USING INTERNAL MySQL/InnoDB B-TREE INDEX NAVIGATION FOR DATA HIDING

Conference Paper · January 2015

DOI: 10.1007/978-3-319-24123-4_11

Using Internal MySQL/InnoDB B-Tree Index Navigation for Data Hiding Purposes

Peter Frühwirt, Peter Kieseberg and Edgar Weippl
{pfruehwirt|pkieseberg|eweippl}@sba-research.org
Favoritenstraße 16, 1040 Vienna, Austria

Abstract

Large databases offer a very interesting environment for data hiding: They store a lot of different data, are riddled with internal mechanisms and data pools for enhancing performance and a lot of complex optimization routines are constantly changing parts of the underlying file environment, thus making them inviting for hiding data. Furthermore, the database itself can be a very valuable target where an attacker wants to manipulate search results of simply hide traces of an attack. Although there exist many reasons for hiding data in databases, research on this topic has been rather low in the past years. In this paper we demonstrate the feasibility of data hiding and propose several techniques in MySQL, as well as demonstrate the impact of the implementation of data deletion with respect to forensic analysis.

Keywords Database Forensics, Data Hiding, Data Deletion, Index, InnoDB, MySQL

1 Introduction

With the rise of BigData-analysis, the amount of data needed to be stored and accessed with high performance grows rapidly. Databases provide means to enhance the performance of operations, especially searching, and act as one of the main software systems enabling reconciliation and linking of large data sets, additionally allowing for many complex operations to be included. Thus, many larger corporations use data warehouses for storing their data, including workflow engines for the automated enrichment of raw source data with operational information, or data from other data streams. These databases are often very large and form the data basis for many important tasks inside the corporation, e.g. billing.

Large databases are a perfect place for hiding data, since they constitute a central and thus usually trusted part of the IT-environment. This trust is usually justified by applying automated control systems, sanity checks or audits on the higher software layers, usually on the input data, as well as on results. The database itself usually works as a black box due to its high level of complexity, amount of data, throughput and intransparent internal operations, often introduced in order to enhance performance. This also results in a lot of background noise when looking at the deeper layer of file operations which effectively prohibits classical forensic approaches. Furthermore, sensitive data often already resides inside the database, thus making it more easy to hide instead of having to extract, store and hide it somewhere else.

When discussing techniques for hiding data, two different types have to be taken into consideration: Data removal and data disguise. While the first deals with matters of effectively making data inaccessible without leaving traces, the latter is concerned with hiding information in other, unsuspicious looking data, e.g. using steganographic techniques. There exists a very important contrast between data hiding and cryptography:

> ”Its goal is not to restrict or regulate access to the host signal, but rather to ensure that embedded data remain inviolate and recoverable.” [1]

The main requirements for data hiding techniques can be summed up as follows [1]:

- They must regulate access to the hidden/embedded data.
- The hidden data must be recoverable.
- The integrity of the hidden data must be ensured.

The main contributions of this paper focus on proposing several techniques for data hiding in database management systems using index manipulation. Furthermore, we propose a novel approach for evaluation of data hiding techniques in database systems. In order to show practical feasibility of our techniques, we give an overview of MySQL / InnoDB index mechanisms and explain the data deletion process in detail.

2 Background and related work

Databases not only store large amounts of information, additionally a lot of meta information is generated in order to facilitate fast searching and other operations on the table data. Thus, a lot of additional space is allocated invisible to the actual user of the database, which, nevertheless, is affected by operations and internal mechanisms. In this section we give a short outline on the previous work regarding the usage of database meta information for hiding data, as well as on the very much related subject of database forensics, especially focussing on forensics on the index structure.

Traditional database forensics is mainly focussed on analyzing the underlying filesystem layer for recovery of changed files [6, 14]. Furthermore, internal mechanisms for guaranteeing correctness of the database and providing rollback functionality can be utilized for forensic purposes as shown in [5, 4].

