DETERMINING TRIGGER INVOLVEMENT DURING FORENSIC ATTRIBUTION IN DATABASES

Werner Hauger, Martin Olivier

Chapter 10

DETERMINING TRIGGER INVOLVEMENT DURING FORENSIC ATTRIBUTION IN DATABASES

Werner Hauger and Martin Olivier

Abstract Researchers have shown that database triggers can interfere with the attribution process in forensic investigations. Triggers can perform actions of commission and omission under the auspices of users without them being aware of the actions. This could lead to the actions being wrongly attributed to the users during forensic investigations. This chapter describes a technique for dealing with triggers during forensic investigations of databases. An algorithm is proposed that provides a simple test to determine if triggers played any part in the generation or manipulation of data in a specific database object. If the test result is positive, a forensic investigator must consider the actions performed by the implicated triggers. The algorithm is formulated generically to enable it to be applied to any relational SQL database that implements triggers. The algorithm provides forensic investigators with a quick and automated means for identifying the potentially relevant triggers for database objects, helping to increase the reliability of the forensic attribution process.

Keywords: Database forensics, database triggers, forensic attribution

1. Introduction

Database triggers are a feature of many relational databases. They were formally incorporated in the ISO/IEC 9075 SQL standard in 1999 and have since been updated [9]. However, the digital forensic community has not paid much attention to database triggers. While developing a new framework for database forensics, Khanuja and Adane [13] recognized that the actions performed by triggers are forensically important. Hauger and Olivier [8] have conducted research on the exact role that triggers play in database forensics. They studied trigger implementa-

tions in a number of proprietary and open-source relational databases. Among other things, Hauger and Olivier discovered that the forensic attribution process is impacted by trigger actions.

Triggers can introduce side effects during the normal flow of operations. The side effects include performing additional actions or preventing the completion of the triggering operations. Certain types of triggers can also manipulate or completely replace the original operations. This means that what an original operation intended to do and what actually happened may not necessarily be the same.

As noted above, triggers can also lead to incorrect conclusions when the attribution of database operations is performed. This is because a trigger performs its actions with the same user credentials as the original operation that caused the trigger to fire. Some databases might log additional information with an operation to indicate that it was performed by a trigger. However, it cannot be assumed that such an extended log always will be available to a forensic investigator.

It is, therefore, important that a forensic professional be aware of triggers during an investigation. The first step is to determine if any triggers are present in the database under investigation. If triggers are indeed present, the next step is to establish if any of the triggers may have influenced the data being analyzed.

A database under investigation can potentially contain hundreds or even thousands of triggers. It is not feasible for a forensic investigator to analyze every single trigger to determine if it had an impact on the data being analyzed. A more efficient technique is required to identify the triggers that must be analyzed as part of an investigation.

This chapter proposes a test to quickly and easily establish if a trigger plays a role in the data of a database table being analyzed. A generic algorithm is presented that can be applied to any SQL database that supports triggers. The algorithm generates a list of triggers that potentially influence the data in a given database table, enabling a forensic investigator to focus only on the identified triggers.

2. Background

This section briefly discusses forensic attribution and database triggers.

2.1 Forensic Attribution

Forensic attribution involves the use of technical means to establish the presence of a person or object at the scene of a crime after the fact [1]. In digital forensics, various techniques can be used to determine which

actor, be it a person or a program, has performed specific actions in a digital system. However, determining the actual person responsible for actions in a digital system is problematic. This is because the physical person is located outside the boundaries of the digital system. Therefore, information from outside the digital system is required to positively link a specific person to the human-digital-system interface.

Attribution is performed during the digital evidence examination and analysis step. This step is part of the investigative processes as defined by the ISO/IEC 27043 draft standard [11]. The standard seeks to provide an overall framework for incident investigations and processing. It encompasses various other standards, each of which define the sub-processes in more detail.

Many digital systems create and maintain audit records and metadata [5]. These include dates, timestamps and file ownership, authentication and program execution logs, output files generated by program execution, network traffic logs and numerous other traces. The artifacts are created as part of normal system operation. They allow for the confirmation and review of activities as well as debugging errors.

A forensic investigator can employ authentication and authorization information to assist with the attribution process [5]. Authentication is implemented in many digital systems to identify users and grant appropriate system access. The access available to a user depends on the authorization afforded by the user's credentials. Unauthorized users are prevented from accessing and operating the system. Authentication information describes the actors in a digital system and authorization information describes the actions that the actors can perform.

