Towards a Forensic-Aware Database Solution: Using a secured Database Replication Protocol and Transaction Management for Digital Investigations

Peter Frühwirt[a], Peter Kieseberg[a], Katharina Krombholz[a], Edgar Weippl[a]

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

Abstract

Databases contain an enormous amount of structured data. While the use of forensic analysis on the file system level for creating (partial) timelines, recovering deleted data and revealing concealed activities is very popular and multiple forensic toolsets exist, the systematic analysis of database management systems has only recently begun. Databases contain a large amount of temporary data files and metadata which are used by internal mechanisms. These data structures are maintained in order to ensure transaction authenticity, to perform rollbacks, or to set back the database to a predefined earlier state in case of e.g. an inconsistent state or a hardware failure. However, these data structures are intended to be used by the internal system methods only and are in general not human-readable.

In this work we present a novel approach for a forensic-aware database management system using transaction- and replication sources. We use these internal data structures as a vital baseline to reconstruct evidence during a forensic investigation. The overall benefit of our method is that no additional logs (such as administrator logs) are needed. Furthermore, our approach is invariant to retroactive malicious modifications by an attacker. This assures the authenticity of the evidence and strengthens the chain of custody. To evaluate our approach, we present a formal description, a prototype implementation in MySQL alongside and a comprehensive security evaluation with respect to the most relevant attack scenarios.

Keywords: MySQL, InnoDB, digital forensics, databases, data tempering, replication, transaction management

1. Introduction

Common ACID-compliant *Database Management Systems (DBMS)* provide mechanisms to ensure system integrity and to recover the database from inconsistent states or failures. Therefore they contain a large amount of internal data structures and protocols. Their main purpose is to provide basic functionality like rollbacks, crash recovery and transaction management, as well as more advanced techniques like replication or supporting cluster architectures. They are solely intended to be used by internal methods of the system to ensure the integrity of the system.

Since databases are typically used to store structured data, most complex systems make use of at least basic database techniques for forensic analysis. Thus, when investigating an arbitrary system, standardized forensic techniques targeting the underlying database allow an investigator to retrieve fundamental information without having to analyze the (probably proprietary) application layer. Database forensics support efficient forensic investigations in order to e.g. detect acts of fraud or data manipulation. However, little attention has been paid on the enormous value of internal data structures to reconstruct evidence during a forensic investigation.

To illustrate the need for guaranteeing that a database is unaltered, the following questions may be useful in the course of some digital investigations:

- Was a data record changed in a certain period of time and at what exact moment?
- Was data manipulated in the underlying file system by bypassing the SQL-interface?
- What statements were issued against the database in a given time frame?
- How have manipulated data records been changed with respect to the time line?
- What transactions have been rolled back in the past?

In this paper, we propose a novel forensic-aware database solution. Our approach is based on internal data structures for replication and transaction that are used by the database for crash recovery. They are in general not human-readable and intended to be read and used only by internal methods of the system. The overall benefit of our method is that no log files such as administrator logs are

Email addresses: pfruehwirt@sba-research.org (Peter Frühwirt), pkieseberg@sba-research.org (Peter Kieseberg), kkrombholz@sba-research.org (Katharina Krombholz), eweippl@sba-research.org (Edgar Weippl)

needed in order to create an entire audit trail as a pre-incident security mechanism. Furthermore it can be used as a post-incident method to locate unauthorized modifications. Our approach is feasible for all ACID-compliant DBMSs, because it solely relies on the standard replication and transaction mechanisms. In addition to that, our approach aims at securing these data structures against retroactive malicious modifications in case of an attack scenario to guarantee the authenticity and integrity of the reconstructed forensic evidence. To demonstrate the feasibility of our approach we provide a formal description and present a prototype implementation in MySQL. Furthermore, we provide a comprehensive security evaluation to demonstrate the benefits for system security and the integrity of the forensic evidence.

The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 provides a description of our approach. In Section 4 we present a showcase implementation of our approach based on MySQL. In Section 5 we evaluate our solution with respect to security and applicability. Finally, Section 6 concludes our work.

2. Related Work

Due to the ever rising importance of incorporating computer systems and equipment in investigations, computer forensics is an emerging field in IT-Security [1]. In this section, we present related scientific work with respect to database forensics and secure logging.

2.1. Database Forensics

In digital forensics, log files on the operation system level have been used as a vital source to collect evidence in the last decades. Still, as several authors demonstrated in the past, the database layer is unpopular when it comes to forensic exploitation, even though it constitutes an integral part of enterprise assets.

In 2009 Martin Olivier provided a thorough review on the then current state of the art in research on database forensics [2] and compared it to the then state of the art of file forensics. Five years later the situation has not changed much.

However, the amount of literature in the area has increased in recent years compared to what was available back then. The topic has been featured at the IFIP WG 11.9 meetings in recent years. The papers featured covered a range of topics relating to database forensics. Beyers et al. discussed the creation of a method to separate the different layers of data and metadata to prepare a database management system for forensic examination [3]. In a similar vein, Fasan et al. demonstrated how a database reconstruction algorithm can be utilized to reconstruct a database allowing an examination to be performed [4]. Pieterse et al. discussed the various techniques that can be used to hide data within a database caused by the complexity of databases and the lack of forensic tools with which to examine databases [5]. Lalla et al. described a model for investigating computer networks through network log files and how the examination of said files could reveal concealed activity [6].

The Digital Investigation Journal has published two papers on database forensics in the last five years: Martin Olivier describes the pertinent differences between file systems and databases and how file system Forensic techniques could possibly be applied to database forensics. The paper also attempts to highlight potential areas of research within database forensics [2]. In 2012 Fasan et al. published an extended version of the respective IFIP publication [7].

Additionally, in [8] the authors discussed research challenges in database forensics and regret the lack of attention to this field until now. Their main concern lied in the absence of practical tools available for forensic analysts. In a very recent work [9], the authors aimed at developing practical techniques to exploit the database layer. One of their techniques aims at providing a better reconstruction method for changed data, another method focuses on providing confirmative evidence on data stored in a database. Furthermore, the authors emphasized the absence of in-depth research regarding database forensics.

2.2. Using Database Internals for Forensic Investigations

In a student work in 2005 [10] it was shown that data stays persistent in the files system layer when using MySQL or PostgreSQL. This work was extended in two subsequent papers [11, 12], both focussing on privacy aspects in database systems. In these papers they pointed out where data is preserved inside the internal structure of the DBMS: Table storage, the transaction log, indexes and other system components.

In a series of practical resources [13, 14, 15, 16, 17, 18, 19, 20], Litchfield demonstrated the possibilities of recovering data from the redo log, dropped objects and other sources for Oracle 10g, release 2 running on a Windows server.

In [21] an analogy to file carving targeting MySQL databases based on internal data files was presented, i.e. recovering stored data, even if it is not available anymore via the SQL-interface.

