# In-database Auditing Subsystem for Security Enhancement

B. Bašić*, P. Udovičić*, O. Orel*
*University of Zagreb, University Computing Center, Zagreb, Croatia
{bjanka.basic, petra.udovicic, ognjen.orel}@srce.hr

***Abstract*—Many information systems have been around for several decades, and most of them have their underlying databases. The data accumulated in those databases over the years could be a very valuable asset, which must be protected. The first role of database auditing is to ensure and confirm that security measures are set correctly. However, tracing user behavior and collecting a rich audit trail enables us to use that trail in a more proactive ways. As an example, audit trail could be analyzed ad hoc and used to prevent intrusion, or analyzed afterwards, to detect user behavior patterns, forecast workloads, etc.**

**In this paper, we present a simple, secure, configurable, role-separated, and effective in-database auditing subsystem, which can be used as a base for access control, intrusion detection, fraud detection and other security-related analyses and procedures. It consists of a management relations, code and data object generators and several administrative tools. This auditing subsystem, implemented in several information systems, is capable of keeping the entire audit trail (data history) of a database, as well as all the executed SQL statements, which enables different security applications, from ad hoc intrusion prevention to complex a posteriori security analyses.**

***Keywords*—audit trail analysis; database forensics; database security; in-database auditing; SQL trigger**

## I. Introduction

In the modern era the creation and consumption of data is constantly growing. Actions must be taken to promote and implement the best practices of data protection. Database auditing allows us to track our users actions and provides the base for database forensics. It is a control mechanism designed to track the use of database resources and authority. When database has some sort of auditing mechanism in place, each audited database operation creates an audit trail that must include information such as what data was impacted, who performed the operation, and when. It is important to keep in mind that auditing is not a goal but a means to achieve more secure environment [1], so audit trails must be maintained over time to allow auditors to perform in-depth analysis of access and data modification patterns. The main purpose of these analyses is to prevent further security threats by detecting violation of security settings, which are in detail discussed in [2], [3].

There are several ways of recording audit trail:

- within a database management system (DBMS): Commercial (or open-source) DBMS typically include some form of trace recording. The trace recording mechanism itself can be implemented as a process that actively observes the same memory blocks in which the DBMS maintains its operational memory structures, or as a by-product of the system operation itself.
- within a database: The implementation of such a trace recording solution it is necessary for developers to add program code that will record the trace with the desired data within each trigger that is triggered on the event to be monitored. If the trigger for the observed event does not already exist, it must be added to the database. All of this makes this trace recording method quite complex to implement and maintain.
- using external auditing software / hardware: Many DBMSs have several common architectural features, such as writing logical logs, using shared memory or implementing other change data capture mechanisms. These features allow the construction of external trace recording systems. External trace recording systems store the recorded trace in isolated locations - in other databases or files that may be on the database server itself or better yet, on another server.

Regarding the events in the database, there are several different kinds that could be audited:

- logging in and out,
- data definition language (DDL) operations,
- SQL errors generated by users,
- changes in database objects (e.g. stored procedures, triggers, permissions, synonyms),
- data change,
- data retrieval,
- audit rules definitions.

In this paper, we present a simple in-database auditing subsystem capable of keeping the entire data history (data change, including audit rules definitions) and all executed SQL statements. It has been proven reliable source of data for the detection of both simple and complex investigations. The main contribution of our work is the in-database code generator, a set of database procedures and triggers which automatically handles all the necessary objects needed for auditing to perform, based on configuration tables. This part of the code is open-sourced by us.

The paper is organized as follows. Chapter II reflects on related work regarding database auditing. The architecture of the auditing subsystem and in-database code generator are discussed in chapter III. Role separation concerns are addressed in chapter IV, while next chapter shows the measures

of performance overhead with in-database auditing in place. Chapter VI brings information of current implementations of presented auditing method and discusses the possible future work. Finally, chapter VIII concludes the paper.

## II. Related work

Commercial or open-source DBMSs (such as Oracle, MS SQL Server, IBM Db2, IBM Informix, PostgreSQL, MySQL, ...) typically include some form of auditing. For example, in PostgreSQL [4] basic statement auditing can be done using the standard logging facility with `log_statement = all`. This is acceptable for monitoring but should be avoided because it produces huge files that need special parsing to get normal audit trails and then still does not provide the level of detail required for an audit. Instead, it is recommended to use free PostgreSQL tool pgAudit [5] which provides detailed session and/or object audit logging via the standard PostgreSQL logging facility.