Regarding the construction of indices, in [10] the authors give very detailed reviews on the internal workings of indices, both, on a very generic level, as well as related to many existing database management systems (DBMSs) and the underlying storage engines. They also go into very much detail regarding the efficient implementation of database indices, which is one of the basic requirements for constructing slack spaces working in real life implementations. A further analysis of the internal workings of index trees can be found in [11], where also the possibility of using these structures for forensic mechanisms is touched slightly. In [9] the authors outline how B-Trees can be used for forensic on a FAT32 file system by searching for remnants of deleted data in the underlying navigation tree. Another approach utilizing the index tree for forensics has been discussed in [12]: Since the B^+-Tree for a given set of elements is not unambiguous, the exact structure is depending on the order the elements were inserted into the tree. Thus the authors of [8] were able to outline several scenarios where manipulations on the indexed data in a database could be detected by studying the structure of the underlying B^+-Tree. Furthermore, based on these results, the authors proposed a new logging mechanism in [7].

In [13] the authors proposed some practical approaches for data hiding techniques in databases on PostgreSQL-specific implementations. Here the authors focus on techniques based on the SQL-interface for hiding structures, which makes them very easy to implement for the attacker, but also rather easy to detect in case of professional investigations. In contrast, this paper focusses on providing techniques for hiding data deeper inside the internal mechanisms, thus making detections much harder. Furthermore, we provide several layers for manipulating the result set of queries, thus allowing for targeted manipulation of e.g. automated queries, without changing the results of manual investigations or audit routines.

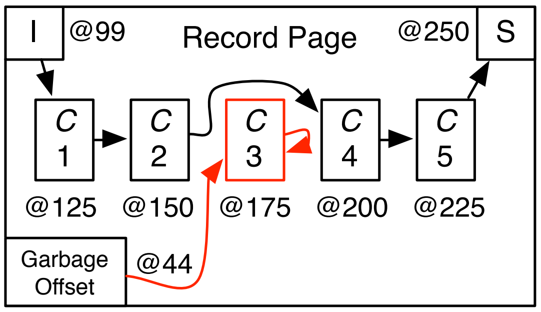
3 InnoDB Index

In InnoDB the data itself is stored in the form of a index tree, based on the primary index, which is thus mandatory for every table. Data and primary index are thus closely intertwined and directly affect each other, which is the major difference to secondary indices, which solely exist for the purpose of speeding up specific searches. When creating a table, InnoDB generates an index for the primary key (if no column is specifically selected, an auto-incremented id-column will be generated) and stores the actual data records directly inside the B^+-tree structure of the index. Furthermore, additional index trees are generated for each secondary key defines which hold pointer to the respective page in the primary key.

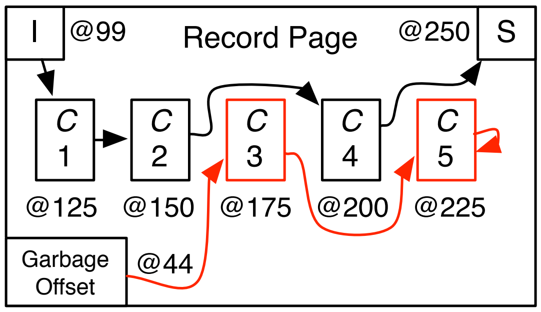
InnoDB uses a $B+$-Tree for locating pages. The first INDEX page within the tablespace (page 3) is called the *root node*, all actual data (keys and data records for the primary, keys and the corresponding links to the primary key page in case of secondary indices) is stored in the leaf nodes of the tree. All other pages, i.e. inner nodes of the tree, are only used for navigation and do not contain any user records. Still, in very small tables the root node itself is the (only) leaf node. All elements in the leaf nodes are sorted, thus implemented as a singly-linked list. For faster navigation within a page InnoDB uses a so called *page directory* that directly links to every fourth to eighth element.

