entries without any padding. In the case of *Update* and *Delete* statements, the remaining `log_undo_insert` log entry specifies the statement completely, whereas in the case of *Inserts*, the `mlog_comp_rec_insert` log entry following the `log_undo_insert` log entry provides information on parameters of the statement.

B. Reconstructing Insert *Statements*

In the case of *Update* or *Delete* statements, most of the information needed is stored in this `mlog_undo_insert` log entry, which is not valid in the case of *Insert* statements. In the course of inserting a new record into a table, InnoDB creates nine log entries in the log files (see Table VIII for an ordered list).

While most of the log entries are not relevant for the forensic analysis outlined in this paper, the `mlog_comp_rec_insert`-log entry (log entry code 0x26) contains a variety of detailed information that can be used to reconstruct the logged *Insert* statement (the identification of the *Insert* statement was done by checking the data manipulation type in the `mlog_undo_insert` entry right before).

Log entry type	Name	Log entry type	Name
0x01	8byte	0x1F	multi_rec_end
0x18	undo_hdr_reuse	0x14	undo_insert
0x02	2byte	0x26	comp_rec_insert
0x02	2byte	0x02	2byte
0x02	2byte		

Table VIII
ALL LOG ENTRIES FOR AN INSERT STATEMENT

Table IX gives a detailed description of the fields found inside the `mlog_comp_rec_insert` log entry for *Insert* statements.

Field nr.	Length	Interpretation
1	1	Log entry type (fixed value: 0x26)
2	◦	Tablespace ID
3	◦	Page ID
4	2	Number of fields in this entry (n)
5	2	Number of unique fields (n_{unique})
6	2	Length of the 1$^{\text{st}}$ unique field (primaryKey).
. . .	2	Length entries for unique fields.
7	2	Length of the last unique field.
8	2	Length of the transaction ID)
9	2	Length of the data rollback pointer
10	2	Length of the 1$^{\text{st}}$ non-unique column.
	. . .	Length definitions for other non-unique columns.
11	2	Length of the last non-unique column.
12	2	Offset
13	◦	Length of the end segment.
14	1	Info and status bits.
15	◦	Origin offset.
16	1	Mismatch index.
17	◦	Length of the 1$^{\text{st}}$ dynamic field like varchar.
	. . .	Length entries for dynamic fields.
18	◦	Length of the last dynamic field.
19	5	Unknown
20	6*	Data for the first unique column.
	. . .	Data for unique columns.
21	7*	Data for the last unique column.
22	8*	Transaction ID
23	9*	Data rollback pointer
24	11*	Data for the last non-unique column.
	. . .	Data for non-unique columns.
25	10*	Data for the first non-unique column.
. . .		

Table IX
MLOG_COMP_REC_INSERT LOG ENTRY FOR INSERT STATEMENTS

The structure of log entries of log entry type `comp_rec_insert` is quite complex. After the first general log entry data fields (log entry type, tablespace ID and page ID), which also define the database table used, two data entries holding information on the columns of the underlying table are provided: n and n_{unique}. n defines the number of data fields that can be expected in this log record, whereas n_{unique} specifies the number of data fields holding primary keys. The number n of data fields is not equal to the number of columns in the table, since definitions for system internal fields like the transaction ID and the data rollback pointer are stored in data fields too.

Following the definition of n_{unique}, the next $2 \cdot n_{\text{unique}}$ bytes are reserved for the definition of the lengths of these unique columns, two bytes for each column. Furthermore, the lengths of data fields holding the transaction ID and the data rollback pointer are defined. The following $2 \cdot (n - n_{\text{unique}})$ bytes hold the length definitions for the columns that do not contain primary keys. It must be taken into account that the length definitions given in the section refer to the lengths defined by the table definition, not the actual length of the inserted data. In case of static data types like *int*, the actual length is always the defined length, however in the case of dynamic data types like *varchar* (containing data of variable length), the above mentioned length definitions only hold the fixed value 0x8000. The actual length of the data to be inserted is defined later in the log entry. Figure 3 shows the context between the length definitions and the data fields.

The following bytes contain various information about the record which is not needed for the reconstruction of the *Insert* statement.