There are also numerous third-party solutions available for database auditing. They offer a number of options for what will be recorded and how to record audit trail.

ApexSQL Audit [6] is a SQL Server database auditing tool for capturing data and schema changes, that is added to existing database and on it implements additional triggers whose sole purpose will be auditing. It stores all auditing information in a central repository table. Triggers are based on templates that can be customized and it is easy to add and maintain large number of them. It is intended exclusively for working with the Microsoft SQL Server database management system.

Oracle Audit Vault and Database Firewall [7] combines auditing and network-based monitoring. It monitors database traffic to detect and block threats, as well as enables compliance reporting by consolidating audit data from databases, operating systems, and directories. The audit vault server contains an Oracle database with all auditing information, and makes it available to reporting tools through a data warehouse [8]. Its advantage is that it can be used with heterogeneous databases and it has support for newer versions of most popular DBMSs.

Nowadays there is no explicit need for outside solutions since all databases provide auditing features. One of the most basic features is triggers that can be adjusted to enable row level auditing of DML statements. Given that triggers are often used for auditing scenarios pertaining to DML queries, it is natural to consider extending it to SELECT queries for data auditing. It is not a simple extension because usually SELECT queries contain enormous amounts of tuples and recording all of them would quickly suffocate audit database. One of the solutions is presented in [9], where number of tuples for which an audit record will be made was reduced by auditing only what was perceived as sensitive data. We decided for a different approach, to let audit administrators define what should be audited and available for further analyses.

In addition to selecting methods used to implement auditing subsystem, it is important to make a sustainable auditing model. As seen in [10], secure database management system, aside from being able to effectively record history of all operations in the database, must also have a good support for querying that history. We will present auditing model that satisfies both requirements. Similar model to ours is given in [11] but with more emphasis put on improving Chain of Custody property, while our focuses more on overall development of effective auditing subsystem.

## III. Architecture

In this section, our audit database model and auditing objects are presented.

### A. Formal description

The auditing database $\mathcal{R}_{\mathcal{DB}}$ is a relational one, and it consists of $n$ relations $R_i$. Each relation $R_i$ has a schema:

$$R_i(a_{i,1}, \ldots, a_{i,m}), \tag{1}$$

where $a_{i,k}$ is $k^{\text{th}}$ attribute of relation $R_i$.

It is possible to perform four basic data operations over relation $R_i$: creating a new tuple, reading it, updating and deleting it (CRUD). These operations map to SQL as INSERT, SELECT, UPDATE and DELETE, respectively:

$$O \in \{\text{select, insert, update, delete}\}. \tag{2}$$

Also, apart from SELECT, the other three operations are part of data manipulation language (DML):

$$M \in \{\text{insert, update, delete}\}. \tag{3}$$

For each relation $R_i$ being audited, database $\mathcal{R}_{\mathcal{DB}}$ is expanded with a new relation $R'_i$ for auditing information pertaining to relation $R_i$. Beside all attributes contained in the original relation, audit relation $R'_i$ contains information describing a specific event affecting the tuple: which operation was performed, who performed it and when.

**Definition 1.** *Let $\mathcal{A}$ be a set of attributes:*

$$\begin{aligned} \alpha := ( & \text{ operation} \\ & , \text{ user} \\ & , \text{ transaction time stamp} \\ & , \text{ session} \\ & , \text{ operation time stamp} \\ & , \text{ operation type}). \end{aligned} \tag{4}$$

*Let $R_i \in \mathcal{R}_{\mathcal{DB}}$. Then audit relation $R'_i$ consists of set of attributes:*

$$(a_{i,1}, \ldots, a_{i,m}) \cup \alpha. \tag{5}$$

The execution of operation $M$ in relation $R_i$ initiates the generation of an audit record in the corresponding audit relation $R'_i$, for all relations that have auditing enabled.

