InnoDB Database Forensics: Enhanced Reconstruction of Data Manipulation Queries from Redo Logs

Peter Frühwirt[a], Peter Kieseberg[a], Sebastian Schrittwieser[a], Markus Huber[a], Edgar Weippl[a]

[a]*SBA Research gGmbH, Favoritenstraße 16, 1040 Vienna, Austria*

Abstract

The InnoDB storage engine is one of the most widely used storage engines for MySQL. This paper discusses possibilities of utilizing the redo logs of InnoDB databases for forensic analysis, as well as the extraction of the information needed from the MySQL definition files, in order to carry out this kind of analysis. Since the redo logs are internal log files of the storage engine and thus cannot easily be changed undetected, this forensic method can be very useful against adversaries with administrator privileges, which could otherwise cover their tracks by manipulating traditional log files intended for audit and control purposes. Based on a prototype implementation, we show methods for recovering *Insert*, *Delete* and *Update* statements issued against a database.

Keywords: MySQL, InnoDB, digital forensics, databases, log files

1. Introduction and Background

When executing a SQL statement, the InnoDB storage engine keeps parts of the statements in several storage locations [1]. 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 [2]. While this fact is well known in computer forensics research and several forensic tools [3] as well as approaches [4, 5, 6, 7] exist to analyze data, the systematic analysis of database systems has only recently begun [8, 9, 10, 11]. 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 [12, 13]. 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 [14][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.

The content of this paper is based on our contribution to the latest ARES-conference [15], where we outlined the basic methods used. In Section 2 we describe the general structure of the log files that are used in the course of our analysis, in Section 3 we detail our approach for identifying recent operations, as well as using the redo information for recovering overwritten data. Furthermore, in addition to the methods outlined in [15], we present methods for gathering the needed basic information on the table structure and keys in Section 4. Section 5 gives a detailed demonstration on the capabilities of our forensic method by analyzing example log entries. In Section 6 we conclude our work and give an outlook to future plans regarding the development of additional methods for recovering more complex statement types.

2. Log File Structure

2.1. 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 [16]. Both files have the same structure and InnoDB rotates between them and eventually overwrites old data. Similar to the data files [17], the log files are separated into several fragments (see Figure 1):

1. One *Header block* containing general information on the log file.

Email addresses: `pfruehwirt@sba-research.org` (Peter Frühwirt), `pkieseberg@sba-research.org` (Peter Kieseberg), `sschrittwieser@sba-research.org` (Sebastian Schrittwieser), `mhuber@sba-research.org` (Markus Huber), `eweippl@sba-research.org` (Edgar Weippl)

[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

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).

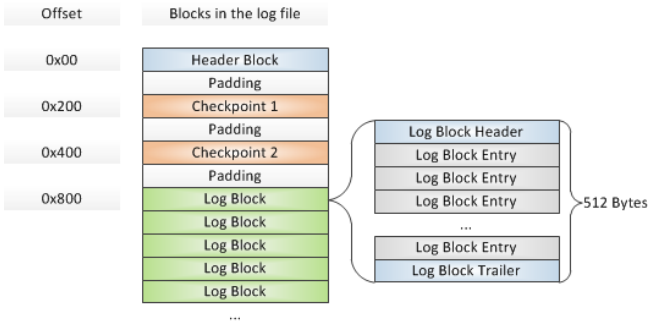

Figure 1: Structure of the log files

2.2. 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. at the beginning of the file header. Table 1 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 1: Interpretation of the header block

2.3. Checkpoints

InnoDB uses a checkpoint system in the log files. It flushes changes and modifications of database pages from the doublewrite-buffer[18, 19, 20] 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 [21, 22] 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 [21].

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 2. When flushing the log data to the disk, the current checkpoint information is written to the currently unfinished log block header.

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()`[23]
0x14	4	Size of the buffer (a fixed value: 2·1024·1024)
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 [24]
0x12C	4	Magic number that tells if the checkpoint contains the field above (added to InnoDB version 3.23.50 [24])

Table 2: Interpretation of the checkpoints

2.4. 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 [25]. 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 3 we demonstrate how to reconstruct previous queries from the actual content of the log blocks.