Recently, new techniques to exploit internal log files for forensic purposes were developed: In [22] the authors developed a forensic approach based on data stored in the internal redo logs used for rollbacks and undos. They demonstrated an efficient way to extract simple INSERT, DELETE and UPDATE statements, including eventually deleted information. Furthermore, they analyzed the overall structure of these redo logs. An enhanced reconstruction of data manipulation queries is presented in [23].

On related terms, in [24] the authors outlined the need to develop forensic-aware databases that allow an efficient and provable extraction of forensic information during investigations in order to develop SOX-conforming database applications. Furthermore, the authors developed a new logging strategy based on the underlying data structure of database indexes in [25] as a first step towards a forensic-aware database.

2.3. Versioning and Archiving Mechanisms for Databases

Versioning and Archiving is often associated with the generation of backups, still there is a major difference between these concepts. Backups are usually not designed for efficient retrieval of past states of single records, since their main purpose lies in providing an exact copy of a former overall snapshot that can be played back efficiently as a whole. This especially holds true for real-life implementations using incremental backups.

Researchers have devised a large range of tools for the versioning of data that preserve complete histories with query capacities [26, 27, 28], but archives and temporal databases are rarely used in practice. The versioning is usually implemented directly in the application layer or in the middleware. The approach outlined in this paper is fundamentally different to the problem of Versioning and Archiving, since the main purpose of our approach lies in the verification of the actual data in the database and not in the restoration of former states. This is especially important, since in this approach not all data required for restoration needs to be stored for providing a verified snapshot. Furthermore, Versioning and Archiving do not provide means against data tampering per se, they only allow simple fallback to a previous state that may, depending on the assumptions of the data owner, be untainted.

2.4. Log File Integrity and Secure Logging

In [29] Schneier et. al. describe a method to provide forward secrecy to log entries, making it impossible for the attacker to read any log entry generated before the machine was compromised. Furthermore, this approach detects modifications and removal of log entries. The main idea behind this approach lies in making each log entry depending on the previous ones by applying a hash chain. Contrary to our approach, the one proposed here additionally encrypts the logged data. Further approaches can be found in [30] and [31]. This is not feasible for our approach, since the internal data structures that are utilized in this paper are not designed for logging, but need to stay readable to the DBMS in order to serve their main purposes.

Regarding log file integrity, Marson et. al. developed an approach for practical secure logging in [32]. Outlining the importance of log files for forensic investigations, the authors discuss the problem of log authentication of locally recorded logs: In case of an intrusion, mechanisms are needed to protect the collected log messages against a manipulation by an intruder. The authors propose two main requirements: (i) These mechanisms need to be forward-secure and (ii) they need to be traceable in order to enable the auditor to verify the integrity and especially the order of the log entries. The authors proposed the *seekable sequential key generator* that fulfills both requirements and can be applied directly in logging.

3. Chained Witnesses Approach

In this section we outline the fundamentals of our chained witnesses approach followed by a brief discussion of the security requirements.

Basically, every ACID-compliant DBMS needs to fulfill requirements like transaction safety, replication and rollbacks in order to ensure atomicity and consistency. Therefore, the relevant data is stored in internal data structures. If these data structures could be changed in a way to protect their authenticity, this would constitute a vital source for forensic investigations. To this end, we provide an approach that prevents undetected manipulations of these internal mechanisms. Therefore, it assures the authenticity of these information vaults as witnesses in course of forensic investigations.

3.1. Functional Prerequisites

In this section, we present the requirements on which our approach is based.

3.1.1. Internal Data Structures

Nowadays all major database management systems provide mechanisms to recover from inconsistent states or internal failures, rollbacks, or to provide distributed states within a short period through replication.

Their generic implementation can further be used to implement internally signed log files. Every large database management system utilizes internal log files for crash recovery, e.g. Oracle [33] or Microsoft SQL-Server [34], that can be used to implement signed log files to verify the integrity and authenticity of stored data. Also the replication protocol can be used for our purpose.

MySQL transaction logs[35] in general consist of redo- and undo logs. The purpose of redo logs in general is to apply changes that were made in the memory, but where not flushed to the permanent table records. They are used in order to recover the database from an inconsistent state provoked by a crash. Undo logs are used in case a user does not complete a transaction with a commit or rollback to undo the previously executed actions. Most database management systems use different internal data structures to ensure the authenticity of the written data. MySQL replication basically relies on binary logging. The updates and changes are treated as events and applied to the slaves based on the assumption that the master is "dumb". In Oracle[33], crash recovery is performed by means of redo log files. Oracle applies redos automatically without user intervention. After the crash of a single instance or the entire database cluster, these redo logs are applied to restore the database. Other ACID-compliant DBMS solutions, such as MSSQL[34], IBM DB2[36] and Teradata[37] provide similar mechanisms. In the following, we outline the most significant differences between the transaction mechanism data and the data replication protocol.

Transaction mechanism. The transaction mechanism is mainly used to facilitate rollbacks and undos to guarantee atomicity. Therefore, it contains all the required information to restore former versions of the data, but it does not store information on events that cannot be undone by a rollback, such as changes to table structures, general *Data Definition Language(DDL)* commands or changes to *large objects* (cf. 5.4).

Data Replication protocol. The data replication mechanism stores data in order to mirror the whole database from a *master* instance to so-called *slave* or *replication* instances, usually in order to implement a redundant storage. Hence, the information stored in this data structure is complete in the sense that the logical structure and the content of the databases is completely identical, also including more sophisticated information objects like *large objects* and metadata (e.g. session information, timestamps, etc.).

Our Chained Witnesses approach is based on the transaction and replication logs discussed in this section. The aim of our approach is to implement an audit trail of these internal mechanisms against external modification, even by the database administrator. Our approach works with both mechanisms, in Section 4 we present a prototype implementation based on MySQL.

3.1.2. Authenticity

Since the database replication and transaction mechanisms contain data that is relevant for forensic investigations, it is of the utmost importance to assure the authenticity and integrity of this data. Furthermore, tampering with the database replication may not only thwart forensic investigations, but will also provoke inconsistencies between the master database and the slaves.
To enable for timeline analysis, a timestamp is added to each information fragment (e.g. an executed query). This timestamp enables the reconstruction of the sequence of changes. In addition to that, it yields a method to align the order via timelines derived from logging mechanisms or events on the (file) system.

For a forensic analysis and to assure consistency of transmitted data between the master and the slaves, data integrity and authenticity must be assured. In order to achieve this, we adapt an approach involving chained hashes based on secure logging (see 3.4).

3.2. Formal Description

Our formal framework works independently from the actual source of the forensic information, i.e. regarding the data sources outlined in Section 3.1.1, the approach can be applied to any arbitrary internal mechanism and log.