Another attribution technique is to order and connect the various traces found in a digital system to create a chain of events [5]. The sequences of these events describe how the system reached a given state. By determining the actions that lead to the events and the actors that performed the actions, the person or program responsible for the actions can potentially be identified.

However, authentication information and the various traces are not infallible. Attackers can bypass authentication or steal the credentials of an innocent user. Traces can be modified or even removed to hide malicious activity. Furthermore, attribution techniques themselves can be abused by malicious actors. Stolen user credentials can be used to create traces that falsely implicate innocent users.

It is, therefore, important for a forensic investigator to attempt to obtain multiple independent traces that portray the same event. The independent traces can then be used to correlate the actions and identify possible manipulations. By excluding the manipulated information or

reconstructing the original information, a forensic investigator can move closer to identifying the real actor or actors.

Olivier [15] has investigated attribution in forensic investigations involving databases. He noted that database forensics can use the same techniques as general digital forensics to perform attribution. The traces in a database are available in various log files and are also stored in system tables. The authentication of database users and the authorization of their operations is built into many databases. The same caveats that apply to general digital forensics also apply to database forensics.

2.2 Triggers

Database triggers originated in the mid 1970s in "active databases." It was not until the 1990s that this technology gained traction and was incorporated in commercial databases. The researchers at the time referred to triggers as event-condition-action rules [16]. The action portion of a rule was executed when the condition was true after being initiated by a triggering event. The condition was an optional part and a rule without a condition was called an event-action rule. The term database trigger was adopted when the technology was incorporated in the SQL standard in 1999.

Triggers are implemented for various reasons. Ceri et al. [2] distinguish between automatically-generated and manually-crafted triggers and suggest different uses for the two types of triggers. One use for manually-crafted triggers is the creation of internal audit trails. Developers now leverage this use case for auditing in their own external applications. When used for external auditing, the triggers normally take data from the original tables and store them in dedicated audit tables. Since the tables are part of the custom application, there is no particular layout or content. In general, the audit tables indicate which users performed operations, what operations were performed and when the operations were performed.

Independent of their usage, all the actions performed by triggers can be classified into two groups: (i) general actions; and (ii) condition-specific actions. A general action trigger performs its actions every time the triggering operation is performed. This makes the trigger suitable for general auditing. For example, a financial institution could use general action triggers to keep track of all the transactions performed on its system. A trigger would record relevant information such as the nature of a transaction, the time it was performed and the user who performed the transaction.

In contrast, condition-specific action triggers only perform their actions under specific circumstances. The circumstances are based either on the data that is manipulated by the triggering operation or on the user who performs the triggering operation. For example, a financial institution may have a legal requirement to report a certain type of monetary transaction that exceeds a specified amount. This requirement can be implemented using a condition-specific trigger that checks all transactions as they are executed. The trigger would only fire when a transaction type matches the type that has to be reported and the associated transaction amount exceeds the legislated amount.

Consider a situation where a financial institution might for security reasons only allow supervisors to execute certain transactions. However, due to the volumes of these transactions and the ratio of supervisors to normal operators, it may not possible to have only supervisors execute the transactions. To address this problem, the financial institution can implement an override that is invoked by a normal operator to enable him to execute a restricted transaction after a supervisor has authorized the transaction with his credentials.

Naturally, the financial institution would want to keep a very close eye on override transactions to identify any irregularities as soon as possible. A condition-specific action trigger can be used in this scenario to audit the restricted transactions. The trigger is created to fire only when a user who is not a supervisor executes a restricted transaction.

At first glance, it might appear that the two types of triggers would not interfere with a forensic investigation. However, one cannot assume that the triggers have not been modified by an attacker to perform malicious actions. For example, an attacker might wish to collect all the credit card numbers entered into a database. The attacker could set up his own general action trigger on the table that stores the credit card information or he could modify an existing audit trigger on the table. The trigger is created or the existing trigger is modified to place the credit card information in a separate location internal or external to the database. Another process is then scheduled to transmit the collected information to a location outside the financial institution from where the attacker can retrieve it at his convenience.

A malicious trigger can also be condition-specific. Consider a business that supports various forms of payment for its products and services. The attacker could add or modify a trigger so that it fires only when the payment method for a transaction is a credit card. This trigger then extracts the credit card information in the same manner as before. The attacker also may wish to perform malicious actions only when a specific user performs operations on the database. This could be because this

particular user's actions are less likely to be scrutinized. Alternatively, the attacker may have a desire to implicate the user in the malicious actions.