2.4.1. 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 (see table 3). 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. [24].
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 2.4.3 for further details).
0x08	4	Number of the currently active checkpoint (see 2.3).
0x0C	2	Hdr-size

Table 3: Interpretation of the log block header

As described in Section 2.3, 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.

2.4.2. Log Block Trailer

The log block trailer only contains a checksum for verification of the validity of the log block (see table 4).

2.4.3. 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 Figure 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.

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` [24].

Table 4: Interpretation of the log block trailer

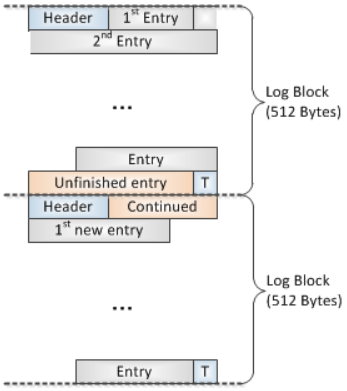

Figure 2: Splitting a log entry over two log blocks

3. 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 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. 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.

3.1. 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 5).

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

Table 5: Distinctive log entries

For every data manipulation statement, InnoDB creates at least one new log entry of the type `mlog_undo_insert`. 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 (see Table 6).

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 6: 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 7.

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.

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

Table 7: Analyzed values for the data manipulation type

3.2. 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 8 for an ordered list).

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

Table 8: All log entries for an Insert statement

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).

Table 9 gives a detailed description of the fields found inside the `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

[3] innobase/include/mtr0mtr.h

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 1st 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 1st 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 1st 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 9: mlog_comp_rec_insert log entry for Insert statements

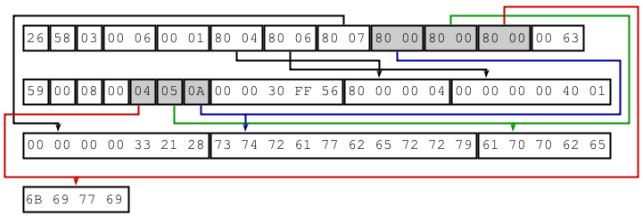

Figure 3: Context between the data fields in a mlog_comp_rec_insert log entry

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 meta information about the record itself which, however, are not needed for the reconstruction of the *Insert* statement. After that, 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) are stored. The following five bytes are additional bytes and flags, which are, again, 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 [17].

3.3. 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.

3.3.1. 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 10: mlog_undo_insert log entry for Update statements

For an interpretation of the first five fields, please refer to Section 3.1.

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.

3.3.2. 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 3.2, 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.

3.4. 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 11 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 3.3.

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 11: mlog_undo_insert log entry for Delete statements

4. Table Reconstruction

In order to be able to reconstruct the queries from the Redo-logs, detailed knowledge on the structure of the underlying tables is of vital importance. In this section we demonstrate, how to reconstruct the table structures from the MySQL table definition files. This analysis is based on the results published in [17].

4.1. General structure of the table definition files

For each table definition inside a database, MySQL generates a separate .frm-file in the databases folder structure called *table name*.frm. The files define most of the characteristics of the table, including key definitions.

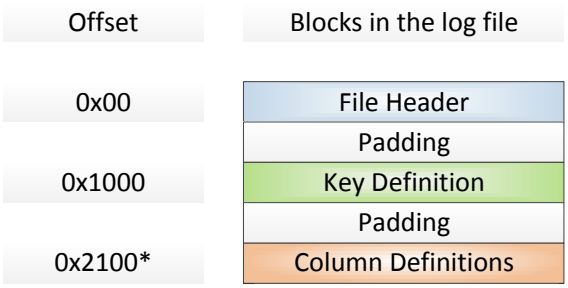

Figure 4: Structure of the .frm-Files

Since the file header does not contain information needed for the reconstruction of the query logs, we will omit it in this paper. A detailed discussion of the contents of the file header can be found in [17].