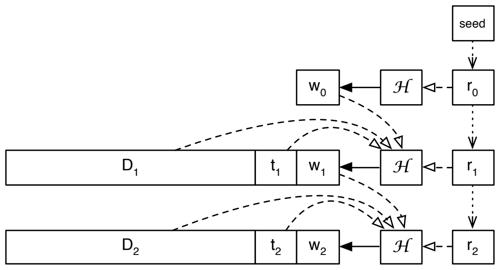

Figure 1: Hash chain on information fragments

We assume that our DBMS is running on an untrusted system $\mathcal{S}$. Let n be the number of data fragments from our source and D_i denote the i^{th} data fragment ($i \leq n$) stored by the respective internal mechanism at time t_i on $\mathcal{S}$.

Let $\mathcal{R}$ be a cryptographically secure pseudo-random number generator (CSPRNG) that was originally initialized with a secret seed s that is known by a trusted third party $\mathcal{T}$ and that is not known to $\mathcal{S}$. Thus $\mathcal{R}$ has to generate random values that pass statistical randomness tests, satisfy the next-bit test and has to withstand state compromise extensions, so that an attacker is not able to reconstruct previous results based on a known value, especially not the secret seed s. Following, r_i denotes the value of the i^{th} iteration of $\mathcal{R}$.

Furthermore, let $\mathcal{H}$ denote a cryptographic one-way hash function. In our approach we assume that the hash function $\mathcal{H}$ and the pseudo-random number generator $\mathcal{R}$ are secure.

In order to guarantee the authenticity of the information stored in our internal mechanism, each data fragment is appended with time information and a *witness* w_i containing the hash of it's data D_i, together with the current iteration of the pseudo-random number generator r_i, a timestamp holding t_i and the witness of the previous data fragment w_{i-1} (see Figure 1). More precisely, this can be expressed as:

$$w_i = \mathcal{H}(w_{i-1}||D_i||t_i||r_i)$$

where $||$ denotes string concatenation. We call the tuple (t_i, w_i) the *signature* of D_i and the tuple (D_i, t_i, w_i) the *verifiable data set* of the i^{th} data fragment. The set of all n such tuples we call the *verifiable data log*.

In the initialization phase, e.g. during installation, a trusted third party $\mathcal{T}$ selects the random seed s and subsequently generates the first output r_0 of the random number generator, the first witness w_0 is defined as $w_0 = \mathcal{H}(r_0)$.

3.3. Verification

In this section, we propose a workflow for the verification of the authenticity of the stored data as shown in Figure 2. Each file change is on the one hand written to the data storage and on the other hand to the internal data structures used for transaction management and/or replication. Therefore all changes are documented and can be used to calculate the current state starting from an older one (e.g. a backup) - a common practice during crash recovery.

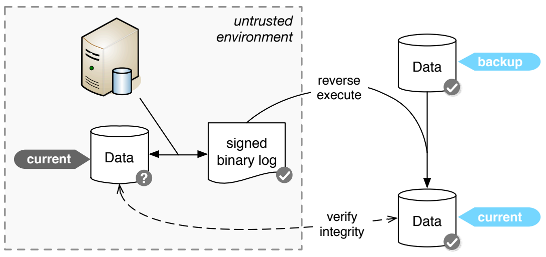

Figure 2: Verification of the authenticity of data

The verification system reads the verifiable data log in reverse order and verifies the witness of the data set using the secret initial vector of the random generator. If the witness is not valid, the verifiable data set was manipulated. After executing all data sets to the old state of the database, it is possible to compare the current state with the calculated state byte-wise. This uncovers all changes that were made directly in the file system. Irreproducible hash values show manipulations of the internal data storage mechanisms and can thus be used for detection.

This workflow verifies the authenticity of the database management system and automatically shows unauthorized changes that can be used for further investigations. Furthermore, besides revealing tamperings with the log, it also shows the exact position of the manipulation in the file system, as well as the changes introduced by the manipulation.

3.4. Security Requirements

Forward Secrecy. A key-agreement protocol is based on a set of long-term keys to derive session keys. The protocol possesses the property of *forward secrecy*, if it is not possible to deduce past session keys in case one of the long-term keys is compromised. Forward secrecy is often targeted in cryptographic applications, e.g. [38] and [39]. This feature is required so that a successful attacker cannot alter evidence of events from before the system was corrupted.

Secure Logging. Logging information must be secured in order to be suitable for forensic analysis, especially if the results of the investigation are used in court [40, 41]. During the last decades, several mechanisms for achieving untampered log files were proposed. The work by Schneier [29] relies on a chaining mechanism (see 2.4). Due to the construction of the verifiable data log, the use-case for secure logging is similar to our approach.

Cryptographically Secure PRNG. Our approach heavily depends on CSPRNG and has to fulfill the following properties:

- The output of the CSPRNG has to be reasonablly long enough and has to pass statistical randomness tests.
- The CSPRNG has to satisfy the next-bit test, which means that the first k bits of a random sequence given, there is no polynomial-time algorithm that can predict the next bit $((k+1)^{th})$ with a probability of success better than 50% [42].
- The CSPRNG has to withstand state compromise extensions. Therefore it is impossible to reconstruct a sequence of random numbers prior to the revelation of a part or all of its states of the CSPRNG.

Hash Function. In order to provide witnesses that cannot be constructed by an adversary, the hash function used to construct them needs to be a cryptographic hash function. This includes resistance against preimage and second preimage attacks, as well as collision resistance.

Further details on these requirements can be found in Section 5.1.1.

3.5. Handling closed source DBMSs

Many relevant state of the art database management systems do not publish the source code. Thus, in order to apply our approach as outlined in Section 3, the vendors of these database management systems would need to extend their existing code bases. While this is highly desirable, it may collide with other interests of the vendor. Thus, we present a modification for our approach that does not rely on code changes. Nevertheless, this modification is currently limited to using data derived by the data replication protocol. Figure 3 provides an overview on this closed source modification.

The main difference is that signing the data replication entries and generating the respective witnesses is not done on the master node. The data replication traffic is captured when it is dumped to the slave nodes and the signing procedure is applied on the transport or session layer: The data stream is simply chopped into chunks of a fixed length appended by the signature. For verification, the witnesses of the signed traffic are checked and the traffic is replayed to a slave in a trusted environment. This slave is in the original state of the time when the traffic was collected. The slave applies the changes derived from the traffic and compares the resulting state with the state of an arbitrary slave node from the untrusted environment. We consider deviations from this state as evidence for modifications.

Our modification is largely depended on the particular DBMS and the underlying infrastructure. This covers the simulation during replay to the slave in a trusted environment, such as hand-shakes, synchronization or encryption, and other implementation-depending characteristics. Furthermore it has to be discussed how attacks on the network level can influence this modification.