The index is physically stored in pages, which are containers with a size of 16 KiB. These pages are stored in a so called *tablespace* that is stored in the ibdataX files (global tablespace) or *.ibd files if the file-per-table feature is active. Each INDEX page contains a *FIL Header* that contains meta-information about the page itself, an *INDEX Header* with many data related to the index, a *FSEG Header* with certain pointers, infimum- and supremum-records and a *FIL Trailer* that contains checksums.

The user records are located right after the different headers within the page and are physically stored in the order of their insertion. *Next pointers* are used for each record to create an ordered singly-linked list, where the *infimum-record* points to the first record in



(a) One deleted record



(b) Two deleted record

Figure 1: Pointer to deleted records within a page

ascending order. The user records use the next pointer field to link to the next entry in ascending order, while the next pointer of the last user record points to the *supremum-record* which is the signal for the index navigation algorithm that all records of the page were read.

4 Data Removal

Due to performance reasons InnoDB does not delete a record physically. Frühwirt et. al showed that data records stay present after deletion by using delete-flags [2]. These garbage records will be overwritten in the future if the space is needed.

4.1 Physical deletion of data records

As previous mentioned in Section 3 InnoDB uses a singly-linked list for navigation within the page. In detail, InnoDB utilize two Index Header Fields: A pointer to the start of the page free record list and a field that stores the number of bytes of deleted records. Figure 1 demonstrates the deletion process of a data record ("@x" denotes a x bytes page offset i.e. the physical data address in the file system). The garbage offset points to the first deleted record on the current page. Similar to the stored data records, InnoDB utilizes a singly-linked list for its deleted records, the last deleted record points to itself, which signalizes the end of the list.

If a record is deleted InnoDB changes the deleted-flag to 1. Furthermore it updates the next pointer of the previous record in ascending order and points to the next record or the supremum if the last record on the page is deleted. In addition the Index Header is updated, i.e. the garbage size field will be increased by the size of the deleted record and the pointer to the last inserted record gets overwritten with 0x00000 (Offset: 0x0A). Internally the last record of the deleted record list will now point to the currently deleted record instead of pointing to itself.

4.2 Forensic impact of the deletion process

Frühwirt et. al demonstrated in previous research that physically deleted records can be recovered by directly reading the file system [2, 4, 3]. In context of reflecting of the index and the actual algorithm further observations are possible.

Timeline Analysis Due to its design it is possible to reconstruct the sequence of deletion using the next pointer of the singly-linked page free record list. A new deleted record will be added to the end of the list.

Data Retention InnoDB replaces deleted records in the page record free list only if certain conditions are matched: First of all a new record must be on the same page as the deleted record which is determined by the structure of the B^+-Tree. When considering an auto-incrementing table this is not likely, because new data records will only be on the last page caused by the incrementing primary key. If a new record is assigned to the page, InnoDB iterates over the page record free list to find a deleted record with the exactly the same size. If one requirement is not met, InnoDB creates a new page and will not overwrite the deleted records. This method is on the one hand very efficient - which is important in a DBMS - on the other hand it creates a long retention time of deleted records, which is good from a forensic point of view. Only a complete table recreation or reorganization will force InnoDB to overwrite deleted records.

Slack Space InnoDB heavily uses pointers for navigation within pages. It is therefore possible to manipulate the pointers to create areas that are not accessed by the storage engine, thus making it possible to create some kind of slack space within the storage files. We will use this characteristic to hide data in Section 5.

5 Data Hiding

In this section we outline, how the specific structure and especially the removal mechanism of the index can be used in order to generate slack space where data can be hidden. Furthermore, we will outline how to recover the hidden data. The data that shall be hidden will be named *secret* data in the following section. We will propose several approaches for hiding the secret data by utilizing the index and discuss their benefits and shortcomings.

5.1 Manipulating search results

In large scale databases data is usually retrieved with the help of secondary (search) indices in order to provide the needed performance. This dependency on the index can be used to hide data by making it invisible to commonly used searches, without actually removing the data from the table. This works by unlinking the index entries that point to the secret data in the table from the rest of the index, without modifying the underlying table. In case this is done for all relevant searches and their indices, the data will not be retrievable in normal operation, but can be accesses by choosing select-statements that do not use any modified or any index at all.