4.2. Reconstructing Keys

Keys, especially indexes, are very interesting in the course of forensic analysis (e.g. [10]). Furthermore, the definition of primary keys can reveil further important aspects. Figure 6 details the structure of the key block which is starting at offset 0x1000 in the log file.

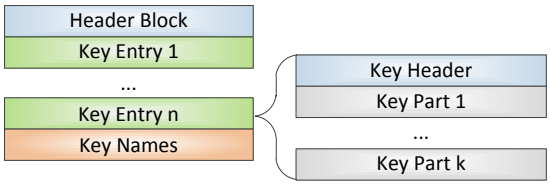

Figure 5: Structure of the Key Block

The key block can be divided in three parts:

1. The header block.
2. The key entries, containing most of the information on the actual key definition
3. A block containing the key names.

Header Block.
The header block is rather short and simple (see Table 12 for a definition). The most important information contained there is k the number of keys in the definition file, which is needed for determining the end of the key block. With this information it is possible to decode the blocks containing the key entries as well as the one containing the key names.

Field nr.	Length	Interpretation
1	1	The number of keys k.
2	2	Flags.

Table 12: The header block

Key Entries.
Since one key can span multiple columns of the table, each key entry is divided into several parts: (i) One key header containing general information on the overall key like id or type and (ii) a key part for each column inside the key (see Figure 6). Table 13 details the content of a key header.

Field nr.	Length	Interpretation
1	2	Flags containing the key-type.
2	2	Key length.
3	1	Number of additional key parts.
4	1	Key algorithm.
5	2	Block size.
6	3	Padding with zeroes.

Table 13: Structure of a key header

The flag field (field one) is of special importance for the forensic analysis of the keys, since it determines the type of the key:

- If the seventh bit of the flag-field is set to one and the name of the key is "PRIMARY", the key definition constitutes the primary key of the table.
- In case the seventh bit is set to one and the name is different from "PRIMARY", the key is of type index.
- If the seventh bit is set to zero, the key defines an unique constraint.

It must be noted, that the key name block is needed for determining the exact type of the key. The second field holds the key length, the third the number of additional key parts for the given key, i.e. the number of columns the key uses minus one. Furthermore, some information on the actual algorithm used for matching the key is stored in field number 4. Still, we will not use this information, as well as the block size in field number 5, in the course of our analysis. The end of the key header is padded with three 0x00-bytes.

The columns affected by the key are defined in the key parts. For each column, there exists a separate key part, the number of these parts per key can be obtained from the key header (field number three). Table 14 details the structure of one key part: The first field holds the id of the used column in the table in an encoded form (column number + 1 + the constant value FIELD_NAME_USED), the second an offset that is not needed for our analysis. The next field gives information on the sort order, followed by a field detailing the type of the key part and the length (both are not needed for our analysis). Each key part is finished with a padding and alignment field holding 0x00.

Field nr.	Length	Interpretation
1	2	Encoded column id.
2	2	Offset.
3	1	Sort order (fixed value 0x00).
4	2	Type of key part.
5	1	Length of key part.
6	1	Padding with 0x00.

Table 14: Structure of a key part

Key Name Block.
The last block contains the names of all keys, the entries are in the same order as the key definitions before, i.e. the first name in this block belongs to the first key entry. The structure of this block is rather simple and straight forward: It starts with a separator (0xFF) and the name of the keys, each separated by 0xFF. The block is finished with a final 0xFF-separator. The key name block is also needed for the definition of the primary key of the table (see Table 13 and the subsequent description).

4.3. Reconstructing the table structure

In the forensic analysis presented in Section 3, detailed information on the actual table structure was needed. In this section, we will give a detailed description on how to retrieve this information.