3. Trigger Identification

This section describes a systematic approach for identifying triggers that are potentially significant to an investigation of a database object. The approach is then formally specified as an algorithm that is database independent.

The simplest way to determine if a trigger has an effect on data contained in an object is to check if the name of the object is mentioned in the trigger. Having identified all such triggers, the next step is to manually check the kinds of SQL statements that refer to the object in question. Since SQL statements can be encapsulated in a user function or procedure, it is not enough to merely search the trigger content. The content of all called functions and procedures also have to be checked. Since additional functions and procedures can be invoked by a function or procedure, the contents of these additional functions and procedures also have to be checked. This process is repeated until the lowest level function or procedure has been reached that invokes no other function or procedure.

This top-down checking is formalized in Algorithm 1. To simplify the definition of the algorithm, procedures are assumed to be equivalent to functions. From a database perspective, there are differences between a function and a procedure. Some of the differences are general while some are database-specific.

Algorithm 1 (Top-Down): Identify all the triggers that directly or indirectly access an object of interest.

1. Assume that a trigger is an ordered pair (t, b_t) where t is the name of the trigger and b_t is the code (or body) of t. Assume that functions exist of the form (f, b_f) where f is the function name and b_f is the body of function f. The body b_x of a trigger or function x consists of primitive statements as well as calls to other functions. Let $f \in b_x$ indicate that f is used in the body of trigger or function x.

2. Let T be the set of all triggers and F be the set of all functions. Let ω be the name of the object that is the target of the search. Let $\omega \in b_x$ indicate that ω is used in the body of trigger or function x. Let R_1 the set of triggers that access ω directly. Then,

$$R_1 = \{t \in T \mid \omega \in b_t\}$$

3. Functions may be called by a trigger, where the function accesses ω, thereby providing the trigger with indirect access to ω. Therefore, the $\in$ notation is

extended to also indicate indirect access. The new relationship is denoted as $\in^*$. Then,

$$f \in^* b_x \Leftrightarrow \left\{ \begin{array}{ll} f \in b_x & \text{direct case} \\ \exists\, b_y \ni f \in b_y \;\&\; y \in^* b_x & \text{indirect case} \end{array} \right\}$$

Therefore, the complete set of functions used directly or indirectly by a trigger t is given by:

$$F^t = \{f \mid f \in^* b_t \ni t \in T\}$$

The subset of the functions that access ω is given by:

$$R_2 = \{t \in T \mid \exists\, f \in F^t \ni \omega \in b_f\}$$

The triggers that directly and indirectly access ω are then combined as:

$$R = R_1 \cup R_2$$

The algorithm produces a set of triggers R that refer directly or indirectly to the object (search target). At first glance, the algorithm seems simple and straightforward. However, a major problem arises when the steps are translated into SQL statements – How does one reliably identify other procedures in the content via string matching? The exact SQL syntax for invoking procedures is database-specific. It is possible to obtain the list of all the user function and procedure names and then search for each name in the content. However, this is not necessarily more reliable. Depending on the naming convention used by the database developers, the names of different object types could overlap (e.g., a table, a view and a procedure could all have the same name).

It is, therefore, necessary to design a new technique for finding the triggers that directly or indirectly refer to the object of interest. Alternatively, it is possible to start with all the triggers, functions and procedures and then narrow them down to only those that refer to the object of interest. Algorithm 2 formalizes this bottom-up technique.

Algorithm 2 (Bottom-Up): Identify all the triggers that directly or indirectly access an object of interest.

1. Assume that a trigger is an ordered pair (t, b_t) where t is the name of the trigger and b_t is the code (or body) of t. Assume that functions exist of the form (f, b_f) where f is the function name and b_f is the body of f. The body b_x of a trigger or function x consists of primitive statements as well as calls to other functions. Let $f \in b_x$ indicate that f is used in the body of trigger or function x.
2. Let T be the set of all triggers and F be the set of all functions. Let ω be the name of the object that is the target of the search. Let $\omega \in b_x$ indicate that ω is used in the body of trigger or function x. Let C be the combined set of all triggers T and functions F. Then,

$$C = T \cup F$$

3. Let U be the set of triggers and functions that access ω directly. Then,

$$U = \{c \in C \mid \omega \in b_c\}$$

Let U^t be the subset of triggers in set U. Then,

$$U^t = \{t \in U \mid t \in T\}$$

The subset U' of U without any triggers is given by:

$$U' = U - U^t$$

Let U_1 be the first iteration of a set of triggers and functions that access the functions in set U' directly. Then,