Our proposed architecture, beside working database $\mathcal{R}_{\mathcal{DB}}$, contains also a historic auditing database ($\mathcal{H}_{\mathcal{DB}}$) for preservation of older audit records. Each database relation $R_i$ has a corresponding audit relation $H_i$ in $\mathcal{H}_{\mathcal{DB}}$, which has the same

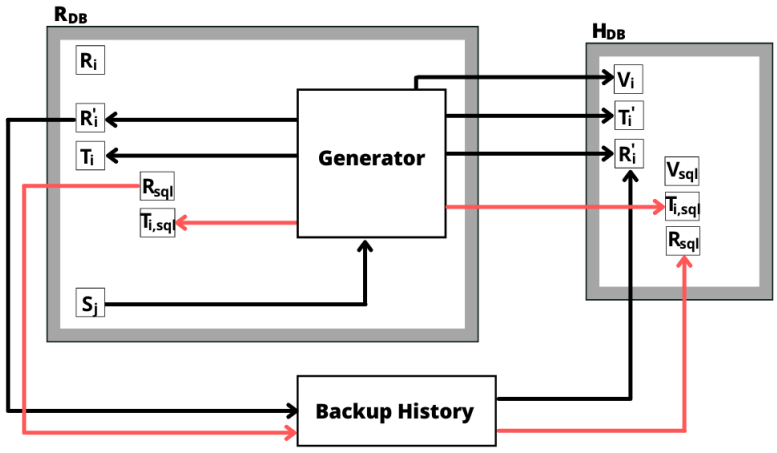


Figure 1. Database auditing subsystem (black arrows represent row-level DML auditing, red arrows represent SQL statement auditing).

relational schema as $R'_i \in \mathcal{R}_{\mathcal{DB}}$. In order to reduce negative impact on performance of the database $\mathcal{R}_{\mathcal{DB}}$, relations $R'_i$ in that database only preserve auditing information of a short period of time, before it is periodically transferred to $\mathcal{H}_{\mathcal{DB}}$ by a *backup history* process. To effectively query all data history, a view is created for each relation $R_i$ in which the whole auditing history of that relation is available.

**Definition 2.** *View $V_i$ of auditing history of $R_i \in \mathcal{R}_{\mathcal{DB}}$ is union of relations $R'_i \in \mathcal{R}_{\mathcal{DB}}$ and $H_i \in \mathcal{H}_{\mathcal{DB}}$:*

$$V_i = R'_i \cup H_i. \tag{6}$$

Most of what was previously described is pertaining to row level auditing of DML operations. Another interesting aspect of auditing is auditing of SQL queries. Since row level auditing writes one entry for each row being affected in any way, it is highly impractical for keeping track of SELECT queries in particular, which may result with great performance overhead. Therefore, in order to keep track of SQL statements (DML, as well as SELECT statements), we introduce auditing of SQL queries. Unlike before, for SQL auditing the database $\mathcal{R}_{\mathcal{DB}}$ is expanded with only one relation $R_{sql}$.

**Definition 3.** *Let $R_{sql}$ be a relation containing auditing information about SQL queries. Its relational schema has attributes:*

$$\begin{aligned} &( \text{session} \\ &, \text{table name} \\ &, \text{operation} \\ &, \text{user} \\ &, \text{host name} \\ &, \text{terminal} \\ &, \text{transaction timestamp} \\ &, \text{operation timestamp} \\ &, \text{sql}). \end{aligned} \tag{7}$$

For all relations that have SQL auditing enabled, the execution of operation $O$ in relation $R_i$ initiates the generation of an audit record in the audit relation $R_{sql}$.

Same as before, there is relation $R_{sql}$ in database $\mathcal{H}_{\mathcal{DB}}$ containing older auditing information and view $V_{sql}$ that combines $R_{sql}$ from both databases.

The configuration of the auditing and its operation is done via a special set of relations, here referred to as system relations, $S_j$. These relations contain information about each relation $R_i$ in the working database: is it enabled for row level DML auditing, is it enabled for SQL auditing and if so, which operations are being audited; exclusion lists for auditing, both user- and server-wise; and is auditing enabled in general or not. We regard $S_j$ as an audit-definition relations. Maintenance of these relations is done through the interface of a utility DbAdmin, application designed to help database administration.

The architecture described is pictured in Fig. 1. Now we will explain in more detail how the generator procedure works.