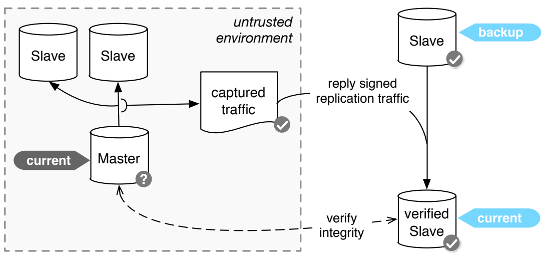

Figure 3: Handling closed source DBMSs using a replication protocol

3.6. Comparing Internal Data Structures

As discussed in Section 3.1.1, there are mainly two mechanisms in most database management systems that can be used to implement our approach: The transaction management system and the replication protocol. In this section we provide a more detailed discussion on the differences between the data collected by the rollback mechanism and the features provided by the data replication protocol, since both mechanisms collect different data and therefore have different advantages during the verification process.

Support of all data manipulation queries. Per definition, the replication mechanism collects all data that is required to generate duplicates of the master database on the slaves. Thus, it is not possible to omit any data objects, since they would be missing. In contrast, the transaction management mechanisms only store data manipulation queries that support rollbacks. On many modern database management systems exist data types that do not support this feature, e.g. BLOBs (binary large objects) in ORACLE and MySQL. Inserting such an object automatically results in a commit-statement, making an entry by the transaction mechanism superfluous.

Query Context. Many database manipulation queries rely on the context the query was executed in, e.g. considering timestamp-sensitive queries, updates that use random numbers, query properties (e.g. auto-increment) or unique checks. The replication mechanism stores all this information in order to push it to the slaves; still, most of this context is not needed for rollbacks, thus it is not stored by the transaction management.

Metadata. Even if the same data is stored in different databases, there are notable differences in meta-information [24]. For example, the insert order or the application of performance enhancing methods like indexing can produce different data structures on the storage side of the DBMS. This is completely transparent to the user and even the database itself, since it yields no changes in the content, nor the context; still, when trying to apply more advanced techniques for database forensics like [25], this can yield differences. Since the main goal of the replication mechanism is to provide slave nodes with the same content, this metadata is not transmitted and therefore not stored by the replication. The transaction mechanism on the other hand stores enough information for crash recovery or complete rollbacks, thus contains significantly more metadata like creation of index pages and record markups.

Interfaces. In order to apply the approach as presented in this paper, the data store of the internal mechanisms needs to be altered. Hence, the question of finding an interface to the respective internal data is of utmost importance. For the replication protocol, interfacing is rather straightforward: Since the changes are pushed to the slave nodes, it is mandatory for every DBMS to provide interfaces to collect the data. Closed source solutions provide possibilities to mirror the data to a slave node and store it together with its witness. The transaction management on the other hand is a purely internal mechanism that needs no connection to the outside world, thus, in common database management systems, no interfaces exist and/or alter the data structure on file level. In order to implement our approach, the developer needs access to the source code of the DBMS, which is a problem in the case of closed source products. Nevertheless, as outlined in Section 4, for open source the utilization of these sources remains perfectly feasible.

Reliability after crashes. While the needed information is stored by the transaction management as soon as possible in order to provide mechanisms for crash recovery, the data collected by the replication protocol is not time critical, thus gets pushed to the slave nodes in batches as well as in reasonable time frames. Therefore, the data derived by the transaction management is more reliable in the case of a crash.

4. Implementation

In this section we present a feasible implementation solution of our approach as an extension of the open source database management system MySQL. As precondition we presume that MySQL implemented a correct and complete replication- and transaction system. To avoid unwanted side effects of our modifications, we minimized the needed source changes and at the same time made changes, e.g. added a signature to a created slack space, which are ignored by the database during daily tasks.

4.1. MySQL Replication

MySQL implements an asynchronous master-slave replication [43] which is shown in figure 4. One server acts as a master and maintains a log of changes that updates the database in one way or another (e.g. updates of data-rows) and events, such as the creation of pages. This log is called *binary log*. In addition it is often necessary to store some metadata (e.g. transaction ID) to reconstruct the context of an update.

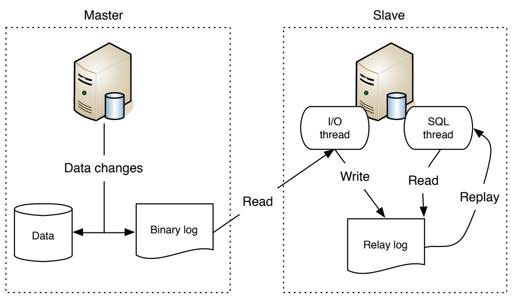

Figure 4: How MySQL replication works

A slave connects to the master, reads the binary log and starts to execute updates according to the binary log. The replication is processed on the client by two threads: The *I/O thread* and the *SQL thread*. The I/O thread downloads the binary log from the master and writes all log entries to temporary files called *relay logs* which are stored locally on the slave. These relay log files are processed by the SQL thread. This thread reconstructs the context of every transaction and executes the updates.

The slave keeps track of the replication process by using two parameters: the current log name and the current log position [44]. If a slave disconnects from the master it will request all log entries starting from the current position. During initial phase the slave requests the first log known to the master.

As mentioned above, the replication of MySQL is asynchronous. In practice, a slave will at some point in the future catch up to the current state of the master, however the master will not wait for the slaves.

MySQL supports two kinds of replication: statement-based replication and row-based replication. The statement-based approach locks every SQL statement that modifies the data. The slave re-executes the logged SQL statements against the same initial data set in the same context. In contrast, the row-based approach logs every row modification which will be later applied to the slaves.

4.2. Binary Log Format

The code that deals with the binary log format can be found in *sql/log_event.cc* and *sql/log_event.h*. Every binary log file starts with the magic number sequence *\xfe\x62\x69\x6e*, which is used to identify the file and to quickly check for completeness. This 4-byte block is followed by the sequence of log entries. Each log entry consists of a header with a fixed amount of fields listed in Table 1, which is written in *Log_event::write_header()* in *sql/log_event.cc*.

The header is followed by the body of the log entry which varies generally according to each type. The body is written by *Log_event::write_data_header()* for a fixed length, event-specific information block followed by a method call of *Log_event::write_data_body()* which writes the actual payload of the event. Each type of log event has it's own implementation of these two virtual methods and therefore its own way to handle these event-type data storage.

Offset	Length	Interpretation
0x00	4	Timestamp of the event
0x04	1	ID of the event (e.g. Write Rows, Start, Create File, etc.)
0x05	4	Server ID which uniquely identifies the server to avoid infinite update loops
0x09	4	Length of the event including header (in bytes)
0x0D	4	Offset of the event in the log (in bytes)
0x11	2	Event flags

Table 1: Binary log entry header

MySQL supplies a tool called *mysqlbinlog* that is able to read binary log files and dumps its content in SQL format with some comments concerning replication. We've developed a prototype based on the implementation of this tool in order to evaluate our approach.