The following fields hold the length information of all columns containing dynamic data types (the length definitions of these columns are filled with the fixed value 0x8000 as mentioned before), each one byte long and in compressed form (see Figure 3). The next five bytes are additional bytes and flags, which are not needed for our forensic approach.

Finally, the content of the inserted record is defined column by column: The first n_{unique} fields hold the data of the primary key columns (lengths of the fields are defined before in the record), followed by one field holding the transaction ID and one field holding the data rollback pointer. These are followed by the $n - n_{\text{unique}} - 2$ fields holding the non-primary key columns, lengths again with respect to the definitions given before at the start of the record. Still, for the correct interpretation of the data fields (especially the data type), knowledge on the underlying table definition is needed, which can be derived from an analysis of the .frm files [9].

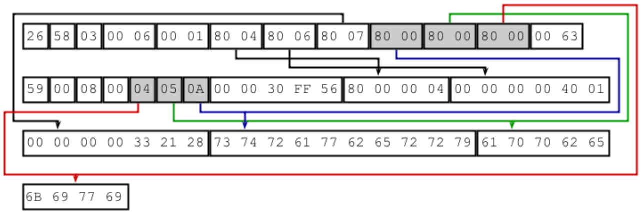

Figure 3. Context between the data fields in a mlog_comp_rec_insert log entry

C. Update

In case of *Update* statements, two log entries are needed for the reconstruction: The `mlog_undo_insert` log entry (which in case of *Insert* statements is only used for determining the statements type) is needed for recovering the data that was overwritten, the following `mlog_comp_rec_insert` log entry is needed for reconstructing the data that was inserted in the course of the *Update*. In this demonstration we focus on *Update* statements which do not change the value of a primary key, since these would result in more log entries and changes in the overall index structure.

1) Reconstruction of the overwritten data: As InnoDB internally stores overwritten data for recovery and rollbacks, we focus on the `mlog_undo_insert` log entry for our forensic purposes.

Field nr.	Length	Interpretation
1	1	Log entry type (fixed value: 0x94).
2	○	Tablespace ID
3	○	Page ID
4	2	Length of the log entry
5	1	Data manipulation type (0x1C = update existing record)
6	2	Table ID
7	6	Last transaction ID on updated field
8	○	Last data rollback pointer
9	1	Length of the primary key
10	9*	Affected primary key
...		
11	1	Number of changed fields
12	1	Field id of first changed field
13	1	Length of first changed field
14	13*	Overwritten data value of first changed field
...		

Table X
MLOG_UNDO_INSERT LOG ENTRY FOR UPDATE STATEMENTS

For an interpretation of the first five fields, please refer to section III-A.

The next two bytes hold a table identifier. This identifier can also be found in the table definition (it is stored in the .frm files at address 0x26). In combination with this information it is possible to derive the name of the table.

The next six bytes hold the transaction identification number and the following compressed field holds the data rollback pointer of the data field. The transaction ID identifies the last executed transaction before the *Update*. By using these references it is possible to reconstruct the complete history holding all changes of a data set, even spanning multiple *Update*s of the same records while maintaining the correct order.

The following fields hold information on the updated primary fields involved. For each primary key, there is a field holding the length of the new value (one byte) and one containing the updated value itself. This is repeated for every primary key of the underlying table, thus it is important to know the number of primary keys for the forensic analysis. The next byte defines the number of non-primary columns affected by the *Update*, therefore the following three fields exist for each updated non-primary column: The id of the changed field, length information on the updated value and the new value for the field.

2) Reconstruction of the executed query: InnoDB creates a mlog_comp_rec_insert log entry containing information on the newly inserted data after the mlog_undo_insert entry, i.e. the updating with new data is logged similar to an *Insert* statement. The created mlog_comp_rec_insert log entry possesses the same structure as the log entry described in Section III-B, thus the only way to distinguish *Update* statements from *Insert*s lies in the evaluation of the mlog_undo_insert entry preceding the mlog_comp_rec_insert entry.

D. Delete

The reconstruction of *Delete* statements is similar to reconstructing *Update* queries. Basically, two forms of *Delete* operations have to be discerned: Physical deletion of a data row and execution of queries, which mark a record as deleted. In the current analysis we only consider the second form, since physical deletion can happen at an arbitrary time.