### B. In-database code generators

Previously described model contains many auditing objects and is complicated to maintain if the whole process is done repeated manually. However, it is automated by implementing a set of procedures which generates SQL code that is used to create all audit objects. The main procedure is called automatically (triggered) after auditing information has been inserted into auditing-definition relations $S_j$.

The generator is a stored procedure that orchestrates others, regarding the audit configuration in $S_j$ relations, and consequently generates all auditing objects in nine steps:

1) generate auditing relation,
2) generate auditing triggers,
3) generate triggers on auditing relation,
4) generate permissions to user for backup history,
5) generate auditing view,
6) generate backup auditing relation (in $\mathcal{H}_{\mathcal{DB}}$),

7) generate indexes for backup auditing relation (in $\mathcal{H}_{\mathcal{DB}}$)
8) generate triggers on backup auditing relation (in $\mathcal{H}_{\mathcal{DB}}$),
9) generate permissions to user for backup history (in $\mathcal{H}_{\mathcal{DB}}$).

The implementation of steps 1, 5 and 6 is straightforward using what we defined in III-A. In step 2 seven types of triggers are created. First, three triggers for all operations $M$ on relation $R_i$, which insert a tuple of the form (5) in $R'_i$. Second, four triggers for operations $O$ on relation $R_i$, which insert a tuple of the form (7) in $R_{sql}$. Steps 3 and 8 are added to keep the integrity of audit relations (prevent change of audit trail), while steps 4 and 9 enable the process of moving current audit data to historic database $\mathcal{H}_{\mathcal{DB}}$. The creation of $R_{sql}$ in both databases is not part of these steps, because it is done only once at the beginning of including auditing in database.

For automation of generating objects to be possible we needed to develop naming conventions for all auditing objects, consisting of original relation name and some sort of prefix or postfix. Other parts of SQL statements are hard-coded and use information from system tables.

Parts of our code, for which we have given description here, are implemented in IBM Informix dialect and open-sourced at [12].

### C. Backup History process

As was already mentioned, Backup History process has a task of moving tuples from auditing relation in database $\mathcal{R}_{\mathcal{DB}}$ to corresponding relation in database $\mathcal{H}_{\mathcal{DB}}$, including a $R_{sql}$ relation. Process is executed in even periods of time, which in our case is one day. It connects to both databases and then repeats steps:

1) read a tuple from auditing relation in $\mathcal{R}_{\mathcal{DB}}$,
2) insert it in relation in $\mathcal{H}_{\mathcal{DB}}$,
3) delete it from $\mathcal{R}_{\mathcal{DB}}$.

For productivity purposes, instead of moving one tuple at the time it moves blocks of tuples.

## IV. Role Separation

Role separation is a way of ensuring audit trail could be trusted, by separating three roles involved in auditing process. One role defines the auditing process, i.e. what should be audited. Second role controls the auditing process, i.e. starts and stops auditing, and the third role is the one that reads the audit trail. None of these roles should not be implemented in a single person.

Our proposed subsystem partially enables role separation, by giving separate permissions to roles (called audit-definition, audit-control and audit-read) in the databases. The audit definition is done by working with data in $S_j$ relations, so CRUD permissions for these tables should be given to audit-definition role. Other roles can only read these relations. Audit control is done by toggling the master audit switch in one of the system relations, which means only the audit-control role should have CRUD permissions on that particular table, while others can have read-only access to it. And lastly, all of the audit trail in $\mathcal{R}_{\mathcal{DB}}$ and $\mathcal{H}_{\mathcal{DB}}$ database should be only read by audit-read role, while all other roles should not have any permission on them. Also, any operations on $S_j$ relations should be audited as well, by default.

The only issue remains tampering with audit objects by a database administrator role (DBA) or a database system administrator role (DBSA). However, if a DBMS supports auditing of its own and it supports role separation, it is possible to combine our method with it. Actions like changing audit triggers, disabling them or dropping audit tables, should be audited by a DBMS audit system. That way, a complete audit trails security could be achieved.