4.3. Transaction Logs

Beside replication there exist other log files which contain a range of context and data of a database management system. Because of the architecture of MySQL it is possible to switch storage engines depending on the desired use case [35]. For example, one storage engine is InnoDB which provides the standard ACID-compliant transaction features, along with foreign key support [45].

InnoDB uses an internal log format to ensure transactions and crash recovery [46]. Every change in the file system will cause at least one log entry that basically consists of a header with a dedicated type, transactionID, position in the modified file and a body which contains a dump of the original file part that was modified [22, 23].

Other (commercial) database management systems like SQL Server [47] are using similar log files to reverse transactions and to recover the database.

> "Although you might assume that reading the transaction log directly would be interesting or even useful, it's usually just too much information." [48]

However, this information contained in the transaction log file - even if it is too much information for manual investigations - can be utilized through our approach in order to enhance the integrity and authenticity of the log information.

4.4. Showcase Implementation

A fundamental feature of our prototype implementation is to minimize the changes in the core of the database management system to ensure maintainability over time (e.g. releases of software and security updates). Furthermore, the introduced changes must not influence the functionality of the DBMS.

MySQL is using pointers for navigation through the log files. Our approach manipulates these navigation pointers to generate slack space between two log entries by changing the length/offset field value of an entry that is used to calculate the pointer to the next entry. Therefore the generated slack space is ignored by the DBMS and does not influence the functionality (see Figure 6 and Figure 7). Thus, we can still guarantee the full functionality of the database while introducing only a small number of code changes, which enhances the probability that the functionality is stable with respect to version changes and updates. In case of InnoDB, the original redo log format was introduced in MySQL 5.0.3 (2005) and not changed until MySQL 5.6.2 (2013), which is the current implementation of the redo logs. However, the code changes (e.g. improved compression) introduced in later versions do not affect our implementation approach. All implementation proposals are based on MySQL 5.6.17 (Community Edition).

InnoDB Transaction Log. InnoDB creates at least one log entry per file change. Every log entry contains a generic header, the location of the origin in the file system and a copy of the overwritten data. The method *log/log0log.cc log_write_low()* is responsible for the creation of the log entry. It receives a byte string which basically contains the dump of the overwritten data and the log header as well as its length. InnoDB creates a container for each log entry with a fixed size of 512 bytes. If a new log entry exceeds the size of the log block, it gets split into two chunks allocated to two log blocks. In our approach we ex-

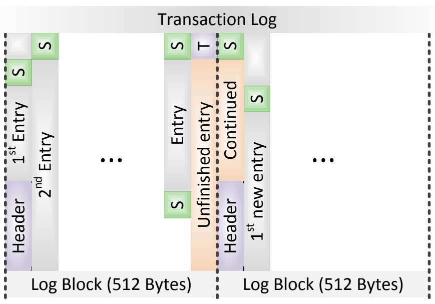

Figure 5: Log Signatures

tended this data with our log signature and modified the length of the written log block. The signature is added to the end of each log entry as trailer (Figure 5). Each log block contains a length definition of itself, therefore InnoDB ignores all added signature data. In addition, these length definitions are used to calculate the offsets which lead to the next log entry. The method *log/log0log.cc recv_parse_or_apply_log_rec_body()* parses the log entry data and returns a pointer to the next log entry. We modified this pointer and added the length of the signature to it. Thus we created on the one hand some kind of slack space that is skipped by InnoDB and does not disturb the functionality of the transaction system (Figure 6), and on the other hand gained the ability to store arbitrary signature data that is used in the verification process.

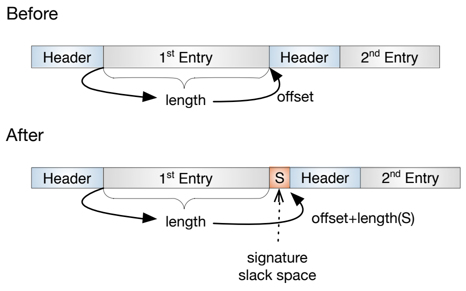

Figure 6: Signature extension of InnoDB transaction logs

MySQL binary log. Similar to the implementation of the signature of the transaction log, it is possible to add our approach to the binary log of MySQL which is used for replication. As mentioned in section 4.2, every log entry needs a log header. This header contains information about the length of the log record and an offset to the next log entry.

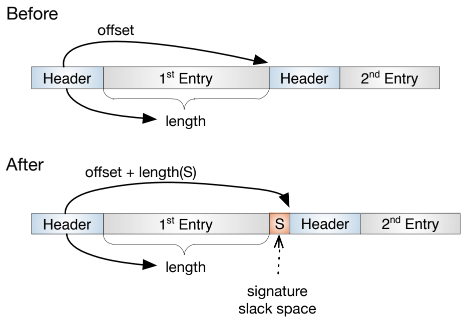

Figure 7: Signature extension of the MySQL binary log

Figure 7 shows how we modified the implementation to add our signature. Similar to our previous modification of the InnoDB transaction log, we generated some slack space for the signature by modifying the pointers. However, in this case it is easier, because only a simple change of the offset enables the opportunity to add a signature to each log entry. This signature is used in combination with a modified version of the *mysqlbinlog*-tool to verify the integrity of the written data records.

5. Evaluation

In this section we evaluate our approach with respect to attack scenarios, general security requirements and performance.

5.1. Attack Scenarios

We identified the following attacker models and assets which are related to specific aspects of our approach.

5.1.1. Attacker Models

In this section we describe the potential attacker in detail. Both attacker models are common in big database applications like modern data warehouse environments that guarantee 24/7 support [49, 50]. Furthermore, the target of this analysis lies in large scale implementations, e.g. in the telecommunication industry [51].

Malicious File System Administrator. This attacker possesses read- and write access to the file system of the data warehouse, especially the data files of the database management system. Thus he is able to change data without invoking any DBMS interface at all. More precisely he has the following abilities:

- modification of arbitrary files belonging to the DBMS, especially those containing all tables and internal data structures (e.g. log files) directly into the file system; result: modifications that are not monitored by any part of the DBMS.
- modification of the OS log system
- no access to the database query interface, especially no administrator rights
- no root on the server on which the actual database instance is running. More precisely, the attacker has no means of directly accessing the RAM of the database server.

Malicious Database Administrator. The database administrator's role lies in managing and supporting the installed database. More precisely, we assume the following rights and limitations:

- administrator rights on the database itself
- only read access to the file system
- able to change logging routines and user rights with respect to the standard methods deployed by the DBMS

5.1.2. Assets

Here we provide a comprehensive description of the identified security assets. In this security evaluation we consider only assets that are interconnected to the fundamental characteristics of our approach. Other aspects are out of the scope of this work and therefore not discussed within this analysis.