$$U_1 = \{d \in C \mid \exists\, e \in b_d \ni e \in U'\}$$

Let U_1^t be the first subset of all the triggers in the first set U_1. Then,

$$U_1^t = \{t \in U_1 \mid t \in T\}$$

The first subset of U_1 without any triggers is given by:

$$U_1' = U_1 - U_1^t$$

The sets U_2, U_2^t and U_2' can be constructed congruently. Specifically, the construction of the sets can be repeated i times ($i = 1, 2, \ldots$). Consequently, the i^{th} iteration of set U_1 is given by:

$$U_i = \{x \in C \mid \exists\, y \in b_x \ni y \in U_{i-1}'\}$$

Furthermore, the i^{th} subset of all the triggers in set U_i is given by:

$$U_i^t = \{t \in U_i \mid t \in T\}$$

Finally, the i^{th} subset of set U_i without any triggers is given by:

$$U_i' = U_i - U_i^t$$

The combined set of all identified triggers $U^t, U_1^t ... U_i^t$ is given by:

$$R = U^t \cup U_1^t \cup ... \cup U_i^t$$

Since C is a finite set of triggers and functions, the algorithm reaches a point where set R no longer grows. Specifically, this is when the n^{th} set U_n becomes empty, in other words $U_n = \phi$. At this point the algorithm terminates.

Algorithm 2 also produces a set of triggers R that refer directly or indirectly to the object of interest. However, it eliminates the need to search for calls to functions, which is difficult and database-specific. Instead, the algorithm repeatedly searches for known function and trigger names in the same way that it searches for the name of the object of interest.

4. Algorithm Implementation

This section discusses the implementation of the proposed algorithm. Implementations are considered for the same databases discussed in [8].

The databases considered are all relational databases, so it can be assumed that data and metadata about triggers, functions and procedures are stored in relations. The most straightforward choice is to implement the algorithm using SQL. Each step of the algorithm is written as a simple SQL statement. SQL statements that are repeated are placed into functions. All the separate SQL steps and functions are then combined into a procedure.

An important point to keep in mind is that the database implementations available from vendors vary considerably. This includes the data dictionary that specifies what information is stored and how it is stored. Thus, in one implementation, retrieving all the triggers may involve just one select statement while in another it may require performing select statements on multiple differently-structured tables. Due to these differences, it is not possible to create template select statements that could be filled in with column and table names corresponding to a particular database. Instead, it is necessary to provide database-specific SQL statements for each database implementation.

A pure SQL implementation has two major drawbacks. The first drawback is that database vendors use different SQL dialects to extend the standard SQL statements. These extended features, such as variables, loops, functions and procedures, are required to implement the algorithm. Oracle uses an extension called PL/SQL that IBM DB2 version 9.7 and later support [3, 6]. Microsoft has a competing extension called Transact-SQL or T-SQL used by SQL Server and Sybase [12, 17]. Other databases such as MySQL more closely adhere to the official SQL/PSM extensions that are part of the SQL standard [10]. These differences would require a separate implementation for every database product. The second drawback is that the implementation has to be stored and executed within the database being investigated. However, this conflicts with the forensic goal of keeping the database uncontaminated.

Another approach is to implement the algorithm using a pure programming language. This provides the advantage of having to create only one application using a single syntax. The application is designed to read the database data from a file that conforms to a specific format. The drawback with this design is that the data first has to be extracted from the database in the correct format. This requires a separate extraction procedure for every database implementation. The extraction

procedure also may have to be stored and executed on the database, which potentially contaminates the database.

The latter implementation choice involves a more involved two-step process. First, the required data is extracted from the database and transformed into the format expected by the application. Next, the standalone application is invoked with the created data file.

Thus, a hybrid approach that combines the advantages of the two choices is a better solution. A single application can be built using a conventional programming language that provides a database-independent framework for accessing and using databases. Next, database-specific formatted SQL statements in the application are used to select and retrieve data from the database under investigation. The database would remain unchanged because the algorithm logic as well as the SQL statements are external to the database.

The authors of this chapter are currently building a prototype that uses the hybrid approach to implement the bottom-up algorithm (Algorithm 2). The prototype comprises a standalone application written in Java that connects to the database being investigated via the JDBC programming interface. A JDBC database driver is required to handle database communications with each database.