## V. Performance overhead

For the purposes of measuring performance overhead, testing was performed using the Apache JMeter [13]. In the test, each SQL statement is being executed on the testing database, by a JMeter testing threads. The testing thread executes statements via JDBC (Java Database Connectivity)

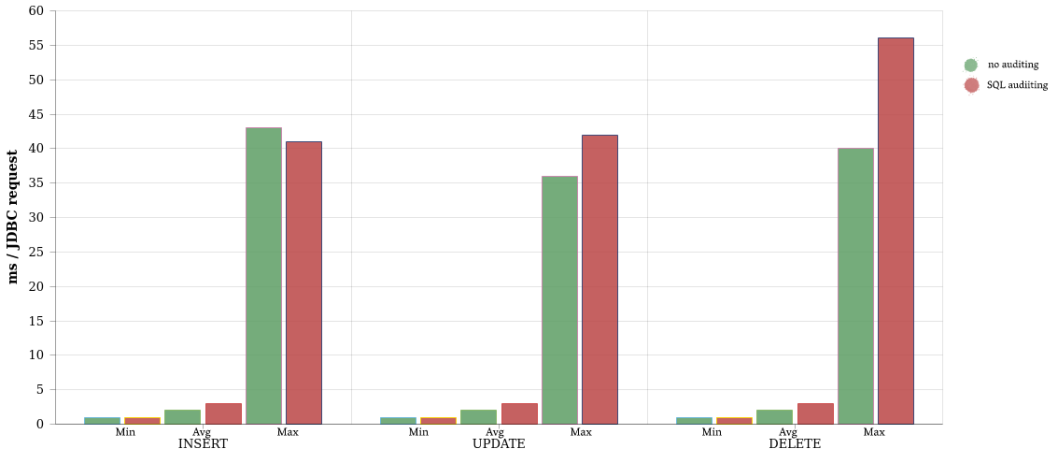


Figure 2. DML auditing test results.

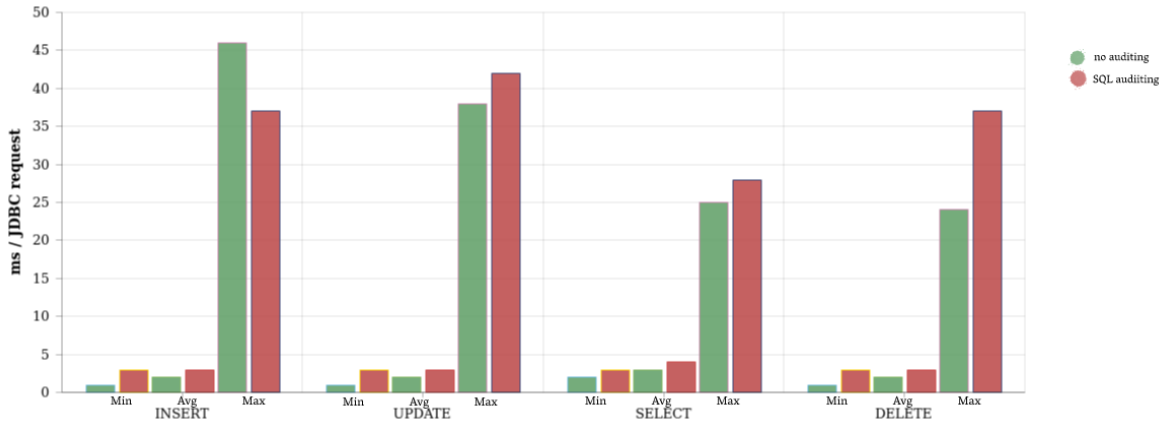


Figure 3. SQL auditing test results, auditing enters one record at a time for each executed statement.

