Analysis of the Internals of MySQL/InnoDB B+ Tree Index Navigation from a Forensic Perspective

Peter Kieseberg
Josef Ressel Center BLOCKCHAINS
Institute of IT Security Research, St. Pölten UAS
St. Pölten, Austria
peter.kieseberg@fhstp.ac.at

Sebastian Schrittwieser
Josef Ressel Center TARGET
Institute of IT Security Research, St. Pölten UAS
St. Pölten, Austria
sebastian.schrittwieser@fhstp.ac.at

Peter Frühwirt
Vienna University of Technology
Vienna, Austria
peter.fruehwirt@broadcom.com

Edgar Weippl
SBA Research & University of Vienna
Vienna, Austria
eweippl@sba-research.org

***Abstract*—In this paper we analyze the structure and generation of InnoDB-indices, as well as navigation in this internal structures with respect to its application in digital forensics. We thus provide an overview on the internal workings of the index that can be used to detect manipulations on the underlying table space. We analyze the physical as well as the logical structure of the index pages and its relation to the very theoretical field of B^+-tree forensics. There we provide a first usable forensic technique targeting a real life open source database system. Furthermore, we discuss several use cases in the course of forensic investigations, as well as applications of the outlined methods, including possible extensions to the field of file system forensics.**

***Index Terms*—databases, forensics, investigation, b-tree**

I. Introduction and Background

Databases are very important parts of modern IT-systems since they allow structured and efficient management of large amounts of data and are used in virtually all large software applications. With the advent of the age of big data, not only the amount of data needed to be incorporated, but also the complexity of the transformation needed to be applied sees a constant growth.

Modern databases need to perform many actions, especially lookups, in a very short timeframe, thus needing to employ various techniques for speeding up the process. The most popular method for enhancing the performance when searching a table is applying indices: A search tree is constructed, tailored for the specific searches that need to get optimized. In case of InnoDB, and most other modern database management systems, a version of the B^+-Tree is employed that is furthermore additionally enhanced for linear searches [1]. $B+$-Trees are also very important in some file systems and forensics in such file systems has been analyzed in [2]–[4]. Utilizing the $B+$-Tree of an index has been proposed in [5], [6], still, this work was rather of theoretical merit: The authors studied how ideal trees, i.e. as defined in the original papers by Bayer et. al. [7], change their structure depending on the order of inserts. While this is interesting from a theoretical point of view, it possesses several limitations when coming to forensics: First, the definition used in these works is a very abstract, mathematical, one that is not employed in real world implementations: Neither did it incorporated the page structure of memory, nor the optimizations for linear searches. Second, it does not include the various optimization routines prevalent in modern databases. Third, many additional effects and artifacts derived from practical implementation are not included, e.g. the fact that deletion in databases usually does not remove the data immediately, but relies on reallocation of pointers while marking the deleted pages as free. Finally, the original approach only works in case of several side parameters: The index must be strict monotonous (ascending or descending) and only INSERT-operations are legal.

In the past, many techniques for providing forensic insight on databases either relied on analyzing the underlying file system for changes [8]–[10] and try to recover old data there [11], [12], or focussed on analyzing external log files provided by the database management system (e.g. ORACLE). For a more comprehensive survey on database forensics see [13], [14].

In this paper we propose new forensic techniques based on the real-life implementation of B^+-trees in the MySQL storage engine InnoDB [15], thus allowing for the development of working tools. Therefore we analyzed the internal workings and mechanisms of the InnoDB index and the underlying B^+-tree, especially considering the effects of insertion and deletion of records from the table the index is built on, as well as general issues regarding internal navigation in the index that can be used for reallocating deleted records. This also includes methods for efficiently hiding data by manipulating the index in order to skip the secret elements when doing ”normal” SELECT-operations.

Summarized, the main contributions of this paper are:

- We demonstrate a detailed view of MySQL / InnoDB index mechanisms and explain internal index navigation.
- We demonstrate practical techniques that can be used for forensics on MySQL databases based on the internal mechanisms of index-structures in InnoDB.