In InnoDB there exist two types of indices, primary indices, where exactly one exists per table and which is applied on the primary key of a table, and so called secondary indices,

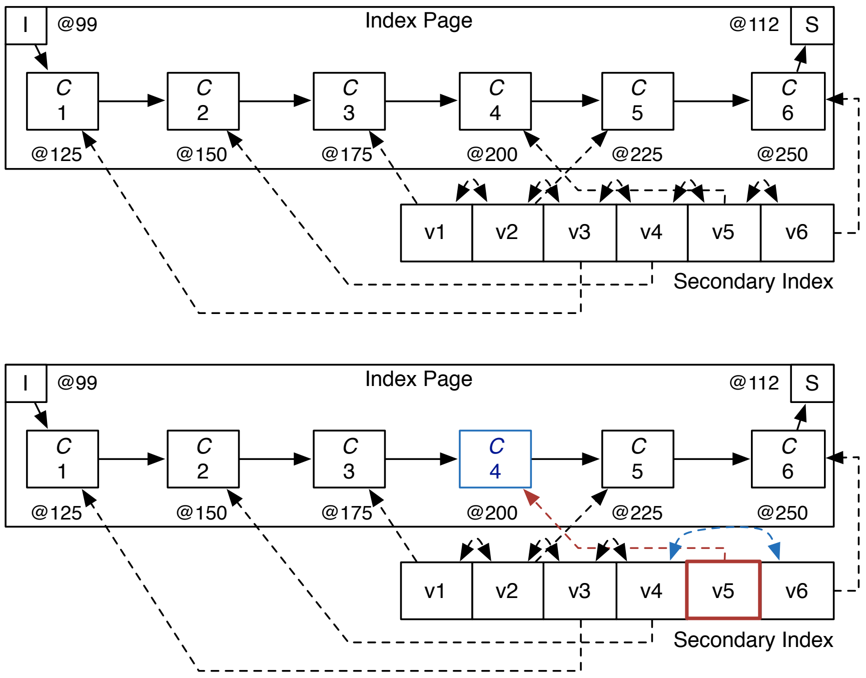


Figure 2: Manipulating search results

which are used for enhancing the performance of search operations. In this approach, the primary index is left unchanged and only secondary indices are modified. This is reasonable, since the primary index is usually an auto-incremented unique field that is not used for actual data retrieval in large databases.

Figure 2 gives an overview on how to manipulate a secondary index for data hiding: The index contains a copy of the indexed columns with pointers to the actual record page, where the records are stored. In order to hide a record, the links in the secondary index leading to and from this record by its neighbors are removed and exchanged for a direct link between the two, e.g. v_8 in Figure 2, will be unlinked from the tree structure (red arrows) and a new link will be set instead, linking the former neighboring records, e.g. linking v_5 and v_6. Thus, the record will no longer exist in the search tree (secondary index), still, the data has not been removed from the primary index.

In order to hide data, the following general approach can be chosen:

1. The table containing the hidden data is generated or selected. The primary index should be chosen in a way that makes it unsuitable for normal searches, e.g. by adding a generic auto-incrementing id-column that possesses the uniqueness property.

2. Secondary indices are generated for all SQL-queries that are used for accessing the table data during normal operation.

3. The secret data is written to the table in the form of table entries.

4. The links to the secret data are removed from the secondary indices by applying the approach outlined before. It is vital that manipulating the page index is not forgotten