DBMS Instance:. This asset covers the actual running instance of the DBMS, including the ability to shut down and restart it.

Configuration:. The ability to read and even change the configuration during runtime or in the file system is referred to as *configuration asset.* The fundamental configuration changes within this asset are, amongst others, changes of the size of internal data structures and configuration of replication- and logging strategies.

Slaves and Network Connection:. Since this asset is not unique to our approach and heavily depends on the actual implementation of the associated techniques as well as the general server and network architecture, it is not considered within this evaluation. However, in case of actual implementations, it must be assured that these attack vectors are mitigated using state of the art techniques.

SQL Interface:. The SQL interface allows the execution of all legal DBMS-commands, despite the name not limited to SQL, but including stored procedures, internal commands and DDL.

Data Files:. Databases store their content in so-called *data files* in the file system. Direct access to these files could be used in order to change data without invoking the DBMS, thus circumventing all logging- and monitoring mechanisms.

Internal Data Structures:. This asset refers to the temporary internal information that is used by the DBMS to guarantee its functionality, e.g. the replication protocols and the transaction management system.

Random Number Generator:. The random number generator is a critical asset concerning our approach, since it gets in place with the unknown component to thwart forging and to establish forward secrecy.

Source Code:. This asset refers to the actual source code of the DBMS, including the possibility to patch and recompile it in order to remove or add functionality.

5.2. Security Evaluation

In this section we compare the assets defined in Section 5.1.2 and the attacker models outlined in Section 5.1.1 and discuss why our approach is secure with respect to the models (as shown in Table 2). The security of the untrusted machine relies on the fact that the database system creates internal log files with an initially shared secret on a trusted verification machine. This secret is used as seed for the cryptographically secure pseudo-random number generator that is utilized to create the signatures for the data particles.

The authenticity and security of the witnesses is based on three fundamental facts:

- The data (i.e. the random number of the n^{th} iteration and the hash value of the previous record) which is a witness for the authenticity is hashed using a one-way hash function.

- Each witness is secured by an individual key generated by a cryptographically secure pseudo-random generator that is derived using a one-way process, which is initialized by a secured and trusted environment (e.g. during installation by a trusted person). Due to the one-way process the shared secret is not derivable. Without the individual key it is not possible to alter the signature.

- Each witness contains the previous entries in form of a hash chain that is secured by the authenticity of all previous witnesses. Therefore if an attacker alters a verifiable data set, all future witnesses have to be altered and recalculated too. This is not possible if the attacker doesn't know each individual key generated by the cryptographically secure random number generator.

5.2.1. Modification of Configuration

In case the attacker has access to change the configuration, he could disturb the logging mechanisms extended by our approach. Thus, he could make changes without witnesses. Still, these changes would be detectable by our verification process (see Section 3.3), since the changes would not be present in the verifiable data log.

5.2.2. Deployment of New Code

The attacker could use his privileges to revert the implemented security features, especially considering our approach. Without root access this would require at least a restart of the whole DBMS, which should trigger alarms (cf. 5.4). Furthermore, these changes would again be visible in the verifiable data log. Still, the attacker could patch the code in order to record the randomly generated sequence numbers. We outlined possible solutions for this scenario in Section 5.4.2.

5.2.3. Denial of Service

Since all internal logging mechanisms have a predefined size and/or lifetime, a vast amount of queries could be used in order to overwrite interesting parts. This can either be solved in our approach by regularly saving the log files to the verification machine (in our closed source approach this happens automatically), or by writing a second log without log rotation and size limitations (see Section 5.4.4).

5.2.4. Execution of Queries

The database administrator is able to execute any kind of legal query, even those which may be considered malicious. Still, in our approach he is not able to deny the execution - as it would be possible without this mechanism, in which case he could e.g. simply delete the query from the execution log. Thus, our approach delivers a new forensic-aware database solution.

5.2.5. Unmonitored Modifications

An attacker possessing write access to the data files of the DBMS could insert or modify data bypassing the DBMS and its logging and security mechanisms (e.g. user permissions). With our approach, these changes remain easily detectable, since they are not included in the verifiable data log. Still, in case of temporary changes, an attacker could insert new data and delete it again between two verification iterations. As a countermeasure against this attack, the verification should be done at a random interval, making it reasonably harder for the attacker to hide such changes.

5.2.6. Modification of the Verifiable Data Log

An attacker (as depicted in the model of the malicious file system administrator) could tamper with the internal data structures that represent the verifiable data log. Still the attacker would not be able to manipulate the entries, since he is not able to generate the required secret random numbers to generate the required witnesses. Furthermore, it is not possible to add a new entry between two existing ones, or delete an existing entry, since the witness of an entry relies on the previous witnesses due to the chained hashing (see Section 3.1.2). In case he is able to brute-force a single random number for one chosen entry, he would need to brute-force all subsequent entries in order to manipulate the chosen one. On the other hand it's possible to find a proper replacement that yields the same signature due to collision attacks. In either case, it is important to choose a cryptographically strong hash function to protect the hash chains.

5.2.7. Deletion of Traces

An attack vector represents the deletion of the verifiable data log. While being possible, it is highly conspicuous and would trigger an alarm in the verification procedure. Additionally, the verifiable data log could be protected by saving it to write-once data storage.

5.2.8. Attacks to Obtain the Random Generator

The security relies on the random generator. If an attacker can obtain a complete state of the random generator, he can calculate the next random numbers and therefore adulterate the log file by calculating the signed hash values. Nevertheless, old log entries can't be modified and stay secure, because a CSPRNG withstands state compromise extensions. There are many real-world physical attacks on memory that enable an attacker to gain access to the random number generator by memory leakage or side channels [52, 53]. To avoid this problem, a random generator implemented in hardware can be used. Chong et al. [54] implemented secure audit logs proposed by Schneier et al. [29] in tamper-resistant hardware. An attacker can't access the implementation of the cryptographically secure pseudo-random generator without physical access to the machine and reverse engineering the random number chip,

File System Administrator	Database Administrator	Assets	Threats
✓ (Services)	✓	DBMS Instance	Restart to change config (5.2.1); Deployment of new code (5.2.2)
✓ (file system; requires restart)	✓ (runtime)	Configuration	Modification of current config (5.2.1); Deactivation of security mechanisms (5.2.1, 5.2.3); Denial of Service (5.2.3)
✗	✓	SQL Interface	Execution of Queries for altering database objects (5.2.4)
✓	✗ (read only)	Data Files	Unmonitored modification (5.2.5); Deletion of data (5.2.5, 5.2.7)
✓	✗ (read only)	Internal data structures	Unmonitored modification (5.2.5, 5.2.6); Deletion of data (5.2.7)
✗	✗	Random number generator	Breaking forward secrecy (5.2.8)
✓	✗	Source code	Unpatching security functionality (5.2.2)

Table 2: Assets, attacker models and threats

which will be noticed due to the caused downtimes of the DBMS.