978-1-7281-5912-6/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSSA48308.2019.00013

Authorized licensed use limited to: FH St. Poelten. Downloaded on May 25,2021 at 07:34:23 UTC from IEEE Xplore. Restrictions apply.

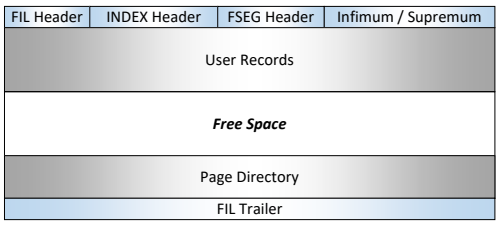

Fig. 1. Physical structure of an INDEX-page

- We analyze the impact of internal data structures like the index on forensic investigations and provide new use cases for future research.
- We adapt a theoretical approach based on the development of the index meta-structure in the course of database operation to the more practically relevant structures used in InnoDB.

II. InnoDB Index

A database index contains many relevant metadata that are crucial in digital investigations. Still today database forensic is not as popular as traditional file based approaches. In this section we take a deeper look into the internal data structures and demonstrate what key data can be obtained using an analysis of the index structures.

InnoDB does not have a separate data row storage structure. It was designed that everything is an "index" thus creates a B^+-Tree for all stored data, first InnoDB creates an index for each primary key and stores the actual row data in this index, further it creates additional indices for each secondary key of each row with its primary key value. All user data are stored in pages of type *INDEX* therefore the index is an important part of InnoDB and this paper takes a deeper look at these structures to enable new techniques for digital investigations.

A. Physical Structure

InnoDB uses two different types of space files [16]: ibdata$X$, which are the system tablespace files, and *.ibd files that are used as a virtual tablespace that is created for each table, i.e. if the *file-per-table* feature[1] is active. These two types have basically the same structure however the system tablespace files contain additional pages that are located at fixed positions (pages 3-7).

All data a table is stored as in a page which is located in the tablespace. The default page size is 16 KiB and contains a FIL-Header (38 Bytes) and a trailer (8 Bytes). These parts, i.e. the header, contains some meta-information about the page i.e. the type of the page which determines the structure of the rest of the page.

Figure 1 gives an overview of the physical structure of an index page. It contains a set of fixed records at a defined position and length (blue), i.e. headers and meta informations, and the actual data (grey) that grows dynamically.

[1]http://dev.mysql.com/doc/refman/5.6/en/innodb-multiple-tablespaces.html

- **FIL Header** (Size: 0x26 Bytes, Offset: 0 Bytes): This header contains meta-information about the page itself (e.g. Log Sequence Number of the last page modification that is very important for other forensic techniques [17], [18] and the space ID), pointers to the next and previous page, which are used during navigation, and checksums.
- **INDEX Header** (Size: 0x24 Bytes, Offset: 0x26 Bytes): The index header contains many data that are related to the index and its record management. Due to its relevance for forensic investigations we will describe all fields of this header in detail below (see Section II-A1).
- **FSEG Header** (Size: 0x14 Bytes, Offset: 0x4A Bytes): The FSEG Header contains pointers to the file segment (fseg) that contents the index.
- **Infimum & Supremum** (Size: 0x1A Bytes, Offset: 0x5E Bytes): The infimum and supremum are special records of the index. The infimum points to the first record of the index in ascending order. On the other hand the last entry of the index points to the supremum, which is used during navigation as a signal for the search algorithm that this was the last data set of the page.
- **User Records** (Offset: 0x78): This section contains all records of this page. The data records are physically persisted unordered but single-linked to each other with a next pointer in an ascending order (see Section II-A2).
- **Page Directory** (Offset: 0x3FF8 Bytes) The page directory grows downwards to the user records starting at the FIL Header. It contains the keys of the records in an ascending order. The number of elements in the page directory is determined by a field in the INDEX Header (see Section II-A3).
- **FIL Trailer** (Size: 0x8 Bytes, Offset: 0x3FF8 Bytes): The FIL Trailer contains a checksum to ensure the integrity of the page and the log sequence number of the newest modification log record to the page (same value as in the corresponding FIL Header)

1) INDEX Header: The INDEX Header is part of an INDEX page and contains data related to the index and its record management, thus can give important information for an forensic investigator. Table I gives a detailed overview of all 0x24 Bytes of the INDEX Header.