Log records of statements which mark records as deleted are very short, they usually only generate four log entries. For forensic reconstruction, only the data in the mlog_undo_insert log entry is needed. Table XI shows the log entry for an executed *Delete* statement which is rather similar to the one generated in the course of an *Update* statement without information on the values of the deleted record, except the primary keys involved. Still, these can be identified by using field number 7, the last transaction id on the deleted record. For an detailed interpretation of the log record, please refer to Section III-C.

As a precondition for a correct analysis the number of primary keys of the table needs to be known. Otherwise it is not possible to calculate the number of affected primary key fields (fields 9 and 10). Note that this log record only gives information on the primary key of the record marked as deleted.

Field nr.	Length	Interpretation
1	1	Log entry type (fixed value: 0x94).
2	◦	Tablespace ID
3	◦	Page ID
4	2	Length of the log entry
5	1	Data manipulation type (0x0E = delete record)
6	2	Table ID
7	6	Last transaction ID on deleted record
8	◦	Last data rollback pointer
9	1	Length of the primary key
10	4	Affected primary key
...		
11	3	Unknown
12	1	Length of primaryKey field
13	4	PrimaryKey of deleted field

Table XI
MLOG_UNDO_INSERT LOG ENTRY FOR DELETE STATEMENTS

IV. Demonstration

In this section we demonstrate the techniques outlined in Section III by analyzing real-life log entries derived from a demonstration database.

A. Demonstration database

All examples in this Section are presented with respect to the following table model (listing in Table 1).

Listing 1. Used table structure

```
CREATE TABLE ‘fruits‘ (
  ‘primaryKey‘ int(10) NOT NULL,
  ‘field1‘ varchar(255) NOT NULL,
  ‘field2‘ varchar(255) NOT NULL,
  ‘field3‘ varchar(255) NOT NULL,
  PRIMARY KEY (‘primaryKey‘)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

We used two simple data types (`integer` and `varchar`) in order to demonstrate the procedure of reconstruction. InnoDB stores values of an *integer* field with a fixed length of 4 bytes. The other fields of the type `varchar` have variable lengths, most other data types can be reconstructed in the same way except for the interpretation of their content. For our forensic analysis, knowledge on the exact table structure is required, which can be reconstructed from the table description file (.frm file) [9].

B. Reconstructing Inserts

In our example we use the excerpt shown in Table XII containing a `comp_rec_insert` log entry. In order to improve the clarity of our example, the blocks inside the log entry are distinguished by colors.

0x00000	26	58	03	00	06	00	01	80
0x00008	04	80	06	80	07	80	00	80
0x00010	00	80	00	00	63	59	00	08
0x00018	00	04	05	0A	00	00	30	FF
0x00020	56	80	00	00	04	00	00	00
0x00028	00	40	01	00	00	00	00	33
0x00030	21	28	73	74	72	61	77	62
0x00038	65	72	72	79	61	70	70	62
0x00040	65	6B	69	77	69			

Table XII
EXAMPLE FOR A COMP_REC_INSERT LOG ENTRY

The first entry (containing the value 0x26) marks the entry as `comp_rec_insert` log entry. The two bytes at offset 0x03 and 0x04 denote the number of data fields in this *Insert* statement (0x0006, i.e. 6 data fields), the two bytes at offset 0x05 and 0x06 the number of unique columns (0x0001, i.e. one unique column). Since two of the data fields are reserved for transaction ID and data rollback pointer, we can derive that four columns were inserted, with one being a column containing unique values. The length of the unique column is given in the two bytes at offset 0x07 and 0x08 (encoded as signed integers, thus 0x8004 represents 4) followed by the length definitions for the transaction ID and data rollback pointer (0x8006 and 0x8007 respectively). The length definitions for the three remaining data columns are set to the key value 0x8000, thus denoting columns of dynamic length — the values of the actual data inserted can be found at offsets 0x19, 0x1A and 0x1B respectively (containing the values 0x04, 0x05 and 0xA). Using the length definitions, the rest of the log entry can be split into the data inserted into the table: An unique column containing the value 0x80000004, a transaction ID (signed integer 0x00000000332128) and a data rollback pointer (value 0x00000000332128), followed by the data in the non-unique columns number 3 (value 0x737472617762657272279), number 2 (value 0x6170706265) and number 1 (value 0x6B697769).

Together with knowledge on the table model extracted from the corresponding .frm files, we can derive the correct interpretation of the data fields: The primary key field holds an integer (4), the non-unique columns one to three ASCII-encoded strings (”kiwi”, ”apple” and ”strawberry”). Thus, it is possible to reconstruct the *Insert* statement (see Listing 2).

Listing 2. Reconstructed Insert Statement