5.3. Performance Evaluation

Since databases are often used in performance critical environments and a lot of database research is targeted at making them faster and more efficient, performance is a critical factor in the evaluation.

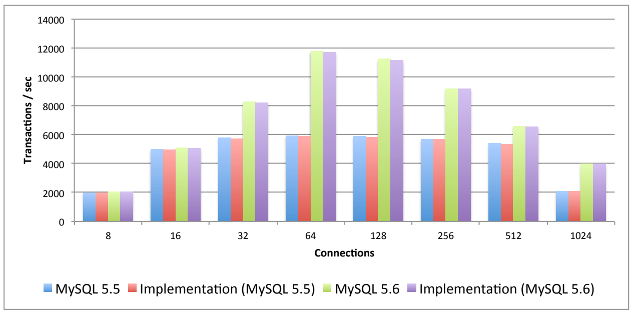

Figure 8: Performance evaluation (*sb_OLTP_RW*)

Runtime overhead. We evaluated the performance of our prototype on a typical DBMS in a large infrastructure (2x Intel Xeon E5-2470 @ 2.30GHz (32 Threads), 128 GB RAM, Oracle Linux 6.2, XFS mounted with "noatime, nodiratime, nobarrier, logbufs=8") obtained on Sysbench[1] workloads. We have simulated heavy RW-workload using *OLTP_RW* in order to simulate a realistic testing environment. Figure 8 shows the result of the performance evaluation of our prototype implementation. It demonstrates that no relevant performance loss caused by our approach is measurable. We have calculated an average performance loss of about 0,3% throughput. To ensure reproducibility a detailed description of our showcase implementation is presented in Section 4. We will publish the working prototype under an open source license[2].

Disk space overhead. The disk space overhead introduced by our approach mostly relies on the selected cryptographic hash function (see Section 3), currently typically 256 bit with an additional three bytes for the timestamp, thus resulting at 280 bit per entry. Compared to the data sizes stored by the transaction management or data replication mechanisms in database management systems, e.g. in the telecommunication industries, this is also neglectable.

5.4. Limitations and Countermeasures

In the current form our approach does face some limitations which we will outline below. Furthermore, we will indicate what kind of countermeasures can be applied in order to mitigate the resulting negative effects.

5.4.1. Availability and Changes in the Code Base

Since our approach relies on the extension of the storage mechanisms for internal data, it is limited to database management systems that provide the source code. Furthermore, the logging methods have to be rewritten with every deployed release of the DBMS. Fortunately our approach solely depends upon the rewriting of a small number of methods and doesn't require a large amount of work for modification. Additionally, in Section 3.5 we provide an alternative approach using the data replication protocol that does not depend on any modification of the source code, thus can be used in closed source systems.

[1]`http://dev.mysql.com/downloads/benchmarks.html`

[2]Note: After publication of this article

5.4.2. Malicious Changes in the Code Base

Since our approach is implemented by applying changes to the source code of the DBMS (which thus has to be available), an attacker could try to patch the method in order to log all generated random numbers. In order to mitigate this risk, secure software development life cycles including signed code bases have to be in place in order to guarantee the execution of an untampered version of the database. Furthermore, a recompilation would lead to a restart of the database management system, which requires a new seed of the trusted party. This again should result in a verification of the signed code caused by an organizational policy.

5.4.3. Root Access

In principle, an attacker with root access is able to directly read the location in the RAM, where the random number generator stores the current value, and misuse it for e.g. manipulating transactions. Furthermore, in case of a continuous monitoring of the random numbers, the attacker is able to change the log file subsequently, at the cost of having to change all subsequent log entries due to the influences of the changes on the chaining mechanism. However, it is not possible to modify log events that were logged *before* the attacker compromised the system, since our approach provides forward secrecy. Finally no cryptographic mechanism can be used to prevent the deletion of the internal log files, which still would be detected by our approach, but could result in non-restorable damage.

> "A few moments' reflection will reveal that no security measure can protect the audit log entries written *after* an attacker has gained control of $\mathcal{U}$ (untrusted machine)." [29]

5.4.4. Lifetime of Internal Data Structures

Lifetime is currently a limiting factor and determined by the configuration. One possible solution for this dilemma lies in the duplication of the method that writes the temporary internal data structures without limitations of the lifetime. This verifiable data log will then be truncated after verification procedures validated its authenticity to avoid disproportional growth. Furthermore, the sizes of the internal data structures are usually configurable and can be extended in order to provide a larger time frame than in the standard configuration.

6. Conclusion

In this paper, we proposed a novel, forensic-aware database management system based on transaction and replication mechanisms, which are mainly used for crash recovery and to assure transaction authenticity. In general, these temporary data structures are not human-readable and intended for internal system methods only. In this work, we showed that this internal information is a vital source to reconstruct evidence during a forensic investigation. Thus, we added forensic non-deniability of transactions to common database solutions by only applying minimal changes to the code base, hence our approach does not rely on additional logging. Moreover, we provided a concept for closed source database management systems. The overall goal of this work was to provide a formal description of our concept and a prototype implementation in MySQL. To demonstrate the benefits for system security, we also provided a comprehensive security evaluation with respect to the most relevant attacker models and assets.

As future work, we plan to extend the approach regarding closed source database management systems as outlined in Section 3.5, especially regarding synchronization amongst others. Furthermore, we plan to build a forensic aware database management system based on the results presented in this paper using an open source database management system as technological basis.

Acknowledgments

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

References

[1] E. Casey, Digital evidence and computer crime: forensic science, computers and the internet, Academic press, 2011.

[2] M. S. Olivier, On metadata context in database forensics, Digital Investigation 5 (3) (2009) 115–123.

[3] H. Beyers, M. Olivier, G. Hancke, Assembling metadata for database forensics, in: Advances in Digital Forensics VII, Springer, 2011, pp. 89–99.

[4] O. M. Fasan, M. Olivier, Reconstruction in database forensics, in: Advances in Digital Forensics VIII, Springer, 2012, pp. 273–287.

[5] H. Pieterse, M. Olivier, Data hiding techniques for database environments, in: Advances in Digital Forensics VIII, Springer, 2012, pp. 289–301.

[6] H. Lalla, S. Flowerday, T. Sanyamahwe, P. Tarwireyi, A log file digital forensic model, in: Advances in Digital Forensics VIII, Springer, 2012, pp. 247–259.

[7] O. M. Fasan, M. S. Olivier, Correctness proof for database reconstruction algorithm, Digital Investigation 9 (2) (2012) 138–150.