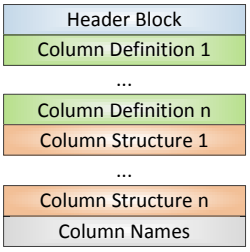

Figure 6: Structure of the column definition block

Figure 6 gives an overview on the structure of the column definitions. Normally, the column definitions start at offset 0x2100 (see Figure 4). Still, in case of many keys, the key definitions don't fit in the space between the offsets 0x1000 (start of key definitions) and 0x2100 (start of column definitions) and need to be moved to a later position. The byte at offset 0x2100 marks, whether the column definitions start here (0x01) or not (else).

Header Block.
The last two bytes of the header block contain the number of columns inside the table, including the number of internal columns set by MySQL itself. Since the header block is not needed any further for the forensic analysis presented in this paper, it is omitted and can be found in [17].

Column Definition Block.
The column definition block can be divided in two sections: First, all internal columns, i.e. columns defined by MySQL itself are defined. Afterwards, all columns defined by the user are appended. This is especially important when looking at the corresponding entries in the column structure block, since these usually don't exist for the internal columns. Table 15 details the structure of the column definition block. This block is mainly concerned with defining the column names and ids, the notation of field lengths follows the same conventions as described in Section 3, i.e. "3*" in the length denotes that the field in question possesses the length defined by the content of field three.

Field nr.	Length	Interpretation
1	1	Column id.
2	1	Unknown
3	1	Length of column name.
4	3*	Column name.

Table 15: The column definition block

Column Structure Block.
The column structure block defines the properties for each column, most of the attributes in this block are concerned

with the data type (see Table 16 for more details).

Field nr.	Length	Interpretation
1	1	Column id.
2	1	Unknown
3	1	Length of column.
4	2	Size of the data type.
5	3	Offset for data pointer.
6	2	Data type property flags and precision.
7	2	Additional property flags.
8	1	Interval id (reference for ENUM and SET).
9	1	SQL-type
10	1	Encoding
11	2	Comment length
12	11*	Comment

Table 16: The column structure block

Field three, the column length, denotes the length of the data in the column (e.g. 10 for int(10), while field four contains the size of the underlying data type (e.g. 4 for int(10)). Field number five contains a pointer to the columns data. Fields six and seven contain flags for the specification of several attributes. The following flags are interesting for our forensic analysis: Bit seven of field six is set to zero in case of unsigned and to one in case of binary data. In case bit nine is set to one, the isZerofilled attribute is set. Bit one of field seven denotes whether null is allowed as a value inside this column. Furthermore, bits three to eight contain the precision in case of numbers, e.g. in case of float(10,3) field three contains 10, field four contains 4 and bits three to eight of field 7 contain 3. Field nine contains the SQL data type as defined in /include/mysql_com.h, field ten the encoding, while fields eleven and twelve deal with addition comments.

Column Names.
At the end of the file, an additional block holding a duplicate of all user-defined column names follows. The structure of this block is very simple, it is lead by one separator-byte holding 0xFF, followed by the names of all columns, each separated by a 0xFF-byte. The block (and thus the whole .frm-file) is completed with the fixed value 0xFF00.

5. Demonstration

In this section we demonstrate the techniques outlined in Sections 3 and 4 by analyzing real-life log entries derived from a demonstration database.

5.1. Retrieving the table structure

The first step of our forensic analysis lies in the retrieval of the table structure and the keys. For this we use a dump of the table "fruits.frm", the relevant parts are shown in the respective paragraphs.

Deriving table rows.
Table 17 shows the parts relevant for the column definition from the file "fruits.frm" (offset 0x2100 to the end of the file). The blocks were colored in order to better visualize their borders.

