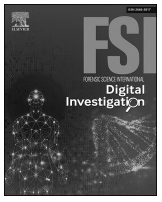
Forensic Science International: Digital Investigation 53 (2025) 301934

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi



DFRWS USA 2025 - Selected Papers from the 25th Annual Digital Forensics Research Conference USA

Exploiting database storage for data exfiltration

James Wagner [a,*], Alexander Rasin [b], Vassil Roussev [a]

[a] University of New Orleans, New Orleans, LA, USA
[b] DePaul University, Chicago, IL, USA

ARTICLE INFO

Keywords:
Data hiding
Database forensics
Digital forensics

ABSTRACT

Steganography is a technique for hiding messages in plain sight – typically by embedding the message within commonly shared files (e.g., images or video) or within file system slack space. Database management systems (DBMSes) are the de facto centralized data repositories for both personal and business use. As ubiquitous repositories that already offer shared data access to many different users, DBMSes have the potential to be a powerful channel to discretely deliver messages through steganography.

In this paper we present a method, Hidden Database Records (`HiDR`), that adapts steganography techniques to all relational row-store DBMSes. `HiDR` is particularly effective for hiding data within a DBMS because it adds data to the database state without leaving an audit trail in the DBMS (i.e., without executing SQL commands that may be logged and traced to the sender). While sending a message in this way requires administrative privileges from the sender, it also offers them much more control enabling the sender to erase the original message just as easily as it was created. We demonstrate how `HiDR` keeps data from being unintentionally discovered but at the same time makes that data easy to access using SQL queries from a non-privileged account.

1. Introduction

Data hiding (or steganography) is the science of hiding the act of communication. To understand the significance of steganography, it is important to differentiate it from cryptography. Cryptography seeks to hide the content of the message, but does not hide the existence of communication – a message may be observed but its content cannot be comprehended. Instead, steganography seeks to hide the act of communication itself rather than the content of a message – a message is hidden, but its content may be reprehensible (i.e., plausible deniability) when observed. Thus, cryptography and steganography are two distinct means for obscuring communication, which can be used together to compliment each other.

A well-designed steganography method makes the concealed data difficult to detect while also making that data easily accessible by the recipients. For example, ubiquitous files (e.g., video, audio, image or text files) are good candidates for message hiding (Conlan et al., 2016; Johnson and Jajodia, 1998). These files can be sent (and observed) among users without raising suspicion. Additionally, these files contain redundant and noisy data, making it easy to find unused or redundant bytes to hide data without affecting the primary function of the file. For example, the least significant bit of the image data (altering the image but imperceptible to a human eye) or a header of a media file can be used to embed a hidden message (Hamid, 2012). There is also a significant amount of work on hiding data in unused portions of metadata in file systems (Garfinkel, 2007a; Kessler, 2007). User privilege can constrain steganography, such as whether a user has the privilege to store the message in a file (write), attach the file to an email (read), or directly alter file system metadata (system administration).

DBMSes are widely used to manage both personal and corporate data. For example, a lightweight DBMS such as SQLite commonly manages personal data stored on mobile phones and web browsers; whereas a DBMS that supports more robust access control and storage management, such as Oracle, PostgreSQL, MySQL, or Microsoft SQL Server, is better suited to manage corporate data. Similar to common files, DBMSes are suitable candidates to hide messages that are shared with many users without raising suspicion. Unlike common files, database files are stationary in the file system, i.e., are never transmitted directly; hence, users access records and values within DBMS using SQL queries.

There is relatively little existing work on DBMS steganography applications – due to the complexity of hiding data within a DBMS and the problem of making message delivery practical. Multiple types of internal DBMS logging (transaction and audit logs) make it difficult to effectively

* Corresponding author.
E-mail addresses: jwagner4@uno.edu (J. Wagner), arasin@cdm.depaul.edu (A. Rasin), vassil@cs.uno.edu (V. Roussev).

https://doi.org/10.1016/j.fsidi.2025.301934

2666-2817/© 2025 The Author(s). Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

hide a message. In general, introducing a hidden message into the database via an SQL command creates an associated log record. Furthermore, while an OS administrator (with access to DBMS files) can bypass logging and hide data in unallocated DBMS space, the data must remain retrievable through typical SQL queries so that elevated privileges are not required to receive the message.

1.1. Example: illustration of DBMS steganography

Alice wants to pass information to Bob (a colleague working in the same company) and Carl (a company customer), but her transporting the data off company premises may attract unwanted attention. Alice is a System Administrator – she has administrative (OS) access to the server where the DBMS runs, but does not have a database account. Bob has access to the DBMS through a regular (non-administrative) account, and Carl has no accounts or access inside the company. To send a hidden message to Bob, Alice introduces a special record into the DBMS state, which can be accessed by Bob through a specially formulated SQL query. Alice could later erase the unusual record – but even if the record is found, no audit or log information will indicate who left this message or when. Similarly, in order to send a message to Carl, Alice adds a regular record (within Carl's customer profile) using the same approach. Carl can then access his customer account through normal means to retrieve Alice's message. Although Alice has easier ways to send a message, this approach lets Carl retrieve the message without performing any unusual and thus detectable actions (i.e., Carl merely logs into his existing account as any other customer).

In this paper, we propose a database steganography method called **Hidden Database Records (`HiDR`)**. `HiDR` directly accesses the database files to add records, bypassing the DBMS itself and hiding this activity from the DBMS as a result. `HiDR` delivers three significant contributions in the domain of database steganography:

- Activity that would be difficult to trace by a third party. All DBMS logging mechanisms, constraints, and indexes are effectively bypassed, leaving no indication of how or when the data was introduced into the database. The sender can also erase the message, leaving only a narrow window for an observer to search for the hidden message in the database.
- Messages can easily and unambiguously be retrieved by the intended recipient either through SQL queries (from a regular non-administrative DBMS account) or via a regular web portal (from a customer account).
- The method is applicable to any (both proprietary and open source) row-store DBMS including IBM DB2, Microsoft SQL Server, Oracle, PostgreSQL, MySQL, SQLite, Firebird, and ApacheDerby.