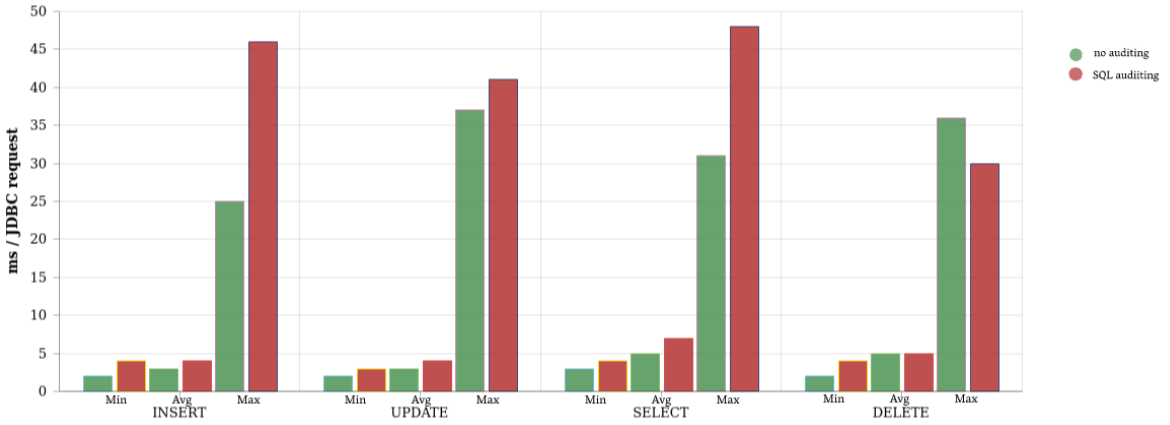


Figure 4. SQL auditing test results, auditing enters one record for the statement that effects 100 records.

requests. The duration of JDBC requests with audit on and off was measured. For DML audit, the duration of DML operations (3) was tested, while for SQL audit we record the duration of all four basic data operations (2). Testing was performed with 1 thread, executing a 1000 operations, which means that for each of the obtained testing results 1000 JDBC requests were conducted. In doing so, we record the minimum, average and maximum duration of the request.

The results of DML audit testing are shown in Fig. 2. The average overhead per operation is independent of the operation itself and increases its duration by 1 ms. Such a result is expected given that in addition to the initially executed command, another record has to be inserted. The DML audit testing did not include SELECT command for the reasons described in Chapter III.

The results of the SQL audit test are shown in Fig. 3. The first thing to notice is a higher overhead than with DML audit. The obtained results correspond to the predictions here as well. Overhead is a consequence of data entry in $R_{sql}$ (Def. 3) and steps 1-9 in more detail described in Chapter III.

Important thing to notice regarding SQL statement auditing results shown in Fig. 3 is that this measurement represents the worst-case scenario. Our testing statements were all effecting only a single record in a relation, and SQL statement auditing enters an audit record once for each executed statement. In comparison, Fig. 4 shows results of the testing with each SQL statement affects a 100 records (and one auditing record entered for each statement). Therefore, for larger operations, SQL auditing shows acceptable overhead. In both cases of SQL auditing the average overhead per audit time is from 1 to 2 ms.

Another comment should be put in place regarding the maximum duration of the operation. Such outliers would periodically emerge in working with DBMSs, with regard to relations taking additional space to store data, index rebuilding due to a B-tree expansion, etc. We did not tune the testing database in any way to mitigate these actions and influence the testing results.

## VI. Current Usage and Future Work

The audit subsystem described here is currently implemented in four major national information systems in Croatia, all of them covering the field of higher education and science

[14]–[17]. Some parts of it are also implemented in some commercial systems as well. The oldest of these systems has row level DML auditing implemented for more than a decade. Throughout this time, data collected in this manner had served as a source for many analyses - from the basic ones requested by users, to much more serious investigations.

Information systems security is increasingly important topic. Gaining knowledge about our systems' and users' behavior could not be possible without considerable amounts of data. The audit trail collected by method presented here will serve as a base for different information security-based research:

- *ad hoc* analyses of incoming audit trail and implementation of reaction mechanism to prevent overstepping of given permissions,
- *ad hoc* analyses of currently performing SQL statements and implementation of reaction mechanism to prevent unwanted queries,
- performance-related analyses of executed SQL statements and proposition for introducing further indexes,
- process mining methods, in order to recognize processes in the system and define security measures,
- on-line workload analyses, in order to forecast the needs for increased hardware resources,
- mechanisms and methods for investigation of complex, multi-user frauds,
- mechanisms for investigating social networks forming in the complex systems, etc.

Keeping in mind that audit trail is a rich source of temporal data, these datasets form a solid base for different kinds of temporal-related research as well, by forming temporal relational snapshots, temporal graphs, etc.