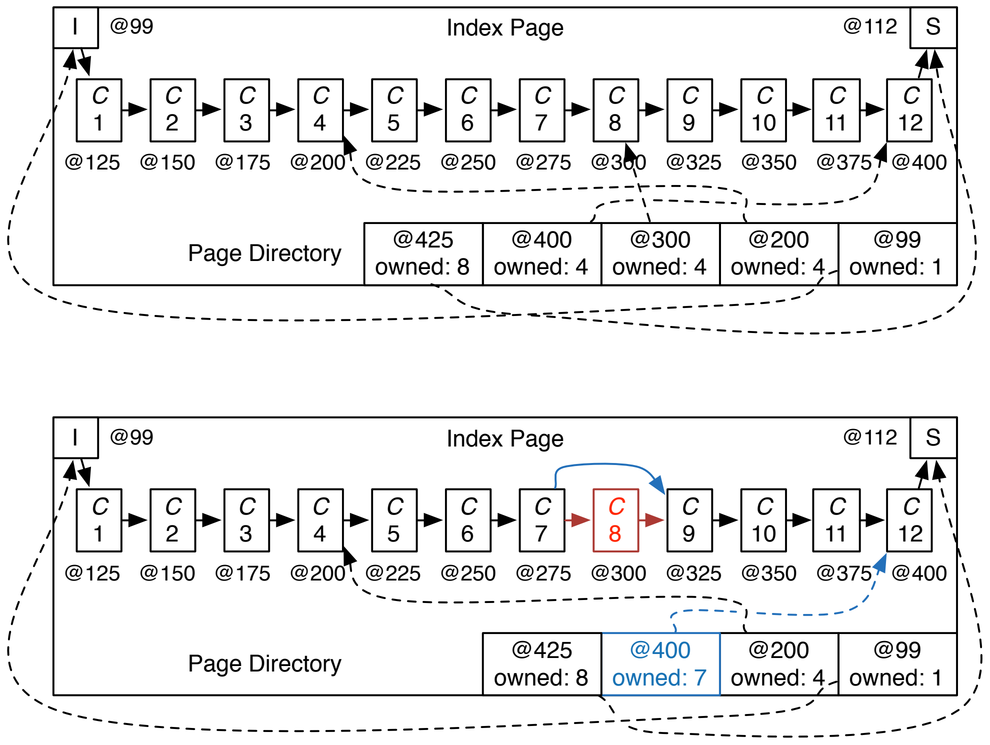


Figure 3: Reorganizing the index

in this step.

5. In case of retrieval, the primary index or an unmanipulated secondary index is used to access the hidden data.

The main drawback of this approach lies in the fact that the data can still be found in the table by searching by the primary index, or with unindexed searches. Still, in many real life applications, e.g. in the area of data warehouses, all relevant queries for extraction workflows are indexed in order to provide the performance needed for e.g. data cubes.

5.2 Reorganizing the index

While the techniques discussed in the previous subsection have their merit, the actual pages holding the table data are still accessible by the database interface and only searching for them by using secondary indices is thwarted. As a countermeasure, simply dropping and recreating indices regularly, or using searches based on non-indexed columns or the primary index can be used. Therefore, we propose alternatives for storing the secret data inside the actual index pages, thus reorganizing the next-pointers to create slack space and force the database to skip the hidden record.

Figure 3 gives an overview on how to manipulate a primary index for data hiding: The index contains the actual data records which are linked with a next pointer. In order to hide a record, the links in the primary index leading to and from this record by its neighbors are removed and exchanged for a direct link between the two, e.g. C_8 in Figure 3, will be unlinked from the tree structure (red arrows) and a new link will be set instead, linking the former neighboring records, e.g. linking C_7 and C_9. Furthermore, the record needs to be removed from the page directory that possesses a direct pointer to every fourth to eighth element for enabling faster searches within the page. In case the hidden record is currently referred by the page directory, the directory has to be reorganized.

In order to hide data, the following general approach can be chosen:

1. The table containing the hidden data is generated or selected.
2. The secret data is written to the table in the form of table entries.
3. The links to the secret data are removed from the primary indices by applying the approach outlined before.
4. In case of retrieval, classic forensic methods like file carving for database records (e.g. [2]) or reorganizing the pointers again are used to access the hidden data.

Note that the hidden records are still accessible through secondary indices and have to be further hidden like proposed in Section 5.1.