The required data is selected and retrieved by executing database-specific SQL statements. These SQL statements, together with the database and JDBC driver configuration, are externalized to a definition file. This allows the pieces to be updated or changed without the need to modify the application. The architecture of the prototype enables it to be extended to support any SQL database with triggers by simply defining the new database, adding the required database and JDBC driver configuration and providing the SQL statements for each part of the algorithm.

5. Implementation Challenges

This section discusses the challenges encountered when implementing the proposed algorithm. Some of the challenges are related to the general SQL syntax while others are related to database-specific implementation decisions.

5.1 Scope and Visibility

Most of the tested databases do not allow direct access to the data dictionary tables (these tables contain, among other things, lists of all the triggers, functions and procedures in a database). Instead, the databases provide access to the data in the tables via system views. The problem

with the views is that they limit the results based on the permissions possessed by the user who accesses the view and on the scope of the database connection. Therefore, the user account employed to query the views must have access to all the database objects. However, even if the user account has the permissions to view all the objects, only the objects that fall within the scope of the connection would be listed. To work around this restriction, it is necessary to obtain a list of all the database scopes and iterate through them, changing the scope of the connection and repeating the query. The results obtained with each individual query are then combined.

5.2 Encryption

Some of the databases considered have the functionality to encrypt the contents of objects, including the triggers. This makes it impossible to search the content for the name of the object of interest. Due to flaws in current encryption implementations, it is possible to retrieve the decrypted content of an encrypted object [4, 14]. However, such workarounds are neither efficient nor practical. In any case, the flaws may be corrected in a future version, rendering the workaround worthless for the updated implementations.

5.3 Case Sensitivity

Since SQL is mostly case insensitive, the name of the object of interest may be spelled in many different ways. Therefore, searching the content for an exact match based on a particular spelling would be a hit and miss affair. The standard way to deal with this problem is to use lower case letters for the content and the object name being searched before executing the query [7]. This is usually done using a built-in function. The approach is feasible for a database with few objects that are to be searched. However, it would be very inefficient and impractical for a database with thousands of objects, each with thousands of lines of content. A better solution is to change the collation of the database to enable case-insensitive comparisons.

5.4 False Positive Errors

Since basic string matching is performed, any string in a trigger, function or procedure would match the object name. This includes comments, variable names and the names of other object types. Therefore, it would be necessary to manually check all the triggers listed by the algorithm and remove all the false positives. Since the SQL parser can identify comments based on the syntax, it is possible to pre-process the

triggers, functions and procedures to remove the comments before initiating the search. However, other sources of matching strings could still produce false positives.

5.5 Data Types

Some of the considered databases do not store the contents of triggers, functions and procedures in table columns with the same data type. Many databases have moved from using the simple VARCHAR data type to one that can hold more data. Microsoft SQL Server, for example, uses the CLOB data type to store content while Oracle uses the older LONG data type. The problem is that all the data types that are used cannot be handled in the same way in the code. A more generic approach is needed to prevent the writing of various code exceptions. The approach would entail querying the database table metadata first to detect the data types that are used and invoke the relevant code to handle the specific data types.

5.6 Recursion

Database functions and procedures are allowed to call themselves either directly or indirectly. The recursion produces an endless loop in Algorithms 1 and 2. In Algorithm 2, R would stop growing at some point because no new elements are added. However, U_n would not become empty, which means that the termination criteria are never met. To address this problem, Algorithm 2 must keep track of all the elements seen in U_i' previously. U_n' should not contain any elements evaluated before and the following equality should hold:

$$U_n' \cap U' \cup U_1' \cup ... \cup U_{n-1}' = \phi$$

To achieve this, the second last line in Algorithm 2 is modified to:

$$U_i' = U_i - U_i^t - \bigcup_{j=1}^{i-1} U_j$$

This ensures that each function and procedure is only evaluated once.

5.7 Performance

The string matching component of the step $\omega \in b_x$ can be performed using the SQL LIKE command. The object name being searched can be located anywhere in the body of the trigger or function being searched. Thus, the SQL wildcard character "%" must be added around the object name as in "%*object_name*%." This makes the execution of the

SQL statement very slow because no index can be utilized. This does not pose much of a problem for a database with a small number of triggers, functions and procedures. However, the SQL statement can take a long time to complete if there are many such objects. This may occur in systems where auditing is performed with triggers or where the application utilizing the database performs a lot of transaction processing in functions and procedures. Depending on the time available to the forensic investigator, this could take a very long time to execute. It is also contrary to the goal of developing a quick test to establish trigger involvement.