2) User Records: User records are added to the page in the order they were inserted. This means that these records are not necessarily physically ordered. There may exist free space in between records, because records were deleted over time. The records are stored as a singly-linked list starting from the infimum and ending with the supremum. All records have a next pointer to the next record in ascending order. Using this list InnoDB can perform a trivial scan trough all users records in ascending order. Note that InnoDB can use the next page pointer in the FIL Header to extend this scan to other pages. We give a detailed explanation of an ascending-order table scan in Section III-C.

Frühwirt et. al described in 2010 the user record format

Authorized licensed use limited to: FH St. Poelten. Downloaded on May 25,2021 at 07:34:23 UTC from IEEE Xplore. Restrictions apply.

Offset	Length	Interpretation
0x00	0x2	Number of slots in the page directory (see Section II-A3)
0x02	0x2	Pointer to the top of the record heap
0x04	0x2	Number of records in the heap
0x06	0x2	Pointer to start of page free record list (first garbage record offset)
0x08	0x2	Number of bytes of deleted records in the page (garbage)
0x0A	0x2	Pointer to the last inserted record, or 0x0000 if this field has been reset by i.e. a delete operation
0x0C	0x2	Last insert direction, i.e. PAGE_LEFT (0x1), PAGE_RIGHT (0x2), PAGE_SAME_REC (0x3)
0x0E	0x2	Number of consecutive inserts to the same direction
0x10	0x2	Number of user records on the page
0x12	0x8	Highest id of a transaction which may have modified a record on the page
0x1A	0x2	Level of the node in an index tree. The leaf level is the level 0.
0x1C	0x4	Index ID where the page belongs

TABLE I
INDEX PAGE HEADER

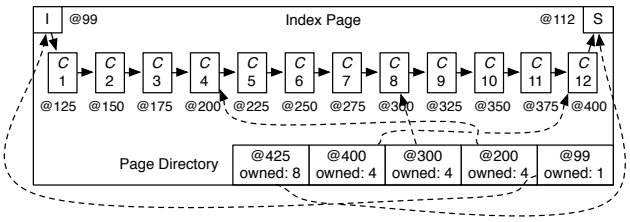

Fig. 2. InnoDB Page Directory

of the InnoDB storage engine [9]. Due to its complexity of the disk format of user records and the limited space we do not give an explanation of the concert record format implementation here.

3) Page Directory: The page directory starts at the beginning of the FIL Trailer and grows upwards (in contrast to the user records which grow downwards to the FIL Trailer). The page directory contains a pointer to every 4-8 user records and a mandatory pointer to the infimum and supremum. The page directory is used for an efficient way to navigate trough the tree. A traversal trough a page by using the next pointer can be very expensive (linear time complexity) especially if you consider that a page may have hundreds of records. Therefore InnoDB stores a pointer to every 4-8 user records in order to use a binary search to find the requested data set (Section III-B). The number of sets in the page directory is defined by the INDEX header at the offset 0x00. Figure 2 shows a schematic overview of the connection of the page directory and the user records ($@x$ denotes a x bytes page offset i.e. the physical data address in the file system).

4) Free Space: The space between the user records and page directory is considered as free space. Thus the user records grows upwards and the page directory downwards these two sections will meet in the middle and exhaust the free space. If no space can be allocated by re-organizing the page by removing the garbage, the page is considered as full.

B. Increasing page size of the B^+ Tree

For better understanding we have calculated the physical size of the B^+ tree. We assume that InnoDB makes perfekt packages, i.e. we achieve this situation by insertion of all values in ascending order without any deletion (write-once) therefore space is wasted with garbage sections. We know that this might not happen quite often in practice, but we use this situation for our discussion. We have created a table with an integer primary key and an additional VARCHAR(32) column. The table does not use any secondary key nor additional indices or constraints. Table II gives an overview of maximum size of the B^+ with a given tree depth.