Table 1 summarizes the remainder of this paper.

Table 1
Summary of the remaining paper.

§	Summary
2	Background information on DB auxiliary objects, constraints, and query access patterns to understand how HiDR messages are effectively hidden.
3	A discussion of related work in steganography and its current limitations with respect to DBMSes.
4	The considerations taken when implementing HiDR to performing external modifications to DBMS files.
5	A description of the HiDR method.
6	An explanation of why HiDR messages are masked, and how the hidden data can be unambiguously retrieved.
7	Using three representative DBMSes (PostgreSQL, MySQL, and Oracle) we demonstrate that HiDR effectively hides data, which is retrieved using SQL.
8	A discussion of prevention and detection measures to counter malicious applications of HiDR.

2. DBMS constraints and queries

`HiDR` exploits database engine operation to both hide messages from regular queries and provide easy retrieval for the intended recipient. This section discusses the relevant concepts needed to understand how this achieved.

2.1. Indexes

A database index stores value–pointer pairs (typically a B-Tree structure) to help locate rows within a table, providing performance benefits. A DBMS can create indexes automatically – for example, constraints (e.g., primary key or UNIQUE) generate an index. Index entries are stored in pages as shown in Fig. 2 –an index is structurally similar to a table that stores (value, pointer) records. It is important for this paper to note that NULL values are not stored in indexes.

Fig. 1 displays an example index page, and how a value references a record. A pointer to a table record is stored with each city value. Here, the pointer stores the page identifier, 8, and the respective row identifier, 25.

2.2. Constraints

Relational databases are designed to represent a relation (table) as a set of tuples (rows). Each row must be unique, enforced through a *primary key* constraint. Primary keys are always unique and by definition can never be NULL ("unknown"); the DBMS blocks any operation that attempts to violate that rule.

Uniqueness of the primary key is further used to enforce integrity through *foreign keys*. A foreign key is a cross-table reference: for example, a loan payment record holds the loan ID (primary key) to which it belongs. *Referential integrity* requires such foreign key references to always be valid – either reference an existing record (e.g., an existing loan ID) or contain a NULL as a placeholder.

Relational database rules furthermore impose a constraint on every table column. Each column must have a well-defined data type such as INTEGER or VARCHAR(15). As with other constraints, the DBMS actively enforces these rules by checking every operation that might potentially modify stored values. Any step found to be in violation of these rules (e.g., inserting a string into an integer column) is blocked.

2.3. Query execution

A DBMS engine has two strategies to fetch data from tables: 1) an *index access* performs a targeted data retrieval (i.e., use an index to fetch relevant rows), or 2) a *full table scan* searches the entire table reading both relevant and irrelevant rows. An index access is only used when an index is available and deemed to be cheaper than a full table scan.

An SQL query that accesses multiple tables combines them with a

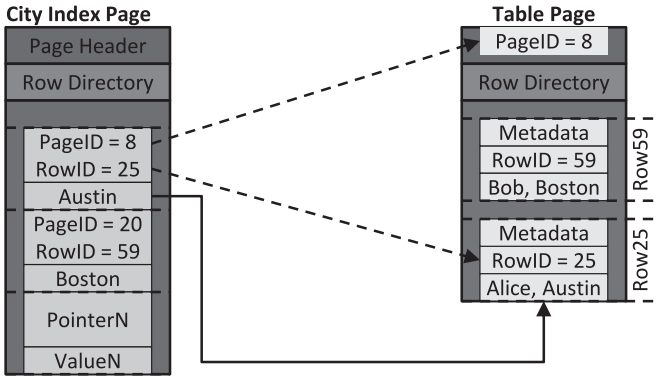


Fig. 1. Example: an index value referencing a record.

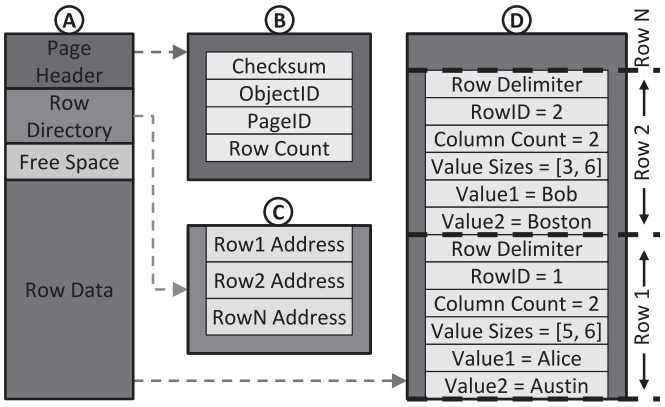


Fig. 2. Page examples: A) high-level, B) header, C) row directory, and D) row data.

join operation. The default join type is an INNER JOIN, which combines only the matching rows. Using our loan example, a report about loans and loan payments includes only loans with existing payments – a loan without any associated payments is ignored. Several other operations such as NATURAL JOIN or subqueries are also executed as an INNER JOIN. An SQL query may explicitly request that unmatched values be included in the result – this operation is known as an OUTER JOIN. An OUTER JOIN in our loan example returns loans without any associated payments, substituting NULLs for missing loan payment data.

3. Related work

3.1. Database forensics

HiDR directly examines and modifies database storage. Therefore, we consider methods that read database data independent of the DBMS. Database page carving (Wagner et al., 2017, 2019, 2020a, 2020b) is a method that reconstructs relational DBMS file contents without relying on the file system or DBMS. Page carving is similar to traditional file carving (Richard and Roussev, 2005; Garfinkel, 2007b) in that data, including deleted data, is reconstructed from disk images or RAM snapshots without using a live system. The work in (Wagner et al., 2015) presented a comparative study of the page structure for multiple DBMSes. Wagner et al. expanded on this work by generalizing database memory structures (Wagner and Rasin, 2020), which was later used for log verification (Wagner et al., 2023). Work in (Wagner et al., 2016; Lenard et al., 2021) described the lifetime of deleted data within an DBMS, and (Wagner and Rasin, 2024) proposed a method to sanitize this data so that it is not forensically recoverable. While a multitude of built-in and 3rd party recovery tools (e.g., (OfficeRecovery, 2017; Percona, 2018; Phoenix, 2018)) can extract database storage, these tools are not can only recover "active" table (non-deleted or un-corrupt) records. Forensic tools, such as Sleuth Kit (Carrier, 2011) and EnCASE Forensic (Garber, 2001), are commonly used by digital investigators, but they primarily reconstruct file system data and are not capable of parsing DBMS files. Therefore, the most practical method for HiDR was to examine and modify storage at the page level with a specialized version of page carving.