5.3 Hiding data inside the index garbage space

Further, we propose alternatives for storing the secret data inside the actual index pages, thus removing them from the data table altogether. These alternatives can be seen as extensions of the original approach to the primary index. The main difference to removal from a primary and/or secondary index is that the primary index constitutes the table content, i.e. all records belonging to a table are indexed by the primary index and any record removed from it is also removed from the table. The main danger behind manipulations of the primary index thus lie in generating inconsistencies that not only allow for the detection of the manipulations, but can destroy the correctness of large parts of the database.

In Section 4 we outlined the internal workings of data removal from the primary index by unlinking the record in the tree and linking it to the list of deleted records, starting with the garbage offset (see Figure 1). Furthermore, in Section 5.1 we manipulated secondary indices for manipulating searches. Thus, there are basically two different starting points for hiding data from the primary index: First, it is possible to manipulate the delete-operation in order to not link the deleted record to the list of deleted records, thus, the space containing the secret data will not be overwritten by the database. Second, the approach of unlinking the record in the secondary indices can be extended to the primary index too, thus linking the neighboring records and removing the links to the secret record. Furthermore, in case of the second approach, it is necessary to remove the links in all secondary indices too, in order not to generate inconsistencies in the database.

This approach can be extended to the primary index data-hiding mechanisms (Section 5.2): The secret record that is stored in the table is unlinked like in the case of deletion, but it is not linked to the garbage collection. This also includes unlinking the record in every secondary index.

The following generic approach can be used for removing the secret data from the primary index and thus from the table:

1. The table containing the hidden data is generated or selected. There are no real requirements regarding the primary index, especially not regarding its use in searches.

2. The record holding the secret data is removed from the table using a modified version of the delete operation (see the next step).

3. When deleting the respective entry in the index tree, a modification of the delete procedure is applied: While the record is unlinked from the tree as it is done normally, it is not linked to the list of deleted records, thus not being marked as available for future use. Since this is the only change with respect to the original deletion mechanism, all secondary indices are updated normally.

4. In case of retrieval, file carving methods are used to access the hidden data.

The drawback of this method lies in the fact that the insert and delete operations leave many traces in internal mechanisms like the transaction log or other places that need to be removed in case a forensic investigation is expected before they are expired. In order to overcome this weakness, in step two the secret data is not stored in the underlying table, but some arbitrary, unsuspicious data is chosen. After having unlinked the record from the index, the free space is then filled with the secret data by using file carving. This has the additional benefit that the data can be changed afterwards, i.e. the above mentioned procedure is only used in order to generate slack space that is not allocatable by the database.

Unfortunately, navigation by file carving is rather tedious and inefficient if many reads need to be done. In order to enhance usability of the slack space, we propose a further enhancement: In analogy to the linked list of free space that starts with the garbage offset, we propose to link all generated slack space inside a page starting with a *hidden page offset* and links to the first hidden record. The last record links to itself in order to signal the end of the list. This allows for easy navigation through the slack space inside a page, since only the hidden page offset needs to be found by file carving.

In order to further enhance searching in the slack space, hidden page offsets can be linked together e.g. by generating a B^+-tree, thus generating a *shadow index*, much like the primary index of a regular table.

5.4 Hiding data inside the free space of an index page

Due to the physical structure of a page there exists some free space that is allocated by the storage engine but currently not used (see Section 3). Figure 4 shows a schematic overview of an index page. New records are written to the user record space towards the FIL Trailer in the order of their insertion. Simultaneously the page directory grows towards the user records. If these two sections meet, the free space of the page is exhausted and the page is considered as full. This free space is not used by the DBMS and can be used to hide data, however it is - in contrast to the other data-hiding techniques - not save against overwriting, because the DBMS considers this space as free.