C. Logical Structure

InnoDB uses a B^+-Tree structure for its indices and navigation. This structure requires a fixed number of reads according to the depth of the tree. This is in particularly efficient if the data does not fit into the memory. According to [7], a B^+-Tree of order n is a balanced tree with the following properties:

- Every non-root node contains between $n/2$ and n elements.
- The root node contains at most n elements.
- An inner node with x elements has got $x + 1$ children nodes.
- All branches of the tree, i.e. all leaf nodes, have the same depth.
- All elements of a leaf are sorted.

An index tree is designed to start at a fixed location with a *root* page, which is used as starting point for navigation. Index pages are called *leaf* pages if they contain actual row data otherwise they were called *non-leaf* (or node) pages. The root page can either be a leaf in case of a very small tree or a non-leaf node. Due to the fact that the tree is balanced, all branches of the tree have the same depth. InnoDB assigned to the leafs level 0 and the level increase on every node going up the tree.

Authorized licensed use limited to: FH St. Poelten. Downloaded on May 25,2021 at 07:34:23 UTC from IEEE Xplore. Restrictions apply.

Depth	Non-leaf pages	Leaf pages	max. Rows	Size
1	0	1	274	16 KiB
2	1	1203	329,90 thousand	18,8 MiB
3	1204	1,45 million	396,86 million	22,1 GiB
4	1,45 million	1,74 billion	1,70 billion	26,0 TiB

TABLE II
$B+$ TREE SIZE AND INCREASING TREE DEPTH

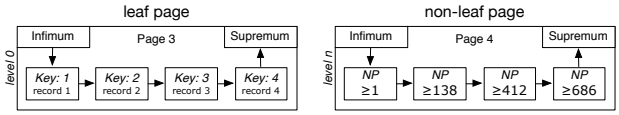

Fig. 3. Comparison of leaf and non-leaf pages

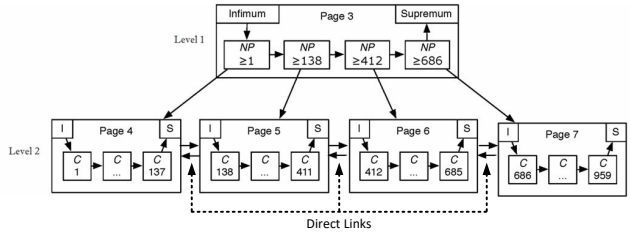

Fig. 4. Index Overview

Each index page contains a special records called *infimum* which is located at a fixed position within the page (offset: 0x63). The infimum record contains a pointer to the first record in the page. In contrast the last record of the page points to another fixed record of the page: the *supremum* (offset: 0x70). Each data record contains a pointer to the next record within the page. A page is a linked list starting with the infimum record and links all records in ascending order by key ending with the supremum record. All records are physically stored in the order of creation, i.e. a record takes the next free space which is available at time of insertion. The order is created by the linking of records within the index.

Figure 3 shows a comparison between the two types of pages. A leaf page contains the actual record data. The records are linked according to their logical order with a next pointer. These pages are assigned to level 0. In contrast the non-leaf pages are located above level 0 and have an identical structure. However these pages contain a pointer to the child page instead of the actual record data.

III. INDEX NAVIGATION

A. Page search

Every search starts basically by locating the page where the queried record is located at. InnoDB uses a B^+-Tree for locating the user records. Every page has a level numbered starting from 0 at leaf pages and incrementing up the tree. All leafs have level 0 and the root node has the highest level of all pages. The root page is located at page 3 of a single tablespace. Figure 4 shows a high-level view of the B^+-Tree structure. Pages on the same level are doubly-linked with their predecessor and successor page ordered by the primary key which is used for range scans (Section III-C). InnoDB is using a simple $B+$-Tree algorithm for the finding page of primary key k using the following steps:

1) Find the root node by loading the first INDEX page.
2) Read the INDEX Header and read level field (Offset: 0x1A). If page is at level 0, exit.
3) Read the infimum record and follow the next pointer
4) Read primary key p_1 and the following primary key p_2 using the next pointer. If $p_1 \leq k < p_2$ or the next pointer points to the supremum, follow the pointer to the next page and return to step 2. Otherwise follow the next pointer and repeat this step.