[8] H. K. Khanuja, D. D. Adane, Database security threats and challenges in database forensic: A survey, in: Proceedings of 2011 International Conference on Advancements in Information Technology (AIT 2011), available at http://www. ipcsit. com/vol20/33-ICAIT2011-A4072. pdf, 2011.

[9] O. M. Adedayo, M. S. Olivier, On the completeness of reconstructed data for database forensics, in: Digital Forensics and Cyber Crime, Springer, 2013, pp. 220–238.

[10] P. Stahlberg, Forensic analysis of database systems–final report, in: Fall seminar report, Department of Computer Science, UMass Amherst, 2005.

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

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

[13] D. Litchfield, Oracle forensics part 1: Dissecting the redo logs, NGSSoftware Insight Security Research (NISR), Next Generation Security Software Ltd., Sutton.

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

[15] D. Litchfield, Oracle forensics: Part 3 isolating evidence of attacks against the authentication mechanism, NGSSoftware Insight Security Research (NISR).

[16] D. Litchfield, Oracle forensics part 4: Live response, NGSSoftware Insight Security Research (NISR), Next Generation Security Software Ltd., Sutton.

[17] D. Litchfield, Oracle forensics part 5: Finding evidence of data theft in the absence of auditing, NGSSoftware Insight Security Research (NISR), Next Generation Security Software Ltd., Sutton.

[18] D. Litchfield, Oracle forensics part 6: Examining undo segments, flashback and the oracle recycle bin, NGSSoftware Insight Security Research (NISR) Publication, Next Generation Security Software.

[19] D. Litchfield, Oracle forensics part 7: Using the oracle system change number in forensic investigations, NGSSoftware Insight Security Research (NISR) Publication, Next Generation Security Software.

[20] D. Litchfield, A forensic investigation of pl/sql injection attacks in oracle, NGSSoftware Insight Security Research (NISR) Publication, Next Generation Security Software.

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

[22] P. Frühwirt, P. Kieseberg, S. Schrittwieser, M. Huber, E. Weippl, Innodb database forensics: Reconstructing data manipulation queries from redo logs, in: 5th International Workshop on Digital Forensic, 2012.

[23] P. Frühwirt, P. Kieseberg, S. Schrittwieser, M. Huber, E. Weippl, Innodb database forensics: Enhanced reconstruction of data manipulation queries from redo logs, Information Security Technical Report.

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

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

[26] G. Ozsoyoglu, R. T. Snodgrass, Temporal and real-time databases: A survey, Knowledge and Data Engineering, IEEE Transactions on 7 (4) (1995) 513–532.

[27] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, Transaction time support inside a database engine, in: Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on, IEEE, 2006, pp. 35–35.

[28] E. McKenzie, R. Snodgrass, Extending the relational algebra to support transaction time, in: ACM SIGMOD Record, Vol. 16, ACM, 1987, pp. 467–478.

[29] B. Schneier, J. Kelsey, Secure audit logs to support computer forensics, ACM Transactions on Information and System Security (TISSEC) 2 (2) (1999) 159–176.

[30] D. Ma, G. Tsudik, A new approach to secure logging, ACM Transactions on Storage (TOS) 5 (1) (2009) 2.

[31] A. A. Yavuz, P. Ning, Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems, in: Computer Security Applications Conference, 2009. ACSAC'09. Annual, IEEE, 2009, pp. 219–228.

[32] G. A. Marson, B. Poettering, Practical secure logging: Seekable sequential key generators, in: J. Crampton, S. Jajodia, K. Mayes (Eds.), Computer Security - ESORICS 2013, Vol. 8134 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 111–128. `doi:10.1007/978-3-642-40203-6_7`.
URL `http://dx.doi.org/10.1007/978-3-642-40203-6_7`

[33] F. M. Alvarez, A. Sharma, Oracle Database 12c Backup and Recovery Survival Guide, Packt Publishing Ltd, 2013.

[34] P. S. Randal, Understanding logging and recovery in sql server, TechNet Magazine.
URL `http://technet.microsoft.com/en-us/magazine/2009.02.logging.aspx`

[35] P. DuBois, MySQL, Pearson Education, 2008.

[36] A. Neagu, R. Pelletier, IBM DB2 9.7 Advanced Administration Cookbook, Packt Publishing Ltd, 2012.

[37] T. Coffing, M. Larkins, Teradata SQL, Coffing Publishing, 2013.

[38] R. Canetti, S. Halevi, J. Katz, A forward-secure public-key encryption scheme, Journal of Cryptology 20 (3) (2007) 265–294.

[39] M. Bellare, S. K. Miner, A forward-secure digital signature scheme, in: Advances in CryptologyÑCRYPTOÕ99, Springer, 1999, pp. 431–448.

[40] B. Carrier, E. H. Spafford, An event-based digital forensic investigation framework, in: Digital forensic research workshop, 2004, pp. 11–13.

[41] M. Kohn, M. S. Olivier, J. H. Eloff, Framework for a digital forensic investigation., in: ISSA, 2006, pp. 1–7.

[42] A. C. Yao, Theory and application of trapdoor functions, in: Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on, IEEE, 1982, pp. 80–91.

[43] C. Bell, M. Kindahl, L. Thalmann, MySQL High Availability: Tools for Building Robust Data Centers, O'Reilly, 2010.

[44] A. S. Pachev, Understanding MySQL Internals, Oreilly & Associates Incorporated, 2007.

[45] M. Reid, InnoDB Quick Reference Guide, Packt Publishing Ltd, 2013.

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

[47] K. Fowler, SQL Server Forenisc Analysis, Pearson Education, 2008.

[48] K. Delaney, Inside Microsoft SQL Server 2005: The Storage Engine, O'Reilly, 2009.

[49] S. Chaudhuri, U. Dayal, An overview of data warehousing and olap technology, ACM Sigmod record 26 (1) (1997) 65–74.

[50] B. Griesemer, Oracle Warehouse Builder 11g R2: Getting Started 2011, Packt Publishing Ltd, 2011.

[51] J. Adzic, V. Fiore, S. Spelta, Data warehouse population platform, in: Databases in Telecommunications II, Springer, 2001, pp. 9–18.

[52] A. Akavia, S. Goldwasser, V. Vaikuntanathan, Simultaneous hardcore bits and cryptography against memory attacks, in: Theory of Cryptography, Springer, 2009, pp. 474–495.

[53] S. N. Chari, V. V. Diluoffo, P. A. Karger, E. R. Palmer, T. Rabin, J. R. Rao, P. Rohotgi, H. Scherzer, M. Steiner, D. C. Toll, Designing a side channel resistant random number generator, in: Smart Card Research and Advanced Application, Springer, 2010, pp. 49–64.

[54] C. N. Chong, Z. Peng, P. H. Hartel, Secure audit logging with tamper-resistant hardware, SEC 250 (2003) 73–84.