6. Conclusions

A database trigger performs its actions under the credentials of the user who executed the database operation that initiated the trigger. Since the actions occur in the background without the user being aware of them, they can be abused by malicious actors. Malicious actions can be attached to legitimate operations and valid operations modified or even omitted. A forensic investigator who is tasked with analyzing these malicious actions could mistakenly attribute them to the wrong party.

Hence, it is important that a forensic investigator consider triggers very carefully when attributing actions. After suspicious data modifications have been identified, the forensic investigator must establish if a trigger interfered with the operations that lead to the data manipulation. Since a database under investigation might contain hundreds or even thousands of triggers, it is not feasible for the forensic investigator to analyze each trigger individually. An automated approach is needed to identify possibly significant triggers, which could then be analyzed in more detail.

The principal contribution of this chapter is an algorithm for identifying possible trigger involvement for a particular database object. The algorithm, which is simple and effective, is formulated in a generic manner so that it can be applied to any SQL database.

The prototype currently implements SQL statements for Microsoft SQL Server and Oracle. While the initial results are promising, more thorough tests are being conducted. Options such as moving the string comparison out of the database and into the prototype to enhance performance are being evaluated. The use of a local embedded database to temporarily store the content of all the triggers, functions and procedures is also being considered.

Future research will focus on optimizing the algorithm. Currently, only one object can be checked for trigger involvement at a time. Repeating the same check for every object is inefficient. Therefore, efforts will be made to search for multiple objects simultaneously in order to improve efficiency.

References

[1] M. Afanasyev, T. Kohno, J. Ma, N. Murphy, S. Savage, A. Snoeren and G. Voelker, Privacy-preserving network forensics, *Communications of the ACM*, vol. 54(5), pp. 78–87, 2011.

[2] S. Ceri, R. Cochrane and J. Widom, Practical applications of triggers and constraints: Success and lingering issues, *Proceedings of the Twenty-Sixth International Conference on Very Large Data Bases*, pp. 254–262, 2000.

[3] Y. Chan, N. Ivanov and O. Mueller, *Oracle to DB2 Conversion Guide: Compatibility Made Easy*, IBM Redbooks, Armonk, New York, 2013.

[4] D. Cherry, *Securing SQL Server*, Elsevier, Waltham, Massachusetts, 2012.

[5] F. Cohen, *Digital Forensic Evidence Examination*, Fred Cohen and Associates, Livermore, California, 2009.

[6] S. Feuerstein and B. Pribyl, *Oracle PL/SQL Programming*, O'Reilly Media, Sebastopol, California, 2014.

[7] P. Gulutzan and T. Pelzer, *SQL Performance Tuning*, Addison-Wesley, Boston, Massachusetts, 2003.

[8] W. Hauger and M. Olivier, The role of triggers in database forensics, *Proceedings of the Information Security for South Africa Conference*, 2014.

[9] International Standards Organization and International Electrotechnical Commission, ISO/IEC 9075-2:2011, Information Technology – Database Languages – SQL – Part 2: Foundation (SQL/Foundation), Geneva, Switzerland, 2011.

[10] International Standards Organization and International Electrotechnical Commission, ISO/IEC 9075-4:2011, Information Technology – Database Languages – SQL – Part 4: Persistent Stored Modules (SQL/PSM), Geneva, Switzerland, 2011.

[11] International Standards Organization and International Electrotechnical Commission, ISO/IEC 27043, Information Technology – Security Techniques – Incident Investigation Principles and Processes (Final Draft), Geneva, Switzerland, 2013.

[12] A. Jones, R. Stephens, R. Plew, R. Garrett and A. Kriegel, *SQL Functions Programmer's Reference*, Wiley Publishing, Indianapolis, Indiana, 2005.

[13] H. Khanuja and D. Adane, A framework for database forensic analysis, *Computer Science and Engineering*, vol. 2(3), pp. 27–41, 2012.

[14] D. Litchfield, *The Oracle Hacker's Handbook: Hacking and Defending Oracle*, Wiley Publishing, Indianapolis, Indiana, 2007.

[15] M. Olivier, On metadata context in database forensics, *Digital Investigation*, vol. 5(3-4), pp. 115–123, 2009.

[16] E. Simon and A. Kotz-Dittrich, Promises and realities of active database systems, *Proceedings of the Twenty-First International Conference on Very Large Data Bases*, pp. 642–653, 1995.

[17] P. Turley and D. Wood, *Beginning T-SQL with Microsoft SQL Server 2005 and 2008*, Wiley Publishing, Indianapolis, Indiana, 2009.