After locating the page InnoDB uses a different algorithm for locating the requested data record (see Section III-B).

B. In-Page Navigation

As mentioned before all user records within a page are linked as singly-linked list in an ascendent order. A linear search using these next pointer can be very expensive if you take into account that a page may have hundreds of records. Therefore InnoDB is using a page directory for an efficient way to navigate within the page. The page directory is optimized for an efficient search by providing a fixed-length data structure containing pointers to the every 4-8 user records in a sorted way. InnoDB uses a binary search to find the queried record starting at the mid-point of the directory and dividing the page directory with every iteration and therefore reducing the search space. With the remaining 4-8 user records InnoDB uses a linear search from here. Note that the page directory is an array therefore it can be traversed in ascending or descending order. These concrete steps are necessary to locate a user recording using the page directory:

1) Load the INDEX Header an receive the amount of slots of the page directory n (Offset: 0x00)
2) Locate the page directory at beginning of the FIL Trailer (Offset: 0x3FF8) and load the page directory as array (2 Bytes per row, Size: $n * 2$).
3) If the remaining page directory array only contains one element proceed to step 5. Otherwise locate the middle of the directory array.
4) Follow the pointer of the current directory pointer and load the value. If the primary key is smaller as the loaded key, use the first part of the array including the middle, else use the second part and continue with step 3.

Authorized licensed use limited to: FH St. Poelten. Downloaded on May 25,2021 at 07:34:23 UTC from IEEE Xplore. Restrictions apply.

5) Follow the pointer of the page directory slot. Store the n_owned value of the record-header as o.
6) If the key matches with the requested primary key, exit and return current user record. Otherwise repeat this step o-times.
7) Return 'record not found'.

C. Ascending-order table scan

Based on the page structure an full ascending-order table scan is implemented in a trivial way by using the following steps:

1) Find the page of the first data record with the lowest key in the index using the algorithm of Section III-A.
2) Read the infimum record (Page offset: 0x78) and follow its next pointer.
3) If the record is the supremum proceed to step 54 otherwise read and process the record data. Follow the next pointer to the next entry and repeat this step.
4) Read the next page pointer from the FIL Header. If the next page pointer is NULL, exit. Otherwise proceed to the page the pointer and return to step 2.

This procedure can be adapted for range scans by stopping at step 3 if the last user record was located.

IV. Index Insert

InnoDB uses the index navigation algorithms for insertion of new entries. Therefore it is important to know the navigation behavior in detail to understand the internal mechanisms that can be used in further investigations. This section gives a brief overview about how a data records it persisted in the file system. Due to space limitations internal database optimizations like the double write buffer that boosts performance and other methods like transaction handling will be ignored. We will only concentrate on the physical storage of the data records. InnoDB uses the following steps to insert a new record of size s bytes:

1) Locate an existing page using the algorithm of Section III-A.
2) Calculate the free space between the heap top position (0x02) and the page directory (0x00). If the size is enough to fit the record, i.e. is larger than s bytes, locate the free space using the pointer to the heap top position (0x02) and write the record to this address. Further update the index header, i.e. heap top position (0x02), last insert position (0x0A), number of records (0x10), etc. Further proceed with step 6.
3) If the garbage space of this page (0x08) is larger than s, navigate to the first record of the garbage (0x06). If the deleted records has the exact size of the new record, overwrite this record and proceed with step 4. Otherwise use the next pointer (see Section III-B) the locate the next deleted record on this page and redo step 3. If the next pointer is NULL (0x00), i.e. the page has no further deleted records, proceed with step 5.
4) Overwrite the deleted record and update the pointer of the linked list of deleted records. Further update the size of the garbage space (0x08) in the Index Header. Proceed with step 6.
5) There is no free space in this page of this record therefore a new page has to be created. Further the B^+-Tree is updated, i.e. the next and previous page pointers of the neighbor pages are updated. If the requirements are met, a new index page level will be created. This triggers a reorganize of the tree. Insert the record to this new page.
6) Update points to the new record and reorganizes the page directory. The n_owned value of the page directory record will be increased by one. If this value exceeds 8 the page directory slot will be split into half. All page directory slots will be moved to the left (away from the trailer in direction to the record heap). Further the n_owned value of the new record is set to 4 and it becomes a new page directory entry.