0x02100	01	00	04	00	5C	00	07	03
0x02108	00	00	04	03	22	00	00	00
0x02110	00	00	00	00	00	00	50	00
0x02118	16	00	00	00	00	00	00	00
0x02120	5C	00	05	04	02	14	29	20
0x02128	20	20	20	20	20	20	20	20
0x02130	20	20	20	20	20	20	20	20
0x02138	20	20	20	20	20	20	20	20
0x02140	20	20	20	20	20	20	20	20
0x02148	20	20	20	20	20	20	20	00
0x02150	04	00	0B	70	72	69	6D	61
0x02158	72	79	4B	65	79	00	05	00
0x02160	07	66	69	65	6C	64	31	00
0x02168	06	00	07	66	69	65	6C	64
0x02170	32	00	07	00	07	66	69	65
0x02178	6C	64	33	00	04	0B	0A	0A
0x02180	00	01	00	00	1B	40	00	00
0x02188	00	03	08	00	00	05	07	48
0x02190	FF	00	05	00	00	00	40	00
0x02198	00	00	0F	08	00	00	06	07
0x021A0	48	FF	00	05	01	00	00	40
0x021A8	00	00	00	0F	08	00	00	07
0x021B0	07	48	FF	00	05	02	00	00
0x021B8	40	00	00	00	0F	08	00	00
0x021C0	FF	70	72	69	6D	61	72	79
0x021C8	4B	65	79	FF	66	69	65	6C
0x021D0	64	31	FF	66	69	65	6C	64
0x021D8	32	FF	66	69	65	6C	64	33
0x021E0	FF	00						

Table 17: File "fruits.frm" from offset 0x2100 to EOF

We start by analyzing the first and last bytes of the header block (red block starting at offset 0x2100). The entry 0x01 at offset 0x2100 identifies the start of the column definitions, the last byte of the header returns the number of columns in the table (0x04).

The following five (four plus one) blocks define the column names and ids (the first column with id 0x02 is an internal column with empty spaces as column name), e.g. (second block) a field with column id 0x04 (offset 0x2150), named "primaryKey" (the byte at offset 0x2152 contains the length of the name (10), the next 10 bytes contain the name itself). Starting at offset 0x217C we find the corresponding column structure entry for column 0x04. From field three we derive that the column length is 10 (0x0A), bit one in field seven tells us that the "NOT NULL" option is set for the column, bits three to eight of the same field

determine that the column possesses no decimal places. Furthermore it is possible to derive the SQL type from filed nine (0x03=int). The column does not possess any comments, thus the field containing the comment length holds the value 0x0000. Repeating this analysis for all entries in the column structure block yields the following SQL-DDL-statement:

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
```

The last block (from offset 0x21C0 to the end) holds a copy of all user-defined column names.

Deriving key definitions.
Furthermore, in order to get a complete description of the DDL-statement, we will analyze the parts of the .frm-file that contain the key definition (see Table 18).

0x01000	01	01	00	00	0A	00	00	00
0x01008	04	00	01	00	00	00	01	80
0x01010	01	00	00	1B	40	04	00	FF
0x01018	50	52	49	4D	41	52	59	FF

Table 18: File "fruits.frm" from offset 0x1000 to 0x1020

The first byte of the header block (offset 0x1000) returns the number of keys for this table (1). The following 11 bytes constitute the key header for this key entry. Since the seventh bit of the first field in the key header is one, we can determine that the key is either a primary key or an index, the third field shows that this key is only containing one single column. The next nine bytes hold the information on the first (and only) key part: With knwledge on the constant FIELD_NAME_USED it is possible to calculate the column id (0x04).

Offset 0x1017 marks the start of the key name block with a 0xFF-separator. The key name is "PRIMARY", together with the information derived from the first field of the key header we can determine that the column with id 4 (and name "primaryKey") is a primary key. Thus we derive the following DDL-statement:

Listing 2: 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`);
```

5.2. Reconstructing Inserts

In our example we use the excerpt shown in Table 19 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 19: 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 0x73747261776265727279), 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 3).

Listing 3: Reconstructed Insert Statement

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

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 20: 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 21: Example of a mlog_undo_insert log entry for a Delete statement

5.3. 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 5.2.

In our example we use the record shown in Table 20. 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".

5.4. Reconstructing Deletes

This example refers to the excerpt shown in Table 21 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 4): The row where the primary key with id one (addresses 0x18-0x0x1B) containing the original value 0x80000001 (addresses 0x20-0x23) was deleted.