## VII. Conclusion

In this paper, a simple, secure, configurable, role-separated, and effective in-database auditing subsystem is presented. It consists of several objects inserted in the relational database (tables, stored procedures and triggers) being audited. These objects control and perform code generation of other database objects which, in turn, perform the row level DML operation auditing and SQL statements auditing as well. Our main contribution, the audit code generator is discussed in detail. Its implementation is open-sourced and it is quite simple, taking only several hundreds of lines of code.

The auditing subsystem described here is easily configurable and effective - each change in configuration is followed by redefining of auditing objects and new audit configuration is in effect immediately. Security and role-separation are simply enforced by the database permissions, and can be enhanced by the underlying DBMS auditing subsystem.

We have measured and shown performance overhead of such an auditing mechanism. Even though there is a significant performance overhead in general, which is quite expected in this kind of implementation, our conclusion is that possible security gain justifies this overhead. Auditing does not have to performed on all relations, however, some relations hold very sensitive data. On such data, every action should be audited. It is up to a auditor to define which relations should be audited, and in which way.

Lastly, current implementations of this audit subsystem and future work are discussed. Audit trail being a valuable source of information, there are many possible directions of future work. In that manner, this paper serves as a base for the presentation of our further research activities regarding various information security topics.

## Acknowledgment

The authors would like to thank Ms. Jasenka Anzil of University Computing Centre, University of Zagreb, for her contribution on parts of the in-database audit code generator.

O. Orel would also like to thank Dr. Slaven Zakošek of Faculty of Electrical Engineering and Computing, University of Zagreb, for his work and mentoring in implementation of Java-based code generators and parsers years ago. That joint work has inspired us to build various in-database code generators.

## References

[1] R. Ben-Natan, Implementing Database Security And Auditing. Burlington, MA: Elsevier Digital Press, 2005.
[2] T. Y. Lin, "Anomaly Detection: A Soft Computing Approach," in Proceedings of the 1994 Workshop on New Security Paradigms, Rhode Island, USA, 1994 pp. 44–53.
[3] T. F. Lunt, "A survey of intrusion detection techniques," Computers & Security, 12(4), 1993 pp. 405–418.
[4] PostgreSQL (https://www.postgresql.org/) [3.2.2021]
[5] pgAudit (https://github.com/pgaudit/pgaudit/blob/master/README.md) [3.2.2021]
[6] ApexSQL Audit (https://www.apexsql.com/sql-tools-audit.aspx) [29.1.2021]
[7] Oracle Audit Vault and Database Firewall: Datasheet (https://www.oracle.com/a/tech/docs/dbsec/avdf-datasheet.pdf) [29.1.2021]
[8] Oracle Audit Vault and Database Firewall: Tehnical report (https://www.oracle.com/a/tech/docs/dbsec/avdf20-technical-report.pdf) [29.1.2021]
[9] D. Fabbri, R. Ramamurthy and R. Kaushik, "SELECT triggers for data auditing," in 2013 29th IEEE International Conference on Data Engineering (ICDE 2013), Brisbane, QLD, 2013 pp. 1141-1152.
[10] B. Kogan and S. Jajodia, "An audit model for object-oriented databases," in Proceedings Seventh Annual Computer Security Applications Conference, San Antonio, TX, 1991 pp. 90–99.
[11] D. Flores and A. Jhumka, "Hybrid Logical Clocks for Database Forensics: Filling the Gap between Chain of Custody and Database Auditing," in 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), Rotorua, New Zealand, 2019 pp. 224-231.
[12] O. Orel, P. Udovičić and B. Bašić, Database audit generators (https://github.com/ognjenorel/ifmx-utils/tree/master/sql/audit) [29.1.2021]
[13] Apache JMeter (https://jmeter.apache.org/) [29.1.2021]
[14] Higher Education Information System (ISVU) (https://www.srce.unizg.hr/isvu) [29.1.2021]
[15] Croatian Qualifications Framework Information System (ISRHKO) (https://www.srce.unizg.hr/en/CROQF-IF) [29.1.2021]
[16] Information System for Support of Evaluation Processes (Mozvag) (https://www.srce.unizg.hr/en/mozvag) [29.1.2021]
[17] Croatian Research Information System (CroRIS) (https://www.srce.unizg.hr/en/croris) [29.1.2021]