3.2. Database steganography

There is much research in steganography at the file system or OS level, and steganography for files (e.g., image files (Ansari et al., 2020), audio file (Cui et al., 2020), QR Codes (Patel and Pragathi, 2022)) that are typically shared. Garfinkel et al. (Garfinkel, 2007a), Conlan et al. (2016), and Dhawan et al. (Dhawan and Gupta, 2021) provided overviews of steganography research. However, all of this work has limited applications for DBMSes. DBMS files cannot be interpreted individually and must use the DBMS to read the files. Furthermore, data hidden in unallocated space must be added in such a way that it can be queried, which is not explored by any of the related work. A few methods for hiding database data were proposed, but we consider them to be limited. These methods are DBMS-specific and require SQL commands to be executed using the DBMS, which creates evidence in two types of DBMS logs (write-ahead and audit).

Pieterse et al. (Pieterse and Olivier, 2012) proposed PostgreSQL DBMS hiding methods, which alter the system catalog tables. This required modifications to the relational schema which included removing constraints, and removing and adding columns to tables using SQL commands. We believe these SQL commands can leave a significant amount of evidence behind and are not inconspicuous. Furthermore, we argue that modifying the database schema would have significant impacts on user database applications and would be rather noticeable. The operations performed were also highly specific to PostgreSQL.

Furhwirt et al. (Fruhwirt, 2015) proposed methods that require a record to be inserted using SQL, and either removing the pointer in the secondary index or the primary key indexes. By removing pointers in indexes, they claim index accesses are common and therefore, their hidden data is less likely to be returned by SQL queries. However, current trends in database systems research argues that secondary indexes are limited to only queries with the highest selectivity due to improvement in I/O efficiency of hardware (Kester et al., 2017). We argue and demonstrate that bypassing constraints (rather than just indexes) makes HiDR more likely to exclude a hidden record in regular query results.

4. DBMS file modification

HiDR directly modifies database files without the DBMS API (e.g., SQL). DBMSes do not provide an API to modify or even directly inspect the storage at the page level. When a DBMS file is modified, correct format and all relevant metadata must be considered to avoid corrupting the page (or the entire database instance). Three things must be considered to perform live database file modifications: 1) page checksum, 2) committed transactions, and 3) dirty pages.

Any page modification requires updating the corresponding checksum, even if the DBMS is shut down. For all data and metadata updated in a page, the checksum is always the last thing to be updated. This is because updating another part of a page may result in a new checksum value. If the checksum value is not correct, the DBMS flags the page or even the entire file as corrupt.

Transactions help manage concurrent access to the DBMS and are used for recovery post-failure. For example, consider a customer who transfers $10 from account A($50) to account B($5). Should the transfer fail mid-flight (after subtracting $10 from A, but before adding $10 to B), transactional mechanism restores account A back to $50. Changes are stored in the transactional log (e.g., <A, $50, $40>, <B, $5, $15>) and are finalized by transactional COMMIT. If modifications are performed in a live database instance, transaction logs can verify that there are no relevant uncommitted transactions.

DBMSes do not immediately write pages back to disk after pages were loaded and modified in the buffer cache – a page that contains pending changes in the cache is a *dirty* page. This is significant because our manually modified DBMS page can be overwritten when a dirty page is flushed to disk. This does not prevent HiDR from working or pose any corruption risk, but can overwrite some of the changes (undoing the changes we made to the page). Therefore the sender might need to verify that the message was not destroyed by internal DBMS activity. Since a COMMIT does not force pages to be written to disk, some DBMSes offer an explicit SQL command to flush the buffer cache contents to disk.

5. HIDR

As previously mentioned, HiDR operates at the page level to

introduce records into the DBMS state. Pages are directly read and modified using a Hex editor or a programming language (for this paper we used Python 2.7). Along with the considerations for DBMS file modifications discussed in Section 4, metadata within the page must be correctly updated to avoid corruption. In this section, we describe how HiDR alters pages. The terminology and concepts apply (but are not limited) to DB2, SQL Server, Oracle, PostgreSQL, MySQL, Apache Derby, SQLite, and Firebird.

5.1. Pages

Before we describe the HiDR specifics, we provide an overview of database pages. The DBMS storage layer partitions all physical objects (e.g., tables, indexes, and system catalogs) into fixed pages with a typical size of 4 or 8 KB. Using a fixed page size across an entire database instance significantly simplifies storage and cache management. Page size can be changed but only through rebuilding all affected database objects: page size cannot be changed for individual tables, at a minimum it is global per tablespace (a logical storage unit).

When data is inserted or modified, the DBMS controls data placement within pages and internally maintains additional metadata. Despite the wide variety of DBMSes from different vendors on the market, many commonalities exist between DBMSes in how page data is stored and maintained. Every row-store (storing record values on the same page) DBMS uses pages with three main components: header, row directory, and row data. Fig. 2A displays a high-level breakdown of a page with all three of these structures. HiDR modifies the page components in the following order: 1) Row Data, 2) Row Directory, and 3) Page Header. This sequence ensures data is correctly added to the database state without corruption.

5.2. Row data

The row data stores the user data along with the metadata that describes each value contained in the records. Fig. 2D shows an example row data structure (there are some minor DBMS-specific variations). In this example, each record stores a row delimiter, row identifier, column count, value sizes, and the user data values. Table 2 summarizes the significant pieces of metadata stored with each record in the row data segment for each DBMS. Each DBMS uses a subset of these metadata items to represent its internal storage; none of the metadata is used in all DBMSes. At the same time, there is significant overlap in how the metadata is used – we have included the DBMSes that use each piece of metadata to demonstrate the similarity in storage choices between DBMSes.