```
INSERT INTO fruits
    (primaryKey, field1, field2, field3)
VALUES (4, 'strawberry','apple','kiwi');
```

C. Reconstructing updated data

In this demonstration, we reconstruct data that was overwritten by an *Update* statement. Since, from the logging point of view, an *Update* can be considered as overwriting a data field together with an additional *Insert* statement, we only demonstrate recovering the overwritten data, a demonstration on recovery of the inserted data can be found in Section IV-B.

0x00000	94	00	33	00	1B	1C	00	68
0x00008	00	00	00	00	40	01	00	00
0x00010	33	21	28	04	80	00	00	04
0x00018	01	04	05	61	70	70	6C	65

Table XIII
EXAMPLE OF A MLOG_UNDO_INSERT LOG ENTRY FOR AN UPDATE STATEMENT

0x00000	94	00	33	00	1E	0E	00	66
0x00008	00	00	00	00	28	01	E0	80
0x00010	00	00	00	2D	01	01	10	04
0x00018	80	00	00	01	00	08	00	04
0x00020	80	00	00	01				

Table XIV
EXAMPLE OF A MLOG_UNDO_INSERT LOG ENTRY FOR A DELETE STATEMENT

In our example we use the record shown in Table XIII. After interpreting the header identifying this log entry as an *Update*, the table ID (0x0068) (which is the Table “fruits” according to the .frm file), the last transaction id on the updated field (0x000000004001) and the last data rollback pointer (0x0000332128) can be retrieved. The byte at address 0x13 identifies the length of the value for the primary key field (0x80000004), which is the signed integer representation of 4, i.e. the primary key field with value 4 was updated. Furthermore, we conclude that one (address 0x00018) data field, the fourth (address 0x00019), got changed and that the old value was 0x6170706C65, i.e. “apple”.

D. Reconstructing Deletes

This example refers to the excerpt shown in Table XIV containing a `mlog_undo_insert` log entry. Again, the blocks inside the log entry are distinguished using colors.

Together with knowledge on the table structure, we can reconstruct the query (see Listing 3): The row where the primary key with id one (addresses 0x18-0x0x1B) containing the original value 0x80000001 (addresses 0x20-0x23) was deleted.

Listing 3. Reconstructed *Delete* statement