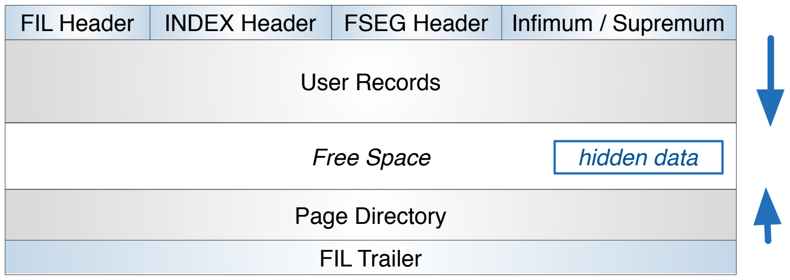


Figure 4: Physical structure of an index page

5.5 Removing a page from the index

InnoDB uses pointers between pages to create a B^+-Tree. These pointers are used to find the page where the requested data is stored. All leaf nodes contain the actual record data in contrast to the non-leaf nodes that only contain pointers to the next pages. All pages on the same level are double-linked to their predecessor and successor. Similar to the idea of reorganizing the index (Section 5.2) it is possible to change the pointers to unlink a page and use it to hide data. However our evaluation showed that this approach is not feasible, because in general a regular page contains a lot of data which is also referred to by secondary indices, which results in many additional updates. Furthermore, the B^+-Tree has to be rearranged, thus creating a massive overhead.

6 Evaluation

According to the internal source documentation of InnoDB, the database storage engine accesses every data record solely by using the primary index. If a records is not accessible via this index, the data record doesn't exist for the DBMS. This architecture makes sense in regards of performance but can be misused for data hiding purposes presented in section 5.

Since we did want to solely rely on the source code documentation, we created a new method for evaluation purpose of our data hiding approaches. The basic idea is to create a set of queries that are executed on a manipulated table space. We monitor if a token that we have hidden before can be retrieved and whether the storage engine crashes due to undocumented behavior of our approach. It is impossible to cover all possible query constellations, because this problem is undecidable. However, by using a fuzzy testing approach that generates a large test set that simulates a realistic environment and due to the fact that the navigation algorithms are limited, we can guarantee with a high certainty that all possible mechanisms are covered.

Technically we are using a SQL Generator for testing SQL servers called *randgen*[1]. This tool implements a pseudo-random data and query generator that is originally designed for functional and stress testing. Test cases are generated by a context-sensitive grammar that is used as input.

In our evaluation scenario we only consider queries that actually rely on the existence of data, i.e. that reveal the existence of the hidden token. The following operations fulfill this

[1]`https://launchpad.net/randgen`

requirement.

- The SELECT Operation
- JOIN Operations
- INSERT Operations with ON DUPLICATE UPDATE

However we didn't use and functions, procedures, triggers, subqueries or views, because they are handled by the same internal functions and mechanisms as the operations above.

As mentioned before we use a context-sensitive grammar as an input for our modified version of the tool *randgen*[2]. As an example the following (simplified) version of context-sensitive grammar of the SELECT syntax is used. In order to generate a valid and adequate test set we used the *where_condition* to force the database management system and the query optimizer to use preferably different types of index navigation, i.e. sql caches, direct access or area searches.

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ] [SQL_CACHE | SQL_NO_CACHE]
    select_expr [, select_expr ...]
    [FROM table_references
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

We used this setup to evaluate our different data-hiding techniques of Section 5. We automatically derived concrete sql statements from the grammar and executed them on a manipulated table containing hidden data. A test case failed if the hidden token could be retrieved. If a test case failed we used MySQL internal tools like *EXPLAIN*-Queries in order to determine the used navigation algorithms and grouped the results into different categories: Full-Table Scans (e.g. *SELECT*-Statements without index or *SELECT*-Statements without *WHERE*-Conditions), direct access using a primary key (i.e. *JOIN*-Operations and *WHERE*-Conditions using primary key fields) and indirect access using secondary indices (i.e. *WHERE*-Conditions using a secondary index) and area scans using conditions, e.g. *BETWEEN*. Table 1 gives an overview of the different approaches and the availability of the hidden data using different queries and whether the hiding technology is persistent and resistent against accidental overwriting caused by the database management system.