HiDR first adds the user data along with all of the necessary metadata expected in the row data for that DBMS. For example, to add a record to the row data for a PostgreSQL DBMS, the following need to be generated: a row identifier, column count, and column sizes.

Table 2
Row data metadata descriptions.

Parameter	Summary
Row Identifier [1, 4, 6, 7]	The internal row identifier pseudo-column, which corresponds to a record's pointer. Sometimes only a subset of the row identifier is stored with the record.
Column Count [1, 5, 6, 7, 8]	The number of columns for a record (this value is fixed for each table page, but stored with each record).
Column Sizes [1, 3, 4, 5, 6, 7]	The size of each column. Typically only the sizes for strings are stored, whereas other datatypes such as integers assume a fixed storage space.
Column Directory [2, 8]	Pointers to each column within the record.

[1] ApacheDerby, [2] DB2, [3] Firebird, [4] MySQL, [5] Oracle, [6] PostgreSQL, [7] SQLite, [8] SQLServer.

5.3. Row directory

The row directory stores pointers to each record – when a record is added to a page, a corresponding pointer is created. Fig. 2C shows an example of how the row directory could be positioned within the page structure. Table 3 summarizes how exactly the row directory is positioned in each DBMS. It is also possible that a DBMS uses a sparse row directory, in which case an address does not necessarily need to be appended to the row directory. When a sparse row directory is used, one pointer is typically created for every 4–6 records. A sparse row directory is rarely used, and we only observed it for the index organized tables used by MySQL.

HiDR appends the page address of the record to the row directory. For example, if record is added to a PostgreSQL DBMS page, a pointer is appended to the bottom of the row directory since this is the next space to be chosen by that DBMS (as per Table 3).

5.4. Page header

The page header contains metadata describing the user records stored in the page. The page header metadata we consider for the purposes of this paper are the checksum, free space pointer, object identifier, page identifier, and record count. Fig. 2B demonstrates how this metadata could be positioned in a page header. Table 4 describes each one of these metadata items for each DBMS.

For HiDR, the relevant metadata in the header is updated: the free space pointer, record count, and checksum. The free space pointer is updated to reflect the next free space following the added record to avoid overwriting of the record by subsequent DBMS operations. Next, the record count is incremented by one. The page checksum is always the last metadata item to be updated. This is because the page checksum computation may (checksum functions vary in different DBMSes) use metadata that was previously updated. Therefore, changing other metadata after the checksum update may result in an incorrect checksum.

5.5. HiDR deployment considerations

The simplest way to modify DBMS files is to shut down the DBMS or place it into a backup mode (which ensures no active transactions can execute). However, a DBMS shutdown is not conducive to hiding a message – and the same techniques can also be easily executed against a live database. To record the hidden message in a live DBMS, we prefer to choose the pages that are not currently being modified and will not be overwritten when the DBMS buffer cache is flushed to disk. Such pages can be identified by inspecting RAM or the DBMS transaction log. However, this process requires additional effort and may still fail to detect some of the pages that will be overwritten. Therefore, Alice can instead opt to create multiple message copies throughout the database file to reduce the likelihood of the message being accidently overwritten and lost.

When performing HiDR, it is useful to know the table to which you are adding the record. The object identifier and page identifier enable us to determine the page identity. While the name of an object (e.g., table name) is not stored in a page, the object identifier can be mapped to the system catalog data to retrieve a plaintext name (e.g., *customer* table).

Table 3
Row directory metadata descriptions.

Parameter	Summary
Top-to-bottom [2, 3, 5, 6, 7]	Row directory addresses are appen-ded from top-to-bottom of the page
Bottom-to-top [1, 4, 8]	Row directory addresses are appended from bottom-to-top

[1] ApacheDerby, [2] DB2, [3] Firebird, [4] MySQL, [5] Oracle, [6] PostgreSQL, [7] SQLite, [8] SQLServer.

Table 4
Page header metadata descriptions.

Parameter	Summary
Checksum [1, 2, 3, 4, 5, 6, 7, 8]	Used by the DBMS to detect corruption within a page; it is updated whenever a page is modified.
Record Count [1, 2, 3, 4, 5, 6, 7, 8]	The number of *active* records within a page. If a record is deleted in a page the record is decremented by one; if a record is added to a page, it is incremented by one.
Free Space Pointer [1, 2, 3, 4, 5, 6, 7, 8]	The address where the next record will be appended to the page.
ObjectID [1[a], 2, 3, 4, 5, 6[a], 8]	Represents the object to which the page belongs.
PageID [2, 3, 4, 5, 6, 8]	DBMS-dependent, a unique number for each page – either for each object within a file, or across all database files.

[1] ApacheDerby, [2] DB2, [3] Firebird, [4] MySQL, [5] Oracle, [6] PostgreSQL, [7] SQLite, [8] SQLServer.
[a] The ObjectID is stored in the row data with each record.

The page identifier is static for any given page, regardless of whether the page is in the buffer cache or in persistent storage. To remain inconspicuous, newly added records should not differ significantly from the existing records in the table.

5.6. Quantification of hidden message throughput

There are no limitations on the number of records that can be added through HiDR. In this paper we only describe adding records to free space in existing DBMS pages – although other mechanisms are available, this one is the simplest. In practice, multiple pages in each table are likely to have free space in them. This is because free space is created by many common user commands, such as DELETE, UPDATE, and rebuild/defragment commands. Moreover, most DBMSes allocate storage to a table in units of an *extent*, which is a group of pages (about 10). It is unlikely for the entire extent of pages to be immediately filled by rows.

If none of the pages in a table have free space, a new page must be manually built, added to the database file, and assigned to the relevant table. The steps to perform such an operation are beyond the scope of this paper and require modifying some metadata that was not discussed in this paper.