Listing 4: Reconstructed *Delete* statement

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

5.5. 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.

To validate the effectiveness of our approach, we measured how long one specific record stays inside one of the log files before it is overwritten by new log data. The methodology of our evaluation setup is as follows: In an iterative process we added records to a simple database table consisting of a MD5 message digest (128 bit stored in a 256 bit `varchar` field) and a primary key of the type `int` with 32 bit length (INSERT statement), followed by a SELECT statement over this table. Note that both statements have an effect on the InnoDB log files as redo logs have to be created for every statement that is executed by the storage engine. We used the default file size of 5 megabytes for InnoDB log files. In a second step, the log files are automatically analyzed and searched for the message digests that previously were added to the table. The described process is repeated until any hash value cannot be found in one of the log files anymore (i.e. when the first inserted database record got overwritten by newer data). In our exemplarily evaluation, we were able to insert approximately 80.000 database records before the first hash value was overwritten in the log files. Depending on the size of inserted, updated or deleted data, the number of recoverable statements in the redo logs can, of course, be lower than in our evaluation. Nevertheless, we were able to show the real-life feasibility of our approach.

The current version of our prototype implementation is not optimized for performance. Due to the choice of programming language we cannot apply optimizations of InnoDB such as pointers within the log file. Nevertheless, our prototype can process a log file with the default size of 5MB in under one minute.

6. Conclusions and Future Work

In this paper we proposed a practical forensic approach for reconstructing basic SQL statements from InnoDB's redo logs based on our work in [15]. This includes the reconstruction of the table's *Create Table* statement together with the defined keys. Thus our method is not only

able to reconstruct the history of tales together with the possibility of restoring deleted or updated values based on a given table definition, but it is also able to retrieve this basic information itself. 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. For verification, we enhanced our existing prototype by integrating all these methods and thus being able to recover information on table creation, as well as emphInsert, *Delete* and *Update* statements.

After enhancing our prototype by implementing methods for retrieving the basic table structure in the course of this paper (as outlined in the future work section in [15]), we plan on adding support for recovering data in more complex scenarios, as well as recovering other DDL statements like *Alter Table*, *Truncate Table* or *Drop Table*. Furthermore, we plan on enhancing the prototype in terms of performance, in order to be able to create a practical, easy-to-use tool set for practical investigations. Another interesting research topic would be the in-depth analysis of InnoDB's B-tree based index due to its forensic value.

Appendix A. InnoDB Data Compression

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 A.22 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 A.22: Compressing unsigned integers

Acknowledgments

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

References

[1] K. Pavlou, R. Snodgrass, Forensic analysis of database tampering, ACM Transactions on Database Systems (TODS) 33 (4).

[2] P. Stahlberg, G. Miklau, 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.

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

[4] P.-H. Yen, C.-H. Yang, 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.

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

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

[7] H. Jin, J. Lotspiech, Forensic analysis for tamper resistant software, 14th International Symposium on Software Reliability Engineering.

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

[9] M. Olivier, On metadata context in database forensics, Digital Investigation 4 (3-4).

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

[11] P. Kieseberg, S. Schrittwieser, L. Morgan, M. Mulazzani, M. Huber, 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, ACM, 2011, pp. 301–304.

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

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

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

[15] P. Fruehwirt, P. Kieseberg, S. Schrittwieser, M. Huber, E. Weippl, Innodb database forensics: Reconstructing data manipulation queries from redo logs, in: The Fifth International Workshop on Digital Forensics, 2012.

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

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

[18] M. Kruckenberg, J. Pipes, Pro MySQL, Apress, 2005.

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

[20] R. Bannon, A. Chin, F. Kassam, A. Roszko, Innodb concrete architecture, University of Waterloo.

[21] H. Tuuri, Crash recovery and media recovery in innodb, in: MySQL Conference, 2009.

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

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

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

[25] P. Zaitsev, Innodb architecture and performance optimization, in: O'Reilly MySQLConference and Expo, 2009.