V. Forensic Impact

The main purpose of this paper lies in raising these new techniques in InnoDB for providing new methods for forensic techniques. While being rather theoretical, the possibility of detecting manipulations of the underlying data structures in databases provides for very potent mechanisms for defending against unnoticed changes and manipulations, especially considering database administrators who usually possess the rights and means for manipulating any files including log files. In case of criminal investigations, as well as internal attack assessments, the information gained from both, the structure and implementation specific characteristics of the index can serve as evidence concerning actions conducted in the past.

A. Digital Investigations

Digital forensics in databases can serve as a valuable technique for thwarting attacks and attempts of fraud against IT-systems, especially industrial services. Still, to this day, database forensics is not that popular compared to the more traditional file based approach. This is also a result of the large complexity involved, especially due to the uncertainty on the internal mechanisms involved. Furthermore, most proposed techniques are too general in order to be applied to a real world database. Based on the results of this work, the following questions common in digital investigations can be tackled::

- Have there been (illegal) deletions or updates of records? In case of many revision secure databases this means detecting any deletes or updates.
- Have there been any manipulations using other interfaces or even unintended techniques like manipulations in the underlying file system for bypassing the SQL-interface?
- How has manipulated data been changed over the time?

B. Uses Cases and Ideas for future work

The ability to detect changes in the underlying tree structure can be utilized in several forms, not all possessing the same objectives and capabilities. We thus give a short overview on the most prominent uses for this type of control.

Authorized licensed use limited to: FH St. Poelten. Downloaded on May 25,2021 at 07:34:23 UTC from IEEE Xplore. Restrictions apply.

a) Enhancing logging mechanisms: : Storing the structure of the B+-Trees of a database can act as additional logging information that gives additional information on the internal structure of the database tables.

b) More exact backups: : In case a database needs to be replicated on a new server, the B^+-tree holds valuable metadata. Using the approach in this paper, this metadata can be extracted and stored in the backup repository. Since the index is depending on a lot of metadata that are hard to forge without producing inconsistencies, this information is especially valuable for a forensic investigator.

c) Reconstruction of old states: : With knowledge on the structure of the tree in the past it is also possible to reconstruct old states of indices. Together with other data sources (e.g. classical backups only containing the data) it is thus possible to reconstruct a more detailed version of the old state.

d) Detection of manipulations: : A shown in previous work ([5], [6]), the theoretical structure of B+-Trees can be used to give information on the sequence the items were inserted into database tables. The main requirements in the theoretical case are that the index is built in a strict monotonous order (ascending or descending), e.g. by using a timestamp. Still, the index does not need to be complete, i.e. it is valid to leave holes in the set of possible index values (e.g. not for all valid timestamps there is a record in the database).

e) Reconstruction of action sequences: : Since we outlined the techniques for generation of and navigation inside the index, it is possible to detect various actions on the table in the past: E.g. it is possible to detect that entry x was inserted after entry y, based on the position the two entries are actually physically stored. Contrary to the manipulation detection outlined in the precious paragraph this works for all kinds of indices, not only for strictly monotonous ones.

f) Extension to file systems: : Since certain file systems (e.g. NTFS, HFS+, AIX, Ext4, etc.) are using tree-like structures for their internal representation [19], the results from this work can act as a valuable starting point for future research on methods tackling digital forensics in this area.

VI. Conclusion and Perspectives

In this paper we discussed the physical and logical structure of the InnoDB-index and provided methods for digital forensics in the course of investigations based on these insights, which could not be derived from purely theoretical approaches. Furthermore, we raised new use cases and research questions regarding database forensics, that even reach out to the more traditional field of file system forensics.