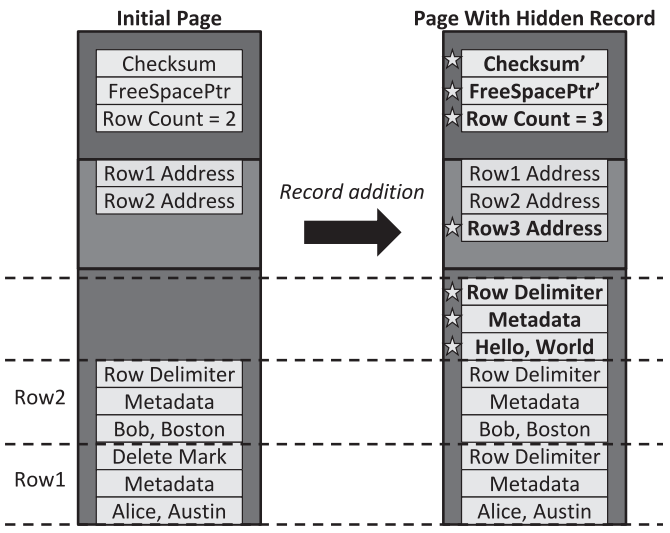


Fig. 3. An illustration of record addition.

5.7. Record addition example

Fig. 3 illustrates HiDR adds records to DBMS state. Here, the record ('Hello', 'World') was added to a page. The updated page attributes are bolded and labeled with a star. Along with the record values themselves, additional row data metadata is updated. Next, a row directory address is appended to the row directory. The page header row count is incremented by one and the free space pointer is updated are then updated. Finally, the checksum is updated.

6. DBMS data hiding

HiDR is logically similar to an SQL INSERT command, in that data becomes part of the database state and can be queried using the DBMS API. In this section, we describe why HiDR is effective for hiding data within a DBMS, and how HiDR data remains retrievable with regular (specifically targeted) SQL queries.

6.1. Message hiding effectiveness

HiDR is effective at concealing a message within a DBMS because it adds the data to the database state unbeknownst to the DBMS. Therefore, all of DBMS access control, logging mechanisms, constraints, and indexes are effectively bypassed.

6.1.1. Access control

HiDR does not require the user (the sender) to have an account in the DBMS where the message is being hidden. Only write access to DBMS files (at the OS level) is required to create the hidden message, and only regular table read access is needed to retrieve the message.

6.1.2. DBMS logging

DBMS logging (both transaction and audit) only records the activity executed through the DBMS API as SQL commands. Since HiDR does not use SQL and the operation is not part of any DBMS transaction, HiDR activity is not logged. Live DBMS files are constantly updated by ongoing user queries and therefore any logging at the file system level would be difficult to attribute to HiDR data.

6.1.3. OS logging

It would generally be difficult to attribute a file system journal entry to specific HiDR activity. A running DBMS performs many writes across its files. In order to better conceal HiDR activity at the OS level, the modifications to DBMS files could be performed by impersonating DBMS process ID (thereby making HiDR writes indistinguishable from other DBMS activity).

6.1.4. Constraints

DBMS constraints are only enforced when commands are executed, and they are not retroactively checked. Therefore, the DBMS never performs a constraint check on the HiDR data. If existing constraints are violated, that would only become apparent if these constraints were dropped and recreated, which is not a typical database activity. Otherwise, no indication will be given that a constraint-violating record exists in the DBMS storage.

HiDR's ability to bypass constraints contributes significantly to data concealment. Current database systems research trends favor full table scans over index accesses due to the improvements in I/O throughput costs over seek costs (Kester et al., 2017). Executing a single table scan is not typical because relational database design normalizes data across multiple tables, requiring table joins to recreate complete data records. While a single table full table scan may return the hidden message, joins involving multiple full table scans are unlikely to return the hidden messages. Joins are usually performed based on the foreign keys, and any record that violates the foreign key constraint will not be returned by the query. Since HiDR can bypass foreign key constraints, HiDR data

can be hidden from queries that use joins.

Fig. 4 illustrates how a foreign key violation better hides data. The added record (*NULL, Message1, -1*) violates the referential constraint because the value for `EMPLOYEE.DeptID`, −1, is not in the set of referenced primary key values. Thus, (*NULL, Message1, -1*) is not returned by an inner join.

6.1.5. Indexes

The DBMS maintains indexes as data is added to tables using SQL. Therefore, the `HiDR` data will not be indexed at the time of addition. This keeps the hidden data from being returned by any queries that use index access. However, it is not uncommon for users (or even the DBMS) to rebuild indexes for performance considerations. To keep the `HiDR` data from being added to the index on rebuild, we propose using the NULL value for all indexed columns (including the primary key). DBMS indexes will not include NULL entries, thus permanently keeping the added record from being indexed. The only way for a DBA to discover such hidden record would be to re-create the key constraint (e.g., dropping and then re-adding the primary key to the table). However, there is never a reason to do so in a DBMS because all constraints (including disallowing NULL in the primary key column) are always strictly enforced for SQL queries.

Fig. 4 illustrates how NULL can be used for any indexed columns. We added NULL to the primary key columns for both tables. By default, the DBMS creates an index to enforce the primary key constraint. Additionally, we bypassed the primary key constraint since NULL is not allowed. These records are not returned by any index access queries that use the `ID` column for either of the tables. Furthermore, if the indexes were rebuilt, NULL would still not be added to the index. This data would only become problematic for the DBMS if the primary key constraints were dropped and recreated, which is unlikely.

6.1.6. Message destruction

Once the message is received (or has not been retrieved within an agreed up time frame), the message creator may want to remove evidence of the message from the DBMS. Just as it is possible to add a record to the database state, it is also possible to wipe that record. This reverse process would require the record in row data to be overwritten with `NULL`, the corresponding row directory pointer to be overwritten with `NULL`, the row count decremented by one, and the page checksum to be recalculated again. Based on our experience, the free space pointer within the page does not need to be updated for this operation.

We believe this message destruction feature is one of the major advantages of using `HiDR` over other types of steganography message sending. For example, while sending a file such a JPEG or PDF via email does not require administrative privileges from the message sender, the sender also loses control over the message once it is sent. The sender of the message cannot delete a PDF file once it is emailed. While the recipient of the message might delete the file, any additional OS copies (e.g., in paging files or in RAM) can still be found, carved using appropriate tools, and read by an observer. In case of `HiDR`, the original message can be erased, and the copy of a message is harder to find and interpret unless an investigator knows exactly what she is searching for.