Note that classical SQL Backups use a full-table scan for generating data backups. Due to its design all information of the index gets lost, therefore some modifications like the manipulation of the search results will not be uncovered in digital investigations. This data tampering attacks can thus only be detected in the live system that is usually not analyzed due to possible side effects on the productive systems.

[2]Due to blinding reasons we will release our modified version and test set after reviews

	full-table	***primary***	**secondary**	***persistent***
manipulating search results (5.1)	yes	yes	no	yes
reorganizing the index (5.2)	no	no	yes	yes
hiding data inside the index garbage space (5.3)	no	no	no	yes
hiding data inside the free space of an index page (5.4)	no	no	no	no
remove a page from the index (5.5)	yes	yes	no	yes

Table 1: Hidden data accessible via SQL interface

7 Conclusion and Perspectives

In this work we demonstrated how indices in InnoDB can be manipulated in order to hide data. We proposed a set of five methods that possess different attributes and benefits. Thus we were able to show that it is possible to hide data, as well as generate free slack space in InnoDB. The main aspect behind the outlined techniques lies in manipulating the underlying index structure. Depending on the technique the data can still be retrieved via the SQL-interface by issuing suitable select-statements, or solely via advanced file carving methods.

Regarding practical application, we showed that it is possible to manipulate secondary indices in a way that selected statements return manipulated results without actually removing the data from the table, thus making it still available in case of investigations and sanity checks. This is especially important considering large data warehouses used for automated workflows. While the statements used for the workflow routines are typically indexed in order to guarantee the needed performance, manual investigations usually do target unindexed searches. Thus it is possible to manipulate a database in a way that removes data from the indexed searches and thus from the actual workflow, while making this manipulations go unnoticed in the case of investigations or sanity checks. This also demonstrates that the result returned from a database through the SQL-interface cannot be trusted.

Furthermore, we showed that it is possible to generate an arbitrary amount of hidden slack space inside the database, which cannot be searched or modified through the SQL-interface, thus being stable with respect to normal database operation. This slack space is especially valuable as it resides directly inside the normal data files, which are target of constant changes during normal operation, thus making any additional changes practically impossible to detect through more traditional means of forensics targeting the file system. For accessing the hidden data, file carving can be utilized which was extensively explained in [2]. Additionally, we proposed some additional structure for enhancing performance of operations on this slack space in order to boost practical usability.

We thus conclude that it is possible to manipulate and hide data inside databases with potentially large impact to operations in data warehouses, as well as traditional file based forensics. For future work we plan to extend our approaches to other prominent database management systems, as well as to conduct a large scale real life case study targeting industry partners.

References

[1] W. Bender, D. Gruhl, N. Morimoto, and A. Lu. Techniques for data hiding. *IBM systems journal*, 35(3.4):313–336, 1996.

[2] 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*, pages 1028–1036. IEEE, 2010.

[3] P. Frühwirt, P. Kieseberg, K. Krombholz, and E. Weippl. Towards a forensic-aware database solution: Using a secured database replication protocol and transaction management for digital investigations. *Digital Investigation*, 11(4):336–348, 2014.

[4] P. Frühwirt, P. Kieseberg, S. Schrittwieser, M. Huber, and E. Weippl. Innodb database forensics: reconstructing data manipulation queries from redo logs. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 625–633. IEEE, 2012.

[5] 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*, 17:227–238, 2013.

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

[7] 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*, 9(1):53–68, 2013.

[8] 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.

[9] 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.

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

[11] 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*, pages 65–73, 2000.

[12] G. Miklau, B. N. Levine, and P. Stahlberg. Securing history: Privacy and accountability in database systems. In *CIDR*, pages 387–396. Citeseer, 2007.

[13] H. Pieterse and M. Olivier. Data hiding techniques for database environments. In *Advances in Digital Forensics VIII*, pages 289–301. Springer, 2012.

[14] 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*, pages 91–102. ACM, 2007.