We conclude that databases are a very promising target from a forensic perspective, not only for theoretical approaches presented in state of the art research, but also in real life implementations using an open source database. Using the results presented in this paper it is possible to read an InnoDB-index and its internal metadata, as well as records marked for deletion. Future work on this area is especially targeting other database management systems, especially finding similar mechanisms in popular closed source products.

Acknowledgements

This research was funded by the Austrian Research Promotion Agency (FFG) COIN project 866880 ”Big Data Analytics (BDA)” as well as the Josef Ressel Center for Blockchain Technologies & Security Management (BLOCKCHAINS) and the the Josef Ressel Center for Unified Threat Intelligence on Targeted Attacks (TARGET). The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

References

[1] T. Lahdenmaki and M. Leach, *Relational database index design and the optimizers*. Wiley-Interscience, 2005.

[2] H. Lu, Y. Y. Ng, and Z. Tian, “T-tree or b-tree: main memory database index structure revisited,” in *Database Conference, 2000. ADC 2000. Proceedings. 11th Australasian*, 2000, pp. 65–73.

[3] P. Koruga and M. Baca, “Analysis of b-tree data structure and its usage in computer forensics,” in *Central European Conference on Information and Intelligent Systems*, 2010.

[4] G. Miklau, B. N. Levine, and P. Stahlberg, “Securing history: Privacy and accountability in database systems.” in *CIDR*. Citeseer, 2007, pp. 387–396.

[5] 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*. IEEE, 2011, pp. 282–285.

[6] 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,” *International Journal of Web Information Systems*, vol. 9, no. 1, pp. 53–68, 2013.

[7] R. Bayer and E. Mccreight, “Organization and maintenance of large ordered indexes,” *Acta Informatica*, vol. 1, pp. 173–189, 1972.

[8] H. Beyers, M. Olivier, and G. Hancke, “Assembling metadata for database forensics,” in *Advances in Digital Forensics VII*. Springer, 2011, pp. 89–99.

[9] P. Frühwirt, M. Huber, M. Mulazzani, and E. R. Weippl, “Innodb database forensics,” in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. IEEE, 2010, pp. 1028–1036.

[10] D. Litchfield, “Oracle forensics part 2: Locating dropped objects,” *NGSSoftware Insight Security Research (NISR) Publication, Next Generation Security Software*, 2007.

[11] A. Grebhahn, M. Schäler, and V. Köppen, “Secure deletion: Towards tailor-made privacy in database systems.” in *BTW Workshops*, 2013, pp. 99–113.

[12] P. Stahlberg, G. Miklau, and B. N. Levine, “Threats to privacy in the forensic analysis of database systems,” in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007, pp. 91–102.

[13] M. S. Olivier, “On metadata context in database forensics,” *Digital Investigation*, vol. 5, no. 3, pp. 115–123, 2009.

[14] P. Frühwirt, P. Kieseberg, K. Krombholz, and E. R. Weippl, “Towards a forensic-aware database solution: Using a secured database replication protocol and transaction management for digital investigations,” *Digital Investigation*, vol. accepted for publication, 2014.

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

[16] N. Son, K. Lee, S. Jeon, S. Lee, and C. Lee, “The effective method of database server forensics on the enterprise environment,” *Security and Communication Networks*, vol. 5, no. 10, pp. 1086–1093, 2012.

[17] P. Frühwirt, P. Kieseberg, S. Schrittwieser, M. Huber, and E. Weippl, “Innodb database forensics: Enhanced reconstruction of data manipulation queries from redo logs,” *Information Security Technical Report*, vol. 17, pp. 227–238, 2013.

[18] ——, “Innodb database forensics: reconstructing data manipulation queries from redo logs,” in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*. IEEE, 2012, pp. 625–633.

[19] B. D. Carrier, “Risks of live digital forensic analysis,” *Communications of the ACM*, vol. 49, no. 2, pp. 56–61, 2006.

Authorized licensed use limited to: FH St. Poelten. Downloaded on May 25,2021 at 07:34:23 UTC from IEEE Xplore. Restrictions apply.