6.2. Hidden data retrieval

We demonstrated how records can be effectively hidden from typical queries, but it is also important to make the hidden data easy to retrieve. Since it is possible for the hidden data to be returned with a set of results, the hidden data must be filtered to distinguish it from regular data. The approach that we describe relies on known constraint violations to filter and retrieve hidden data.

We previously discussed primary key and foreign key constraint violations. Queries could then be written to find all data that violates internal constraints. Additionally, a third constraint type that can be violated using record addition is the domain constraint. For example, if a column is declared as a variable length string of 10 characters, VARCHAR(10), and the hidden message is 11 characters long, then a query that filters on all values greater than 10 characters will only return the hidden message.

Fig. 4 illustrates how domain constraint violation can be used to identify a hidden message. If the domain for `EMPLOYEE.Name` is VARCHAR(6), then 'Message1' violates the domain constraint since it is 8 characters. With this knowledge, 'Message1' could easily be retrieved with a query that selects all strings greater than six characters. The DBMS will reject any INSERT or UPDATE query that violates a domain constraint. Therefore, the following query only returns 'Message1':

```
SELECT Name FROM EMPLOYEE
WHERE LEN(Name) > 6;
```

When retrieving hidden messages through a web portal (e.g., Carl in Example 1), it may be impossible to formulate the requisite custom SQL queries. In that case, Alice could add "normal" DBMS records that would automatically show up in Carl's web portal (e.g., create an additional order record or a new user profile entry for Carl's account).

7. Experiments

The objective of this experiment is to demonstrate that `HiDR` can be

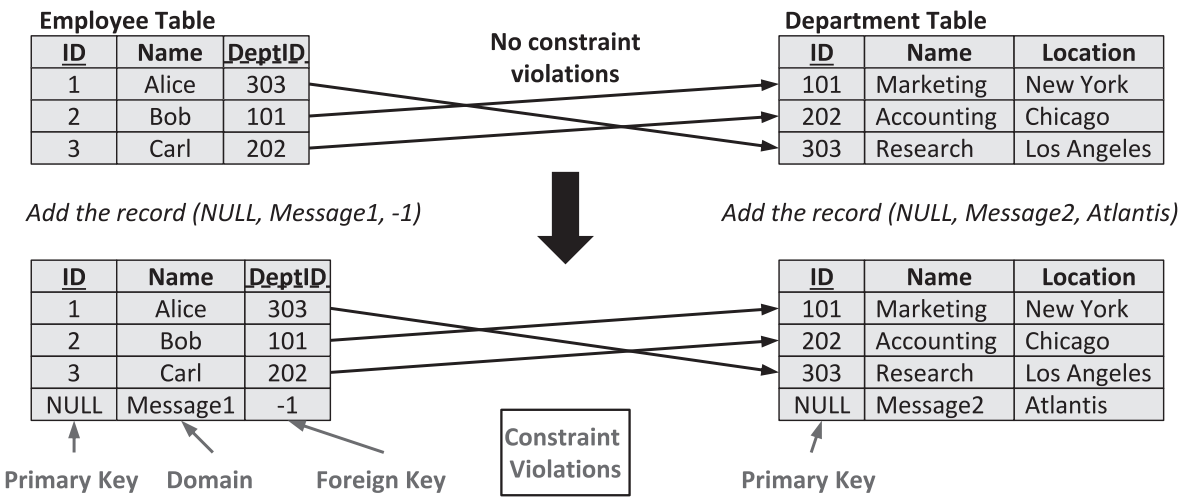


Fig. 4. An example of a record addition to hide a message.

used in practice. We performed HiDR on three representative DBMSes: PostgreSQL 9.6, MySQL 5.6, and Oracle 12c to illustrate that our approach is applicable to all row-store DBMSes that use pages as described in Section 5. PostgreSQL and MySQL are commonly used open-source DBMSes and Oracle is the most widely used commercial DBMS. Table 5 summarizes the three parts to this experiment.

To setup our three DBMS instances, we created the five tables (DATE, SUPPLIER, CUSTOMER, PART, and LINEORDER) from the Star Schema Benchmark (SSBM) (Neil et al., 2009; Lenard et al., 2020), and populated the tables with Scale-4 (∼2.5 GB) data. This benchmark is widely used in the database systems research community. It combines a realistic distributed data (maintaining data types and cross-column correlations) with a synthetic data generator. The primary keys for each table, including (LO_Orderkey, LO_Linenumber) on LINEORDER, created an index for the respective columns by default. Additionally, the LINEORDER used the following foreign key columns: LO_Custkey references CUSTOMER, LO_Partkey references PART, LO_Suppkey references SUPPLIER, and LO_Orderdate references DATE.

We added the record shown in Fig. 5 to the database files containing the LINEORDER table. Our hidden message in this record is 'Hello_World'. The composite primary key is underlined (solid line), the foreign keys are underlined (dashed line), and all values that bypassed a constraint are highlighted in gray.

Checksum. The checksum for each DBMS required a different implementation. Since both PostgreSQL and MySQL are open-source DBMSes, the checksum function was available in the source code. Oracle does not make the checksum available – however, Oracle checksum function was described by Stawiarski (2018).

7.1. Foreign key constraint

In this part, we demonstrate that violating referential integrity excludes a record from typical queries. All SSBM queries (Neil et al., 2009) perform joins using the foreign key columns in LINEORDER. For example, Query 1.1 joins LINEORDER and DATE using LO_Orderdate:

```
SELECT SUM(LO_Extendedprice*LO_Discount)
FROM Lineorder, Date
WHERE LO_Orderdate = D_Datekey
AND D_Year = 1993 AND LO_Quantity < 25
AND LO_Discount BETWEEN 1 AND 3;
```

The hidden record uses a value of −1 for LO_Orderdate which does not match any values in the D_Datekey column from DATE table. The record bypassed referential integrity (normally impossible), and will never be returned by any query that performs an inner join on DATE. Similarly, the record bypassed referential integrity for LO_Custkey, LO_Partkey, and LO_Suppkey setting the value to −1. None of the SSBM queries returned the record with the hidden message because they perform at least one join, thus hiding the message from accidental discovery.