```
DELETE FROM fruits
WHERE primaryKey=1;
```

E. Prototype implementation

We validated our approach described in this paper with a prototype implementation written in Java. Our tool first analyzes the structure of an InnoDB table based on the its format stored in the table definition file (`.frm`). As described in the paper, the table's structure is ultimately required for further analysis of the redo log files as it is used for calculating offsets in the log files, which are parsed in the second analysis step performed by our tool. We assume a static table structure, thus, `Alter table` statements are not supported in the current version of the tool. The result of the analysis is a history of *Insert*, *Delete* and *Update* statements. Additional types of SQL statements can be added easily because of the modular architecture of the tool. It allows deep insights into the history of a InnoDB table, thus it's main application area is the forensic investigation of a MySQL database using InnoDB as storage engine.

V. Conclusion

In this paper we defined forensic methods for reconstructing basic SQL statements from InnoDB's redo logs. Our techniques make it possible to gain deep insights in the complete history of tables, including the possibility of restoring deleted or updated values. Since InnoDB stores log information for every single transaction, these methods are to be considered powerful allies in the task of reconstructing whole timelines and table histories for forensic purposes. We verified our methods by implementing a prototype for recovering *Insert*, *Delete* and *Update* statements.

For future research we aim at raising the prototype version of our forensic tool to a more enhanced state, which includes the ability of recovering data in more complex scenarios with DDL (Data Definition Language) statements such as *Create Table/Alter Table/Drop Table*.

Appendix

InnoDB uses a special compression method for writing unsigned integers (smaller than 2^{32}), where the most significant bits (msbs) are used to store the length of the data. Table XV gives an overview on the encoding-modes.

First byte	Compressed data
0[rest]	The first byte is interpreted as number smaller than 128.
10[rest]	The first byte is xored with 0x80, this and the second byte are interpreted as number.
110[rest]	The first byte is xored with 0xC0, this and the following two bytes are interpreted as number.
1110[rest]	The first byte is xored with 0xE0, this and the following three bytes are interpreted as number.
1111[rest]	The first byte is omitted, the following 4 bytes are interpreted as number.

Table XV
Compressing unsigned integers

Acknowledgments

The research was funded by COMET K1 and grant 825747 by the FFG - Austrian Research Promotion Agency.

References

[1] Innodb checkpoints (11.08.2010). http://dev.mysql.com/doc/mysql-backup-excerpt/5.0/en/innodb-checkpoints.html.

[2] Mysql performance blog - innodb double write (11.08.2010). http://www.mysqlperformanceblog.com/2006/08/04/innodb-double-write/.

[3] Storage engines (28.07.2010). http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html.

[4] Using per-table tablespaces (11.08.2010). http://dev.mysql.com/doc/refman/5.1/en/multiple-tablespaces.html.

[5] Changes in release 5.1.x (production). http://dev.mysql.com/doc/refman/5.1/en/news-5-1-x.html, 2008.

[6] R. Bannon, A. Chin, F. Kassam, and A. Roszko. Innodb concrete architecture. *University of Waterloo*, 2002.

[7] G. Francia and K. Clinton. Computer forensics laboratory and tools. *Journal of Computing Sciences in Colleges*, 20(6), June 2005.

[8] G. Francia, M. Trifas, D. Brown, R. Francia, and C. Scott. Visualization and management of digital forensics data. In *Proceedings of the 3rd annual conference on Information security curriculum development*, 2006.

[9] P. Frühwirt, M. Huber, M. Mulazzani, and E. Weippl. Innodb database forensics. In *Proceedings of the 24th International Conference on Advanced Information Networking and Applications (AINA 2010)*, 2010.

[10] H. Jin and J. Lotspiech. Forensic analysis for tamper resistant software. *14th International Symposium on Software Reliability Engineering*, November 2003.

[11] P. Kieseberg, S. Schrittwieser, L. Morgan, M. Mulazzani, M. Huber, and E. Weippl. Using the structure of b+-trees for enhancing logging mechanisms of databases. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, pages 301–304. ACM, 2011.

[12] P. Kieseberg, S. Schrittwieser, M. Mulazzani, M. Huber, and E. Weippl. Trees cannot lie: Using data structures for forensics purposes. In *Intelligence and Security Informatics Conference (EISIC), 2011 European*, pages 282–285. IEEE, 2011.

[13] M. Kruckenberg and J. Pipes. *Pro MySQL.* Apress, 2005.

[14] J. T. McDonald, Y. Kim, and A. Yasinsac. Software issues in digital forensics. *ACM SIGOPS Operating Systems Review*, 42(3), April 2008.

[15] M. Olivier. On metadata context in database forensics. *Digital Investigation*, 4(3-4), March 2009.

[16] K. Pavlou and R. Snodgrass. Forensic analysis of database tampering. *ACM Transactions on Database Systems (TODS)*, 33(4), November 2008.

[17] P. Stahlberg, G. Miklau, and B. N. Levin. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2010.

[18] H. Tuuri. Crash recovery and media recovery in innodb. In *MySQL Conference*, April 2009.

[19] H. Tuuri. Mysql source code (5.1.32). /src/storage/innobase/log/log0log.c, 2009.

[20] H. Tuuri. Mysql source code (5.1.32). /src/storage/innobase/include/log0log.h, 2009.

[21] M. Widenius and D. Axmark. *MySQL reference manual: documentation from the source*. O'Reilly, 2002.

[22] P. Wright and D. Burleson. *Oracle Forensics: Oracle Security Best Practices (Oracle In-Focus series)*. Paperback, 2008.

[23] P.-H. Yen, C.-H. Yang, and T.-N. Ahn. Design and implementation of a live-analysis digital forensic system. In *Proceedings of the 2009 International Conference on Hybrid Information Technology*, 2009.

[24] P. Zaitsev. Innodb architecture and performance optimization. In *O'Reilly MySQLConference and Expo*, April 2009.