The value NULL could have also been used for the foreign key values. This would exclude the record from any inner joins, and this would not have violated referential integrity. However, we used −1 because this is not an expected value, whereas NULL could be expected by the user.

Table 5
Experiment summary.

§	Summary
7.1	Bypassing the foreign key constraint effectively hides the added records from the typical user queries.
7.2	Using < *preclass* = *"listings"* > *NULL* < */pre* > as the primary key keeps the record from being discovered by an index accessand any unintentional conflicts of future inserted data.
7.3	The intentional domain constraint violation simplifies hidden message retrieval.

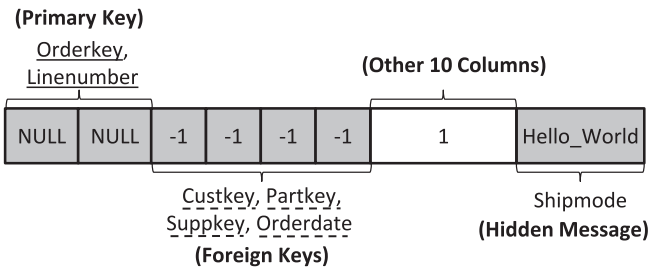


Fig. 5. The record containing a hidden message.

Furthermore, we used −1 because this not only did not match any values in the referenced columns, but it is not likely to match any future referenced values. For example, it is unlikely to assign a customer with an ID of −1 or for an order date to be −1.

7.2. Primary key

The primary key constraint says that all values must be unique and not NULL. By default, all DBMSes create an index on the primary key column(s). By using (NULL, NULL) in the key value (again, normally impossible), we keep the hidden record out of the primary key index and make it less likely to be unintentionally retrieved. As with other indexes, a primary key index rebuild will exclude NULL values.

7.3. Domain constraints

The LO_Shipmode column was declared as a variable length string of 10 characters (VARCHAR(10)). Since 'Hello_World' is 11 characters long, this value violates the domain constraint. The hidden message was easily retrieved by returning only the values that violated domain constraints. All legitimately inserted values for LO_Shipmode must be a string of no more than 10 characters. For example, following query returned our message and nothing else:

```
SELECT LO_Shipmode FROM Lineorder
WHERE LENGTH(LO_Shipmode) > 10;
```

7.4. Experiment conclusion

We claim that it is unlikely for the hidden record to be accidently discovered. If a full table scan returned all 24 million records from LINEORDER, the observer would need to know what to look for to find the hidden record. While we used values that stand out to demonstrate HiDR, in practice the message sender/hider would choose less conspicuous values that do not stand out from the rest. An auditor is also unlikely to search for the hidden record because HiDR bypassed all DBMS logging (as verified in our experiments), leaving no indication that a message was hidden.

8. Database storage validation

While HiDR is presented as a tool designed for ethical purposes, nothing is stopping an individual from using it for malicious applications. This section discusses potential countermeasures and their challenges for such malicious applications.

8.1. Detecting database file tampering

Wagner et al. previously proposed methods to detect database file tampering (Wagner, 2018; Rasin et al., 2018). These methods detect storage inconsistencies by comparing user data in tables and additional data stored in auxiliary structures (e.g., indexes, materialized views);

tampering is detected by identifying an unexplained difference in different data structures (e.g., B-Trees used for indexes, log entries). One of the major limitations of this work is that it requires auxiliary structures such as indexes to be built on the user data stored in tables. Without optional auxiliary structures, there is little information available for comparison to detect inconsistencies. Another challenge in this work is the rate at which detection is performed. Over time, data can be overwritten and data structures could be rebuilt (e.g., B-Tree reorganization).

8.2. Constraint checking

Recall that the DBMS engine does not retroactively check data constraints. Under normal circumstances (i.e., through SQL operations) the DBMS engine enforces these constraints only as data is inserted or updated; however, HiDR bypasses DBMS constraint enforcement. We propose that a database user could run a comprehensive set of queries to check for constraint violations. For example, the following query finds all records (i.e., Message1) that violate referential integrity (i.e., the foreign key constraint) in Fig. 4:

```
SELECT *
FROM Employee LEFT OUTER JOIN Department
ON Employee.DeptID = Department.ID
WHERE Department.Name IS NULL
AND Employee.DeptID IS NOT NULL;
```

Similar queries can be written to validate all standard SQL constraints which could be leveraged for message hiding. In regards to the possible applications of HiDR, the following constraints require validation: domain (i.e., column data types), row-level check constraints, NOT NULL (including primary key set to NULL), and referential integrity constraints. Other types of constraints, such as UNIQUE or multi-table constraints (i.e., constraints enforced by triggers) can be violated through direct storage modification, but do not offer the targeted data retrieval as the HiDR use cases. For example, adding a duplicate value into a column under a UNIQUE constraint does not help retrieve a hidden message. There is no query that can quickly find a row with a duplicate value. Alternatively, setting a column to NULL when a NULL is not allowed (e.g., in a primary key) allows to quickly identify such row as an anomaly.

To automate this approach, a list of data constraints and table definitions can be retrieved from the DBMS system tables. This list of constraints can then be iterated over to write queries that check constraints useful for hidden message passing:

For each datatype constraint, such as VARCHAR(20), a query can be formulated to quickly identify columns in violation of the constraint:

```
SELECT [column name]
FROM [table name]
WHERE LENGTH([column name]) > 20;
```

For each NOT NULL constraint:

```
SELECT [column name]
FROM [table name]
WHERE [column name] IS NULL;
```

For each foreign key constraint:

```
SELECT *
FROM [primary key table name]
      RIGHT OUTER JOIN [foreign key table name]
WHERE [primary key] IS NULL
AND [foreign key] IS NOT NULL;
```

9. Conclusion

In this paper, we presented HiDR, a mechanism for adapting steganography techniques to relational DBMSes. HiDR adds data to the DBMS state by directly modifying the database files rather than going through DBMS API by executing SQL commands. HiDR has three significant advantages: 1) it bypasses all DBMS access control, logging mechanisms, and constraints; 2) messages are retrievable using regular SQL queries; and 3) it is applicable to all row-store DBMSes including Apache Derby, IBM DB2, Firebird, MySQL, Oracle, PostgreSQL, SQLite, and Microsoft SQLServer.

We recognize that HiDR can also be used for malicious activity. Steganography techniques could be used for nefarious purposes, and DBMS storage tampering can be used for activities such as falsifying or tampering with bank transactions. Wagner et al. (Wagner, 2018) discussed general techniques for detecting tampering with database internal storage. In Section 8, we discuss a simple strategy that can validate database storage and detect any of the inconsistencies that are useful for sending steganographic messages via database storage.

Acknowledgments

This work was partially funded by the Louisiana Board of Regents Grant LEQSF(2022-25)-RD-A-30 and by US National Science Foundation Grant IIP-2016548.

References

Ansari, A.S., Mohammadi, M.S., Parvez, M.T., 2020. A multiple-format steganography algorithm for color images. IEEE Access 8, 83926–83939.
Carrier, B., 2011. The Sleuth Kit. URL. http://www.sleuthkit.org.
Conlan, K., et al., 2016. Anti-forensics: Furthering Digital Forensic Science through a New Extended, Granular Taxonomy.
Cui, W., Liu, S., Jiang, F., Liu, Y., Zhao, D., 2020. Multi-stage residual hiding for image-into-audio steganography. In: ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, pp. 2832–2836.
Dhawan, S., Gupta, R., 2021. Analysis of various data security techniques of steganography: a survey, Information Security. Journal: Glob. Perspect. 30 (2), 63–87.
Fruhwirt, P., 2015. Using Internal Mysql/innodb B-Tree Index Navigation for Data Hiding, p. 179 other.
Garber, L., 2001. Encase: a case study in computer-forensic technology. IEEE Comput. Magaz. Jan.
Garfinkel, S., 2007a. Anti-forensics: Techniques, Detection and Countermeasures. ICIW, pp. 77–84.
Garfinkel, S.L., 2007b. Carving Contiguous and Fragmented Files with Fast Object Validation.
Hamid, N., 2012. Image Steganography Techniques: an Overview. Other, pp. 168–187.
Johnson, N.F., Jajodia, S., 1998. Exploring steganography: seeing the unseen. Computer 31 (2).
Kessler, G.C., 2007. Anti-forensics and the digital investigator. In: Australian Digital Forensics Conference. Citeseer, p. 1.
Kester, M.S., Athanassoulis, M., Idreos, S., 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe? ICDE.
Lenard, B., Wagner, J., Rasin, A., Grier, J., 2020. Sysgen: system state corpus generator. In: Proceedings of the 15th International Conference on Availability, Reliability and Security, pp. 1–6.
Lenard, B., Rasin, A., Scope, N., Wagner, J., 2021. What is lurking in your backups?. In: IFIP International Conference on ICT Systems Security and Privacy Protection. Springer, pp. 401–415.
Neil, P.O., et al., 2009. The star schema benchmark and augmented fact table indexing. In: Performance Evaluation and Benchmarking. Springer, pp. 237–252.
OfficeRecovery, 2017. Recovery for Mysql. URL. http://www.officerecovery.com/mysql/.
Patel, R.F., Pragathi, Y.S., 2022. Steganography of encrypted messages inside valid qr codes using wavelet transforms. J. Eng. Sci. 13 (11).
Percona, 2018. Percona Data Recovery Tool for Innodb. https://launchpad.net/percona-data-recovery-tool-for-innodb.
Phoenix, S., 2018. Db2 Recovery Software. URL. http://www.stellarinfo.com/database-recovery/db2-recovery.php.
Pieterse, H., Olivier, M., 2012. Data Hiding Techniques for Database Environments. IFIP.
Rasin, A., Wagner, J., Heart, K., Grier, J., 2018. Establishing independent audit mechanisms for database management systems. In: 2018 IEEE International Symposium on Technologies for Homeland Security (HST). IEEE, pp. 1–7.
Richard III, G.G., Roussev, V., 2005. Scalpel: a frugal, high performance file carver. In: DFRWS.
Stawiarski, K., 2018. Oracle Database Block Checksum Xor Algorithm Explained. URL. http://blog.ora-600.pl/2018/01/28/oracle-database-block-checksum-xor-algorithm-explained/.
Wagner, J., 2018. Auditing dbmses through forensic analysis. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, pp. 1704–1708.

Wagner, J., Rasin, A., 2020. A framework to reverse engineer database memory by abstracting memory areas. In: International Conference on Database and Expert Systems Applications. Springer, pp. 304–319.
Wagner, J., Rasin, A., 2024. Forget about it: Batched database sanitization. In: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, pp. 1441–1452.
Wagner, J., Rasin, A., Grier, J., 2015. Database forensic analysis through internal structure carving. Digit. Invest. 14, S106–S115.
Wagner, J., Rasin, A., Grier, J., 2016. Database image content explorer: carving data that does not officially exist. Digit. Invest. 18, S97–S107.
Wagner, J., Rasin, A., Malik, T., Heart, K., Jehle, H., Grier, J., 2017. Database forensic analysis with dbcarver. In: CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research.
Wagner, J., Rasin, A., Heart, K., Jacob, R., Grier, J., 2019. Db3f & df-toolkit: the database forensic file format and the database forensic toolkit. Digit. Invest. 29, S42–S50.
Wagner, J., Rasin, A., That, D.H.T., Malik, T., Grier, J., 2020a. Odsa: open database storage access. In: 21st International Conference on Extending Database Technology.
Wagner, J., Rasin, A., Heart, K., Malik, T., Grier, J., 2020b. Df-toolkit: interacting with low-level database storage. Proc. VLDB Endowment 13 (12), 2845–2848.
Wagner, J., Nissan, M.I., Rasin, A., 2023. Database memory forensics: identifying cache patterns for log verification. Forensic Sci. Int.: Digit. Invest. 45, 301567.